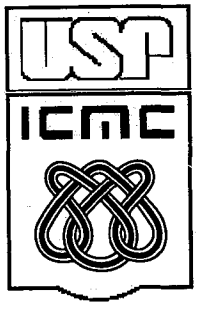
UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Escalonamento de processos: uma abordagem dinâmica e incremental para a exploração de características de aplicações paralelas

Luciano José Senger



São Carlos - SP

Escalonamento de processos: uma abordagem dinâmica e incremental para a exploração de características de aplicações paralelas

Luciano José Senger

Orientador: ***Prof. Dr. Marcos José Santana***

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional.

USP – São Carlos
Dezembro de 2004

Aluno: *Luciano José Senger*

A Comissão Julgadora:

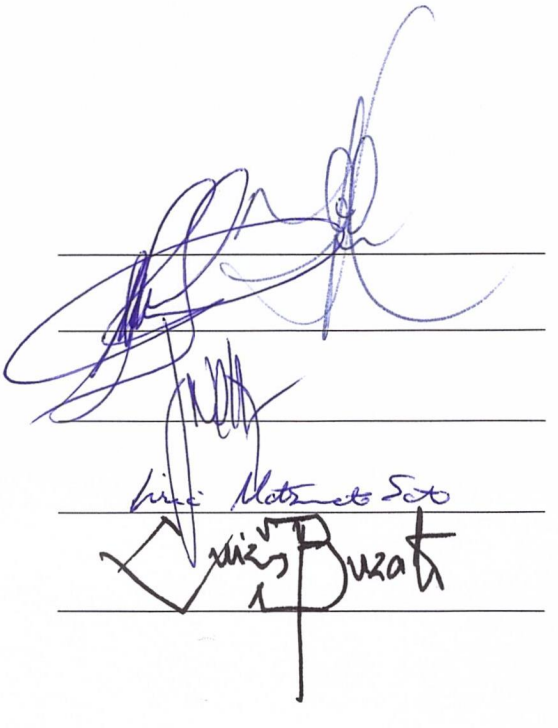
Prof. Dr. Marcos José Santana

Prof. Dr. Rodrigo Fernandes de Mello

Prof. Dr. João Cesar Neto

Profa. Dra. Liria Matsumoto Sato

Prof. Dr. Luiz Eduardo Buzato



À Lilian, minha amada esposa

Agradecimentos

Ao professor Marcos José Santana, pela orientação deste trabalho.

Aos professores e alunos do grupo de sistemas distribuídos e programação concorrente que estiveram dispostos em contribuir de alguma forma com este trabalho, principalmente durante os seminários do grupo. Agradecimento especial ao professor Rodrigo Fernandes de Mello, pelas contribuições e sugestões valiosas.

Ao professor Zhao Liang, pela ajuda com as arquiteturas de redes neurais baseadas na teoria da ressonância adaptativa.

À Warren Smith, por gentilmente fornecer seu software de predição de tempo de execução de aplicações paralelas, e à Allen Downey, por disponibilizar o código fonte de seu simulador de escalonamento para máquinas paralelas.

À Reagan Moore, Victor Hazelwood (*San Diego Supercomputer Center*), *Cornel Theory Center* e Curt Canada (*Los Alamos National Laboratory*) por fornecerem os traços de execução utilizados nesta tese.

À Patrick Worley, pela boa vontade de informar detalhes da aplicação paralela PSTSWM.

Agradecimentos especiais à Lilian, minha esposa, pelo carinho, apoio e ajuda na revisão do texto desta tese.

À CAPES e à Universidade Estadual de Ponta Grossa, pelo auxílio financeiro.

À Universidade de São Paulo, pela estrutura fornecida para a pesquisa e pelo apoio ao aluno de pós-graduação.

Resumo

Esta tese apresenta uma abordagem dinâmica e incremental para a exploração de características de execução e de submissão de aplicações paralelas visando o escalonamento de processos. Modelos de classificação e de predição de características de aplicações são construídos, utilizando algoritmos de aprendizado de máquina adaptados como ferramentas para a aquisição de conhecimento sobre a carga de trabalho. Os paradigmas conexionista e baseado em instâncias orientam a aquisição de conhecimento e os algoritmos e suas extensões permitem a atualização do conhecimento obtido, à medida que informações mais recentes tornam-se disponíveis. Esses algoritmos são implementados e avaliados utilizando informações obtidas através da monitoração da execução de aplicações paralelas e da utilização de traços de execução representativos da carga de trabalho seqüencial e paralela de diferentes centros de computação. A avaliação é conduzida visando observar o desempenho nas tarefas de aquisição de conhecimento e classificação, assim como o desempenho computacional das implementações dos algoritmos. Algoritmos de escalonamento são definidos e avaliados para observar as vantagens da utilização do conhecimento adquirido, considerando cenários de execução baseados em máquinas paralelas e sistemas distribuídos. Os resultados obtidos com este trabalho justificam a importância da utilização desse conhecimento nas decisões do software de escalonamento em sistemas computacionais distribuídos e contribuem para a definição de mecanismos adequados para a aquisição e utilização desse conhecimento em sistemas paralelos reais.

Abstract

This thesis presents a dynamic and incremental approach for exploring execution and submission characteristics of parallel applications aiming at improving process scheduling. Models for classifying and predicting applications behavior are developed, considering machine learning algorithms as tools for acquiring knowledge about workload. The neural network and instance-based paradigms guide the knowledge acquisition and the algorithms and their extensions allow knowledge updating when new information occurs. The algorithms are implemented and evaluated using information obtained from monitoring the parallel application execution and using representative sequential and parallel workload traces acquired from different computing centers. The evaluation is conducted aiming at observing the knowledge acquisition and classification tasks performance as well as the algorithms implementation computing performance. Scheduling algorithms are developed and evaluated for observing the knowledge utilization improvements, using parallel machines and distributed systems as an execution platform. The results obtained in this thesis justify the importance of employing knowledge for software scheduling decisions on distributed computing systems and allow developing suitable mechanisms for acquiring and using application knowledge in real parallel systems.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas e Siglas

AMIGO *DynAMical FlexIble SchedulinG EnvirOnment*

ART-1 Versão da rede ART que trabalha apenas com padrões representados por vetores compostos estritamente de valores binários

ART-2A Versão algorítmica da rede ART-2

ART-2 Versão da rede ART que permite a classificação de padrões de entrada representados por vetores de valores contínuos

ART *Adaptive ressonance theory*

CORBA *Common object request broker architecture*

COWs *Cluster of workstations*

DPWP *Dynamic policy without preemption*

EMMM *Execution mode machine model*

EP Elemento de processamento

FCFS *First-come, first-served*

GAS *Genetic algorithm scheduling*

IBL *Instance-based learning*

LBM *Load balance manager*

LIM *Load information manager*

MIMD *Multiple instruction multiple data*

MIPS Milhões de instruções por segundo

MISD *Multiple instruction single data*

MPI *Message passing interface*

MPP *Massive parallel processing*

NOMs *Network of machines*

NOWs *Networks of workstations*

PVM *Parallel virtual machine*

R^2 Coeficiente de determinação

RNAs Redes neurais artificiais

SIMD *Single instruction multiple data*

SISD *Single instruction single data*

TLBA *Tree load balancing algorithm*

UCP Unidade central de processamento

CAPÍTULO

1

Introdução

A ciência da computação tem se caracterizado nos últimos anos pela grande conectividade dos recursos computacionais. Essa conectividade, impulsionada pelo desenvolvimento das redes locais de computadores a partir da década de 1980, tem permitido o compartilhamento de recursos com um bom desempenho e a concepção de sistemas distribuídos (Coulouris *et al.*, 1994). Tais sistemas são caracterizados por um conjunto de recursos compartilhados através de uma rede de interconexão e pela busca de características como transparência de acesso aos recursos, tolerância a falhas, possibilidade de crescimento em escala e desempenho. Essas características, aliadas à possibilidade de um baixo custo na aquisição e manutenção, tornam viáveis a utilização dos sistemas distribuídos em vários domínios.

Um dos domínios que se beneficia das vantagens dos sistemas distribuídos é a computação paralela. Antes restrita a domínios compostos de computadores específicos com processamento paralelo, a computação paralela, em busca de melhor desempenho e uma melhor relação custo/benefício, tem sido constantemente empregada em sistemas computacionais distribuídos. Esse emprego permite associar as vantagens dos sistemas distribuídos com os objetivos da área de computação paralela. Nessa abordagem, o sistema computacional distribuído é visto pelas aplicações paralelas como uma máquina paralela virtual, com os elementos de processamento espalhados pela rede de comunicação.

Máquinas paralelas virtuais têm algumas diferenças significativas em relação às máquinas verdadeiramente paralelas (Anderson *et al.*, 1995). Máquinas verdadeiramente paralelas têm vários elementos de processamento, geralmente homogêneos, interconectados por uma rede de comunicação dedicada e de alta velocidade. Por outro lado, máquinas paralelas virtuais são formadas por elementos de processamento potencialmente heterogêneos, tanto em relação à sua configuração quanto em diferenças arquiteturais, além de uma rede de comunicação de

menor velocidade e compartilhada por diferentes usuários e aplicações, que influenciam substancialmente o desempenho do sistema. Essas diferenças criam novas necessidades para o software distribuído, muitas vezes não existentes em sistemas paralelos baseados em máquinas verdadeiramente paralelas.

Comumente, a computação paralela em sistemas distribuídos é realizada através de ferramentas de software, que permitem o desenvolvimento de aplicações paralelas, provendo recursos para a distribuição e a comunicação no ambiente distribuído. Muitas ferramentas de software foram adaptadas a partir de ferramentas de software existentes em computadores paralelos; outros ambientes surgiram especificamente para sistemas distribuídos. Essas ferramentas implementam as primitivas de comunicação, sincronização, gerência de recursos e gerência de aplicações paralelas. Exemplos dessas ferramentas são o PVM (*Parallel Virtual Machine*) (Beguelin *et al.*, 1994; Senger, 1997), o MPI (*Message Passing Interface*) (MPI Forum, 1997), e mais recentemente, as implementações do modelo de objetos da especificação CORBA (*Common Object Request Broker Architecture*) (Santos *et al.*, 2001).

Tarefas importantes na área de computação paralela, que podem ser absorvidas pela aplicação do usuário ou pelo software distribuído, são a distribuição das aplicações paralelas e suas tarefas entre os elementos de processamento e o compartilhamento dos elementos de processamento entre as aplicações paralelas (Schnor *et al.*, 1996). Esse aspecto reflete-se diretamente no desempenho do sistema, no que diz respeito à utilização dos recursos e à satisfação dos usuários. As atividades de gerenciamento de aplicações paralelas e dos recursos computacionais é chamada de escalonamento (Casavant & Kuhl, 1988). Decisões comuns que devem ser tomadas pelo software que realiza o escalonamento são quais aplicações obterão acesso aos recursos computacionais, a quantidade de recursos que serão destinados para as aplicações e a localização desses recursos. Essas decisões são afetadas por diferentes fatores, como por exemplo a carga de trabalho do sistema, a diversidade de aplicações e as necessidades dos usuários. A complexidade de fatores envolvidos em sistemas computacionais distribuídos torna o escalonamento nesses sistemas uma área desafiante e em constante pesquisa.

O escalonamento pode ser implementado internamente à aplicação paralela, internamente ao sistema operacional ou como um software em nível de aplicação. Quando implementado internamente à aplicação paralela, as decisões de escalonamento são tomadas pelo software paralelo. Isso aumenta a complexidade do código e diminui a flexibilidade das aplicações, pois a portabilidade das aplicações fica restrita ao cenário de execução considerado pelo programador. A implementação interna ao sistema operacional e como um software de escalonamento, ou ambiente de escalonamento, em nível de usuário, permite que as decisões de escalonamento sejam isoladas das aplicações paralelas. Dessa forma, as aplicações paralelas têm sua portabilidade entre diferentes sistemas melhorada e o programador fica liberado da implementação do escalonamento. Assim, cabe ao software de escalonamento a gerência dos elementos de

processamento e alguma forma de monitoração da carga do sistema, visando oferecer um serviço adequado aos usuários.

O escalonamento de aplicações paralelas tem sido amplamente estudado pela comunidade científica (Feitelson *et al.*, 1997; Souza, 2000). Vários trabalhos têm sido propostos, direcionados para diferentes plataformas computacionais e com diferentes objetivos. Grande parte dos trabalhos dessa área refletem adaptações dos conceitos já empregados em sistemas uniprocessados. Embora a pesquisa na área de escalonamento esteja substancialmente consolidada em máquinas verdadeiramente paralelas, em sistemas paralelos baseados em sistemas computacionais distribuídos, a pesquisa encontra-se em constante evolução. Conceitos e técnicas comuns em sistemas baseados em máquinas verdadeiramente paralelas muitas vezes possuem sua adeqüabilidade reduzida quando empregados em sistemas distribuídos de propósito geral. Isso indica que, no contexto de sistemas distribuídos, vários conceitos e técnicas precisam ser revisados e construídos para garantir um bom desempenho na computação.

Um ponto importante na atividade de escalonamento é o conhecimento sobre as características de execução das aplicações paralelas que são executadas no sistema. Vários autores demonstram que o escalonamento, realizado com o auxílio do conhecimento existente sobre as aplicações paralelas, produz um melhor desempenho que o escalonamento realizado sem o auxílio desse conhecimento (Harchol-Balter & Downey, 1997; Smith *et al.*, 1998b; Krishnaswamy *et al.*, 2004; Apon *et al.*, 1999; Brecht & Guha, 1996; Naik *et al.*, 1997; Sevcik, 1989). Tal conhecimento engloba a carga de trabalho do sistema computacional e o padrão de comportamento, analisado isoladamente, das aplicações seqüenciais e paralelas. Fontes comuns desse conhecimento são o usuário (ou programador) que submete a aplicação paralela, através de uma linguagem de descrição do comportamento e requisitos da aplicação, o histórico da execução de todas as aplicações e os seus perfis de execução das aplicações. Dentre essas fontes, o histórico da execução de aplicações paralelas e seus perfis de execução têm demonstrado um grande potencial na revelação de muitas informações, que servem de base para a construção desse conhecimento (Smith *et al.*, 1998b).

Apesar de vários autores apresentarem contribuições que evidenciam a importância do emprego desse conhecimento, poucos trabalhos têm sido direcionados para a aquisição e utilização desse conhecimento (Gibbons, 1997; Silva & Scherson, 2000; Corbalan *et al.*, 2001; Krishnaswamy *et al.*, 2004). Além disso, a maioria dos trabalhos são direcionados ao contexto de máquinas com processamento paralelo, onde fatores como influências externas de usuários que usam interativamente o sistema e heterogeneidade dos recursos não se fazem presentes. Essa lacuna pode também ser observada nas implementações de software de escalonamento (Kaplan & Nelson, 1994; Thain *et al.*, 2004). Embora o software de escalonamento, visando melhorar suas decisões, permita que o usuário especifique as características das aplicações paralelas, pouco esforço tem sido notado na definição de métodos práticos para obter e utilizar esse conhecimento,

em sistemas distribuídos de propósito geral e na associação de tal conhecimento com as decisões do escalonador.

O problema central estudado nesta tese de doutorado é a exploração de características de aplicações paralelas no escalonamento de processos em sistemas computacionais distribuídos. O objetivo principal é o estudo e desenvolvimento de métodos que possam adquirir conhecimento a partir das características de execução das aplicações paralelas e da carga de trabalho comum ao sistema. As características de execução estão relacionadas com a utilização dos recursos computacionais pelos processos que compõem as aplicações e as características de submissão da carga de trabalho são utilizadas para definir aplicações paralelas similares e gerar predições. Os métodos realizam o aprendizado de máquina de maneira incremental, pois permitem que o conhecimento adquirido sobre as aplicações possa ser atualizado à medida que novas informações ocorram. Nesse sentido, algoritmos de aprendizado de máquina são adaptados, implementados e utilizados para adquirir conhecimento através de informações coletadas através da monitoração de execução das aplicações e de cargas de trabalho representadas por traços execução.

Os algoritmos de aprendizado utilizados visam obter conhecimento e gerar predições de informações que sirvam aos propósitos do software de escalonamento. Nesse sentido, dois tipos de informações são analisadas: o comportamento das aplicações durante a utilização de recursos do sistema e o tempo de execução das aplicações. Aplicações seqüenciais também são consideradas pelos algoritmos de aprendizado, embora o foco principal seja a aquisição de conhecimento sobre aplicações paralelas. Observa-se que ambientes de execução direcionados para o processamento paralelo também são utilizados para a execução de aplicações seqüenciais.

Visando a utilização do conhecimento sobre as aplicações, dois algoritmos de escalonamento são propostos. O primeiro algoritmo é orientado a um cenário de execução composto de uma máquina paralela real, sendo gerenciada por um software de escalonamento único. O segundo algoritmo é orientado a um cenário de execução heterogêneo e distribuído. A avaliação desses algoritmos permite verificar o ganho obtido com a utilização do conhecimento sobre as aplicações paralelas nas decisões do software escalonador.

Visando atingir esses objetivos, o restante desta tese está organizado através de 8 Capítulos e 1 Apêndice. O Capítulo 2 apresenta a revisão da literatura sobre computação paralela e escalonamento, abordando também os trabalhos mais relevantes da área de escalonamento apoiado com a utilização de informações sobre as aplicações paralelas.

O Capítulo 3 apresenta os algoritmos de aprendizado de máquina utilizados: o algoritmo de aprendizado baseado na arquitetura de redes neurais ART-2A (Carpenter *et al.*, 1991a) e o algoritmo de aprendizado baseado em instâncias (Aha *et al.*, 1991). Nesse capítulo são descritos os detalhes dessas técnicas de aprendizado e dos algoritmos implementados.

O Capítulo 4 apresenta a carga de trabalho empregada nas tarefas de aquisição de conhecimento. Essa carga de trabalho é composta por traços de execução de diferentes centros

de computação de alto desempenho e de um conjunto de aplicações paralelas reais e sintéticas. Nesse capítulo, são descritos também detalhes do ambiente de execução utilizado para os experimentos conduzidos com as aplicações reais e sintéticas.

O Capítulo 5 descreve a forma de aquisição de informações de execução sobre as aplicações paralelas executadas através de um conjunto de tarefas, mapeadas no ambiente de execução como processos do sistema operacional. A forma de aquisição emprega técnicas de monitoração e instrumentação do sistema operacional, visando obter características mais detalhadas sobre a execução das aplicações. A implementação dessas técnicas e sua avaliação de intrusão no desempenho do sistema, empregando a execução de *benchmarks*, são também descritos nesse capítulo. Nesse capítulo, é apresentado o modelo de aquisição de conhecimento sobre o comportamento de aplicações paralelas na utilização de recursos. Esse modelo emprega um algoritmo de aprendizado baseado na arquitetura de redes neurais ART-2A , que utiliza as informações providas pelos mecanismos de monitoração. Estados de utilização de recursos das aplicações e um diagrama de transição de estados são identificados por esse modelo, permitindo classificar as aplicações paralelas quanto ao seu comportamento na utilização de recursos. Um algoritmo de rotulação é proposto, com o objetivo de identificar os estados identificados pelo modelo. Esse modelo é definido e aplicado em um conjunto de aplicações paralelas reais e sintéticas, e os resultados obtidos sobre o desempenho de classificação são descritos.

No final do Capítulo 5, é apresentada a implementação e avaliação desse modelo de aquisição de conhecimento junto ao núcleo do sistema operacional, visando a classificação de processos. Nesse sentido, são discutidas as modificações necessárias na implementação do modelo e os resultados obtidos com experimentos, que têm como objetivo principal a avaliação do desempenho computacional do algoritmo proposto. A avaliação de desempenho, realizada através da execução de *benchmarks*, permite verificar a intrusão do modelo no desempenho do sistema computacional.

O Capítulo 6 apresenta o modelo de aquisição de conhecimento que tem como objetivo principal a predição de características de execução de aplicações paralelas, através do histórico de execução de todas as aplicações paralelas executadas no sistema. Esse modelo explora as características da carga de trabalho comum ao sistema, através da utilização de atributos das aplicações submetidas para encontrar conjuntos de aplicações que compartilham as mesmas características. Esse modelo é aplicado em um conjunto de traços de execução de ambientes de produção relevantes, e os resultados são avaliados e comparados com outras técnicas apresentadas na literatura. Nesse sentido, o desempenho computacional e de aquisição de conhecimento do algoritmo de aprendizado são avaliados.

No Capítulo 7, são apresentados os dois algoritmos de escalonamento propostos. O primeiro algoritmo, denominado `BACKFILL-IBL`, visa melhorar as decisões do software de escalonamento em máquinas paralelas através da utilização de predições sobre o tempo de execução das aplicações paralelas. Esse algoritmo é avaliado através de simulação e comparado

com os resultados obtidos com outros algoritmo de escalonamento comuns ao mesmo cenário de execução. O segundo algoritmo, denominado GAS, utiliza técnicas de algoritmos genéticos para encontrar os recursos mais adequados para a execução das aplicações paralelas. Esse algoritmo visa utilizar o conhecimento obtido através dos modelos propostos no Capítulos 5 e 6. O algoritmo GAS é avaliado e comparado com outros algoritmos de escalonamento através de simulação, e os resultados obtidos são descritos.

O Capítulo 8 apresenta as considerações e conclusões finais obtidas com o desenvolvimento deste trabalho de doutorado.

O Apêndice A apresenta detalhes dos resultados obtidos e suas tabelas estatísticas, que descrevem a análise dos resultados obtidos com os experimentos.

CAPÍTULO

2

Computação Paralela e Escalonamento de Processos em Aplicações Paralelas

2.1 Considerações iniciais

Este capítulo apresenta a revisão dos conceitos e os trabalhos relacionados aos objetivos desta tese de doutorado. Nesse sentido, a Seção 2.2 apresenta os conceitos da área de computação paralela, enfatizando a utilização de sistemas computacionais distribuídos como ambiente de execução. A Seção 2.3 apresenta os conceitos e trabalhos da área de escalonamento de processos em aplicações paralelas. A Seção 2.4 é dedicada aos trabalhos correlatos na área de escalonamento e avaliação de desempenho, cujo o objetivo é adquirir conhecimento sobre as aplicações que são executadas no sistema computacional. A Seção 2.5 apresenta as considerações finais sobre os assuntos abordados neste capítulo.

2.2 Computação paralela distribuída

Computação paralela ou processamento paralelo constitui-se na exploração de eventos computacionais concorrentes, através de unidades de processamento que cooperam e comunicam-se (Almasi & Gottlieb, 1994). O objetivo principal do processamento paralelo é a busca de melhor desempenho para aplicações computacionais que necessitam de maior potência computacional, muitas vezes não alcançada em arquiteturas seqüenciais.

A exploração de eventos concorrentes pode ser realizada em diferentes níveis. Em um nível mais próximo ao hardware, o paralelismo pode existir nas unidades funcionais que

compõem a UCP (Unidade Central de Processamento). Nesse nível, a granulação[1] empregada é fina, com o paralelismo sendo explorado através de instruções de máquina que são executadas em paralelo. Essa forma de execução paralela tem sido empregada com sucesso em arquiteturas de hardware que têm múltiplas unidades funcionais e empregam técnicas de *pipeline*[2]. Outra maneira de obter paralelismo em nível de instruções de máquina é através da utilização de ambientes de paralelização automática, onde o compilador é responsável por gerar um programa paralela a partir de um programa seqüencial.

Em um nível intermediário, o paralelismo pode ser explorado através de procedimentos ou subrotinas de programas paralelos que são atribuídos aos elementos de processamento e executados concorrentemente. Geralmente, o paralelismo através de procedimentos necessita do suporte explícito de linguagens de programação paralela.

Embora seja possível aumentar o desempenho no processo computacional utilizando o paralelismo em níveis de instruções de máquina e procedimentos, uma solução altamente empregada é a utilização de granulação grossa. Nesse nível mais alto, o trabalho a ser realizado é dividido em processos ou tarefas, que são executadas concorrentemente. Seguindo essa idéia, uma aplicação paralela é um conjunto de processos que interagem entre si para realizar um determinado trabalho.

A construção de aplicações paralelas pode ser orientada por dois modelos principais: o paralelismo funcional e o paralelismo de dados (Quinn, 1994). No paralelismo funcional, os processos da aplicação realizam diferentes funções, cada um responsável por uma etapa do trabalho. O paralelismo funcional é também conhecido como MPMD (*Multiple Program Multiple Data*). Um modelo mais simples é empregado no paralelismo de dados, onde os processos têm a mesma função operando sobre dados diferentes. O paralelismo de dados é referenciado como SPMD (*Simple Program Multiple Data*). A aplicação de um determinado modelo está fortemente relacionado com o problema a ser resolvido e, de acordo com a solução, os dois modelos podem ser considerados concomitantemente. Além disso, essa escolha é influenciada pelo hardware que será empregado. Características como modos de execução, organização da memória, heterogeneidade dos recursos e desempenho da rede de interconexão afetam diretamente o desenvolvimento de aplicações e a granulação do paralelismo empregada.

O suporte em hardware para a computação paralela pode ser provido por diferentes arquiteturas. Recentemente, sistemas baseados em computadores com memória distribuída têm apresentado desempenho excelente na execução de muitas aplicações que empregam intensivamente os recursos computacionais, com a promessa de crescimento em sua potência computa-

[1]Granulação ou granulosidade (*granularity*) representa o tamanho da unidade de trabalho destinada aos processadores (Almasi & Gottlieb, 1994), sendo diretamente relacionada ao hardware e medida em número de instruções.

[2]A técnica *pipeline* representa uma forma de paralelismo temporal, caracterizado quando existe a execução de eventos sobrepostos no tempo. A tarefa a ser executada é dividida em subtarefas, cada uma destas sendo executada por um estágio de hardware especializado, que trabalha de maneira concorrente com os demais estágios envolvidos na computação.

cional mantendo uma boa relação custo/benefício. Computadores de memória distribuída estão disponíveis através de máquinas com processamento altamente paralelo (MPP - *Massive Parallel Processing*), onde um número expressivo de elementos de processamento são interligados por uma rede de comunicação de alto desempenho. Outra maneira de se obter uma máquina com memória distribuída é através da utilização de sistemas distribuídos, onde computadores de propósito geral interligados por uma rede de comunicação e equipados com software distribuído atuam como uma máquina paralela virtual.

Características comuns em sistemas distribuídos, como transparência de acesso aos recursos computacionais, segurança, confiabilidade e interoperabilidade têm sido cada vez mais almejadas na área de computação paralela. Esses fatores, aliados aos problemas de custo e manutenção das máquinas paralelas, têm direcionado a utilização de sistemas distribuídos como uma plataforma distribuída para a execução de aplicações paralelas.

A computação paralela realizada em sistemas computacionais distribuídos, ou computação paralela distribuída, é uma área desafiante. Técnicas e modelos comuns para construção, partição e mapeamento de programas paralelos utilizados no passado, muitas vezes têm a sua viabilidade reduzida em sistemas distribuídos. Características intrínsecas desses sistemas, como heterogeneidade, fraco acoplamento dos recursos, presença de aplicações com diferentes necessidades computacionais e satisfação de usuários que utilizam interativamente o sistema, criam necessidades adicionais para o software paralelo.

2.2.1 Arquiteturas paralelas

As arquiteturas paralelas podem ser organizadas através da taxonomia de Flynn & Rudd (1996). De acordo com essa taxonomia, as arquiteturas são classificadas através dos fluxos de dados e instruções em quatro classes: SISD, SIMD, MISD e MIMD. As máquinas SISD (*Single Instruction Single Data*), tradicionalmente encontradas em computadores que têm apenas uma UCP, apresentam um fluxo único de instruções operando sobre um único fluxo de dados. A categoria SIMD (*Single Instruction Multiple Data*) engloba máquinas com processamento vetorial e paralelo (empregam grande número de processadores específicos), onde o fluxo de instruções é único, operando um fluxo múltiplo de dados. Máquinas MISD (*Multiple Instruction Single Data*) representam arquiteturas onde um fluxo múltiplo de instruções opera sobre um fluxo múltiplo de dados. Sem representantes na prática, essa classe orienta apenas componentes específicos da arquitetura, como por exemplo os *pipelines*, não representando a filosofia de operação global da arquitetura. Finalmente, as máquinas MIMD (*Multiple Instruction Multiple Data*) representam os computadores paralelos que têm um múltiplo fluxo de instruções, operando sobre múltiplos fluxo de dados.

O paradigma MIMD pode ser obtido através da utilização de vários EPs interconectados por uma rede de comunicação de alta velocidade. Diferente de máquinas de processamento

paralelo que seguem o paradigma SIMD, máquinas MIMD têm um número menor de EPs, mas com maior potência computacional (Almasi & Gottlieb, 1994). Outra característica de máquinas MIMD é a possibilidade de crescimento em escala, de modo que a capacidade de processamento pode ser incrementada de acordo com as necessidades dos usuários.

Atualmente, máquinas MIMD podem ter instruções da classe SIMD para aplicações específicas que utilizam várias mídias e que envolvem a utilização massiva de elementos gráficos. Por exemplo, a família de processadores Intel, iniciando com o processador Pentium II, tem instruções da classe SIMD adicionadas, com o objetivo de melhorar o desempenho de aplicações que necessitam que a mesma operação seja executada em dados diferentes. Essas instruções são chamadas de MMX(*MultiMedia eXtension*) (Wilkinson & Allen, 2004).

Em máquinas MIMD, a memória pode ser compartilhada ou distribuída entre os processadores da máquina paralela. Exemplos de máquina MIMD com memória distribuída são o Intel Paragon e o IBM CM-5 (Anderson *et al.*, 1995). Soluções de software geralmente empregados por esses sistemas são o PVM e implementações da especificação MPI (vide Seção 2.2.2), que permitem o desenvolvimento de aplicações paralelas portáteis entre plataformas diferentes, através do modelo de passagem de mensagens.

Existe atualmente uma convergência entre máquinas MPP e computadores pessoais e estações de trabalho. Computadores pessoais e estações de trabalho estão sendo interligados por redes de comunicação cada vez mais velozes, enquanto que máquinas MPP têm sido construídas empregando processadores idênticos àqueles empregados em estações de trabalho. Um aspecto interessante dessa convergência é o tempo de projeto de máquinas paralelas. Máquinas paralelas têm apresentado uma defasagem de tempo, em relação ao microprocessador empregado, de um ano em relação às estações de trabalho (Tabela 2.1). Como o avanço na tecnologia de microprocessadores permite uma melhoria de desempenho de 50 a 100% por ano, uma defasagem dessa magnitude pode distanciar em mais do dobro o desempenho dos processadores aplicados em estações de trabalho em relação àqueles empregados em máquinas MPP (Anderson *et al.*, 1995).

Tabela 2.1: Comparação entre máquinas MPP e estações de trabalho com microprocessadores equivalentes

Máquina Paralela (MPP)	Processador	Lançamento	Estação de Trabalho Equivalente
T3D	150 MHz Alpha	1993-94	1992-93
Paragon	50 MHz i860	1992-93	$\approx$ 1991
CM-5	32 MHz SS-2	1991-92	1989-90

Além da defasagem com relação ao mercado de microprocessadores, outra fraqueza das máquinas MPP é o nicho ocupado por elas. Embora sejam altamente atrativas para a implementação de aplicações em domínios específicos, máquinas MPP têm se mostrado inapropriadas para

outras aplicações, como por exemplo servidores de arquivos ou aplicações interativas. Outra desvantagem na utilização de máquinas MPP é o custo elevado de aquisição, manutenção e atualização.

Apesar dos problemas de custo, defasagem de tecnologia e nicho de aplicação, as máquinas MPP apresentam duas características importantes para a computação paralela: visão global de uma máquina paralela e alto desempenho da rede de interconexão. A alta velocidade de comunicação deriva do emprego de roteadores de mensagens rápidos e implementados em uma única pastilha. A interface de rede é geralmente conectada ao barramento comum entre a memória e o processador, ao invés de ser conectada ao barramento padrão de E/S. Essa forma de implementação permite a redução significativa da latência de comunicação. A visão de um sistema global significa que a aplicação paralela executa em uma coleção de EPs como uma única entidade, ao invés de uma coleção arbitrária de processos distribuídos. Esse agrupamento de processadores provê facilidades para o gerenciamento dos recursos e aplicações.

Redes de estações de trabalho e computadores pessoais buscam atualmente atingir tais características. Trabalhos têm sido desenvolvidos para prover valores menores de latência e sobrecarga de comunicação, através do desenvolvimento de software de gerenciamento de comunicação em nível de usuário, como o modelo de mensagens ativas (von Eicken *et al.*, 1992), e implementações de ferramentas de software (vide Seção 2.2.2) otimizadas para diferentes arquiteturas (Chiola & Ciaccio, 2000; Lauria & Chien, 1997). A visão global de um sistema único em sistemas distribuídos é ainda um desafio. Várias ferramentas têm sido propostas para o gerenciamento de aplicações paralelas e recursos, embora ainda exista a necessidade de ferramentas com maior flexibilidade e transparência para o acesso aos recursos computacionais.

Sistemas computacionais compostos de computadores pessoais e estações de trabalho direcionados à computação paralela distribuída são referenciados como COWs (*Cluster of Workstations*) (Anastasiadis & Sevcik, 1997; Chiola & Ciaccio, 2000; Zhang *et al.*, 2000), NOWs (*Networks of Workstations*) (Anderson *et al.*, 1995; Arpaci *et al.*, 1995) e NOMs (*Network of Machines*) (Zaki *et al.*, 1995).

Arranjos computacionais (*clusters*) são sistemas computacionais de alto desempenho compostos por computadores interligados por uma rede local (Mello, 2003). Um arranjo pode ser dedicado para a solução de um mesmo problema computacional ou ser utilizado como uma máquina paralela compartilhada entre vários usuários. Esses arranjos visam fornecer ao programador e ao usuário a visão de um sistema computacional único, como uma máquina paralela real, mas com custo de aquisição e manutenção mais reduzido.

Sistemas computacionais baseados em redes de estações de trabalho (NOWs) têm como objetivo aproveitar os recursos computacionais existentes em uma rede de computadores de propósito geral. Nesse sentido, os computadores são empregados por usuários que utilizam interativamente estações de trabalho e, concomitantemente, por aplicações paralelas. Cabe ao

software de escalonamento gerenciar adequadamente os recursos distribuídos e as requisições dos usuários do sistema, de forma eficiente (Anderson *et al.*, 1995).

Máquinas paralelas (MPP), arranjos computacionais (COWs) e sistemas baseados em redes de estações de trabalho (NOWS) podem ser interconectados através de redes não locais e geograficamente distantes, formando um aglomerado de recursos computacionais em grande escala. Esse aglomerado de recursos, chamado de sistemas computacionais em grade (*grid computing*), permite que recursos computacionais distribuídos sejam compartilhados remotamente (Thain *et al.*, 2004). Sistemas computacionais em grade devem implementar políticas de acesso e segurança para os recursos (Foster & Kesselman, 1997).

A utilização de computadores paralelos, computadores pessoais e estações de trabalho, de modo a obter um grande sistema computacional, constitui um sistema heterogêneo. Essa heterogeneidade surge em níveis arquiteturais e configuracionais. Na heterogeneidade arquitetural, os EPs têm diferentes arquiteturas, com diferentes formas de representação de dados e potência computacional. A heterogeneidade configuracional surge quando existem EPs de mesma arquitetura, porém com diferentes capacidades de processamento, quantidade de memória principal e diferentes latências na rede de comunicação.

Ekmecic *et al.* (1996) apresentam uma taxonomia para sistemas computacionais heterogêneos, chamada de EM^3 (EMMM *Execution Mode Machine Model*). Nessa taxonomia, sistemas heterogêneos são classificados em duas direções ortogonais: modo de execução e modelo da máquina. O modo de execução é caracterizado pelo tipo de paralelismo encontrado na arquitetura (por exemplo MIMD ou SIMD). Esse modo ainda pode ser organizado em temporal, quando alterna os modos de execução através de instruções de máquina, ou espacial, quando os modos de execução são diferenciados por unidades de hardware. O modelo da máquina define as diferenças entre a arquitetura e o desempenho da máquina, por exemplo uma estação de trabalho e um computador pessoal que têm diferentes arquiteturas. Além disso, duas máquinas com a mesma arquitetura, mas com relógios (*clocks*) e/ou quantidades de memória principal diferentes produzem desempenhos diferentes e também são considerados como modelos de máquina diferentes. A heterogeneidade baseada em modelos de máquina é naturalmente espacial. A classificação dos sistemas heterogêneos é realizada através da contagem do número de modos de execução (EM-*execution modes*) e do número de modelos de máquina (MM-*machine models*) (Tabela 2.2).

Tabela 2.2: Taxonomia de Ekmecic *et al.* (1996) para sistemas computacionais heterogêneos

Classe	Descrição	Exemplo
SESM	modo de execução único e modelo de máquina único	MPP com EPs homogêneos
SEMM	modo de execução único e modelos de máquina múltiplos	arquiteturas NOW
MESM	modos de execução múltiplos e modelo de máquina único	máquinas paralelas específicas
MEMM	modos de execução e modelos de máquina múltiplos	máquinas paralelas e NOWs

A heterogeneidade nos recursos introduz uma grande variação da potência computacional. Essa variação, aliada à possibilidade de utilização de diferentes arquiteturas traz novas dificuldades que precisam ser gerenciadas pelo software paralelo. Modelos de avaliação de desempenho e índices de desempenho tradicionais em sistemas homogêneos têm sua viabilidade reduzida em sistemas heterogêneos. Além disso, o software paralelo necessita incorporar mecanismos eficazes para a gerencia dos recursos que são, potencialmente, distintos sob as características de arquitetura e configuração. Uma solução empregada para viabilizar a computação paralela em sistemas heterogêneos distribuídos é a utilização de ferramentas de software que permitam a distribuição de processos e a gerência de recursos de modo transparente, liberando o programador do conhecimento dos detalhes envolvidos. A próxima seção trata dessas ferramentas, disponíveis para vários sistemas computacionais distribuídos heterogêneos, permitindo a utilização orquestrada de várias arquiteturas, como computadores pessoais, estações de trabalho e máquinas MPP, interligados através de tecnologias diferentes de comunicação.

2.2.2 Ferramentas de software

O modelo arquitetural proposto por von Neuman assume que o processador tem a capacidade de executar seqüências de instruções (Foster, 1995). Uma instrução pode especificar, além de várias operações aritméticas, o endereço de dados a serem lidos da memória e o endereço da próxima instrução que será executada. Na programação seqüencial, esses recursos são empregados através de uma linguagem de montagem, ou, de uma maneira mais usual, através de linguagens de alto nível como C, Pascal ou Fortran. Essas linguagens permitem o uso de abstrações (por exemplo *if*, *else*, *while*), que são traduzidas automaticamente em código executável pelo compilador.

A programação paralela necessita de recursos não disponíveis diretamente nessas linguagens. Essa necessidade é gerada pela inclusão de novas fontes de complexidade à programação seqüencial. Na programação paralela, são necessários mecanismos para definir quais tarefas serão executadas concorrentemente, mecanismos para a ativação e finalização dessas tarefas e métodos para coordenar a interação entre elas.

Segundo Almasi & Gottlieb (1994), as ferramentas para a construção de programas paralelos podem ser organizadas em três grupos: ambientes de paralelização automática, linguagens concorrentes e extensões paralelas para linguagens imperativas [3].

[3] As linguagens de programação podem ser classificadas como imperativas ou declarativas. As linguagens imperativas representam a maioria das linguagens de programação hoje empregadas, como por exemplo C, Fortran e Pascal. Essas linguagens caracterizam-se pela forte ligação com as arquiteturas seqüenciais e pela seqüência de comandos que regem a manutenção dos dados utilizados pelo programa. As linguagens declarativas são aquelas que enfatizam o problema em si, e não em qual arquitetura o programa será executado. Exemplos de linguagens declarativas são as linguagens para programação funcional (como orientada a fluxo de dados) e para programação lógica (como Prolog).

Os ambientes de paralelização automática tentam absorver o paralelismo existente em um código fonte, escrito geralmente em uma linguagem imperativa, para que seja gerado, de maneira automática, um programa paralelo. A grande vantagem dessa abordagem é a utilização de códigos seqüencias legados, reduzindo o trabalho do programador.

As linguagens concorrentes permite a criação de programas paralelos através de construções próprias da linguagem. Um exemplo dessa categoria é a linguagem *Occam*, baseada no paradigma CSP, que permite a criação de aplicações paralelas com o paralelismo procedural em máquinas com memória distribuída (Almasi & Gottlieb, 1994).

Extensões paralelas para linguagens imperativas são bibliotecas que permitem a criação, comunicação e sincronismo entre processos. Essas extensões são implementadas geralmente através de bibliotecas de passagem de mensagens, que viabilizam a computação paralela em sistemas de memória distribuída. Exemplos dessas extensões são o PVM e o MPI.

O PVM (*Parallel Virtual Machine*) é um conjunto de bibliotecas e ferramentas de software que permite a criação de um sistema computacional dinâmico, flexível e de propósito geral (Beguelin *et al.*, 1994; Sunderam *et al.*, 1994; Senger, 1997). Desenvolvido inicialmente para estações de trabalho UNIX, hoje o PVM encontra-se implementado para uma variedade de plataformas, desde computadores pessoais executando sistemas Linux e Windows, até máquinas com processamento paralelo. As características principais do PVM são:

- **coleção de máquinas (*host pool*) configurada pelo usuário:** as aplicações paralelas são executadas em um conjunto de máquinas configuradas, dinamicamente, pelo usuário. O usuário tem total controle para a criação e manutenção de sua máquina paralela virtual;

- **transparência de acesso ao hardware:** a aplicação pode tratar o hardware como uma coleção de elementos de processamento virtuais. Assim, a aplicação pode atribuir tarefas para arquiteturas mais apropriadas, sem a necessidade de distinção ou adaptação nos métodos de criação e comunicação entre tarefas;

- **computação baseada em processos:** a unidade de paralelismo PVM é uma tarefa, que altera sua execução entre computação e comunicação. É possível a execução de mais de uma tarefa em um elemento de processamento virtual e essas tarefas são mapeadas localmente através de processos do sistema operacional dos EPs (vide Seção 2.3.1);

- **paradigma de passagem de mensagens e memória global associativa:** as tarefas que estão sendo executadas cooperam entre si através da troca de mensagens ou através de uma memória global compartilhada associativa [4];

[4] A partir da versão PVM 3.4, é possível também a utilização do modelo de memória associativa, similarmente ao modelo considerado pelo ambiente LINDA (McBryan, 1994). Seguindo essa idéia, a comunicação e sincronização entre tarefas é possível através de operações de escrita e leitura em uma memória associativa. Outras funcionalidades existentes a partir dessa versão são: os manipuladores de mensagens (*message handlers*), que permitem o disparo automático de rotinas da aplicação do usuário em resposta ao recebimento de mensagens; os contextos de

- **utilização de ambientes heterogêneos:** o PVM oferece suporte à heterogeneidade em nível das arquiteturas dos elementos de processamento virtuais, assim como em nível de rede de comunicação. Assim, mensagens podem ser trocadas entre máquinas com diferentes formas de representação de dados.

O modelo computacional PVM considera que uma aplicação é composta de várias tarefas, de forma que cada tarefa é responsável por uma parte do trabalho a ser realizado. No PVM, a construção de uma aplicação paralela pode ser feita através de 3 métodos: paralelismo funcional, paralelismo de dados e o paralelismo híbrido, com algumas tarefas da aplicação seguindo o paralelismo de dados e outras o paralelismo funcional.

A implementação PVM é composta por uma biblioteca de comunicação e por uma coleção de processos *daemon*, que são executados nos elementos de processamento virtuais. A biblioteca de comunicação disponibiliza para a aplicação paralela, através de uma interface de programação, primitivas de comunicação e sincronização, enquanto os processos *daemon* atuam na gerência da máquina paralela virtual e no roteamento de mensagens. Em arquiteturas baseadas em estações de trabalho, o PVM emprega a pilha de protocolos TCP/IP para prover a comunicação na rede de comunicação (Souza *et al.*, 1997). Em máquinas multiprocessadas com memória distribuída, as primitivas de comunicação são mapeadas diretamente nas chamadas nativas do sistema. Em máquinas multiprocessadas com memória compartilhada, as primitivas são implementadas utilizando *buffers* residentes em memória.

O MPI (*Message Passing Interface*) é uma especificação de rotinas de comunicação para ambientes de passagem de mensagens, criado por um fórum de pesquisadores da área de computação de alto desempenho. Diferente do PVM, que constitui uma biblioteca de comunicação, o MPI é uma especificação que define como será a interface de programação e a semântica das rotinas de comunicação (MPI Forum, 1997).

O objetivo principal do MPI é permitir que aplicações paralelas sejam portáteis entre diferentes plataformas, sejam máquinas MPP ou sistemas distribuídos. Além disso, o MPI oferece a possibilidade de executar, de modo transparente, em sistemas heterogêneos [5]. Assim, diferentes implementações podem explorar as diferentes características da arquitetura alvo. Exemplos de implementações são: MPICH [6], CHIMP [7], LAM [8], e UNIFY [9]. Existem também outras implementações específicas para as máquinas paralelas, por exemplo Cray T3D, IBM SP-2, NEC Cinju e Fujitsu AP1000.

---

mensagens, que fornecem mecanismos para implementar escopos para mensagens, evitando a perda ou recebimento indevido de mensagens; e os grupos estáticos, que visam diminuir o impacto no desempenho criado por servidores dinâmicos de tarefas (Senger & de Gouveia, 2000).

[5] Entretanto, o MPI não proíbe que implementações sejam destinadas apenas para sistemas homogêneos e não define que as implementações existentes sejam interoperáveis (MPI Forum, 1997).

[6] ftp://ftp.info.mcs.anl.org/pub/mpi (visitado em 04/09/2001)

[7] ftp://ftp.epcc.ed.ac.uk/pub/chimp/release/chimp.tar.Z (visitado em 04/09/2001)

[8] ftp://ftp.tbag.osc.edu/pub/lam (visitado em 04/09/2001)

[9] ftp://ftp.erc.msstate.edu/unify (visitado em 04/09/2001)

O padrão MPI possui várias rotinas para o envio de mensagens, suporte para grupo de processos e criação de contextos para mensagens, e as aplicações paralelas podem ser escritas nas linguagens C e Fortran. Diferente do PVM que possui basicamente 3 semânticas [10] para comunicação entre processos, o MPI possui várias formas de comunicação entre processos, que podem ser otimizadas para a arquitetura alvo.

Outra abordagem para o desenvolvimento de aplicações paralelas é a utilização de objetos distribuídos. A grande motivação para essa abordagem é a possibilidade de reuso do código paralelo, que muitas vezes não é possível através do emprego restrito de ambientes de passagem de mensagens. Um exemplo de *middleware* para a gerência de objetos distribuídos é o CORBA (Xu & He, 2000).

A arquitetura CORBA (*Common Object Request Broker Architecture*) é uma especificação da OMG (*Object Management Group*), que permite a execução de aplicações distribuídas construídas de acordo com o paradigma de orientação a objetos. Nesse contexto, uma aplicação CORBA é constituída de um conjunto de componentes de software independentes, chamados de objetos CORBA. A infraestrutura de comunicação entre os objetos é gerenciada pelo gerente de requisições ORB (*Object Request Broker*), que permite a invocação direta entre os objetos. As operações que um objeto pode realizar são especificadas através de uma linguagem de definição de interface (IDL).A arquitetura CORBA é independente da linguagem na qual os objetos são implementados. Por exemplo, um objeto pode ser implementado na linguagem C++ e pode ser chamado por um cliente implementado na linguagem Java.

O paralelismo em aplicações CORBA pode ser obtido no escopo dos objetos, com objetos implementados através de várias *threads*. Essa forma de paralelismo não requer modificações na especificação CORBA e permite a exploração de eventos concorrentes em vários processadores, que têm acesso a mesma memória compartilhada. Entretanto, essa forma de paralelismo é limitada, pois, à medida que cresce o número de objetos sendo executados na mesma máquina, crescem também problemas de disputa durante o acesso à memória principal.

Implementações recentes exploram, através de extensões na interface de definição e mecanismos assíncronos de invocação de objetos, o paralelismo existente entre objetos residentes em diferentes máquinas. Exemplos dessas implementações são o Cobra (Beaugendre *et al.*, 1997) e o PARDIS (Keahey & Gannon, 1997), que permitem o desenvolvimento de aplicações paralelas de acordo com o modelo SPMD, e o PCB (Xu & He, 2000), que permite o desenvolvimento de aplicações orientadas aos modelos SPMD e MPMD. Com o mesmo objetivo, Santos (2001) apresenta, através de modificações em uma implementação da especificação CORBA, uma extensão eficiente dos mecanismos de distribuição de processos e sua integração com o software de escalonamento, sem alterações na interface de definição. Esse trabalho confirma a viabilidade da utilização do modelo CORBA para o desenvolvimento de aplicações paralelas, demonstrando

[10] *send* bloqueante assíncrono, *receive* bloqueante assíncrono e *receive* não bloqueante.

que o desempenho alcançado, empregando implementações eficientes desse modelo, é comparável ao desempenho obtido com ferramentas construídas especificamente para a computação paralela (Santos *et al.*, 2001).

2.2.3 Medidas de desempenho

As medidas de desempenho surgem da necessidade de quantificar o desempenho obtido em um sistema computacional. Essas medidas estão diretamente relacionados com os objetivos da avaliação e têm sua adeqüabilidade dirigida para diferentes situações.

Uma medida comumente empregada na área de computação paralela é o *speedup*. Essa medida reflete o ganho no desempenho na execução de uma aplicação paralela utilizando P elementos de processamento. Formalmente, o *speedup* em sistemas homogêneos (i.e., onde todos os EPs têm a mesma organização arquitetural e mesma potência), pode ser definido como (Almasi & Gottlieb, 1994):

$$S(P) = \frac{Tseq}{Tpar} \tag{2.1}$$

Similarmente, a *eficiência* (Ep) obtida com o processamento paralelo, utilizando P elementos de processamento, pode ser definida como:

$$E(P) = \frac{S(P)}{P} \tag{2.2}$$

O *speedup* ideal é igual ao número P de elementos de processamento [11]. Exemplificando, uma aplicação seqüencial que gasta 300 unidades de tempo para ser executada em um único EP e 100 unidades de tempo na execução de sua versão paralela em 4 EPs, possui um *speedup* igual a 3 e a *eficiência* obtida é igual a 0, 75, ou 75%. Nesse caso, o *speedup* ideal seria igual a 4 (Figura 2.1).

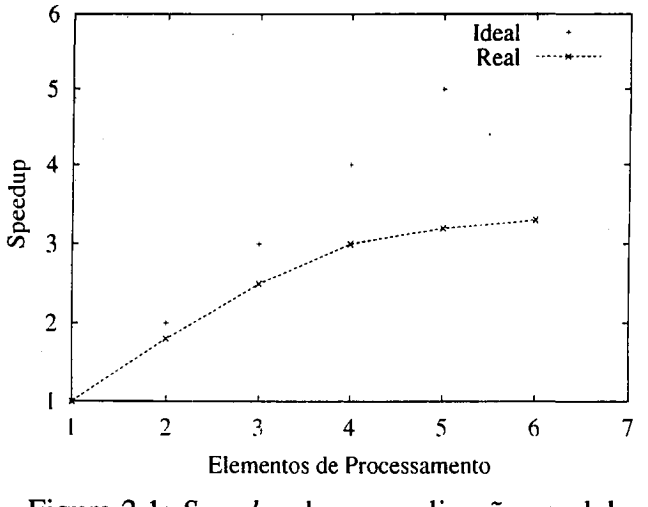


Figura 2.1: *Speedup* de uma aplicação paralela

[11] Em algumas situações, o *speedup* pode ser maior que o número de elementos de processamento considerados, situação que é chamada de anomalia de *speedup* ou *speedup* super-linear (Foster, 1995).

O *speedup* é geralmente inferior ao ideal devido à parcela do código que deve ser obrigatoriamente executada seqüencialmente e ao tempo de processamento envolvido para a criação, comunicação e sincronização entre processos. A *Lei de Amdahl* define que o *speedup* máximo ($S_{max}(P)$) que pode ser alcançado, utilizando P elementos de processamento e uma aplicação paralela que possui uma fração f ($0 \leq f \leq 1$) de código que deve ser, obrigatoriamente, executado seqüencialmente é:

$$S_{max}(P) \leq \frac{1}{f + \frac{1-f}{p}} \tag{2.3}$$

Isso indica que um número pequeno de operações seqüenciais pode limitar significantemente o *speedup* obtido através da execução paralela. Por exemplo, se a aplicação paralela possui uma parcela de 10% de operações que necessitam ser executadas seqüencialmente, o *speedup* máximo que pode ser obtido é igual a 10, não importando a quantidade de EPs empregados (Quinn, 1994). Entretanto, essa lei não contempla certas aplicações paralelas, onde a fração seqüencial do programa é reduzida à medida que o tamanho do problema aumenta. Esse fenômeno, conhecido como *Efeito Amdahl*, permite que o *speedup* aumente linearmente em reflexo ao incremento no tamanho do problema (Quinn, 1994).

Em sistemas heterogêneos, a seleção do EP onde é medido o tempo de execução seqüencial da aplicação ($Tseq$) é importante. Um sistema heterogêneo pode ser modelado como um grafo $G = (V, E; \alpha)$, onde os nós V são os EPs, as arestas E são as conexões entre os EPs e onde $\alpha : V \mapsto [0, 1]$ é o desempenho normalizado do EP. Sendo t_{v0} o tempo gasto por um EP para executar uma aplicação seqüencial e t_v o tempo necessário em um elemento de processamento v, então $\alpha_v = t_{v0}/t_v$ é o desempenho de v em relação a v_0. Através desse modelo, o *speedup* pode ser definido como:

$$S(V) = \frac{t_{v0}}{t_V} \tag{2.4}$$

onde t_V corresponde ao tempo gasto em V elementos de processamento. Como pode ser observado, nesse caso o *speedup* depende do elemento de processamento de referência v_0. A *eficiência* pode ser definida, ao invés da razão entre o *speedup* e o número de elementos de processamento, dividir o *speedup* pelo desempenho total dos elementos de processamento considerados:

$$E(V) = \frac{S(V)}{\sum \alpha_{v \in V}} \tag{2.5}$$

As medidas de desempenho *speedup* e *eficiência* permitem observar o desempenho na execução de aplicações paralelas relacionando a quantidade de EPs atribuídos para a aplicação, não medindo diretamente diferentes fatores que influenciam o desempenho, como por exemplo a carga paralela do sistema no momento da execução. Além disso, em determinadas situações o alvo da avaliação de desempenho não é a aplicação paralela e sim o software paralelo responsável por gerenciar os recursos do sistema. Nesse contexto, outras medidas de desempenho podem ser empregadas, como por exemplo o *tempo de resposta*, *throughput*, *power* e *slowdown*.

O *tempo de resposta* pode ser definido como sendo o intervalo de tempo desde a solicitação de um serviço até o término da execução desse serviço pelo sistema (Jain, 1991). Na área de computação paralela, essa medida geralmente indica o tempo gasto pelo sistema para finalizar a execução da aplicação paralela, considerando dessa forma a aplicação paralela como uma requisição única. Uma variação dessa medida é o *tempo médio de resposta*, que indica o desempenho durante a execução de aplicações paralelas, dentro de um intervalo de tempo, em um sistema multiprogramado. O *tempo médio de resposta* ($\overline{T}$), considerando n aplicações paralelas finalizadas, pode ser formalmente definido como:

$$\overline{T} = \lim_{n \to \infty} \frac{1}{n} \sum_{k=1}^{n} (\delta_k - \alpha_k) \tag{2.6}$$

onde α corresponde ao instante de tempo que a solicitação foi feita e δ representa o instante de tempo em que a solicitação foi completada pelo sistema. Essa formalização indica que o tempo médio de resposta cresce à medida que aumenta a utilização do sistema. Esse comportamento pode ser observado na Figura 2.2.

Outra medida comumente empregada é o *throughput*, que representa a taxa na qual as requisições são atendidas pelo sistema. Novamente, na área de computação paralela as requisições representam as aplicações paralelas. Assim, o *throughput* (ρ) pode ser definido, de acordo com a equação 2.7, como a razão entre a quantidade de aplicações paralelas (N) finalizadas em um intervalo de tempo ($\Delta\, t$) (Figura 2.3).

$$\rho = \frac{N}{\Delta\, t} \tag{2.7}$$

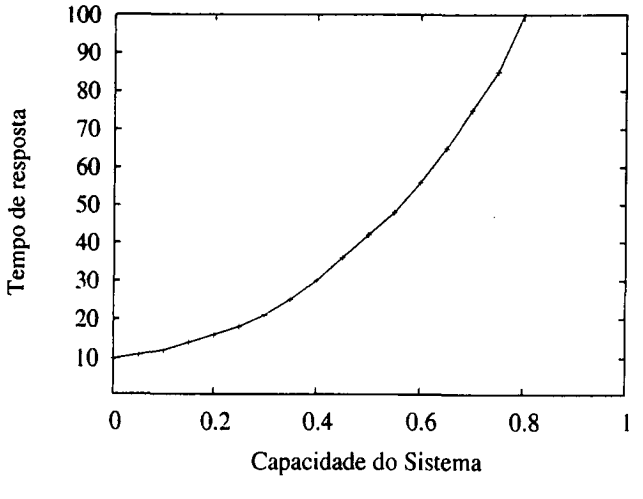


Figura 2.2: Tempo de resposta em relação à utilização do sistema

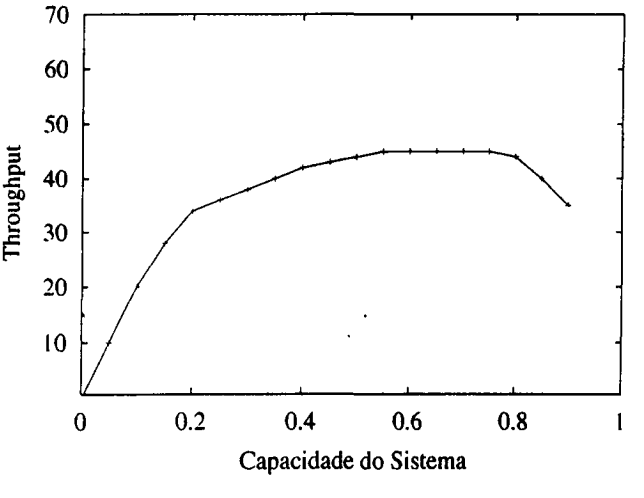


Figura 2.3: *Throughput* em relação à utilização do sistema

A medida de desempenho *power* é a razão do *throughput* pelo *tempo médio de resposta* do sistema:

$$power = \frac{\rho}{\overline{T}} \tag{2.8}$$

Quando o *throughput* aumenta (i.e., mais aplicações são processadas por segundo) com um *tempo médio de resposta* fixo, ou quando o *Tempo médio de resposta* diminui com um *throughput* fixo, o *power* do sistema aumenta. A Figura 2.4 exemplifica a medida *power*, relacionando o tempo de resposta e o *throughput* apresentados nas Figuras 2.2 e 2.3.

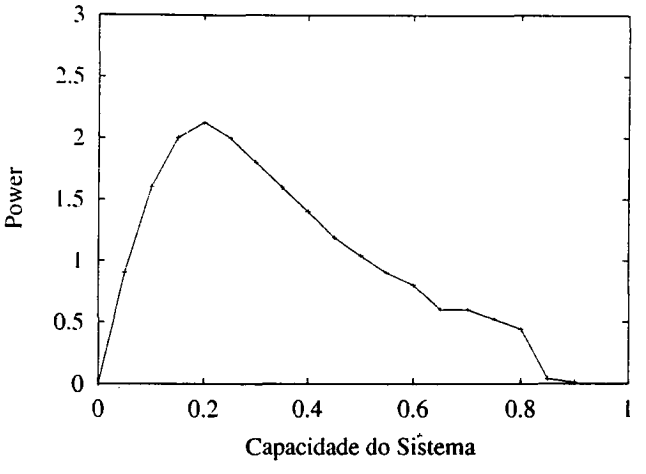


Figura 2.4: *Power* em relação à utilização do sistema

A medida *slowdown* (Harchol-Balter & Downey, 1997) caracteriza o atraso imposto pelo sistema computacional para uma aplicação paralela ou um processo, relacionando o tempo agregado de processamento (*wall time*) e o tempo de utilização do elemento de processamento. Sendo T_{wall} o tempo de processamento agregado e T_{ucp} o tempo de utilização efetiva do elemento de processamento, o *slowdown* (S) é igual a:

$$S = \frac{T_{wall}}{T_{ucp}} \tag{2.9}$$

A medida *slowdown* é, geralmente, empregada para verificar a carga de um determinado EP. A situação ideal para os processos da aplicação paralela, é quando o *slowdown* é igual a 1, situação que ocorre quando todo o tempo de processamento é gasto com os processos. Um EP saturado impõe valores maiores de *slowdown* para os processos em execução. Por exemplo, um EP que possui *slowdown* igual a 2 indica que metade do tempo de execução de um processo é causado por sobrecargas adicionais causadas pelo sistema computacional na gerência de outros processos.

Alguns autores (Corbalan *et al.*, 2001; Feitelson & Rudolph, 1998) consideram a medida *slowdown* como sendo a razão entre o tempo de resposta da aplicação (coletado durante a execução de um experimento) e o tempo de resposta em uma máquina dedicada, com um número de EPs que satisfazem os requerimentos da aplicação. Nesse caso, é necessário o conhecimento prévio do tempo de resposta da aplicação paralela em condições ideais, informação que muitas vezes não é disponível.

2.3 Escalonamento de processos em aplicações paralelas

Um dos grandes desafios existentes em sistemas computacionais paralelos e distribuídos é o desenvolvimento de técnicas eficientes para a distribuição de processos de aplicações paralelas entre os elementos de processamento. Essa atividade de distribuição, conhecida como escalonamento de processos (*Job Scheduling*), visa atingir um conjunto de objetivos relacionados com medidas de desempenho, por exemplo minimizar o tempo médio de resposta e/ou maximizar a utilização dos recursos. Esses objetivos são muitas vezes conflitantes e o escalonador, software responsável pela atividade de escalonamento, deve tomar decisões que são influenciadas por diversos fatores.

Alguns dos fatores que influenciam as decisões do escalonador são: a carga de trabalho paralela do sistema, a presença de aplicações com diferentes características, o hardware da rede de comunicação, o sistema operacional nativo e o hardware dos EPs. Além disso, o escalonamento torna-se mais importante à medida que cresce a utilização de COWs e NOWs em domínios como sistemas de base de dados, servidores WEB e multimídia. Esses domínios apresentam aplicações com diferentes necessidades de qualidade de serviço, não relacionadas apenas com medidas de desempenho como tempo de resposta ou capacidade de trabalho do sistema. Essas aplicações apresentam divergentes características de computação, comunicação e operações de E/S, ressaltando ainda mais a importância do escalonamento.

O escalonamento é realizado através de mecanismos de escalonamento ou algoritmos de escalonamento. Algoritmos de escalonamento seguem políticas de escalonamento, recebendo parâmetros que definem a reação do escalonador para diferentes situações. As políticas representam decisões como "permitir que aplicações com tempo menor de execução sejam executadas prioritariamente" ou "permitir que aplicações interativas tenham maior prioridade", embora os termos "política" e "mecanismo" sejam, muitas vezes, empregados como sinônimos.

O escalonamento em sistemas computacionais distribuídos pode ser organizado através de duas etapas. Na primeira etapa, o escalonador é responsável por decidir, entre as várias aplicações que são submetidas ao sistema, o conjunto de aplicações que terão acesso aos recursos computacionais, a localização dos recursos e quantidade de recursos que serão disponibilizados à aplicação. Na segunda etapa, os processos que compõem a aplicação são mapeadas localmente nos EPs, através de processos ou *threads*.

Casavant & Kuhl (1988) apresentam uma taxonomia para o escalonamento em sistemas distribuídos de propósito geral. De acordo com essa taxonomia, a primeira etapa, que recebe as aplicações paralelas e realiza a atribuição de recursos, é chamada de escalonamento global. A segunda etapa, responsável pela multiplexação da potência computacional local aos EPs, é chamada de escalonamento local. De acordo com o tempo em que as decisões do escalonador são tomadas, essa taxonomia também organiza o escalonamento global em estático e dinâmico. No escalonamento estático, as decisões são tomadas em tempo de compilação da aplicação,

assumindo que as informações sobre o sistema estão disponíveis nesse ínterim. Nesse contexto, é comum a modelagem da aplicação através de grafos, onde os nós são representados pelos processos e os arcos representam as dependências (relacionadas com transferência de informações) entre os processos. Para a resolução desse modelo são empregados algoritmos que buscam dentro do espaço de estados uma solução ótima ou, quando algumas informações não estão disponíveis, sub-ótima para o problema. Heurísticas podem também ser empregadas na busca de uma solução aceitável para o problema.

Diferente da forma estática, o escalonamento dinâmico assume que muito pouca ou nenhuma informação sobre o comportamento e necessidades da aplicações quanto aos recursos é previamente conhecida. As decisões do escalonamento dinâmico podem estar centralizadas ou distribuídas entre os diversos EPs. Quando as decisões são distribuídas, o escalonamento pode ser cooperativo, quando os EPs interagem durante a alocação de recursos, ou não cooperativo, quando os escalonadores trabalham isoladamente.

2.3.1 Escalonamento local

O escalonamento local é, geralmente, implementado pela imagem do sistema operacional que executa no EP. Para isso, o sistema operacional representa os programas em execução através de processos. O processo constitui o programa em execução e suas informações, como por exemplo contador do programa (*program counter*) e conteúdo dos registradores. Sob essa definição, processos são entidades dinâmicas, representando a imagem do programa em execução (Ritchie & Thompsom, 1974). Uma abordagem alternativa aos processos é a utilização de *threads* (Tanenbaum, 1992; Coulouris *et al.*, 1994), que é uma solução implementada através de bibliotecas em nível de usuário ou de *kernel*, onde os programas em execução podem solicitar várias linhas de controle. Nesse caso, busca-se reduzir o impacto durante a troca de contexto e a melhoria no desempenho de servidores de recursos de E/S.

No escalonamento local, a política de escalonamento predominante em diversos sistemas é o escalonamento de curto prazo (*short term scheduling*) (Leffler *et al.*, 1989). Essa política visa favorecer processos interativos, pois aumenta a prioridade de processos que residem no estado de bloqueado [12] por muito tempo e reduz a prioridade de processos que acumulam quantidades significantes de tempo de execução. Essa política possui implementações através de algoritmos

[12]De uma maneira geral, um processo pode residir em 3 diferentes estados: executando, pronto e bloqueado. Um processo encontra-se no estado de bloqueado quando executa um pedido ao sistema operacional, geralmente um recurso de E/S, e fica aguardando até que o sistema possa atender ao pedido. Um processo vai para o estado de pronto, quando o escalonador verifica que o seu *quantum* expirou ou quando sai do estado de bloqueado. Um processo reside no estado executando quando o escalonador atribui o elemento de processamento ao processo, que antes encontrava-se no estado de pronto, para que este possa ser executado. Em sistemas com apenas uma UCP, apenas um processo encontra-se no estado executando em um determinado instante de tempo, mas vários processos nos estados pronto e bloqueado podem existir. Com essa multiprogramação e preempção, os usuários têm a impressão de que o sistema executa todos os processos em paralelo, mas o que existe é um pseudoparalelismo entre os processos concorrentes (Tanenbaum, 1992).

que empregam filas múltiplas, que são organizadas através de prioridades. O escalonamento dentro de cada fila é, geralmente, realizado através de um algoritmo *round-robin*, ou em algumas situações, através de uma disciplina FCFS (*First-come, first-served*). Exemplos de sistemas operacionais que implementam esses algoritmos são o UNIX 4.3 BSD, o Solaris, IRIX e Linux.

No sistema operacional UNIX 4.3 BSD (Leffler *et al.*, 1989), os processos podem estar em execução nos modos usuário e *kernel*. O processo executa em modo usuário e entra em modo *kernel* quando executa chamadas ao sistema (*system calls*), que é modo privilegiado para acesso aos recursos. O algoritmo de escalonamento segue a política de curto prazo, sendo que as informações sobre os processos são armazenadas em estruturas chamadas de tabelas de processos (*proc tables*). O escalonamento dentro das filas múltiplas é realizado de acordo com a disciplina *round-robin*, com um *quantum* de 100 milissegundos. As estruturas do sistema responsáveis por armazenar informações sobre o processo são atualizadas a cada 4 pulsos do relógio programável.

Assim como o UNIX 4.3 BSD, o sistema Solaris (Zhang *et al.*, 2000) emprega filas múltiplas, com níveis de prioridade variando entre 0 e 59 e *quantum* variando entre 20 e 200 milissegundos. Uma abordagem similar é empregada pelo sistema IRIX (Barton & Bitar, 1995). Nesse sistema, a política de curto prazo é considerada, sendo que o sistema organiza os processos em 4 classes: *kernel*, tempo real, tempo compartilhado e processamento *background*. Em sistemas com mais de uma UCP, o IRIX utiliza o mecanismo *gang scheduling* (vide Seção 2.3.2) e heurísticas para uma melhor utilização da *cache* do sistema.

O sistema operacional Linux (Bovet & Cesati, 2000) estabelece 4 estados possíveis para os processos, além do estado executando: pronto, bloqueado, *stopped* e *zombie* [13]. O Linux estabelece as classes de processos tempo real e normal. O algoritmo de escalonamento emprega o algoritmo *round-robin* para processos da classe normal, e o algoritmo *round-robin* ou FCFS para processos da classe tempo real [14].

Quando as primeiras versões de sistemas operacionais multiprogramados foram desenvolvidas, processos interativos restringiam-se a aplicações simples como editores de texto com interação através do teclado (Ritchie & Thompsom, 1974). Atualmente, os algoritmos de escalonamento que implementam a política de curto prazo e seus parâmetros têm sido estudados visando melhorar o tempo de resposta de processos interativos atuais. Processos interativos atuais envolvem a utilização de outros dispositivos de interação com o computador e aplicações com características multimídia, englobando áudio e vídeo. A versão mais recente do núcleo do sistema operacional Linux (núcleo 2.6) traz algumas melhorias, por exemplo a alteração da taxa do relógio programável de 100 para $1000Hz$, visando melhorar o tempo de resposta de aplicações interativas (Etsion *et al.*, 2004).

[13] Um processo entra no estado *stopped* quando recebe algum sinal (*signal*) do sistema (por exemplo SIGSTOP), e retorna para o estado de pronto após receber um novo sinal (por exemplo SIGCONT). Geralmente, um processo encontra-se nesse estado quando está sendo depurado. Um processo no estado *zombie* é aquele que finalizou a sua execução, mas por algum motivo, suas estruturas, criadas e mantidas pelo sistema operacional, ainda estão ativas.

[14] Processos da classe tempo real têm prioridade maior que a prioridade de qualquer processo da classe normal.

2.3.2 Escalonamento global

Um sistema computacional distribuído é constituído de vários recursos, que devem ser compartilhados entre os vários usuários do sistema. A arbitragem dos recursos é realizada por software de gerência, responsáveis por atender as requisições de acesso aos recursos.

O escalonador atua como um gerente de recursos. Sob essa perspectiva, o escalonador é responsável por permitir o acesso aos EPs, estabelecendo políticas para que os consumidores (representados pelas aplicações) obtenham recursos de maneira ordenada. Através de uma política, o escalonador atua no sistema computacional. As políticas de escalonamento podem ser organizadas através duas dimensões, em relação ao tempo (*time sharing*) e em relação ao espaço de EPs (*space sharing*).

Compartilhamento do tempo (*time sharing*)

O compartilhamento do tempo é uma extensão direta do compartilhamento do tempo em sistemas uniprocessados (vide Seção 2.3.1), que pode ser, empregando a taxonomia de Casavant & Kuhl (1988), centralizado ou distribuído. O compartilhamento de tempo centralizado existe através da utilização de uma fila global (*global queue*) compartilhada pelos EPs do sistema. A grande vantagem do compartilhamento centralizado é que o compartilhamento de carga entre os EPs é automático, desde que nenhum EP fica ocioso esperando por um novo processo (Feitelson & Rudolph, 1998). Por outro lado, o acesso dos EPs à fila global pode causar disputa, problema que cresce com o número de EPs. Além disso, outro problema que surge dessa abordagem é que os processos são potencialmente executadas, em cada *quantum*, em diferentes EPs. Isso causa ineficiência na utilização da memória principal e da memória *cache* (Feitelson & Rudolph, 1998). O compartilhamento de tempo centralizado geralmente requer modificações no sistema operacional, fato não desejável, e, muitas vezes não possível, em sistemas distribuídos de propósito geral.

O compartilhamento de tempo distribuído ocorre quando os processos são atribuídas aos EPs do sistema, de forma que os escalonadores locais têm total liberdade e tomam decisões isoladas com relação aos demais EPs. Essa abordagem pode ser implementada em nível de usuário, com um software de escalonamento (vide Seção 2.3.3) sendo responsável pela distribuição dos processos e os escalonadores locais, responsáveis pela gerência do compartilhamento de tempo.

Diferente do modo centralizado, no modo distribuído os EPs estão sujeitos a diferenças de carga de trabalho, devido a possível atribuição demasiada de processos para determinados EPs e a subutilização de outros. Em sistemas distribuídos, as diferenças de carga também surgem pela influência externa (ao software de escalonamento), causada por usuários que submetem arbitrariamente suas aplicações seqüenciais e/ou interativas. Esse fato cria a necessidade do

compartilhamento de carga ou balanceamento de carga, que é um objetivo para o software de escalonamento para sistemas distribuídos.

Uma forma de obter balanceamento de cargas é através de migração de processos entre os EPs, abordagem que permite melhorar o tempo de resposta do sistema, diminuindo o trabalho de EPs sobrecarregados e utilizando ciclos de processamento de máquinas ociosas. Para cumprir esse objetivo, é importante definir como será o relacionamento entre os EPs do sistema, durante a troca de informações sobre a carga de trabalho. Shivaratri *et al.* (1992) organiza os componentes dos algoritmos de distribuição de carga em quatro classes: política de transferência, política de seleção, política de localização e política de informação. Cabe à política de transferência definir os EPs participantes (transmissor e receptor) para a transferência de carga, baseando-se na carga de trabalho do EPs. A política de seleção define o conjunto de processos que deve ser migrado, observando parâmetros como o tempo de execução acumulado e dependências locais dos processos. A criação de um relacionamento entre o transmissor e o receptor de carga está sob a responsabilidade da política de localização. A política de informação define quando e quais informações sobre a carga do sistema serão coletadas, assim como a sua origem.

Alguns ambientes de escalonamento, por exemplo o Condor (vide Seção 2.3.3), empregam a migração de processos com o objetivo principal de preservação da autonomia do computador. Nessa abordagem, o computador é utilizado para o processamento paralelo apenas se está ocioso. Quando um usuário passa a utilizar esse computador, a migração de processos é utilizada, com os processos sendo atribuídos para outros EPs. Assim, busca-se aumentar a satisfação dos usuários durante a utilização interativa do sistema.

O custo para migração de processos em sistemas distribuídos é significativo. Esse custo envolve o salvamento do contexto do processo, através de técnicas de *checkpointing*. *Checkpointing* corresponde à ação de efetuar o salvamento das informações sobre o estado atual de uma aplicação (ou processo) em um meio de armazenamento confiável, e, na presença de falha ou após a migração para outra máquina, recuperar a execução da aplicação em um estado consistente, ao invés de reiniciar a execução da aplicação. Essa ação pode ser configurada para ser executada em intervalos de tempo fixos ou em pontos distintos no código do programa, e pode ser realizada pelo programador, pelo compilador ou pelo hardware. Geralmente, as informações que empregadas são o espaço de endereçamento e de pilha do processo, arquivos abertos, sinais pendentes, estado do processador e outras informações pertinentes ao objetivos do salvamento (Ziv & Bruck, 1996).

O custo envolvido na transferência dos processos pela rede de comunicação não é único. O tratamento de dependências locais ao EP, o novo estabelecimento de comunicação entre os processos e a heterogeneidade de recursos também são fatores importantes que devem ser considerados para a implementação da migração de processos. Além disso, é importante o conhecimento da carga de trabalho paralela do sistema, principalmente se for possível estabelecer uma distribuição de probabilidade para o tempo de vida dos processos. Com informações desse nível, é possível amortizar os custos envolvidos na migração de processos, migrando preferencialmente

processos que provavelmente exijam maiores tempos de execução. Nesse sentido, Harchol-Balter & Downey (1997) apresenta a análise de cargas de trabalho reais e um modelo de predição de tempo de vida de processos. Através desse modelo, identificam-se quais são os processos que irão executar por mais tempo, e, assim, mais aptos para serem migrados. Mostra-se assim a importância da utilização do conhecimento sobre a carga de trabalho do sistema na tomada de decisões do software de escalonamento.

Escalonamento cooperativo (*co-scheduling*)

Um problema que surge em políticas de compartilhamento de tempo distribuídas é a falta de sincronização entre os EPs que compõem o sistema. A falta de sincronização entre os EPs durante a execução das aplicações produz uma degradação considerável no desempenho, principalmente em aplicações constituídas por processos que interagem freqüentemente entre si. Arpaci *et al.* (1995) observam, através de simulação, que com um tamanho de *quantum* igual a 100 milissegundos e com desvio de sincronização (*coscheduling skew*) entre os EPs igual a 10 milissegundos, é imposto um atraso (*slowdown*) de 10% na execução de certas aplicações paralelas [15].

Uma forma de garantir a sincronização entre os escalonadores locais em políticas *time sharing* é através de mecanismos de escalonamento cooperativo (*coscheduling*), que permitem que os processos de uma aplicação paralela sejam escalonados em seus respectivos EPs ao mesmo tempo (Corbalan *et al.*, 2001). Variações dessa forma de escalonamento são os mecanismos *gang scheduling* e *family scheduling*. A abordagem *gang scheduling* (Zhang *et al.*, 2000) é baseada na observação de que alguns processos da aplicação interagem freqüentemente, e, apenas estes, devem ser escalonados ao mesmo tempo. Na estratégia *family scheduling* (Feitelson & Rudolph, 1995b), o escalonamento cooperativo é realizado mesmo quando existe um número menor de EPs que o número de processos, através de um segundo nível de compartilhamento de tempo.

De maneira geral, o escalonamento cooperativo e suas variações, requerem alguma forma implícita ou explícita de sincronização entre os escalonadores locais. O custo de implementação da sincronização explícita associada a uma fila global em arquiteturas fracamente acopladas é, muita vezes, proibitivo (Feitelson & Rudolph, 1995b). Mensagens de sincronização enviadas por um EP mestre, podem chegar atrasadas nos demais EPs, devido ao tráfego na rede de comunicação. Além disso, mesmo se fosse possível tal sincronização, os desvios de relógio (*clock skews*) que ocorrem nos processadores dos EPs, ainda existiriam (Lamport, 1990).

[15] Arpaci *et al.* (1995) consideram diferentes aplicações paralelas, com diferentes comportamentos quanto a sincronização: *cholesky*, que implementa fatorização LU em matrizes simétricas esparsas; *em3d*, que simula a propagação de ondas eletromagnéticas em três dimensões; *sample*, que implementa um algoritmo de ordenação; e *connect*, que usa um algoritmo aleatório para encontrar componentes conectados em um grafo arbitrário.

Algumas estratégias exploram a sincronização implícita entre as processos que compõem a aplicação. Nessa abordagem, os escalonadores locais buscam cooperar através da exploração de eventos locais, sem sincronização explícita. Nesse sentido, Arpaci-Dusseau (2001) apresenta o mecanismo *two-phase spin-blocking*, onde os processos que estão aguardando por uma mensagem esperam uma determinada quantidade de tempo (*spin time*) antes de entregar o EP para outro processo [16]. O *spin time* deve ser escolhido em função de vários dos parâmetros do sistema e da rede de comunicação, e pode ser definido automaticamente com a ajuda de *benchmarks* específicos (Arpaci-Dusseau *et al.*, 1998).

Compartilhamento do espaço (*space sharing*)

Nas políticas *space sharing*, os EPs são organizados em conjuntos disjuntos que são atribuídos para as aplicações, na forma de um conjunto por aplicação. Nessas políticas, os conjuntos de EPs são chamados de partições e, de acordo com a maneira que as partições são criadas e mantidas, podem ser fixas, adaptativas ou dinâmicas.

As partições fixas são pré-estabelecidas durante a configuração do sistema e não podem ser alteradas em tempo de execução. Naik *et al.* (1993) mostram, através de simulação, que o desempenho obtido por essa estratégia é modesto, sendo interessante apenas para situações específicas de carga (poucas aplicações com tempo de execução pequeno). Uma variação dessa abordagem são as partições fixas com utilização de prioridades. Nesse contexto, as partições fixas são organizadas de acordo com classes de prioridade, sendo que partições com maiores números de EPs são disponibilizadas para aplicações com maiores prioridades.

Assim como em sistemas com partições fixas, sistemas que empregam partições adaptativas apenas liberam a quantidade de EPs atribuídos à aplicação quando esta é finalizada. A diferença reside no modo que as partições são definidas, que é feito considerando o estado do sistema. Políticas adaptativas tendem a atribuir um grande número de EPs quando a carga do sistema é baixa e um número reduzido de EPs à medida que a carga do sistema aumenta. Tanto parâmetros observáveis (por exemplo número de aplicações no sistema e número de aplicações na fila) quanto parâmetros controláveis (grau de paralelismo em aplicações adaptativas) considerados para a atribuição de EPs podem ser utilizados (Anastasiadis & Sevcik, 1997; Naik *et al.*, 1993; Setia & Tripathi, 1993). Uma variação do esquema de partições adaptativas é a utilização de prioridades, de forma que são definidos limites máximos para o número de EPs que serão alocados pela política (Naik *et al.*, 1997). A grande vantagem das partições adaptativas, em relação às partições fixas, é a adaptação do sistema para diferentes situações de carga.

[16]Para que o processo possa aguardar em espera ocupada (*busy waiting*) por uma mensagem, é necessária a implementação do subsistema de comunicação em nível de usuário, por exemplo através do mecanismo de mensagens ativas (von Eicken *et al.*, 1992; Chiola & Ciaccio, 2000). Nesse mecanismo, o manipulador de dispositivo encontra-se incorporado no código do usuário, de forma que a comunicação é realizada sem a intervenção do sistema operacional.

Políticas que empregam partições dinâmicas definem o tamanho da partição à medida que as aplicações chegam ao sistema, de acordo com o estado do sistema, e o tamanho da partição pode ser alterado em tempo de execução da aplicação (Naik *et al.*, 1993). Embora essa adaptação possa ser realizada eficientemente em sistemas fortemente acoplados com memória compartilhada, o custo dessa adaptação é, na maioria das vezes, proibitivo em sistemas fracamente acoplados com memória distribuída. Essa abordagem necessita que as aplicações estejam preparadas para adaptarem-se em tempo de execução às alterações no número de recursos atribuídos a ela. A maioria das aplicações não tem tal adaptabilidade, e necessitam que a quantidade de recursos inicialmente alocados à aplicação permaneça inalterada.

As políticas *space sharing* são um extensão direta do escalonamento em lote em sistemas paralelos. Essas políticas são simples de ser implementadas e reduzem o custo da troca de contexto nos EPs. Entretanto, a utilização de políticas *space sharing* isoladas pode levar a uma fraca utilização dos tempos de processamentos do EPs, principalmente pela falta da multiprogramação quando os processos realizam operações de E/S.

Embora seja aplicável em sistemas homogêneos, o mapeamento um para um realizado por políticas *space sharing* não é eficiente em sistemas heterogêneos, onde a potência computacional varia entre diferentes EPs. Esse fator, associado à crescente disponibilidade de estações de trabalho multiprocessadas, indica que mais de um processo pode ser atribuído por EP, reduzindo o escopo de aplicação das políticas *space sharing* em sistemas distribuídos. Observa-se que políticas de escalonamento *space sharing* são aplicadas em ambientes distribuídos controlados e geograficamente não distribuídos, principalmente em arranjos computacionais (*clusters*). Tais arranjos são organizados como uma máquina de processamento paralelo, gerenciada por um escalonador global. A próxima seção trata com detalhes o software de escalonamento.

2.3.3 Ambientes de escalonamento

As técnicas e políticas de escalonamento podem ser implementadas internamente ou externamente à aplicação. A implementação interna à aplicação paralela permite total liberdade em como serão atribuídas os processos aos EPs. Apesar disso, quando as técnicas de escalonamento são empregadas internamente à aplicação, a portabilidade dessas aplicações para diferentes sistemas paralelos é reduzida e o tempo gasto com o desenvolvimento e depuração torna-se mais elevado. Assim, é preferível que o escalonamento seja executado externamente a aplicação.

Externamente, a implementação pode ser realizada dentro do sistema operacional ou através de ambientes de escalonamento. Os ambientes de escalonamento atuam em um sistema computacional, estabelecendo quais aplicações terão acesso aos recursos e quais recursos serão disponibilizados para determinadas aplicações. Assim, é de responsabilidade do software

de escalonamento o gerenciamento da máquina paralela virtual, estabelecendo políticas para a execução das aplicações e para o acesso aos recursos.

Grande parte dos ambientes de escalonamento são baseados em um escalonador mestre com filas múltiplas. Nesse modelo, o escalonador gerencia várias filas, que são estabelecidas através de regras específicas, sendo que cada fila é criada para atender aplicações com determinadas características. As características geralmente empregadas são: o tempo de execução da aplicação, quantidade e tipo de recursos requisitados pela aplicação. A submissão da aplicação em uma fila que satisfaça seus critérios é geralmente de responsabilidade do usuário. É comum nos ambientes baseados em filas múltiplas a utilização de prioridades, assim como a possibilidade de restringir o acesso dos recursos para determinados usuários. Geralmente, existe uma partição associada a cada fila, representando um subconjunto dos EPs. Dessa forma, a potência computacional é distribuída proporcionalmente entre as filas, de acordo com as políticas estabelecidas pelo sistema.

O `LSF` (*Load Sharing Facility*) (Kaplan & Nelson, 1994) é um exemplo desses ambientes. Esse ambiente permite a execução remota e o escalonamento de aplicações paralelas em redes de estações de trabalho heterogêneas. O `LSF` segue o princípio da distribuição de carga entre os EPs. O usuário pode especificar as necessidades e características da aplicação, que são consideradas pelo escalonador durante a atribuição de processos. Exemplos de características são: arquitetura dos EPs, quantidade de memória principal, quantidade de memória de troca e quantidade de espaço em disco disponível.

O escalonamento nesse ambiente segue essas duas etapas. Inicialmente, o ambiente seleciona um conjunto de EPs, baseado nas necessidades da aplicação. Após essa seleção, o ambiente escolhe o melhor subconjunto de EPs dentro do conjunto previamente selecionado. Durante a seleção de EPs, o `LSF` observa o tamanho da fila dos EPs, assim como a utilização da UCP, taxa de paginação, quantidade de memória principal e quantidade de espaço em disco. Medidas de desempenho adicionais podem ser inseridas pelo administrador do sistema, flexibilizando as decisões do escalonador. Em caso de falha no escalonador mestre, um EP pode ocupar sua posição tornando-se mestre. Isso é realizado através de um algoritmo eletivo (Tanenbaum, 1992). As filas do ambiente e políticas podem ser configuradas através de uma interface de programação. Um exemplo da flexibilidade do `LSF` pode ser verificado em Parsons & Sevcik (1997), que descrevem a implementação de uma extensão desse ambiente, empregando diversos algoritmos de escalonamento.

Alguns sistemas associam perfis (*profiles*) aos EPs para realizar o casamento entre as necessidades da aplicação o usuário com as características dos EPs, sem a existência de múltiplas filas centralizadas. Nessa abordagem, o usuário pode especificar características de sua aplicação e o sistema é, dinamicamente, responsável pela busca de EPs que satisfaçam às necessidades da aplicação, como por exemplo quantidade de memória ou espaço em disco. Exemplos dessa abordagem são o `Sun Grid Engine` e o `AMIGO`.

O ambiente de escalonamento `Sun Grid Engine`, versão do software `CODINE` (*COmputing in DIstributed Network Environments*), visa a utilização ótima dos recursos computacionais em ambientes distribuídos, particularmente conjuntos grandes de estações de trabalho integrados com máquinas paralelas de processamento vetorial (Kaplan & Nelson, 1994). O `Sun Grid Engine` encontra-se implementado em várias plataformas, particularmente em sistemas operacionais baseados no UNIX (por exemplo SunOS, Solaris, IBM AIX e Linux). A implementação do `Sun Grid Engine` é estruturada através de um conjunto de filas e de um escalonador mestre (*master scheduler*), que recebem a aplicação do usuário. O sistema mantém uma lista de perfis de todos os recursos disponíveis, incluindo UCP, memória principal, carga de trabalho permitida, número de aplicações permitidas e disponibilidade com relação ao tempo. O usuário pode submeter aplicações, podendo selecionar uma arquitetura específica, quantidade de memória, sistema operacional, número de filas, tempo de UCP, intervalo de *checkpointing*, intervalo para migração e quantidade de espaço em disco. Prioridades podem ser solicitadas e o administrador pode configurar restrições de acesso aos usuários do ambiente. O sistema realiza a busca dos EPs apropriados para a aplicação, com a ajuda de processos *daemons* que executam em cada EP. Caso não sejam encontrados EPs que satisfaçam às necessidades da aplicação, essa aguarda até que os recursos sejam liberados.

No `Sun Grid Engine`, quando o escalonador mestre não consegue contatar um dos EPs dentro de um período de tempo pré-determinado, o escalonador não realiza a atribuição de aplicações para esse EP. Desde que o `Sun Grid Engine` emprega *checkpointing* e migração de processos, o trabalho pode ser potencialmente recuperado, no caso de falha nos EPs. Além disso, buscando também incrementar a tolerância a falhas, o `Sun Grid Engine` permite a utilização de um escalonador mestre auxiliar (*shadow master scheduler*), que automaticamente assume as atribuições do escalonador mestre, no caso de falha neste. Nesse ambiente, as políticas, assim como as filas e períodos de verificação da carga das máquinas, podem ser configuradas em tempo de execução, sem ser necessária a interrupção da execução do ambiente.

O `AMIGO` (DynAMical FlexIble SchedulinG EnvirOnment) é um ambiente que agrupa várias políticas e considera diversos fatores para realizar o escalonamento de aplicações paralelas em sistemas computacionais heterogêneos e distribuídos executando o sistema UNIX (Souza *et al.*, 1999). A implementação do `AMIGO` é baseada em um escalonador mestre e vários processos *daemons*, responsáveis por obter informações sobre a carga dos EPs. Sua implementação é estruturada em duas camadas. A camada superior, composta de uma interface gráfica e arquivos de configuração, permite o cadastro de classes de software, políticas, medidas de desempenho e *benchmarks*. Através dessa camada, o administrador do sistema pode relacionar as classes de software com algoritmos mais adequados, permitindo o escalonamento orientado à classe da aplicação. A camada inferior, responsável pela integração do ambiente com as bibliotecas de passagem de mensagens [17], realiza o escalonamento e atribuição de processos baseando-se nas

[17] Atualmente, o `AMIGO` realiza o escalonamento de aplicações PVM, MPI e CORBA.

informações providas pelos de arquivos de configuração. As aplicações paralelas não precisam ser modificadas ou adaptadas, de forma que o escalonamento é realizado de maneira transparente, escondendo os detalhes e algoritmos empregados. O AMIGO difere dos ambientes de escalonamento que empregam filas múltiplas, organizadas de acordo com as características das aplicações. Neste ambiente, essas filas são substituídas pelos relacionamentos entre classes de software e algoritmos de escalonamento. No AMIGO, não existe um escalonador mestre auxiliar ou mecanismos, como *checkpointing*, que implementem tolerância a falhas.

O ambiente DJM (*Distributed Job Manager*), desenvolvido pelo *Minnesota SuperComputing Center* e disponível como software de domínio público[18], é implementado exclusivamente para máquinas de processamento paralelo (Kaplan & Nelson, 1994). Este ambiente oferece suporte para usuários interativos, balanceamento de carga entre os EPs do computador paralelo, configuração flexível e tolerância a falhas. Esse ambiente DJM busca realizar o balanceamento da carga atribuindo a aplicação para a partição de EPs menos saturados e que satisfaz às necessidades da aplicação. Prioridades e restrições a determinados usuários podem ser especificadas. Em caso de falha em uma das partições, a aplicação é atribuída para outra partição que atende aos requerimentos da aplicação. Toda configuração é armazenada em um arquivo texto, freqüentemente consultado pelo escalonador. Assim, as políticas e parâmetros de configuração podem ser consultados e alterados em tempo de execução.

O ambiente DQS (*Distributed Queueing System*) (Duke *et al.*, 1994) foi desenvolvido inicialmente pelo *SuperComputer Computations Research Institute*, na Universidade da Flórida, e tornou-se bastante popular por ter sido o primeiro ambiente de escalonamento, não comercial e de domínio público [19]. O DQS implementa várias filas, nas quais os usuários submetem aplicações. O DQS tem versões para diversos sistemas operacionais, incluindo o Linux, IBM AIX e FreeBSD. O escalonamento e a atribuição de processos são realizados de acordo com a carga média observada, de forma que os EPs mais ociosos recebem a aplicação. O usuário pode especificar quais são os requisitos de sua aplicação. As filas do ambiente podem possuir atributos (*queue complexes*), que podem incluir características como quantidade de memória, arquitetura dos EPs e características da aplicação. Restrições podem ser aplicadas para usuários e as filas podem possuir restrições quanto ao número de recursos (por exemplo tempo de UCP, tempo *wall clock*, tamanhos dos dados, tamanho da pilha e tamanho da aplicação) que caracterizam a aplicação.

Similarmente ao *Sun Grid Engine*, o DQS possui um escalonador auxiliar (*shadow master scheduler*), que assume as atividades do escalonador mestre (*master scheduling*) em caso de falha. Se um EP falha, o escalonador não atribuí processos adicionais a ele. Os processos atribuídos a um EP que falhou são reiniciados no momento em que o EP volta a operar normalmente.

[18] ftp://ec.msc.edu.pub/LIGHTNING (visitado em 20/08/2001)
[19] ftp://ftp.physics.mcgill.ca/pub/Dnqs/ (visitado em 20/08/2001)

As políticas de escalonamento podem ser configuradas em tempo de execução, através de uma interface gráfica ou através de linha de comando.

A abordagem centralizada não é a única. Outros ambientes assumem o modelo cliente/servidor, como por exemplo o `Task Broker` (vide Seção 2.3.3), onde os clientes solicitam potência computacional, que é provida por servidores configuráveis distribuídos pelos EPs. Servidores recusam pedidos de aplicações, caso decidam que não podem atender a requisição por algum motivo, seja pela carga no EP ou necessidades da aplicação. Os serviços são pré-estabelecidos e caso nenhum servidor responda a requisição do cliente, este deve aguardar em um fila. O administrador do sistema é responsável pela criação de serviços, assim como pela configuração dos recursos que serão associados a cada serviço. Não existe um servidor central para o escalonamento, de forma que não existe um ponto central de falha. Toda a configuração pode ser realizada em tempo de execução, através de interfaces gráfica ou linha de comando.

Construído de acordo com a idéia de que algumas máquinas da rede desperdiçam tempo de UCP que pode ser aproveitado por aplicações paralelas, o `Condor` implementa a distribuição de aplicações em sistemas computacionais fracamente acoplados (Litzkow *et al.*, 1988; Bricker *et al.*, 1991). Nesse ambiente, o princípio de integridade da estação de trabalho é mantido, de forma que o trabalho do usuário interativo que utiliza a estação de trabalho não é prejudicado. Visando obedecer esse princípio, o `Condor` realiza migração de processos.

O `Condor` emprega a monitoração das atividades de todos os EPs participantes. Os EPs que são classificados como inativos são inseridos em um banco de processadores (*processor bank*). O banco de processadores é uma entidade dinâmica: à medida que os EPs tornam-se ocupados, esses são removidos do banco de processadores. Prioridades são implementadas através do algoritmo *up-down*, que incrementa a prioridade de aplicações que esperam a mais tempo por recursos e reduz a prioridade de aplicações que utilizaram vários recursos em instantes de tempo passados. A implementação do `Condor` é baseada em um gerente centralizado (*machine manager*), que realiza as tarefas de escalonamento e atribuição de processos. O usuário pode selecionar o EP, a arquitetura do EP, a quantidade de memória, a quantidade de memória de troca (*swap space*) e o sistema operacional. Adicionalmente, o usuário pode especificar a prioridade de sua aplicação, mas não são aplicadas restrições em usuários. Desde que *checkpointing* é realizado, o trabalho da aplicação pode ser recuperado em caso de falhas no sistema computacional.

A política considerada para a atribuição de processos é baseada na atividade do usuário e não pode ser modificada em tempo de execução. A definição de disponibilidade no `Condor` não é agressiva (Arpaci *et al.*, 1995). Um EP é considerado disponível apenas após 15 minutos de baixa utilização de UCP e sem atividade de teclado. Definições mais agressivas consideram um minuto de baixa utilização e pouca atividade de teclado para caracterizar um EP como ocioso. Quando a estação de trabalho começa a ser empregada interativamente, as aplicações paralelas que estão sendo executadas são inicialmente suspensas; caso a atividade interativa permaneça, os contextos das aplicações paralelas são salvos e o `Condor` realiza a migração dessas aplicações

para computadores ociosos. O Condor está disponível como software de domínio público e existem listas de discussão para auxiliar usuários e administradores de sistema [20].

O Condor pode ser integrado com a ferramenta GLOBUS, de forma que as aplicações possam ser executadas empregando um ambiente de computação em grade (*grid computing*). A ferramenta GLOBUS define um conjunto de protocolos para a comunicação entre diferentes domínios e para o acesso padronizado para diferentes ambientes de escalonamento (Foster & Kesselman, 1997). O Condor, através do software Condor-G, conecta-se à malha computacional e permite que aplicações sejam submetidas, visando compartilhar recursos geograficamente distribuídos (Thain *et al.*, 2004).

Concebido para sistemas UNIX, o Load Balancer (Kaplan & Nelson, 1994) distribui aplicações em redes de estações de trabalho, melhorando o desempenho do sistema computacional através da utilização de EPs ociosos e reduzindo a carga em EPs sobrecarregados. Nesse ambiente, as aplicações são submetidas em uma fila baseada em prioridades e a medida que os recursos se tornam disponíveis, é realizada a atribuição de processos. Para realizar a atribuição de processos, o Load Balancer observa diversas medidas de desempenho como por exemplo carga média do EP, diferenças relativas de desempenho entre os EPs, disponibilidade de memória e espaço de troca, número de UCP existentes nos EPs, número de sessões interativas abertas nos EPs e a possibilidade dos EPs atenderem as necessidades da aplicação do usuário. O usuário pode especificar para suas aplicações a arquitetura, quantidade de memória e quantidade de espaço de troca. Os EPs são cadastrados pelo administrador por características, por exemplo unidades de disco, unidades de fita, impressoras e bibliotecas de software existentes no EP. As políticas podem ser alteradas através de uma arquivo de configuração, que é consultado a cada 5 segundos pelo escalonador. Restrições podem ser aplicadas para usuários, limitando o número de aplicações que podem ser submetidas por usuário. Nesse ambiente, não existe um escalonador auxiliar. Em caso de falha em um EP, a aplicação volta a ser executada quando o EP voltar a funcionar normalmente.

O LoadLeveler (IBM, 1996) é um ambiente de escalonamento distribuído, baseado no Condor (Thain *et al.*, 2004). O LoadLeveler permite que o usuário especifique características da aplicação e a atribuição de processos é realizada considerando a carga de trabalho dos EPs. O princípio de conservação de integridade é mantida, através da migração de processos quando existe a interação do usuário com o EP e as políticas de escalonamento podem ser alteradas através de uma coleção de utilitários ou através de uma interface gráfica.

O NQE (Kaplan & Nelson, 1994) é um ambiente composto de uma coleção de processos *daemon* que coletam informações sobre a carga dos EPs e as enviam para um processo mestre, responsável por realizar o escalonamento de aplicações. As aplicações que são submetidas ao ambiente são inseridas em filas específicas, de acordo com as características da aplicação

[20] ftp://ftp.cs.wisc.edu/condor (visitado em 20/08/2001)

fornecidas pelo usuário. Recursos importantes para o escalonador (por exemplo bibliotecas de software, arquitetura do EPS e unidades de disco) podem ser configurados pelo administrador, assim como as políticas de escalonamento e o número de filas do ambiente, que podem ser alterados em tempo de execução. Em caso de falha nos EPs, a política pode adaptar-se, mas não existe um escalonador mestre auxiliar como existe no *Sun Grid Engine*.

Seguindo o modelo de filas múltiplas, o *Portable Bach System* (PBS) é um ambiente que permite o escalonamento de aplicações em redes de estações de trabalho e máquinas com processamento paralelo (Bode *et al.*, 2000). O escalonamento é realizado através de políticas configuradas pelo usuário, utilizando prioridades e as informações sobre a aplicação, providas pelo usuário.

Outra possibilidade é a utilização de um software que possui características de um escalonador global, de maneira integrada a um sistema operacional único e distribuído. Exemplos de trabalhos nessa linha são os sistemas V-System, Sprite e o Mosix. O V-System é um dos primeiros ambientes que oferecem suporte para a criação e gerência de processos em um ambiente de execução formado por um arranjo computacional de estações de trabalho. Nesse sistema, o único suporte para a comunicação entre processos oferecido é a utilização de chamadas do sistema operacional para a comunicação entre processos (Theimer *et al.*, 1985). O Sprite (Douglis & Ousterhout, 1991) é um sistema similar ao V-System.

O Mosix é um software de gerenciamento de arranjos computacionais criados a partir de computadores pessoais executando o sistema operacional Linux (Barak & La'adan, 1998). Sua idéia é manter a mesma interface de programação para criação de processos existente no Linux. Além disso, o Mosix permite que aplicações desenvolvidas em MPI ou PVM sejam executadas e que possam ser migradas, visando o balanceamento de cargas (Amir *et al.*, 2003).

2.3.4 Síntese do software de escalonamento

A Tabela 2.3 ilustra as características e diferenças entre os ambientes de escalonamento. A comparação é realizada de acordo com estas características: *heterogeneidade*, *mensagens* (possibilita a utilização de ambientes de passagem de mensagens como PVM ou MPI), *checkpointing*, *migração de processos*, *balanceamento* (ambiente que permite a distribuição de carga para melhorar o desempenho), *sem recompilação* (realiza o escalonamento sem a necessidade de recompilação do código da aplicação), *EP exclusivo* (uso exclusivo do EP por apenas um processo), *interface gráfica* para configuração, *linha de comando*, *políticas dinâmicas* (que indicam a possibilidade de reconfiguração do ambiente em tempo de execução) e *domínio público* [21].

A análise das características do software de escalonamento permite identificar duas categorias, de acordo com a integração com o sistema operacional que é executados nos EPs. O Mosix, V-System e Sprite correspondem aos ambientes de escalonamento mais fortemente

[21] Legenda: (•) característica completamente suportada; (o) característica parcialmente suportada.

Tabela 2.3: Comparação entre os ambientes de escalonamento

Característica	Sun Grid	Condor	DQS	Load Balancer	Load Leveler	LSF	NQE	PBS	Task Broker	Amigo	V-System/ Sprite	Mosix
heterogeneidade	•	•	•	•	•	•	•	•	•	•		
mensagens	•	•		c		•	c			•		•
checkpointing	•	•	o	c	•	o	o	o				
mig. de processos	•	•		o	•		•				•	•
balanceamento	•		•	•	•	•	•	•	•	•	•	•
sem recompilação	o		o	•	o	•	•	•	•	•	•	•
EP exclusivo	•		•	•	•	•	o	•	o			
interface gráfica	•		•	•	•	•	•		•	•		
linha de comando	•	•	•	•	•	•	•	•	•	•		
pol. dinâmicas	•		•	•	•	•	•	•	•	•		
domínio público	•	•	•	•	•			•		•	•	•

acoplados ao sistema operacional. Essa integração permite que os detalhes sobre o sistema computacional fiquem ocultos ao usuário, e a submissão de aplicações é feita da mesma forma que seria realizada em um sistema uniprocessado. Tais ambientes dependem que todos os EPs executem o mesmo sistema operacional, visando formar a imagem de um sistema único e distribuído. Com isso, o suporte a heterogeneidade de software e de hardware fica reduzido, pois não é permitida a coexistência de diferentes sistemas operacionais no ambiente de execução.

O software de escalonamento integrado ao sistema operacional aproxima a característica de visão de um sistema único, comum em máquinas paralelas. Apesar disso, dentre esses ambientes apenas o `Mosix` possui oferece suporte para a execução de aplicações desenvolvidas através de bibliotecas paralelas como o PVM e o MPI. O objetivo principal desses ambientes é a criação de um sistema operacional único. O `Mosix`, o `Sprite` e o `V-System` realizam a migração de processos visando melhorar o desempenho do sistema através do balanceamento de cargas.

Os demais ambientes de escalonamento compõem uma segunda categoria e seguem um modelo onde existe um software escalonador global que centraliza as decisões. Esses ambientes são dedicados mais especificamente para aplicações paralelas e existe o suporte para a heterogeneidade de recursos, desde que o nível de integração com sistema operacional local aos EPs é mais reduzido. Essa integração reduzida é demonstrada pela necessidade de recompilação da aplicação para que os mecanismos de *chekpointing* disponibilizados pelo ambiente sejam empregados.

A migração de processos é realizada nos ambientes `Sun Grid Engine`, `NQE`, `LoadLeveler` e `Condor`. Neste último, a migração é realizada visando preservar as características interativas das estações de trabalho, enquanto que nos demais, a migração possui o objetivo principal de realizar o balanceamento de cargas visando melhorar o tempo de resposta do sistema. O `Condor` e o `Amigo` diferem também dos demais ambientes dessa categoria por permitirem mais de uma

processo sendo executado em cada EP, inclusive a utilização interativa dos EPs. Esses permitem a execução de aplicações desenvolvidas utilizando ambientes de passagem de mensagens. A possibilidade da utilização de diferentes políticas de escalonamento está presente na maioria dos ambientes dessa categoria, com a única exceção sendo o ambiente `Condor`.

2.4 Utilização do conhecimento sobre as aplicações paralelas no escalonamento

Para que o escalonamento cumpra seus objetivos, é importante o conhecimento amplo dos recursos do sistema computacional, da carga de trabalho do sistema e das aplicações paralelas que serão escalonadas. Esse conhecimento permite que a atribuição e a utilização dos recursos sejam otimizadas, assim como permite uma experimentação de melhores tempos de resposta pelo usuário (Sevcik, 1989; Anastasiadis & Sevcik, 1997).

O conhecimento sobre as aplicações paralelas pode ser organizado em dois níveis: global e isolado (Figura 2.5). O nível global agrupa o conhecimento sobre todas as aplicações paralelas que são executadas em um determinado sistema. Características comuns nesse nível são o tempo de execução das aplicações e a função do tempo de execução, que mapeia o tempo de execução de uma aplicação em função da quantidade de recursos que são atribuídos à ela. Tais características podem ser agrupadas e organizadas em classes, por exemplo o nome do usuário ou nome da aplicação, ou uma combinação dessas (Apon *et al.*, 1999; Gibbons, 1997; Smith *et al.*, 1998b).

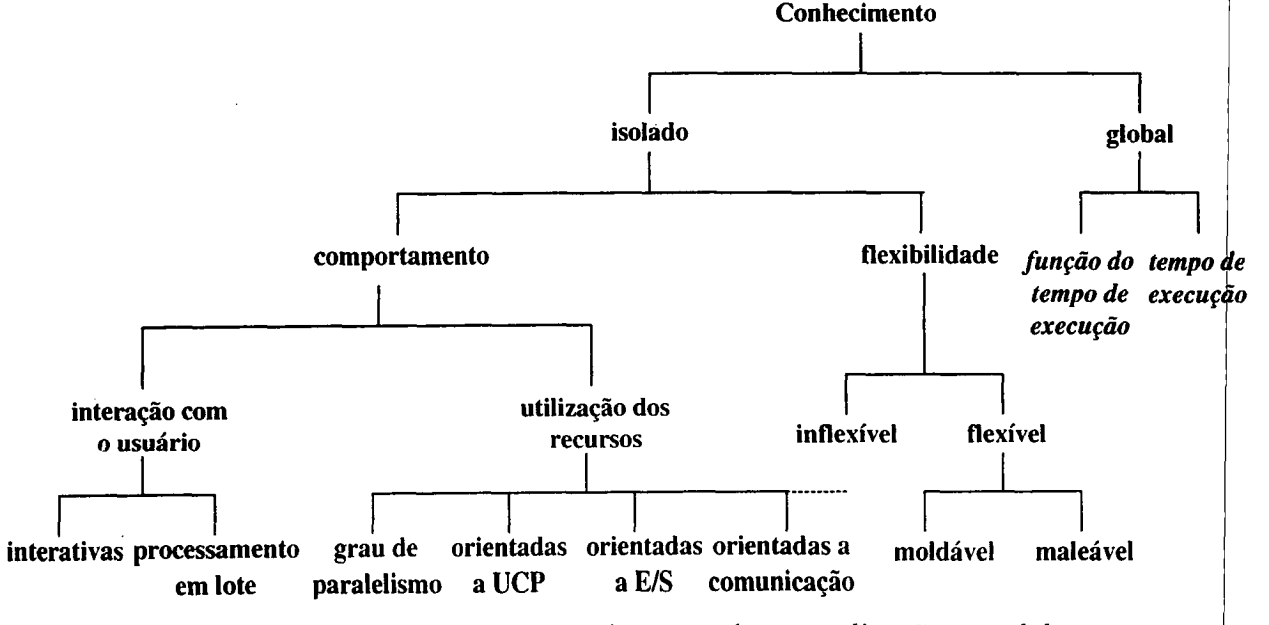


Figura 2.5: Classificação do conhecimento sobre as aplicações paralelas

Em um nível isolado, o conhecimento sobre as aplicações pode ser caracterizado individualmente, através de seu padrão de comportamento, em relação à interação com o usuário, em relação à utilização de recursos e através de sua flexibilidade. O comportamento em relação à

interação do usuário pode classificar as aplicações em classes interativas, que majoritariamente necessitam de interação com o usuário, e de processamento em lote, que têm baixa interação com o usuário. O comportamento predominante em relação à utilização de recursos pode ser representado através das classes orientadas a UCP (*CPU Bound*), orientadas a E/S (*I/O Bound*) e orientadas a comunicação (*Communication Intensive*). Além disso, a utilização dos recursos pode ser caracteriza através do grau de paralelismo (que representa a quantidade de EPs que maximiza a *eficiência* da aplicação), pelo paralelismo mínimo (que indica a quantidade mínima EPs que deve ser obrigatoriamente atendida pelo sistema), pelo paralelismo máximo (que indica o número máximo de EPs que a aplicação pode utilizar com eficiência) ou pela função do tempo de execução.

O conhecimento específico das aplicações pode incluir a sua flexibilidade em relação às decisões de atribuição de recursos do sistema (Feitelson *et al.*, 1997). Aplicações podem ser escritas de acordo com o modelo inflexível, onde a aplicação requisita uma quantidade fixa de EPs que deve ser obrigatoriamente atendida pelo sistema. Geralmente, aplicações escritas de acordo como o modelo de paralelismo funcional apresentam um comportamento inflexível. Aplicações flexíveis podem ser escritas através dos modelos moldável ou maleável. O modelo moldável assume que a aplicação pode adaptar-se, no início de sua execução, a uma quantidade de EPs diferente daquela solicitada. Similarmente, aplicações maleáveis apresentam grande flexibilidade de adaptação, não restrita ao início de sua execução, mas extensível durante toda a sua execução. Aplicações flexíveis são geralmente construídas através do modelo de paralelismo de dados (Silva & Scherson, 2000).

Fontes comuns para obtenção do conhecimento são os usuários que submetem a aplicação, através de alguma forma de descrição da aplicação, e o histórico de execução das aplicações. Embora o conhecimento sobre as aplicações paralelas possa ser perfeito em situações específicas, como por exemplo sistemas fechados e com o emprego do escalonamento estático, em sistemas mais realistas o nível de conhecimento é freqüentemente imperfeito, ou até mesmo inexistente.

Vários autores têm demonstrado que o escalonamento orientado com o conhecimento sobre as aplicações paralelas possui melhor desempenho quando comparado ao caso que não emprega tal conhecimento. Sevcik (1989) demonstra, através de modelos resolvidos analiticamente e por simulação, que o emprego do paralelismo da aplicação, expresso através de diferentes características, pode melhorar substancialmente o escalonamento de aplicações paralelas em relação ao escalonamento que não emprega essas características. Empregando políticas *space sharing* em máquinas paralelas homogêneas, as aplicações paralelas são caracterizadas através de perfis, que englobam características como paralelismo mínimo, paralelismo máximo, fração seqüencial da aplicação (vide Seção 2.2.3) e paralelismo médio (número médio de EPs empregados durante a execução da aplicação).

Para a identificação dos estados de utilização de recursos de processos seqüenciais, Devarakonda & Iyer (1989) propõem uma técnica da área de reconhecimento de padrões para a

predição do tempo de UCP, operações de entrada/saída e utilização de memória de um programa no início de sua execução, dada a identidade desse programa. Essa técnica emprega uma forma de categorização estatística, através do algoritmo *k-means*, para a identificação de estados de utilização de recursos de todos os processos executados em um sistema *UNIX* uniprocessado. Essa técnica é também explorada, de maneira similar, por outros autores (Dimpsey & Year, 1991, 1995). Essas técnicas não contemplam a análise do comportamento das aplicações paralelas em relação a utilização da rede de comunicação.

Brecht & Guha (1996) empregam as características das aplicações, expressas através do grau de paralelismo e do tempo de execução, para melhorar as decisões em políticas da classe *space sharing* adaptativas. Os autores concluem que políticas que usam estimativas apresentam um desempenho próximo àquele observado quando o comportamento da aplicação é previamente conhecido.

Feitelson & Nitzberg (1995) mostram que características das aplicações, como a função do tempo de execução, podem ser estimadas em determinadas situações. Os autores concluem, no contexto de máquinas verdadeiramente paralelas, que aplicações com maiores índices de paralelismo médio tendem a possuir tempos de execução mais elevados. Além disso, os autores verificam que o tempo de chegada e o tempo de execução das aplicações têm um alto coeficiente de variação e que execuções repetidas da mesma aplicação, empregando o mesmo número de EPs, apresentam coeficiente de variação pequeno, o que indica que o histórico de uma aplicação pode ser empregado para melhorar as decisões do escalonador. Adicionalmente, os autores verificam que aproximadamente 2/3 das aplicações paralelas são executadas mais de uma vez, sendo que algumas com um número significante de repetições.

Feitelson *et al.* (1997) observam que execuções repetidas da mesma aplicação tendem a apresentar padrões similares de consumo de recursos e que muitas informações, relativas ao comportamento das aplicações, podem ser descobertas sem a cooperação explícita do usuário, através da observação do histórico de execuções passadas.

Gibbons (1997) relata a implementação de um histórico do sistema organizado através das seguintes características: nome da aplicação (arquivo executável), usuário que iniciou a aplicação, número de EPs empregados, tempo total de resposta e quantidade de memória utilizada. Nesse trabalho, essa organização produziu tempos médios de resposta com coeficientes de variação menores em relação ao coeficiente de variação de todo o sistema, sem organização em classes. Com a utilização desse histórico, o autor define algoritmos que empregam o conhecimento sobre execuções passadas para melhorar o escalonamento de aplicações paralelas. Esse estudo é direcionado para máquinas verdadeiramente paralelas homogêneas, com políticas de escalonamento da classe *space sharing* e a informação derivada do histórico refere-se apenas ao tempo de execução da aplicação e a função do tempo de execução. O autor conclui que o uso das características das aplicações derivadas de um histórico pode melhorar substancialmente, através de menores tempos de resposta experimentados, as políticas de escalonamento.

A predição do tempo de execução de aplicações paralelas é estudada também no trabalho de Smith *et al.* (1998a). Nesse trabalho, traços de execução são tratados como uma base de experiência a partir da qual, através de algoritmos de busca, são definidas categorias mais adequadas para gerar predições. Tal metodologia é seguida também por Krishnaswamy *et al.* (2004), que realiza predições sobre o tempo de execução de aplicações paralelas através de algoritmos específicos, e por Downey (1997), que categoriza todas as aplicações previamente executadas através do identificador da fila do escalonador e cria modelos através de distribuições de probabilidade. Esses modelos são empregados para realizar a predição do tempo de execução.

Iverson *et al.* (1999) empregam um algoritmo estatístico para a predição do tempo de execução de aplicações paralelas, com o objetivo de melhorar as decisões do escalonamento estático em um cenário heterogêneo. Similarmente, Kapadia *et al.* (1999) avaliam os algoritmos dos vizinhos mais próximos, vizinhos mais próximos ponderados pela distância e regressão localmente ponderada para a predição do tempo de execução de ferramentas de simulação. Nesses trabalhos, a avaliação das técnicas propostas restringem-se à utilização de cargas de trabalho sintéticas e traços de execução de ferramentas específicas e não consideram a carga do sistema computacional como um todo. Apesar disso, esses trabalhos mostram que é possível utilizar algoritmos de aprendizado de máquina para adquirir conhecimento a partir de traços de execução e informações sobre o sistema computacional.

Anastasiadis & Sevcik (1997) investigam como diferentes níveis de conhecimento sobre a aplicação e sobre a carga de trabalho do sistema computacional podem melhorar as decisões do escalonador, através da modelagem das aplicações com o uso de uma função do tempo de execução, que relaciona algumas características da aplicação. O estudo permite concluir que políticas de escalonamento que exploram as informações existentes sobre as aplicações paralelas apresentam melhores desempenhos que políticas que não empregam tais informações.

Arpaci-Dusseau *et al.* (1998) propõem, visando substituir a sincronização explícita, um mecanismo de escalonamento cooperativo baseado nas informações coletadas durante a execução das aplicações paralelas. A informação coletada é a quantidade de mensagens que chegam aos processos, para decidir pelo *spin time* ou para decidir pelo bloqueio ou não dos processos.

Apon *et al.* (1999) apresentam um esquema de aprendizado de máquina, com o conhecimento sendo representado estatisticamente, que permite predizer a quantidade ideal de EPs que deve ser atribuída para aplicações moldáveis, baseando-se no estado atual do sistema. Esse estado é representado através da quantidade de aplicações paralelas existentes na fila e da quantidade de aplicações que estão sendo executadas. Esse esquema visa obter a quantidade de EPs homogêneos que devem ser atribuídos à aplicação de acordo com uma política de escalonamento da classe *space sharing*. O conhecimento sobre o comportamento das aplicações paralelas não é explicitamente empregado, e sim o conhecimento global sobre o desempenho do sistema em execuções anteriores, representado pela medida de desempenho *power*.

Silva & Scherson (2000) coletam a informação sobre a execução de aplicações paralelas durante a execução de chamadas ao sistema e trocas de contexto, visando melhorar o *throughput* no escalonamento *time-sharing* cooperativo. Os autores propõem esquemas de classificação da aplicação entre orientada a UCP, orientada a E/S e orientada a comunicação, através do emprego de estimadores bayesianos e modelagem através de conjuntos nebulosos (*fuzzy sets*) (Cortes, 2004). Essa classificação é empregada para decidir qual é o próximo processo a ser escalonada ou qual é o *spin time* que melhor atende à carga atual do sistema.

Corbalan *et al.* (2001) defendem que o escalonador deve tomar decisões, com respeito à quantidade de recursos atribuídos à aplicação, baseado não apenas nas requisições do usuário, mas também nas informações providas pela execução das aplicações, visando garantir uma utilização eficiente dos recursos. Os autores empregam essas informações e exploram a maleabilidade da aplicação para propor melhorias em mecanismos de escalonamento cooperativo.

Algumas ferramentas têm sido desenvolvidas para a coleta de informações sobre aplicações paralelas. Perfis de execução de aplicações são gerados através de monitoração e a instrumentação do código fonte das aplicações paralelas. A ferramenta *Pablo* (Reed *et al.*, 1993, 1996; Madhyastha & Reed, 2002) captura informações sobre as operações de E/S de aplicações, como tempo, duração e tamanho. Essas informações podem ser analisadas de maneira *off-line*, ao término da execução da aplicação paralela, através de software específicos para a visualização.

Na mesma linha, a ferramenta *Charisma* (Nieuwejaar *et al.*, 1996) permite a análise de E/S de cargas de trabalho paralelas. Tais ferramentas necessitam de recompilação das aplicações, não contemplam operações na rede de comunicação e são direcionadas para monitorar especificamente para o acesso à unidades de armazenamento. Um exemplo de aplicação dessas ferramentas é descrito no trabalho de Madhyastha & Reed (2002). Nesse trabalho, são empregados algoritmos de aprendizado de máquina para classificar o padrão de acesso de aplicações paralelas em arquivos. Os esquemas propostos utilizam como algoritmos redes neurais baseadas em perceptrons multi-camadas e modelos de Markov ocultos.

Harchol-Balter & Downey (1997) mostram como o conhecimento sobre o tempo de vida dos processos no sistema pode ser empregado para melhorar as decisões do escalonamento que visa o balanceamento de cargas. Esses autores analisam traços de execução de um sistema computacional baseado em estações de trabalho e modelam a distribuição de probabilidade para o tempo de vida de aplicações seqüenciais. Através desses modelos, o algoritmo de balanceamento de cargas emprega o conhecimento sobre os processos seqüenciais para decidir quais são os processos mais adequados para serem migrados entre os EPs. Esse trabalho é estendido pelo trabalho de Mello & Senger (2004), que analisam também a utilização de UCP para a tomada de decisões.

2.5 Considerações finais

O paralelismo é um conceito intrínseco ao hardware. As unidades funcionais dos elementos de processamento, circuitos de controle de barramento, memória e periféricos atuam paralelamente, embora esse paralelismo não transpareça na maioria das vezes diretamente para o usuário e para o programador. Sob esse ponto de vista, toda a computação pode ser considerada como sendo paralela, embora o termo *Computação Paralela* seja empregado em um nível mais alto, onde busca-se a execução paralela de um conjunto de instruções de máquina.

Embora o suporte em hardware para a computação paralela possa ser provido por diferentes arquiteturas, máquinas paralelas de memória distribuída, compostas por redes de estações de trabalho e computadores pessoais têm várias características atrativas para o processamento paralelo, como baixo custo de aquisição, manutenção e atualização. Além disso, essas arquiteturas, empregadas em conjunto com ferramentas de software ou ambientes de passagem de mensagens, são as mais apropriadas para as exigências atuais do software paralelo. O contexto atual exige que as aplicações paralelas possam ser executadas em diferentes plataformas e, preferencialmente, com a possibilidade de aproveitamento do código seqüencial legado e reuso do código paralelo.

O desempenho obtido com a computação paralela pode ser quantificado através de diferentes medidas, que devem ser empregadas de acordo com os objetivos da avaliação de desempenho. O *speedup* e a *eficiência* refletem o ganho obtido com o processamento paralelo. Medidas como *tempo de resposta*, *throughput*, *power* e *slowdown* são constantemente empregadas em trabalhos de avaliação de desempenho, onde, através de simulação ou aferição, busca-se representar o comportamento do sistema computacional sob diferentes nuances de carga. Tais medidas e índices de carga utilizadas com sucesso em arquiteturas homogêneas, muitas vezes têm sua adeqüabilidade reduzida para sistemas arquiteturalmente e configuracionalmente diferentes. A diversidade de recursos, juntamente com as diferentes necessidades dos usuários, criam vários desafios para o software distribuído.

Dentre as lacunas que precisam ser preenchidas pelo software encontra-se o escalonamento de aplicações paralelas (*Job Scheduling*). Embora muitos trabalhos tenham sido desenvolvidos para o escalonamento em sistemas de memória compartilhada e/ou sistemas de memória distribuída homogêneos, apenas recentemente o problema de escalonamento em ambientes computacionais heterogêneos tem sido pesquisado.

O escalonamento de aplicações paralelas em sistemas computacionais distribuídos tem sido uma área de pesquisa importante e desafiante. Essa importância deve-se ao fato de que as decisões do escalonador, em sua atividade de atribuição de recursos entre as aplicações, refletem-se diretamente no tempo de resposta e capacidade de trabalho do sistema. Os desafios da área estão relacionados à capacidade do escalonador de lidar com diferentes fatores, que são naturalmente dinâmicos, tais como as características das aplicações, carga de trabalho e heterogeneidade dos recursos computacionais.

As políticas para o gerenciamento de recursos pode sem organizadas em dois planos: espaço (*space sharing*) e tempo (*time sharing*). A abordagem *space sharing* minimiza a troca de contexto entre os processos e reduz a perda de desempenho devido ao custos de sincronização, pois todos os processos de uma aplicação executam simultaneamente em diferentes EPs. Apesar disso, essa abordagem não é eficiente para todos os tipos de aplicações e para sistemas heterogêneos, desde que cria problemas de fragmentação [22] e subutilização dos recursos computacionais.

O compartilhamento do tempo é atrativo para sistemas distribuídos, pois os EPs podem executar mais de um processo de uma aplicação ou de aplicações diferentes, permitindo um bom aproveitamento dos recursos computacionais. A falta de sincronização têm sido remediada através do escalonamento cooperativo, que fornece mecanismos para a sincronização dos escalonadores locais.

Vários ambientes de escalonamento têm sido propostos. Muitos ainda herdam características do software de escalonamento implementados em máquinas paralelas, enquanto outros são direcionados para sistemas distribuídos. Embora existam ambientes de escalonamento de bom desempenho para sistemas homogêneos, ainda não existe um "vencedor" para sistemas distribuídos heterogêneos (Kaplan & Nelson, 1994). Características como o tratamento da carga de trabalho em sistemas heterogêneos, suporte para ferramentas de software, possibilidade de reconfiguração dinâmica de políticas e mecanismos de escalonamento, monitoração do desempenho e transparência ainda precisam ser incorporadas ao software de escalonamento.

Muitos trabalhos de avaliação de desempenho relatam que o uso do conhecimento existente sobre aplicações é vital para um melhor desempenho no escalonamento. Implementações de ambientes de escalonamento como o NQE, *Sun Grid Engine*, LSF e o AMIGO, permitem que o usuário caracterize a sua aplicação explicitamente, através de algum tipo de linguagem de controle ou relacionamento com algoritmos de escalonamento, ou implicitamente, selecionando a fila em que a aplicação será submetida. Apesar disso, muitas vezes o usuário não possui conhecimento suficiente para caracterizar a sua aplicação quanto ao comportamento desta, ou ainda, a caracterização pode ser errônea.

Apesar da importância do uso do conhecimento existente das aplicações para o escalonamento de aplicações paralelas em sistemas computacionais distribuídos, poucos trabalhos têm sido direcionados para métodos práticos de obtenção e utilização desse conhecimento. Além disso, grande parte dos trabalhos desenvolvidos para a obtenção e o emprego do conhecimento sobre as aplicações paralelas para o escalonamento são geralmente direcionados para sistemas paralelos compostos de máquinas verdadeiramente paralelas, com elementos de processamento homogêneos e sob a gerência exclusiva de um ambiente de escalonamento. Embora sejam aplicáveis em domínios específicos, tais restrições diminuem a adeqüabilidade dessas soluções para sistemas distribuídos, onde os EPs são potencialmente heterogêneos e influenciados por

[22] A fragmentação ocorre quando o sistema não possui uma quantidade suficiente de EPs disponíveis para atender a requisição da aplicação paralela. Nesse caso a aplicação paralela deve aguardar até que mais EPs sejam liberados.

usuários com diferentes necessidades de computação, muitas vezes sem um controle total de um software de escalonamento.

Outra restrição dos trabalhos analisados é a falta de uma proposta de organização para o repositório de informações ou histórico das aplicações paralelas. Nesse contexto, grande parte dos trabalhos assume um conhecimento sobre as aplicações é representado estatisticamente, relacionando valores do tempo de resposta da aplicação. Nesse ponto, é importante, além do tempo de execução da aplicação, caracterizar as aplicações de maneira isolada através de perfis, com respeito ao seu comportamento e flexibilidade na utilização de recursos, considerando ambientes distribuídos e heterogêneos.

Esta tese é dedicada à exploração de características de aplicações paralelas, visando representar o comportamento das aplicações paralelas e predizer suas características de execução. Para isso, inicialmente são selecionados um conjunto de aplicações paralelas reais e sintéticas, que representam características reais de execução de aplicações paralelas, e um conjunto de traços de execução de centros de computação de alto desempenho. Essas aplicações e os traços de execução são empregados por algoritmos de aprendizado de máquina, visando explorar as características de execução e da carga de trabalho comum ao sistema para a aquisição de conhecimento, classificação e predição de características de aplicações paralelas. O próximo capítulo trata das técnicas e algoritmos de aprendizado de máquina utilizados para atingir os objetivos deste trabalho.

CAPÍTULO

3

Aprendizado de Máquina

3.1 Considerações iniciais

Aprendizado de máquina é uma área da inteligência artificial que consiste no desenvolvimento e utilização de técnicas computacionais capazes de extrair conhecimento a partir de amostras de dados (Mitchell, 1997). Nesse contexto, o conhecimento refere-se à informação armazenada ou aos modelos utilizados para interpretar, predizer e responder adequadamente ao problema considerado. O aprendizado consiste na aquisição e na habilidade de utilização desse conhecimento (Witten & Frank, 1999).

Os algoritmos computacionais que implementam técnicas de aprendizado de máquina têm como objetivos encontrar e descrever padrões a partir dos dados obtidos do ambiente. O aprendizado pode ser realizado de diferentes formas, de acordo com um paradigma. Exemplos de paradigmas são: simbólico, estatístico, baseado em instâncias, conexionista e genético. A tarefa de aprender através desses paradigmas consiste em escolher ou adaptar os parâmetros de representação do modelo. Independentemente do paradigma, os algoritmos de aprendizado têm a tarefa principal de aprender um modelo a partir do ambiente e manter esse modelo consistente de modo a atingir os objetivos de sua aplicação. Explicar os dados e fazer predições são objetivos comuns da aplicação desses algoritmos.

Este capítulo apresenta as técnicas de aprendizado de máquina utilizadas neste trabalho. A primeira técnica descrita utiliza o paradigma conexionista e emprega a arquitetura de redes neurais ART e é apresentada com detalhes na Seção 3.2. Na segunda técnica, é empregado o paradigma baseado em instâncias, que é apresentado na Seção 3.3.

Essas técnicas são utilizadas para a definição de dois algoritmos de aprendizado incremental. Esses algoritmos são utilizados nas tarefas de aquisição de conhecimento sobre

aplicações paralelas, empregando características de execução dos processos em aplicações paralelas e características de submissão de aplicações, gravadas em traços de execução de sistemas computacionais reais.

3.2 Redes neurais ART

Redes neurais artificiais RNAs são sistemas computacionais que emulam estruturas neurais biológicas (de P. Braga *et al.*, 2000; Fausett, 1994). Tais sistemas são compostos por unidades de processamento simples, chamadas de neurônios, nós, células ou unidades, que calculam certas funções matemáticas. Essas unidades de processamento são interligadas por conexões, geralmente unidimensionais, nas quais associam-se pesos, que servem para armazenar o conhecimento representado pela rede e para ponderar a entrada recebida por cada neurônio da rede.

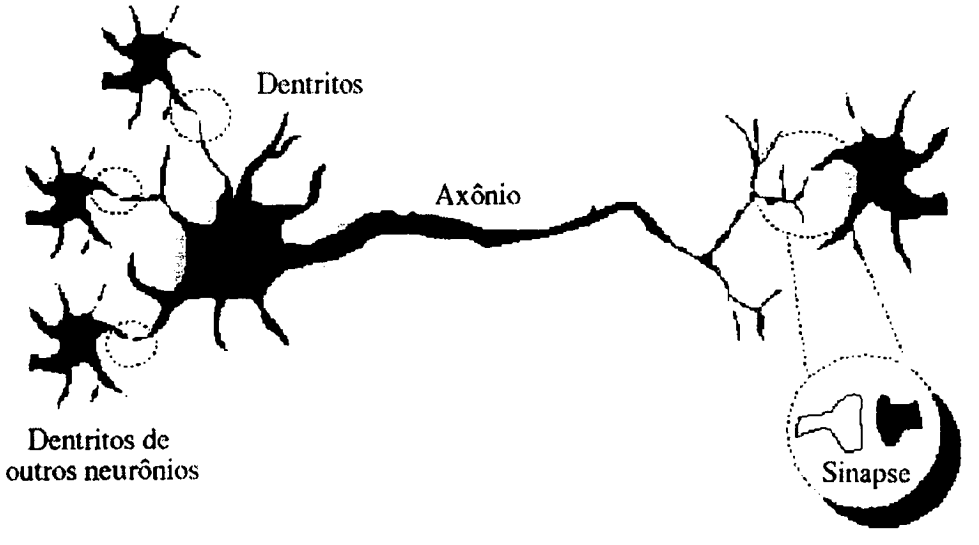


Figura 3.1: Neurônio simplificado

Em sistemas computacionais baseados em redes neurais, cada neurônio apresenta um estado interno, chamado de ativação, que é função das entradas que o neurônio recebe. Geralmente, um neurônio envia sua ativação como sinal para os outros neurônios da rede. A Figura 3.2 ilustra a ativação de um neurônio Y. Esse neurônio recebe sinais dos neurônios X_1, X_2 e X_3, que têm respectivamente os sinais de saída x_1, x_2 e x_3. Os pesos sinápticos das conexões entre os neurônios são respectivamente w_1, w_2 e w_3. O sinal de entrada $y_{entrada}$, que chega ao neurônio Y, corresponde a soma ponderada dos sinais x_1, x_2 e x_3:

$$y_{entrada} = w_1 x_1 + w_2 x_2 + w_2 x_2 \tag{3.1}$$

A ativação T_y do neurônio Y é dada por uma função da entrada $y_{entrada}$, i.e. $T_y = f(y_{entrada})$.

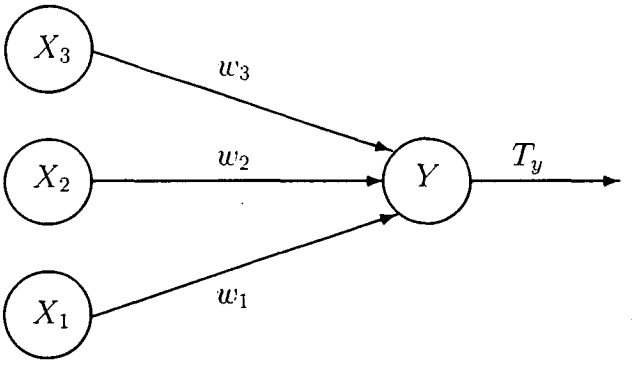


Figura 3.2: Neurônio artificial Y conectado a outros neurônios

Na Tabela 3.1, estão descritos exemplos de funções de ativação. As funções de ativação são escolhidas de acordo com a organização dos neurônios dentro da rede.

A arquitetura da rede descreve a forma pela qual os neurônios são organizados e interconectados. Freqüentemente os neurônios são organizados em camadas, de forma que o número de camadas possa variar em apenas uma camada ou várias, dependendo da arquitetura em questão. Entre os neurônios de uma mesma camada podem existir conexões sinápticas.

Tabela 3.1: Funções de ativação principais

Função de ativação	**Descrição**
$f(x) = x$	função identidade
$f(x) = \alpha x$	função linear, onde α é o coeficiente angular da reta
$f(x) = \begin{cases} +\gamma, & \text{se } x \geq +\gamma \\ x, & \text{se } \lvert x \rvert < +\gamma \\ -\gamma, & \text{se } \lvert x \rvert \leq -\gamma \end{cases}$	função linear restrita ao intervalo $[-\gamma, +\gamma]$
$f(x) = \begin{cases} +\gamma, & \text{se } x > 0 \\ -\gamma, & \text{se } x \leq 0 \end{cases}$	função passo para no intervalo$[-\gamma, +\gamma]$
$f(x) = \frac{1}{1+exp(-x)}$	função sigmóide logística

A solução de um determinado problema através da utilização de RNAs necessita de uma fase inicial de aprendizagem, na qual um conjunto de exemplos de treinamento são apresentados à rede, que extrai automaticamente as características necessárias para representar o conhecimento sobre o domínio. A capacidade de aprender a partir de exemplos e generalizar o conhecimento obtido é o atrativo principal das redes neurais artificiais. A generalização é a capacidade de uma rede neural aprender através de um conjunto reduzido de exemplos e, posteriormente, informar respostas coerentes para dados não conhecidos. Redes neurais podem ser utilizadas em problemas onde busca-se a classificação de dados, em problemas que necessitam de agrupamento (*clustering*) de dados e em problemas onde busca-se encontrar um modelo que represente

funções multi-variáveis, com custo computacional reduzido. Tais tarefas são encontradas com freqüência em áreas como processamento de sinais, controle de dispositivos mecânicos e digitais, reconhecimento de padrões, reconhecimento de voz e aplicações comerciais, como avaliação de risco de crédito (Senger & Caldas Jr., 2001; Fausett, 1994).

Uma das mais importantes características de redes neurais artificiais é a sua capacidade de generalização do conhecimento obtido. A generalização é uma propriedade que permite que padrões de entrada não conhecidos anteriormente no aprendizado da rede sejam corretamente classificados pela rede. Quando uma rede é treinada com um conjunto de padrões, espera-se que a rede produza as saídas corretas para quaisquer padrões que sejam apresentados futuramente. À medida que o tempo passa, o conhecimento sobre o domínio pode alterar-se, observando-se diferentes padrões de entrada que representam a situação atual do mundo real. Para esse tipo de aplicações, em que o conhecimento sobre o domínio do problema é dinâmico, o desempenho da rede pode decair gradativamente, à medida que o conhecimento sobre o problema evolui. A capacidade de uma arquitetura de rede adaptar-se indefinidamente aos padrões de entrada é chamada de *plasticidade* (Carpenter & Grossberg, 1988).

Um abordagem para fazer com que a rede adapte-se aos novos padrões de entrada seria treinar a rede novamente com os padrões de entrada recentes, ou treinar novamente a rede com os padrões de entrada iniciais juntamente com os padrões de entrada atuais (de P. Braga *et al.*, 2000). Entretanto, tal abordagem poderia levar a perda de informações aprendidas anteriormente, além de ser inviável em determinadas situações, principalmente quando os padrões de entrada iniciais não estão disponíveis ou quando existem restrições quando ao tempo de resposta da rede. O aprendizado deve ser não somente *plástico*, mas também *estável*. Uma algoritmo de aprendizado é estável quando permite que o conhecimento seja atualizado, sem perder o conhecimento previamente armazenado. Tal conflito é conhecido como dilema da *estabilidade/plasticidade*. Para a solução desse conflito, busca se a definição de uma arquitetura de RNA incremental, de forma que não seja necessário recomeçar o treinamento da rede a cada vez que seja apresentado à rede um novo padrão de entrada e que o conhecimento obtido previamente seja conservado e estendido. Uma arquitetura de RNAs desenvolvida visando resolver tal dilema é a arquitetura ART.

A família de rede neurais ART (*Adaptive Ressonance Theory*) consiste em arquiteturas que aprendem em tempo real códigos de representação estáveis em resposta a uma seqüência arbitrária de padrões de entrada (Carpenter & Grossberg, 1988). Nas arquiteturas ART, o aprendizado é tratado como uma ação dinâmica, de forma que a rede possa continuamente adaptar-se aos novos padrões de entrada. O aprendizado nas redes ART é não-supervisionado [1] e por competição.

[1]Embora existam formas de aprendizado supervisionado nessas arquiteturas, este capítulo trata basicamente da versão original dessa arquitetura, a versão não supervisionada

Aprendizado não-supervisionado significa que a rede consegue aprender tendo como entrada padrões não rotulados, ou seja, sem empregar um mecanismo supervisor externo, ao contrário de redes com aprendizado supervisionado, em que a rede recebe um conjunto de treinamento previamente classificado e rotulado. Sistemas biológicos, nas fases iniciais de desenvolvimento da visão e audição, são exemplos de aprendizado não-supervisionado (de P. Braga *et al.*, 2000). Nessa forma de aprendizado, a rede tem a habilidade de formar representações internas para codificar as entradas através de um conjunto de unidades de saída ou representação. É comum nessa forma de aprendizado arquiteturas formadas por uma camada de entrada, uma camada de saída ou representação e um conjunto de conexões ou vetores de pesos entre essas camadas. A camada de entrada realiza o processamento inicial do conjunto de entrada e a camada de saída é responsável pela representação dos dados, através da criação de grupos, onde cada grupo descreve (codifica) um subconjunto dos padrões de entrada. Assim, o aprendizado consiste em encontrar uma classe para representar a entrada e em modificar repetidamente tais vetores de pesos em resposta a entrada, visando adaptar continuamente a rede às entradas.

O aprendizado por competição consiste em um dos métodos para a implementação do aprendizado não-supervisionado, em que as unidades da camada de saída disputam entre si para serem ativadas em resposta ao padrão de entrada. Essa disputa descreve uma competição entre as unidades de representação para decidir qual delas será a vencedora. Apenas a unidade de representação vencedora é escolhida para representar o padrão de entrada e tem os seus respectivos vetores de pesos modificados (adaptados).

As arquiteturas ART são projetadas de forma que o usuário possa controlar o grau de similaridade entre os padrões agrupados na mesma unidade de saída. Esse controle permite que a rede seja mais ou menos sensível às diferenças existentes entre os padrões de entrada, e consiga gerar mais ou menos grupos em resposta a esse controle. Além disso, o aprendizado nas redes ART nunca termina: a rede continuamente pode adaptar-se aos dados de entrada, criando novas unidades de processamento para aprender os padrões, se necessário.

3.2.1 Arquitetura básica das redes ART

As redes neurais ART (*Adaptive Ressonance Theory*) consistem em uma família de arquiteturas para o aprendizado definida por Carpenter & Grossberg (1989a). Essas arquiteturas são biologicamente motivadas, no sentido em que as ações coletivas de um grupo de células podem produzir um comportamento que é análogo ao fenômeno cognitivo em humanos e animais. Neste capítulo, mantém-se o foco na funcionalidade e são apresentados os aspectos das redes ART que aplicam-se aos problemas de aprendizado e classificação. A organização básica da arquitetura ART envolve dois componentes principais: o subsistema de atenção (*attentional sub-system*) e o subsistema de orientação (*orienting sub-system*), como mostra a Figura 3.3.

Subsistema de atenção

O subsistema de atenção é composto por uma camada de pré-processamento das entradas (F_0), de uma camada de representação das entradas (F_1) e de uma camada de representação das categorias ou classes (F_2). A função principal das camadas F_0 e F_1 é o processamento inicial do vetor de entrada. Tal processamento pode ser simples ou envolver uma série de operações de normalização e filtragem de ruído, dependendo da arquitetura da rede. A quantidade de unidades de processamento das camadas F_0 e F_1 também depende da arquitetura da rede. A camada F_2 tem como função principal a representação ou agrupamento das entradas. O número de de neurônios nessa camada é igual ao número de categorias ou classes aprendidas através dos padrões de entrada. A quantidade de unidades de processamento da camada F_2 é igual a quantidade de grupos ou protótipos criados para representar os dados de entrada. As unidades de processamento da camada F_2 são dinâmicas, de forma que novas unidades de processamento possam ser criadas à medida que for necessário.

As camadas F_1 e F_2 são ligadas através de dois conjuntos de conexões direcionadas. A camada F_1 envia seus sinais de ativação para a camada F_2 através de pesos *bottom-up*. O peso *bottom-up* que conecta o i-ésimo neurônio da camada F_1 até o j-ésimo neurônio da camada F_2 é chamado de b_{ij}. Tais conexões são também chamadas de pesos *feedforward*. Os neurônios da camada F_2 enviam seu sinais para a camada F_1 através de pesos *top-down*, chamados de t_{ji} ou conexões *feedback*. As conexões *bottom-up* e *top-down* são adaptativas e têm um papel importante no aprendizado da rede e são chamadas de memória de longa duração (*short term memory*). De fato, o conhecimento obtido nas arquiteturas ART é armazenado na memória de longa duração.

Subsistema de orientação

O subsistema de orientação é formado por um mecanismo que controla o nível de similaridade entre os padrões armazenados no mesmo neurônio de saída, chamado de *reset*. Esse mecanismo é importante durante o processo de aprendizado, pois controla a dinâmica de movimentação de dados pela rede.

3.2.2 Aprendizado nas redes ART

O aprendizado na rede ART é competitivo. Quando um padrão de entrada é apresentado à rede, é realizado um pré-processamento desse padrão pela camada F_0. A seguir, a camada F_1 recebe o padrão tratado e é calculada a ativação dos neurônios da camada F_2. O neurônio J com maior atividade da camada F_2 torna-se um candidato para codificar o padrão de entrada. Nesse ponto, os outros neurônios tornam-se inativos e a camada de entrada combina a informação entre o padrão de entrada e o neurônio candidato. O neurônio torna-se vencedor e aprende o

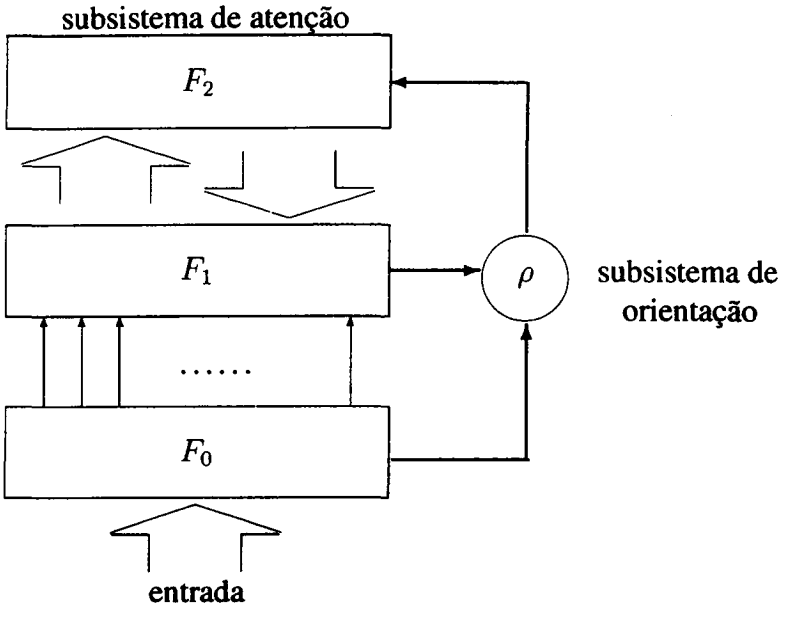


Figura 3.3: Arquitetura básica das redes ART

vetor de entrada, dependendo da similaridade entre o vetor de entrada e o vetor de pesos t_{Ji}. Essa decisão é tomada pela unidade de *reset*, que compara os sinais provenientes da camada F_1, verificando a similaridade entre o vetor de entrada e o vetor de pesos do neurônio candidato. Se a similaridade entre o vetor de entrada e o vetor de pesos for menor que um determinado limiar, o neurônio candidato é marcado como inibido e um novo candidato é escolhido. Tal seqüência é mantida até que encontre-se um neurônio que possa representar o padrão ou até que todos os neurônios da camada de saída estejam inibidos. Nesse caso, a rede pode criar um novo neurônio para armazenar o padrão ou informar que o padrão não pode ser representado pela rede. O Algoritmo 3.1 apresenta o funcionamento básico do aprendizado para a rede ART.

O nível de similaridade que é necessário para que um padrão de entrada seja aprendido por um neurônio é controlado por um parâmetro controlado pelo usuário, chamado de *vigilância* e representado pela letra grega ρ. Sua função é controlar o estado de cada neurônio da camada F_2. Os neurônios da camada F_2 podem estar em três estados diferentes: ativo, inativo e inibido. O neurônio encontra-se no estado ativo quando torna-se um candidato para aprender o padrão de entrada; inativo, quando perde a competição para ser um candidato; e inibido, quando o neurônio foi previamente escolhido como candidato para o padrão de entrada mas não mostrou-se similar o suficiente para aprender o padrão. No estado inibido, o neurônio não poderá competir novamente para aprender o padrão corrente. A condição de parada (linha 2 do Algoritmo 3.1) define a quantidade de ciclos [2] de treinamento para a rede. O número de ciclos pode ser fixo, ou pode-se verificar se houve diferença nos valores dos vetores de pesos após a execução do ciclo anterior. Caso não tenha sido observada tal diferença, não é necessário apresentar novamente os padrões

[2]Um ciclo de treinamento corresponde a apresentação de um conjunto de padrões a rede. Freqüentemente, o treinamento corresponde a apresentação cíclica de um conjunto de padrões, até que o valores dos pesos não apresentem uma alteração significativa ao fim do ciclo.

à rede. O sinal de *reset* é calculado de acordo com a similaridade entre o vetor de entrada e o vetor de pesos do neurônio candidato. Tal sinal define se o candidato será aprovado ou não. Se o candidato for aprovado, o vetor de pesos do neurônio vencedor é adaptado, combinando-se com o vetor de entrada para produzir outro vetor de pesos.

Algoritmo 3.1 Treinamento básico nas redes ART

```
1: inicialize os parâmetros
2: enquanto condição de parada for falsa faça
3:    para cada padrão de entrada faça
4:       Atualize a camada F1
5:       enquanto condição de reset é verdadeira faça
6:          encontre um neurônio candidato a aprender a entrada corrente (i.e., o neurônio na
            camada F2, não inibido, com a maior ativação)
7:          combine as saídas das camadas F1 e F2
8:          testar a condição de reset
9:          se reset é verdadeiro então
10:            rejeita-se o neurônio candidato (neurônio candidato é inibido)
11:         senão
12:            aceite o candidato como vencedor para aprender o padrão
13:         fim se
14:      fim enquanto
15:      aprender o padrão, modificando os pesos do neurônio vencedor de acordo com equações
         diferenciais (ressonância)
16:   fim para
17: fim enquanto
```

Na teoria da ressonância adaptativa, as mudanças nos pesos e nas ativações dos neurônios são governadas por equações diferenciais (Fausett, 1994). A rede está continuamente mudando, mas o processo pode ser simplificado, desde que as ativações mudam muito mais rapidamente que os pesos. Após um neurônio da camada de saída ser escolhido para aprender um determinado padrão, os valores dos sinais *bottom-up* e *top-down* são mantidos por um período de tempo, no qual os valores dos pesos são modificados para adequar-se ao novo padrão. Nesse ponto, ocorre a *ressonância* que dá o nome a rede.

Duas formas de aprendizado podem ser usadas em redes ART. No aprendizado rápido (*fast learning*), assume-se que a atualização dos pesos na ressonância ocorre rapidamente, de forma que os pesos alcançam o equilíbrio em cada apresentação de um padrão. Ao contrário, o aprendizado lento (*slow learning*) assume que os pesos não alcançam o equilíbrio em apenas uma apresentação do padrão e mais apresentações do padrão à rede são necessárias. A diferença básica entre as duas formas de aprendizado é a taxa de aprendizado (*learning rate*) que é aplicada durante a atualização dos pesos. A seguir são descritos os detalhes do aprendizado de 3 versões das redes ART: ART-1, ART-2 e ART-2A.

3.2.3 Rede ART-1

A rede ART-1 permite o agrupamento de padrões codificados através de vetores com valores binários e que o usuário controle diretamente a similaridade entre os padrões. Tal controle resulta em uma maior ou menor quantidade de neurônios ativados na camada de saída. O processo de aprendizado nessa rede á rápido, de forma que os pesos alcançam o equilíbrio a cada apresentação dos padrões.

A arquitetura da rede ART-1 é composta pela camada F_1, pela camada F_2 (camada de saída), e uma unidade de *reset*, que permite o controle de similaridade entre os padrões aprendidos pelo mesmo neurônio de saída. A ausência da camada F_0 na rede ART-1 deve-se ao fato de que como a entrada é binária, não é necessário o pré-processamento inicial dos padrões.

Algoritmo

A Tabela 3.2 mostra a notação empregada nesse algoritmo da rede ART-1 (Algoritmo 3.2). Um vetor com elementos binários serve como entrada. A função de ativação da camada F_1 é a identidade, de forma que os sinais iniciais dos neurônios (representados pelo vetor x) da camada F_1 correspondem aos valores dos componentes do vetor de entrada. Esses sinais são enviados para a camada F_2, multiplicados pelos pesos *bottom-up*. Baseando-se nesses sinais e empregando uma função de ativação identidade, os neurônios da camada F_2 competem para encontrar um neurônio candidato J com a maior entrada. O neurônio com a maior entrada é marcado como ativo (1) e os demais neurônios são marcados como inativos (0). O sinal do neurônio J, multiplicado pelos pesos *top-down*, é enviado para a camada F_1.

A norma do vetor x corresponde ao número de componentes em que os pesos *top-down* do neurônio candidato J e o vetor de entrada s são 1. Se a razão entre $\|x\|$ e $\|s\|$ for maior ou igual ao parâmetro de vigilância ρ, o neurônio torna-se o vencedor para aprender o padrão e seus pesos são ajustados. No caso contrário, em que a razão é menor que ρ, o neurônio J é rejeitado e marcado como inibido (-1) e outro neurônio deve ser escolhido. Como o neurônio candidato é inibido, ele não compete novamente para o mesmo padrão. Nesse caso, a camada recebe novamente como entrada o vetor s e a busca por um novo candidato é reiniciada.

Tal processo continua até que seja encontrado um neurônio para aprender o padrão ou até que todos os neurônios da camada F_2 estejam inibidos. Algumas possíveis ações que podem ser executadas nesse caso são: reduzir o valor do parâmetro de vigilância, permitindo que padrões não muito similares sejam aprendidos por um mesmo neurônio; aumentar o número de neurônios na camada F_2, permitindo que o padrão seja aprendido por um novo neurônio; e simplesmente ignorar o padrão, tratando-o como um ruído.

Ao fim da apresentação de um padrão de entrada, os neurônios da camada de representação (F_2) são marcados como inativos, mas disponíveis para competir e aprender um outro padrão

de entrada. A condição de parada do algoritmo 3.2 pode consistir no término de um número específico de ciclos de treinamento (no qual os padrões de entrada são apresentados várias vezes a rede) ou na verificação da estabilidade (não alteração) dos pesos. A Tabela 3.3 apresenta os parâmetros da rede, seus intervalos permitidos e exemplos de valores (Fausett, 1994). A notação $b_{ij}(0)$ e $t_{ji}(0)$ indica os valores iniciais dos pesos *bottom-up* e *top-down*, respectivamente.

Tabela 3.2: Notação básica empregada no algoritmo de treinamento da rede ART-1

Símbolo	Descrição
m	número de componentes do vetor de entrada
n	número máximo de agrupamentos a serem formados
b_{ij}	pesos *bottom-up*
t_{ji}	pesos *top-down*
ρ	vigilância
s	vetor de entrada
x	vetor de ativação para a camada F_1
$\|x\|$	norma do vetor x, que corresponde a soma dos componentes do vetor

Tabela 3.3: Parâmetros da Rede ART-1

Parâmetro	Valores permitidos	Exemplo
L	$L > 1$	2
ρ (vigilância)	$0 < \rho \leq 1$	0,9
$b_{ij}(0)$ (pesos *bottom-up*)	$0 < b_{ij}(0) \leq \frac{L}{L-1+n}$	$\frac{1}{1+n}$
$t_{ji}(0)$(pesos *top-down*)	$t_{ji}(0) = 1$	1

3.2.4 Rede ART-2

A rede ART-2 foi projetada para o agrupamento de vetores de entrada compostos por valores contínuos (Carpenter & Grossberg, 1989b). As diferenças entre as versões ART-1 e ART-2 estão relacionadas às modificações necessárias, principalmente nas camadas F_0 e F_1, para acomodar os padrões representados por valores contínuos (Figura 3.4). Na rede ART-2, a camadas F_0 e F_1 têm funções de normalização e supressão de ruído.

A camada F_0 tem 4 unidades de processamento (U, V, W e X) e a camada F_1 têm seis tipos de unidades de processamento: W, X, U, V, P e Q. Uma unidade suplementar de processamento, localizada entre as unidades W e X recebe o sinal da unidade W, calcula a norma do vetor w e envia seu sinal para a unidade X. Outras unidades suplementares existem entre as unidades P e Q e entre as unidades V e U.

Algoritmo 3.2 Treinamento na rede ART-1

1: inicialize os parâmetros
   $L > 1$,
   $0 < \rho \leq 1$
2: inicialize os pesos
   $0 < b_{ij} < \frac{L}{L-1+n}$,
   $t_{ji} = 1$
3: **enquanto** condição de parada for falsa **faça**
4: **para** cada padrão de entrada **faça**
5: marque as ativações do neurônios da camada F_2 como zero
6: marque o vetor de ativação x da camada F_1 com o vetor de entrada s
7: calcule a norma do vetor de entrada s
   $\|s\| = \sum_i s_i$
8: **para** cada neurônio Y da camada F_2 que não está inibido **faça**
9: **se** neurônio y_j não estiver inibido **então**
10: $y_j = \sum_i b_{ij} xi$
11: **fim se**
12: **fim para**
13: **enquanto** *reset* é verdadeiro **faça**
14: encontre um neurônio candidato J de forma que $y_J \geq y_j$, para aprender a entrada corrente (i.e., o neurônio na camada F_2, não inibido, com a maior ativação)
15: recalcule as ativações da camada F_1
   $x_i = s_i t_{Ji}$
16: calcule a norma do vetor x:
   $\|x\| = \sum_i x_i$
17: testar a condição de *reset*:
18: **se** $\frac{\|x\|}{\|s\|} < \rho$ **então**
19: neurônio candidato J é inibido:
   $Y_J = -1$
20: *reset* é verdadeiro
21: **senão**
22: *reset* é falso
23: **fim se**
24: **fim enquanto**
25: aprender o padrão, modificando os pesos do neurônio vencedor:
   $b_{ij}^{novo} = \frac{Lx_i}{L-1+\|x\|}$
   $t_{Ji}^{novo} = x_i$
26: **fim para**
27: **fim enquanto**

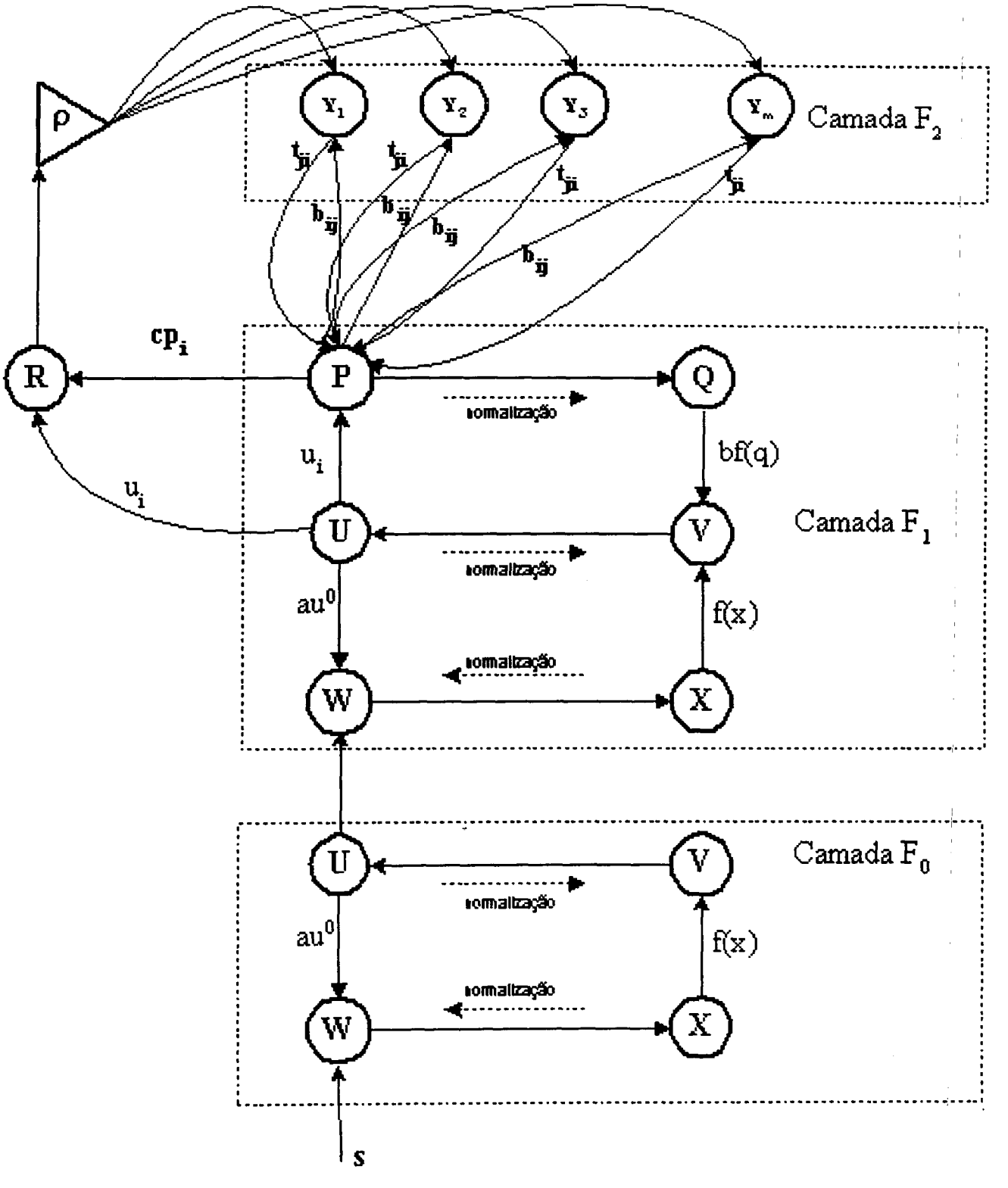


Figura 3.4: Estrutura da rede ART-2 (adaptada de Carpenter *et al.* (1991a), figura 1)

Algoritmo

O algoritmo da rede ART-2 envolve os mesmo passos do algoritmo básico da arquitetura ART. As diferenças ocorrem no tratamento do sinal de entrada contínuo, que é realizado pela camada F_1. Os pesos inicias *top-down* devem ter seus valores de forma que seja garantido que o sinal de *reset* não ocorra no primeiro padrão que será armazenado na unidade de saída. Um exemplo de valor é $t_{ji}(0) = 0$.

Os pesos iniciais *top-down* devem ser escolhidos de forma a satisfazer a equação:

$$b_{ij}(0) = \frac{1}{(1-d)\sqrt{n}} \tag{3.2}$$

Tal restrição previne a possibilidade de um novo vencedor ser escolhido durante a ressonância, que é o instante de tempo em que a rede está aprendendo o padrão. Os demais parâmetros do algoritmo estão relacionados na Tabela 3.4. A função de ativação $f(x)$ com um limiar θ é igual a:

$$f(x) = \begin{cases} x & \text{se } x > \theta \\ 0 & \text{caso contrário} \end{cases} \tag{3.3}$$

Tal equação é responsável pela supressão de ruído, de forma que valores menores que o limiar θ são tratados como zero.

O treinamento na rede ART-2 é ilustrado de acordo com o Algoritmo 3.3. Um ciclo de treinamento consiste na apresentação de um vetor de entrada, que continuamente é enviado à rede. No início do treinamento, todas as ativações da rede são marcadas como zero. A primeira etapa do treinamento consiste na atualização das unidades da camada F_0 e F_1 (passos 4 e 5 do algoritmo). A seguir, a unidade P envia seus sinais para a camada F_2, onde ocorre a competição para a busca de um neurônio candidato. Após a escolha de um neurônio candidato, as unidades U e P enviam seus sinais para a unidade de *reset* (passo 9 do algoritmo).

O mecanismo de *reset* é verificado de acordo com esses sinais. De acordo com esse teste, o neurônio candidato é escolhido ou não para aprender o padrão correspondente. Se o neurônio candidato é rejeitado, tal neurônio é marcado como inativo e inapto a participar novamente de outra competição para armazenar esse mesmo padrão. Esse processo continua até que um neurônio candidato seja aceito para armazenar o padrão.

No aprendizado lento, apenas uma iteração nas equações de atualização de pesos ocorre para cada ciclo de treinamento (passo 18). Nesse caso, é necessário um número elevado de apresentações dos mesmos padrões de entrada, pois os pesos convergem lentamente para a codificação dos padrões. Os padrões podem ser apresentados novamente em diferente ordem e não existe a necessidade de que todos os padrões sejam apresentados novamente. No aprendizado lento, o laço de repetição descrito na linha 17 do algoritmo é executado apenas uma vez.

No aprendizado rápido, a atualização dos pesos continuam até que os pesos alcancem o equilíbrio. Assim o laço de repetição (linha 17) deve ser executado até que seja verificada a es-

tabilização dos pesos *top-down* e *bottom-up*. Poucas épocas de treinamento são necessárias, mas um número maior de interações na atualização dos pesos deve ser considerado. O aprendizado rápido é mais adequado quando existem restrições quanto ao tempo de treinamento da rede.

Tabela 3.4: Parâmetros da Rede ART-2

Parâmetro	Descrição	Exemplo de valor
n	número de unidades de entrada	7
m	número de unidades de representação	15
a, b	pesos fixos, que devem ser diferentes de zero, da camada F_1	$a = 10; b = 10$
c	pesos fixo usado no teste de *reset*	$c = 0,1$
d	ativação do neurônio vencedor J	$d = 0,9$
e	parâmetro para prevenir a divisão por zero	$e = 1$
θ	parâmetro de supressão de ruído	$\theta = \frac{1}{\sqrt{n}}$
α	taxa de aprendizado	$\alpha = 0,1$
ρ	parâmetro de vigilância	$\rho = 0,9$

3.2.5 Rede ART-2A

A rede neural ART-2A (Carpenter *et al.*, 1991a) reproduz eficientemente o comportamento essencial da rede ART-2, permitindo a categorização de padrões representados por vetores com valores binários ou contínuos. No entanto, a versão ART-2A reproduz a dinâmica do aprendizado rápido e implementa um método eficiente para simular o aprendizado lento, de modo a permitir uma categorização mais rápida do padrões de entrada. Essa versão opera de 2 a 3 vezes mais rápido que a rede ART-2 (Frank *et al.*, 1998; Peper *et al.*, 1993).

Algoritmo

O Algoritmo 3.4 ilustra o processamento realizado pela rede no aprendizado. Os parâmetros empregados pela rede ART-2A estão ilustrados na Tabela 3.5. A condição de parada para o algoritmo pode ser o término de um número de ciclos de treinamento ou a verificação de estabilização dos pesos da rede. As várias unidades de processamento existentes na camada F_1 da rede ART-2 são substituídas por um pequeno conjunto de normalizações e filtragem dos dados na rede ART-2A.

A rede ART-2A pode trabalhar apenas com valores não negativos como entrada. Inicialmente, o padrão de entrada I é normalizado à unidade euclidiana através do operador $\aleph$. Tal normalização reduz os vetores de entrada para a mesma magnitude, preservando o ângulo formado por eles. Essa normalização é importante, pois dessa forma, a rede define a busca pela similaridade entre o padrão e os protótipos (pesos dos neurônios da camada de representação F_2) através do ângulo formado entre eles. Nesse pré-processamento, Carpenter *et al.* (1991a)

Algoritmo 3.3 Treinamento na rede ART-2

1: inicialize os parâmetros
$a, b, \theta, c, d, e, \alpha, \rho$
2: **enquanto** condição de parada for falsa **faça**
3: **para** cada padrão de entrada **faça**
4: marque como zero as ativações das unidades u, p, q e
atualize as ativações da camada F_0
$u_i = 0, \quad w_i = s_i,$
$p_i = 0, \quad x_i = \frac{s_i}{e+\|s\|},$
$q_i = 0, \quad vi = f(x_i).$
5: atualize as ativações da camada F_1
$u_i = \frac{v_i}{e+\|s\|}, \quad w_i = s_i + \alpha u_i,$
$p_i = u_i, \quad x_i = \frac{w_i}{e+\|w\|},$
$q_i = \frac{p_i}{e+\|p\|}, \quad v_i = f(x_i) + bf(q_i).$
6: calcule os sinais para as unidades da camada F_2
$y_i = \sum_i b_{ij} pi$
7: **enquanto** *reset* é verdadeiro **faça**
8: encontre um neurônio candidato J de forma que $y_J \geq y_j$, para aprender a entrada corrente (i.e., o neurônio na camada F_2, não inibido, com a maior ativação)
9: testar a condição de *reset*:
$u_i = \frac{v_i}{e+\|v\|}, \quad p_i = u_i + dt_{Ji},$
$r_i = \frac{u_i + cp_i}{e+\|u\|+c\|p\|}.$
10: **se** $\|r\| < (\rho - e)$ **então**
11: neurônio candidato J é inibido:
$Y_J = -1$
12: *reset* é verdadeiro
13: **senão**
14: *reset* é falso
$w_i = s_i + au_i,$
$x_i = \frac{w_i}{e+\|w\|},$
$q_i = \frac{p_i}{e+\|p\|},$
$v_i = f(x_i) + bf(q_i),$
15: **fim se**
16: **fim enquanto**
17: **repita**
18: aprender o padrão, modificando os pesos do neurônio vencedor:
$b_{ij}^{novo} = \alpha d u_i + \{1 + \alpha d(d-1)\} t_{iJ}$
$t_{Ji}^{novo} = \alpha d u_i + \{1 + \alpha d(d-1)\} b_{iJ}$
19: atualize as ativações da camada F_1:
$u_i = \frac{v_i}{e+\|v\|},$
$w_i = s_i + \alpha u_i,$
$p_i = u_i + dt_{Ji},$
$x_i = \frac{w_i}{e+\|w\|},$
$q_i = \frac{p_i}{e+\|p\|},$
$v_i = f(x_i) + bf(q_i).$
20: **até** a condição final para atualização dos pesos seja alcançada
21: **fim para**
22: **fim enquanto**

Tabela 3.5: Parâmetros empregados pela rede ART-2A

Parâmetro	Descrição	Exemplo de valor
n	dimensão dos padrões de entrada	7
m	número máximo de classes	15
θ	parâmetro de supressão de ruído	$\theta = \frac{1}{\sqrt{n}}$
β	taxa de aprendizado	$\beta = 0,7$
ρ	parâmetro de vigilância	$\rho = 0,9$

sugerem ainda um método adicional para a supressão de ruídos. Esse método, representado no Algoritmo 3.4 pela função $F^0(x)$, marca como zero os componentes do vetor de entrada que não excedam um certo valor de limiar θ. A supressão de ruídos apenas faz sentido se os componentes do vetor de entrada, que são importantes para o categorização realizada pela rede, estão codificados exclusivamente nos valores mais elevados (i.e., que ultrapassam o limiar θ) (Frank *et al.*, 1998). O valor do limiar deve estar contido no intervalo $0 \leq \theta \leq \frac{1}{\sqrt{n}}$. A forma de atualização dos pesos no aprendizado da rede ART-2A (linha 16 do Algoritmo 3.4) é similar a atualização dos pesos na rede ART-2 (linha 18 do Algoritmo 3.3.

A constante α ou parâmetro de escolha define a profundidade máxima de busca por um neurônio candidato. Com $\alpha = 0$, todos os neurônios da camada F_2 são verificados antes que um neurônio comprometido seja escolhido como vencedor. O valor dessa constante deve ser escolhida obedecendo-se o intervalo $0 \leq \alpha \leq \frac{1}{\sqrt{m}}$.

A ressonância e a adaptação ocorrem se o neurônio candidato J for um neurônio não comprometido (i.e., não escolhido anteriormente para aprender um outro padrão), ou se a ativação do neurônio candidato T_J for menor ou igual ao parâmetro de vigilância ρ.

O parâmetro β descreve a taxa de aprendizado da rede. Tal parâmetro deve respeitar o intervalo $0 \leq \beta \leq 1$. Com $\beta = 0$, a rede opera no modo de classificação. Nesse modo, a rede devolve um neurônio vencedor J, mas os pesos não são atualizados, de forma que o padrão apresentado não é aprendido pela rede. Esse modo de operação é interessante quando busca-se a classificação de um conjunto de valores de entrada sem que exista a atualização da rede aos padrões. Com $\beta = 1$, a rede opera no modo de aprendizado rápido, onde o vetor de pesos do neurônio J é substituído, sem adaptação, ao padrão de entrada. Valores dentro do intervalo $0 < \beta < 1$ fazem com que a rede opere no modo de aprendizado lento, de forma que os pesos podem ser gradativamente adaptados aos padrões de entrada. Valores de β mais próximos de 0 representam uma adaptação mais suave, enquanto que valores de β mais próximos de 1 reproduzem uma adaptação mais agressiva ao padrão de entrada. O valor do parâmetro β pode variar durante a apresentação de um conjunto de padrões, dependendo do número de ciclos de treinamento. Esse valor não deve ser usado da forma $\beta \cong 1$, pois os protótipos tendem a

"pular" entre os padrões de entrada ao invés de convergir para uma média dos padrões agrupados na mesma classe.

O parâmetro de vigilância ρ define o número de classes que serão criadas pela rede. O valor desse parâmetro forma uma fronteira de decisão circular de raio igual a $\sqrt{2(1-\rho)}$ em torno do vetor de protótipos de cada classe (He *et al.*, 2003a). Com $\rho = 0$, todos os padrões de entrada são agrupados na mesma classe. Com $\rho = 1$, é criada uma classe para cada padrão de entrada diferente apresentado à rede.

Algoritmo 3.4 Treinamento na rede ART-2A

1: marque *reset* como verdadeiro e inicialize os parâmetros:
$\theta, \alpha, \beta, \rho$
2: **enquanto** condição de parada for falsa **faça**
3: **para** cada padrão de entrada **faça**
4: execute o pré-processamento no vetor de entrada I^0 (camada F_0)
$I = \aleph(F_0(\aleph(I^0)))$
onde $\aleph$ e F^0 descrevem as seguintes operações:
$\aleph(x) \equiv \frac{x}{\|x\|}$
$$F^0(x) = \begin{cases} x & \text{se } x > \theta \\ 0 & \text{caso contrário} \end{cases}$$
5: calcule os sinais para as unidades da camada F_2
$$T_j = \begin{cases} Iw_{ij} & \text{se } j \text{ é um neurônio comprometido} \\ \alpha \sum_j I_j & \text{caso contrário} \end{cases}$$
6: **enquanto** *reset* é verdadeiro **faça**
7: encontre um neurônio candidato J de forma que $y_J \geq y_j$, para aprender a entrada corrente (i.e., o neurônio na camada F_2, não inibido, com a maior ativação)
8: testar a condição de *reset*:
9: **se** $y_j > \rho$ **então**
10: *reset* é falso
11: **senão**
12: neurônio candidato J é inibido:
$Y_J = -1$
13: *reset* é verdadeiro
14: **fim se**
15: **fim enquanto**
16: aprender o padrão, modificando os pesos do neurônio vencedor:
$$w_{Ji}^{novo} = \begin{cases} I & \text{se } j \text{ é um neurônio não comprometido} \\ \aleph(\beta \aleph \Psi + (1-\beta) z_J^{velho}) & \text{caso contrário} \end{cases}$$
$$\Psi_i \equiv \begin{cases} I_i & \text{se } w_{Ji}^{velho} > 0 \\ 0 & \text{caso contrário} \end{cases}$$
17: marque o neurônio como comprometido
18: **fim para**
19: **fim enquanto**

3.3 Aprendizado baseado em instâncias

Aprendizado baseado em instâncias (IBL - *Instance-based learning*) é um paradigma de aprendizado realizado através de algoritmos que armazenam e utilizam um conjunto de experiências de uma base (ou memória) (Mitchell, 1997). Cada experiência, também chamada de instância, consiste em um conjunto de atributos, que podem ser classificados como atributos de entrada ou de saída. Atributos de entrada descrevem as condições em que a experiência foi observada e os atributos de saída representam o que ocorreu durante essa experiência. Durante a generalização, esses algoritmos calculam a similaridade entre uma consulta e a base de experiências, através de alguma função de distância, para determinar as experiências previamente observadas mais próximas à consulta, e assim predizer o atributo de saída para a consulta em questão. A generalização nesses algoritmos é somente iniciada quando uma nova instância necessita ser classificada. Assim, o treinamento nesses algoritmos consiste apenas no armazenamento das instâncias na base de experiências. A generalização é disparada no momento da consulta.

Um método utilizado para o aprendizado baseado em memória é o k-vizinhos mais próximos (*k-nearest neighbor learning*). Esse método assume que todas as instâncias são pontos em um espaço de dimensão n. Os vizinhos mais próximos são calculados através de uma função de distância. Sendo uma instância arbitrária x descrita como um vetor:

$$(a_1(x), a_2(x), \ldots, a_n(x)) \tag{3.4}$$

onde $a_i(x)$ consiste no valor do $i$ésimo atributo de uma instância x.

A função de distância empregada é a distância euclidiana heterogênea (Wilson & Martinez, 1997a). Essa função é chamada de heterogênea por permitir valores contínuos, discretos e nominais nos atributos. Tal função, chamada de $E(x, y)$, é definida como:

$$E(x_q, x) = \sqrt{\sum_{i=1}^{n} d(a_i(x_q), a_i(x))^2} \tag{3.5}$$

onde x_q é o ponto de consulta e x é uma instância na base de experiências, ambos tendo um número de componentes igual a n. A função de distância $d(x, y)$ entre dois atributos de dois pontos é definida como:

$$d(x, y) = \begin{cases} d_1(x, y) & \text{se os atributos são nominais} \\ d_2(x, y) & \text{se os atributos são discretos ou contínuos} \end{cases} \tag{3.6}$$

$$d_1(x, y) = \begin{cases} 0 & \text{se } x = y \\ 1 & \text{caso contrário} \end{cases} \tag{3.7}$$

$$d_2(x, y) = ||x - y|| \tag{3.8}$$

Para garantir que a distância entre atributos contínuos de uma mesma instância estejam na mesma faixa (entre 0 e 1), alguma forma de normalização é necessária. A mais comum é

a normalização linear. Sendo max_{x_i} o valor máximo observado na base de experiência para o atributo x_i e min_{x_i} o valor mínimo observado para o mesmo atributo, a função de normalização linear $h(x_i)$ é definida como:

$$h(x_i) = \frac{x_i - min_{x_i}}{max_{x_i} - min_{x_i}} \tag{3.9}$$

A normalização permite que os atributos das instâncias estejam na mesma escala durante o cálculo da distância. A normalização é efetuada nas instâncias no momento em que são inseridas na base de experiências.

No aprendizado baseado nos k-vizinhos mais próximos, a função objetivo pode ser um valor discreto ou contínuo. O Algoritmos 3.5 e 3.6 ilustram a operação da técnica para funções objetivo com valores discretos, na forma $f : \Re^n \rightarrow V$, onde V é um conjunto finito $\{v_1, \ldots, v_s\}$. O valor $f'(x_q)$ retornado como uma estimativa da função objetivo $f(x_q)$ é o valor mais comum de f entre os k exemplos de treinamento próximos a x_q. Esse algoritmo pode ser facilmente adaptado para aproximar funções objetivo contínuas. Para esse tipo de funções, o algoritmo calcula o valor médio dos k vizinhos ao invés de calcular o valor mais comum. Assim, para aproximar uma função contínua $f : \Re^n \rightarrow \Re$, pode-se substituir a última linha do Algoritmo 3.6 pela equação 3.10:

$$f'(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k} \tag{3.10}$$

Algoritmo 3.5 Treinamento no método baseado em instâncias

adicione cada amostra de dados $(x, f(x))$ à base de experiências

Algoritmo 3.6 Classificação k-vizinhos mais próximos

1: Dado um ponto de consulta x_q a ser classificado
2: Seja $x_1 \ldots x_k$ as k instâncias da base de experiências mais próximas ao ponto x_q que compõem um conjunto de instâncias vizinhas V,
3: retorne $f' \leftarrow argmax_{v \in V} \sum_{i=1}^{k} \delta(v, f(x))$
onde:

$$\delta(a,b) = \begin{cases} 1 & \text{se } a = b \\ 0 & \text{caso contrário} \end{cases}$$

Uma melhoria no algoritmo dos k-vizinhos mais próximos é ponderar a contribuição de cada vizinho, de forma que os vizinhos mais próximos tenham uma maior contribuição no cálculo da função f'. Dessa forma, cada vizinho recebe um valor de peso w_i e a função estimativa f', considerando atributos de saída contínuos é modificada para:

$$f'(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \tag{3.11}$$

A função para o cálculo dos pesos w_i deve ter valor máximo quando a distância entre x e x_q for igual a zero e aproximar-se de zero quando a distância entre x_q e x tender ao infinito. A função para ponderação utilizada é a gaussiana, centrada em um ponto x_q:

$$w_i(x) = e^{\frac{-E(x_q,x)}{t}} \tag{3.12}$$

Essa função é também chamada de função de *kernel* e o denominador t chamado de parâmetro de suavização (Aha, 1998). A Figura 3.3 mostra o comportamento da função gaussiana, considerando diferentes valores para a t e distâncias de diferentes pontos em relação à origem. À medida que aumenta o valor de t, maior é a contribuição dos pontos vizinhos no cálculo da função f'.

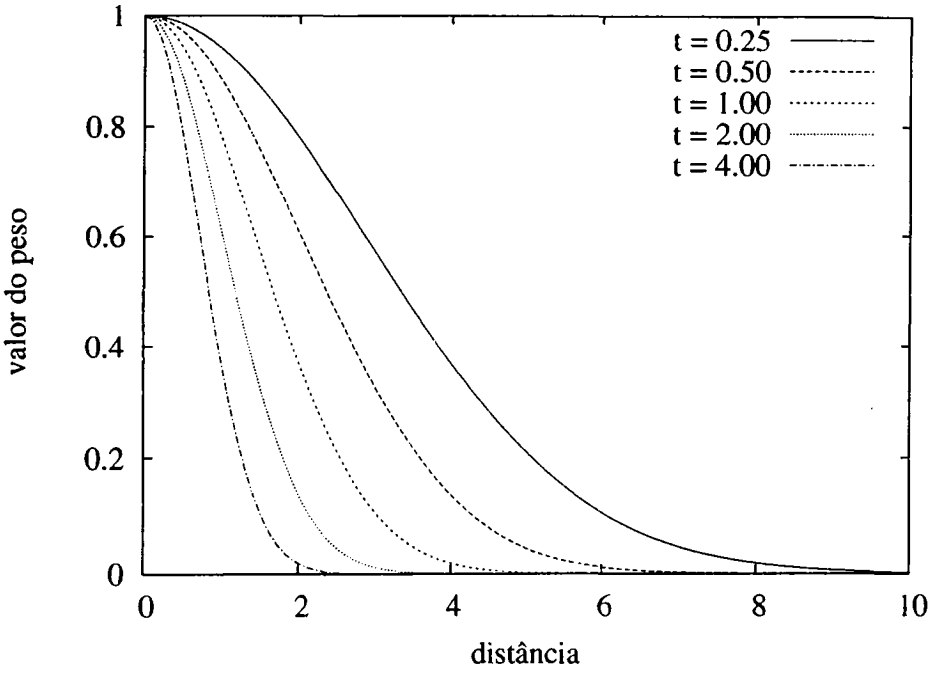


Figura 3.5: Função gaussiana com 4 valores diferentes de t

Com o método de ponderação por distância, todos os exemplos existentes no conjunto de treinamento podem ser empregados para a classificação, pois as instâncias distantes à x_q têm pouco efeito em $f'(x_q)$. A única desvantagem em considerar todos os exemplos de treinamento é que a classificação torna-se mais lenta. Se todos os exemplos são considerados para classificar uma nova instância, o método é chamado de global. Se apenas os k vizinhos mais próximos são considerados, o método é chamado de local (Mitchell, 1997).

O algoritmo k-vizinhos mais próximos com ponderação implementa um método eficiente de inferência para muitos problemas práticos. Esse algoritmo é robusto quando existe ruído no conjunto de treinamento, desde que exemplos muito distantes ao ponto de consulta tem pouca influência na função estimativa. Um ponto importante no algoritmo dos k-vizinhos mais próximos é que a distância é calculada baseando-se em todos os atributos que compõem as instâncias. Essa peculiaridade contrasta-se com outros métodos de aprendizado de máquina,

que selecionam um subconjunto dos atributos para formar a hipótese. Se as instâncias contém um grande número de atributos, a distância entre os pontos pode ser dominada por atributos irrelevantes. Esse problema é conhecido como *curse of dimensionality*. Uma maneira de resolver esse problema é ponderar cada atributo de maneira diferente para calcular a distância entre duas instâncias. Isso corresponde ao ajuste dos eixos no espaço euclidiano, encurtando os eixos que correspondem aos atributos irrelevantes e aumentando os eixos que correspondem aos atributos mais relevantes. Assim, cada atributo pode conter um valor de peso z_j, onde os valores $z_1, \ldots, z_n$ podem ser obtidos de modo empírico através de técnicas de validação cruzada. Nesse ponto, é muito importante o conhecimento sobre o domínio do problema, que pode auxiliar na escolha dos atributos mais relevantes. Outra alternativa, no caso de atributos muito irrelevantes, é ponderar o valor de z igual a 0. Dessa forma, o valor de um atributo irrelevante não é considerado no cálculo da distância. Esse mecanismo pode reduzir consideravelmente o impacto de atributos irrelevantes no algoritmo de classificação.

Uma melhoria na implementação do algoritmo é a aplicação de um mecanismo mais eficiente para a indexação da memória. Sem um mecanismo de indexação, a classificação pode tornar-se lenta, desde que todo o processamento é realizado em tempo de classificação. Outra melhoria possível é a redução do número de instâncias na base de experiências (Aha *et al.*, 1991; Wilson & Martinez, 1997b, 2000), optando por armazenar apenas as instâncias consideradas úteis para novas classificações. Armazenar um número muito grande de instâncias resulta em problemas de espaço de armazenamento e reduzem o tempo de execução da classificação.

O algoritmo desenvolvido, chamado de IBL, utiliza a normalização linear para armazenar as instâncias na base de experiências, durante o treinamento (passos 1 a 2 do Algoritmo 3.7). Para a classificação, emprega-se um vetor de pesos z que permite intensificar a contribuição de cada atributo no cálculo da distância e uma função de *kernel* gaussiana é usada para ponderar as instâncias vizinhas em relação a distância, cujo formato é definido pelo parâmetro t (passos 3 a 7 do Algoritmo 3.7).

3.4 Considerações finais

Este capítulo descreveu os algoritmos de aprendizado de máquina utilizados neste trabalho. Tais algoritmos são incrementais, pois permitem a atualização do conhecimento previamente obtido à medida que novos padrões de entrada tornam-se disponíveis. Essa característica é importante para os objetivos desta tese, pois os algoritmos de aquisição de conhecimento devem adaptar-se à medida que a carga do sistema apresente variações. Os algoritmos selecionados são a rede ART-2A e o IBL.

A família da redes neurais ART permitem o aprendizado incremental, com bom desempenho computacional e de classificação. O paradigma de aprendizado conexionista empregado

Algoritmo 3.7 IBL

1: normalize os atributos de cada amostra de dados
$h(x_i) = \frac{x_i - min_{x_i}}{max_{x_i} - min_{x_i}}$
2: armazene cada amostra de dados $(x, f(x))$ na base de experiências
3: inicialize os parâmetros k e t
4: defina o valor dos pesos dos atributos através do vetor z
5: dado um ponto de consulta x_q, encontre um conjunto de tamanho k de instâncias mais próximas ao esse ponto, através da função de distância euclidiana heterogênea:
$E(x_q, x) = \sqrt{\sum_{i=1}^{n} d(a_i(x_q), a_i(x))^2 \times z_i}$
onde:

$$d(x,y) = \begin{cases} 1 & \text{se } x \text{ ou } y \text{ são desconhecidos} \\ d_1(x,y) & \text{se os atributos são nominais} \\ d_2(x,y) & \text{se os atributos são discretos ou contínuos} \end{cases}$$

$$d_1(x,y) = \begin{cases} 0 & \text{se } x = y \\ 1 & \text{caso contrário} \end{cases}$$

$d_2(x,y) = ||x - y||$
6: Calcule f'
$f'(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$
onde:
$w_i(x) = e^{\frac{-E(x_q,x)}{2 \times t^2}}$

nessas arquiteturas permite representar o conhecimento obtido através das conexões sinápticas existentes entre os neurônios da rede. Tais conexões podem ser atualizadas, em resposta à mudanças no ambiente no qual a rede neural está sendo empregada. O mecanismo de atualização das conexões permite que o conhecimento armazenado previamente seja mantido, de maneira estável. As tarefas de aprendizado das redes ART permitem a categorização de uma amostra de dados, sem a necessidade de um treinamento prévio da rede.

Essas características permitem que as arquiteturas de redes ART sejam utilizadas em vários domínios de problemas, por exemplo: controle de reatores nucleares (Keyvan & Rabelo, 1992), interpretação dinâmica de sensores nucleares (Whiteley & Davis, 1996, 1993), reconhecimento de imagens de satélite (Baraldi & Parmiggiani, 1995; Carpenter *et al.*, 1997), processamento de imagens (Vlajic & Card, 2001), detecção de minas terrestres (Filippidis *et al.*, 1999), controle sensorial de robôs (Bachelder *et al.*, 1993), reconhecimento de caracteres chineses (Gan & Lua, 1992; He *et al.*, 2002, 2003b) e classificação de falhas em sistemas de controle (Benítez-Pérez & García-Nocetti, 2002). A arquitetura ART utilizada depende do tipo do problema a ser resolvido.

Várias arquiteturas da rede ART têm sido propostas na literatura (Frank *et al.*, 1998). Além das versões ART-1, ART-2 e ART-2A, descritas previamente neste capítulo, outras arquiteturas que têm inspiração na teoria da ressonância adaptativa foram desenvolvidas. Exemplos dessas arquiteturas são as redes ARTMAP, que permitem o aprendizado supervisionado, e as

redes FuzzyART (Carpenter *et al.*, 1991b) e FuzzyARTMAP (Carpenter *et al.*, 1992), que são respectivamente similares às versões ART-1 e ARTMAP e realizam o pré-processamento dos dados através da teoria de conjunto difusos (*fuzzy sets*). A arquitetura ART-2A é selecionada neste trabalho por permitir o aprendizado incremental, com bom desempenho computacional e de classificação, considerando amostras de dados compostas de atributos contínuos.

O paradigma baseado em instâncias permite a construção de classificadores onde o aprendizado é rápido e nenhuma informação é perdida. O algoritmo IBL, que utiliza o algoritmo dos k-vizinhos mais próximos, permite encontrar um conjunto de instâncias similares a um ponto de consulta, construindo uma estimativa de um atributo de saída através dessas instâncias. Além de ser incremental, esse algoritmo tem como treinamento apenas o armazenamento das instâncias em uma base de experiências e a classificação é realizada no momento da consulta. A classificação nesse algoritmo envolve encontrar as instâncias mais próximas ao ponto de consulta e o cálculo da estimativa, empregando uma função para ponderar a contribuição das instâncias vizinhas ao ponto de consulta.

O desempenho dos algoritmos ART-2A e IBL dependem de seus parâmetros que não são modificados durante o aprendizado. Os parâmetros do algoritmo IBL são o tamanho da vizinhança k e o tamanho do kernel t. O tamanho da vizinhança define a quantidade de instâncias que serão consideradas pelo algoritmo. Um tamanho de vizinhança pequeno pode depreciar o desempenho de classificação, enquanto um tamanho de vizinhança demasiadamente grande pode depreciar o desempenho computacional. Assim, deve haver um equilíbrio entre os desempenhos de classificação e computacional para que seja encontrado um valor de vizinhança adequado a essas duas características. O parâmetro de tamanho da curva da função gaussiana t permite definir os pesos que as instâncias vizinhas ao ponto de consulta obterão para o cálculo da estimativa. Esse parâmetro afeta apenas o desempenho de classificação, e não tem impacto no desempenho computacional.

Os parâmetros do algoritmo ART-2A são a vigilância ρ e a taxa de aprendizado β. A vigilância, por controlar a similaridade entre os padrões de entrada e os padrões já aprendidos pela rede, define a quantidade de categorias que serão formadas pela rede. Seu valor depende do tipo de problema a ser resolvido, que reflete-se nas características intrínsecas dos dados apresentados à rede. A taxa de aprendizado define a velocidade em que os pesos da rede aproximam-se dos padrões de entrada e tem relação direta com a quantidade de ciclos de apresentação dos padrões. Quanto menor o valor de β, maior a quantidade de ciclos necessária para que a rede estabilize o valor de seus pesos. Um valor de $\beta = 0$ indica que os pesos da rede não serão alterados durante a apresentação dos padrões. A taxa de aprendizado influencia os desempenhos de classificação e computacional.

CAPÍTULO

4

Carga de Trabalho

4.1 Considerações iniciais

Atualmente, a computação paralela tem sido empregada em diversas áreas do conhecimento, visando resolver em um tempo mais reduzido problemas que demandam maior potência computacional. Dentre as formas de processamento paralelo, o desenvolvimento de aplicações paralelas e a sua execução em arquiteturas paralelas de memória distribuída apresenta-se como uma alternativa cada vez mais empregada para a solução de problemas computacionais diversos.

Aplicações paralelas e seqüenciais são caracterizadas por diferentes variáveis que controlam sua execução e determinam o desempenho obtido. Variáveis como o tipo do algoritmo utilizado, tamanho do problema, parâmetros de entrada, linguagens de programação e paradigmas, bibliotecas de comunicação e a arquitetura paralela utilizada tem efeitos significantes no comportamento das aplicações (Ferschweiler *et al.*, 2002). Além disso, a forma pela qual as aplicações paralelas são submetidas a um determinado sistema computacional define a carga de trabalho típica do sistema, que é importante para o entendimento do desempenho do sistema.

A carga de trabalho pode ser analisada de maneira isolada, observando as características intrínsecas dos processos em aplicações paralelas, ou de maneira global, através da observação conjunta de todas as aplicações paralelas que são executadas em um sistema computacional. A maneira pela qual a carga de trabalho é analisada define a quantidade e o tipo das informações que podem ser obtidas sobre as aplicações.

A utilização de uma carga de trabalho representativa constitui a primeira etapa para a exploração das características de execução e de submissão das aplicações paralelas. Nesse contexto, este capítulo descreve as aplicações, *benchmarks* e traços de execução de ambientes de produção, representados pelos *logs* do software de escalonamento desses ambientes, utilizados

como carga de trabalho paralela e seqüencial. Para a caracterização intrínseca da carga de trabalho, é utilizado um conjunto formado por aplicações paralelas reais, *benchmarks* e uma aplicação sintética. Esse conjunto é selecionado visando representar as operações comuns encontradas em aplicações paralelas, em relação à utilização de recursos computacionais. Esse conjunto e o ambiente de execução considerado nos experimentos são descritos na Seção 4.2. Os *logs* de traços de execução de aplicações paralelas e seqüenciais em ambientes de produção, utilizados para a caracterização global da carga de trabalho, são descritos na Seção 4.3.

4.2 Aplicações paralelas

A Tabela 4.1 apresenta as 10 aplicações utilizadas nesta tese. A aplicação CV foi desenvolvida no grupo de sistemas distribuídos e programação concorrente (LASDPC-ICMC) e utilizada previamente nos trabalhos de Cortes (1999), Souza (2000) e Santos (2001). A aplicação sintética A foi desenvolvida neste trabalho, empregando uma parcela do código desenvolvido por Cortes (1999). A aplicação PSTSWM é distribuída com o pacote ParkBench (Hockney & Berry, 1994; Hey & Lancaster, 2000) e as demais aplicações são distribuídas no pacote NAS (Bailey *et al.*, 1994). Tais pacotes agregam uma quantidade de *benchmarks*, orientados para a avaliação de sistemas paralelos.

Benchmarks são ferramentas computacionais utilizadas para a avaliação de desempenho em arquiteturas paralelas, através da execução de programas em ambientes reais (Cortes, 2004). De acordo com a complexidade, os *benchmarks* podem ser classificados em diferentes níveis (Emilio, 1996; Jain, 1991). Em um nível mais baixo, são medidas as propriedades físicas da máquina, por exemplo velocidade de acesso à memória principal, latência da rede de comunicação e velocidade de execução de operações aritméticas. Em um nível intermediário, chamado de nível de núcleo (*parallel kernel*), são executados os códigos que são comuns em várias aplicações científicas, por exemplo resolução de integrais, equações diferenciais, multiplicação de matrizes e resolução de sistemas lineares. Em um nível mais alto, chamado de aplicações compactas (*compact applications*), encontra-se o código e estruturas de dados que visam solucionar problemas reais. Exemplos de aplicações compactas são os programas computacionais que resolvem problemas da área de dinâmica dos fluidos e meteorologia.

Seguindo essa taxonomia, os programas paralelos utilizadas nesta tese são classificados em dois grupos (coluna "classe" da Tabela 4.1): aplicações compactas (A) e *benchmarks* de núcleo (B). As aplicações, os *benchmarks* de núcleo e o ambiente de considerado para a execução são descritos respectivamente nas seções 4.2.1, 4.2.2 e 4.2.3.

Tabela 4.1: Aplicações utilizadas

Aplicação paralela	Descrição	Classe	Linguagem de programação	Ambiente de mensagens	Número de linhas
PSTSWM	*Shallow water model*	A	Fortran/C	PVM	35.134
LU	*Lower-upper gauss-seidel*	A	Fortran	MPI	5.194
BT	*Block tridiagonal*	A	Fortran	MPI	5.632
SP	*Scalar pentadiagonal*	A	Fortran	MPI	4.892
CV	Caixeiro viajante	A	C	PVM	781
A	Aplicação sintética	A	C	PVM	408
EP	*Embarrassing parallel*	B	Fortran	MPI	346
MG	*Multi grid*	B	Fortran	MPI	2.540
CG	*Conjugate gradient*	B	Fortran	MPI	1.841
IS	*Integer sort*	B	Fortran	MPI	1.097

4.2.1 Aplicações compactas

Aplicações compactas representam programas paralelos utilizados para a solução de um determinado problema real. Ao contrário dos *benchmarks* de núcleo, que constituem bibliotecas de código, as aplicações compactas possuem um número maior de linhas de código, bibliotecas paralelas embutidas e todas as estruturas de dados necessárias para a solução de um problema.

Por exemplo, um programa paralelo que apenas resolve um sistema de equações lineares através do método LU não é considerado uma aplicação compacta. De outra forma, se uma aplicação utiliza uma biblioteca com o método LU para a solução de um problema específico, tal aplicação é considerada compacta. Além disso, aplicações compactas possuem o código testado e documentado. O termo "compacta" está relacionado com as simplificações que são realizadas na resolução do problema para que a aplicação seja mais facilmente distribuída e compilada por diferentes usuários. As simplificações permitem que o usuário defina, sem a necessidade de alteração no código, características de execução, como o tamanho do problema e quantidade de EPs que serão empregados.

PSTSWM

A aplicação compacta PSTSWM (*Parallel Spectral Transform Shallow Water Model*) consiste em um conjunto de algoritmos que resolvem equações de águas rasas (*shallow water equations*) na rotação de uma esfera usando transformações espectrais (Worley & Toonen, 1995; Worley *et al.*, 1998; Worley, 1998, 2000; Worley *et al.*, 2002; Worley, 2002). Equações de águas rasas descrevem o fluxo de uma camada fina de fluido em duas dimensões e tem sido usadas em métodos numéricos que visam solucionar problemas atmosféricos e oceânicos (Slobodcicov, 2003). Essa aplicação foi desenvolvida para avaliar diferentes algoritmos para a resolução do

método de transformação espectral, na forma que tais métodos são empregados em modelos de circulação atmosférica global (Foster *et al.*, 1996; Brehm *et al.*, 1998; Drake *et al.*, 1999).

As equações de águas rasas descrevem a evolução no tempo de três variáveis de estado: vorticidade [1], divergência horizontal e perturbação de uma média geopotencial [2]. As velocidades horizontais são calculadas a partir dessas variáveis, durante cada intervalo de tempo de execução da aplicação. A cada intervalo de tempo, as variáveis que representam o estado atual da resolução do problema são transformadas do domínio físico, onde a maioria das forças físicas são calculadas, para o domínio espectral, onde os termos das equações diferenciais são calculados. A transformação a partir de coordenadas físicas para coordenadas espectrais envolve a solução de uma transformada real rápida de Fourier (FFT - *Fast Fourier Transform*) para cada linha de uma latitude constante, seguida de uma integração numérica da latitude, que visa aproximar a transformada de Legendre (LT) para obter os coeficientes espectrais. A transformação inversa corresponde ao cálculo da soma dos harmônicos espectrais e das transformadas FFT.

Os algoritmos paralelos implementados para a resolução das equações são baseados nas decomposições dos domínios computacionais físicos e espectrais sobre uma malha lógica de processadores de tamanho $P = P_x \times P_y$. A eficiência do algoritmo é determinada primariamente pela eficiência dos algoritmos paralelos que visam a solução das transformadas FFT e LT e por qualquer desbalanceamento de carga causado pela escolha do algoritmo de decomposição. A execução da aplicação PSTSWM é descrita através das seguintes fases (Worley & Toonen, 1995):

1. cálculo do produto não linear e dos termos forçantes;
2. cálculo da transformada de Fourier para os termos não-lineares;
3. cálculo da transformada de Legendre;
4. atualização dos coeficientes espectrais para as variáveis de estado;
5. cálculo da soma dos harmônicos espectrais, de maneira simultânea com o cálculo das velocidades das variáveis de estado atualizadas;
6. cálculo da transformada inversa de Fourier par as variáveis de estado e velocidades.

A aplicação é implementada através da linguagem de programação Fortran 77 e com um número pequeno de diretivas implementadas através da linguagem C. Diferentes ambientes de passagem de mensagens, por exemplo MPI e PVM, são utilizados e devem ser selecionados em tempo de compilação. O algoritmo a ser empregado, o tamanho de problema e o número de elementos de processamento podem ser selecionados em tempo de execução. Seis tamanhos

[1] circulação do vetor velocidade de um fluido em movimento
[2] potencial do campo gravitacional da Terra

diferentes de problemas são providos, cada um com soluções para referência e opções de análise de erro.

A configuração da aplicação segue a configuração padrão provida pelo pacote ParkBench, com uma malha de lógica de processadores 2×2, algoritmo paralelo distribuído para FFT e algoritmo distribuído para LT.

LU

O *benchmark* LU (*lower-upper symmetric gauss-seidel*) resolve sistemas de equações lineares através do método LU, empregando um tamanho de bloco igual a 5×5. Esse problema representa os cálculos associados a uma classe de aplicações de dinâmica dos fluidos, chamada de INS3D-LU (Fatoohi, 1989, 1992).

BT

O problema resolvido pelo *benchmark* BT (*Block Tridiagonal*) é o cálculo da solução numérica de um sistema de cinco equações diferenciais parciais não lineares, que representam algumas das características principais de equações de Navier-Stokes. Esse método é empregado em muitos programas computacionais da área de dinâmica dos fluidos implementados para a solução numérica de equações de Euler/Navier-Stokes utilizando a discretização de malhas estruturadas (Naik, 1995).

SP

O *benchmark* SP (*Scalar Pentadiagonal*) resolve sistemas independentes de equações pentadiagonais utilizando um tamanho de malha $N \times N \times N$. Os *benchmarks* SP e BT constituem cálculos representativos da aplicação ARC3D (Berry *et al.*, 1989; Pirsig, 1993) e são similares, mas diferem em sua relação comunicação/computação (Naik, 1993; Barszcz *et al.*, 1993).

As aplicações compactas LU, SP e BT são baseadas em aplicações da área de dinâmica dos fluidos, que visam encontrar a solução numérica para a discretização em três dimensões de uma equação de Navier-Stokes. Tais aplicações são simplificações de programas paralelos reais dessa área. Apesar das simplificações, elas reproduzem os cálculos e movimentações de dados essenciais da área, e são mais complexas que os *benchmarks* de núcleo, por possuírem detalhes das estruturas de dados das aplicações de dinâmica dos fluidos. Três tamanhos de problema são considerados: W, A e B, que correspondem respectivamente aos seguintes tamanhos da malha: $36 \times 36 \times 36$, $64 \times 64 \times 64$ e $102 \times 102 \times 102$. O tempo de execução e a quantidade de memória utilizada são diretamente proporcionais ao tamanho da malha utilizada.

Aplicação sintética (A)

A aplicação sintética A visa emular as características mais comuns encontradas em programas paralelos. Essa aplicação é composta pelas 4 fases ordenadas:

1. Comunicação: troca de informação inicial entre as tarefas;
2. Processamento: conjunto de operações em ponto flutuante;
3. Escrita em disco: escrita em dispositivos de armazenamento não voláteis;
4. Comunicação: troca de informação entre as tarefas.

A comunicação e a escrita em disco podem ser parametrizadas diretamente, através da definição prévia do volume de informação que será transferido entre as tarefas e ao dispositivo de armazenamento. A fase orientada ao processamento utiliza o método do trapézio composto para solucionar integrais definidas (Cortes, 1999). O algoritmo emprega uma propriedade fundamental das integrais: a integral de uma função $f(x)$, que pode ser integrada nos pontos de a até b, é igual ao somatório de n outras integrais definidas pelos pontos contidos nesse intervalo. Variando-se a quantidade de pontos dentro do intervalo a e b, obtém-se uma precisão maior no valor da integral da função $f(x)$ e, conseqüentemente, um tempo de processamento mais elevado. Essa aplicação é escrita através da linguagem C, empregando o PVM como ambiente de passagem de mensagens.

Caixeiro viajante (CV)

A aplicação caixeiro viajante (CV) resolve o problema de encontrar a melhor rota que um viajante deve percorrer de maneira que visite todas as cidades e ainda consiga retornar à cidade de origem, minimizando uma função de custo. Exemplos de função de custo são tempo e distância. Tal problema é NP completo.

A formulação do problema, de acordo com Terada (1991), é: Seja $G = (VG, aG)$ um grafo orientado com custos c_{ij} positivos associados às arestas (i, j). Caso uma aresta c_{ij} não exista, convenciona-se que c_{ij} seja infinito. Uma viagem em G é um circuito (orientado) que contém cada vértice em VG apenas uma vez. A soma total dos custos das arestas que foram utilizadas na viagem corresponde ao custo total da viagem. Assim, a solução para o problema consiste em encontrar uma viagem de custo mínimo, entre todas as viagens possíveis em G.

A solução computacional utilizada para esse problema emprega a técnica de programação dinâmica (Terada, 1991). Essa solução reduz significantemente o número de permutações dos n vértices e conseqüentemente o tempo de execução final da aplicação. A solução supõe que uma viagem começa e termina no vértice 1, após visitar cada um dos vértices $2, 3, \ldots, n$ apenas uma vez. Qualquer viagem é constituída por uma aresta $(1.k).2 \leq +k \leq n$, e um caminho M de

k até 1, sendo que o caminho M visita cada um dos vértices em $VG - \{1, k\}$ apenas uma vez. Se a viagem considerada é de custo mínimo, o caminho M deve ser necessariamente de custo mínimo.

A implementação da aplicação CV é realiza através de linguagem C, utilizando o PVM como ambiente de passagem de mensagens. Os parâmetros principais da aplicação são o número de cidades e a quantidade de EPs empregados. Maiores detalhes sobre a implementação dessa aplicação podem ser encontrados no trabalho de Santos (2001).

4.2.2 Benchmarks de núcleo

Benchmarks de núcleo são menos complexos que aplicações compactas, constituindo bibliotecas de código comumente empregadas em aplicações paralelas. Os *benchmarks* empregados são: EP, MG, CG e IS.

EP

O *benchmark* EP (*Embarrassing Parallel*) fornece uma estimativa da capacidade máxima de processamento em ponto flutuante, sem utilizar uma quantidade significante de comunicação entre processos. O problema resolvido é a geração de números aleatórios, que é um problema típico de simulação computacional. Em sua execução, é executado um laço de repetição onde um par de número aleatórios é gerado e testado de acordo com um método específico (White *et al.*, 1994; Morimoto *et al.*, 1999). A comunicação é realizada ao fim da execução, onde é calculada a soma dos valores obtidos pelas tarefas. Essa aplicação é chamada de *embarrassing parallel* pelo método de partição do problema, que permite que as dependências entre os dados sejam reduzidas e que a comunicação seja minimizada.

MG

O *benchmark* MG (*Multigrid*) é a implementação de um algoritmo multi-malha para obter a solução u de um problema discreto de Poisson $\nabla^2 u = v$ em uma malha tridimensional. O particionamento da malha é realizado de forma que a malha seja sucessivamente dividida, começando pela dimensão z, passando pela dimensão y e finalmente pela dimensão x. Tal divisão é repetida até que o número especificado de tarefas, expresso obrigatoriamente em potência de dois, seja atingido. O algoritmo representa um método de convergência rápida para a solução de sistemas esparsos de equações lineares e equações diferenciais parciais por iteração. Esse algoritmo é o núcleo de muitas aplicações científicas numéricas e permite encontrar uma solução mais rápida de problemas discretos pela aproximação do problema original até que seja encontrada uma resposta suficientemente adequada para o problema original (Chamberlain *et al.*, 2000).

CG

O *benchmark* CG (*Conjugate Gradient*) emprega o método gradiente conjugado para encontrar uma aproximação para o maior autovalor de uma matriz simetricamente positiva, grande e esparsa. Esse *benchmark* possui operações típicas de cálculos em malha, relacionados à solução de equações lineares.

IS

O benchmark IS (*Integer Sort*) resolve o problema de ordenação de uma lista de inteiros. Tal ordenação é importante em algoritmos que empregam o método das partículas (*particle methods*). Sua execução permite avaliar o desempenho no cálculo de operações em inteiros e o desempenho da rede de comunicação. Essa aplicação não realiza avaliação das operações em ponto flutuante.

4.2.3 Ambiente de execução

O ambiente de execução é composto por um conjunto de computadores pessoais do Laboratório de Sistemas Distribuídos e Programação Concorrente (LASDPC). Esses computadores estão interligados em uma rede Ethernet de 100 Mbits e o software operacional utilizado é o Linux.

As versões de núcleo do sistema operacional Linux empregadas são 2.4.5 e 2.4.22. A Tabela 4.2 apresenta os computadores empregados. Cada computador é identificado por uma sigla, de acordo com o padrão EP_i, onde i é o índice que representa o computador.

Tabela 4.2: Computadores empregados nos experimentos

Computador	**Configuração**	
	Processador	**Memória (MB)**
EP_1	Pentium 166 MHz	32
EP_2	Pentium 166 MHz	32
EP_3	Pentium 166 MHz	48
EP_4	Pentium 200 MHz	32
EP_5	Pentium 233 MHz	128
EP_6	K6 II 400 MHz	164
EP_7	Athlon XP 2.1 GHz	256

4.3 *Logs* de ambientes de produção

A Tabela 4.3 ilustra os *logs* de ambientes de produção utilizados nesta tese. Esses *logs* são referenciados neste trabalho através de suas abreviações: SDSC95, SDSC96, SDSC2000, CTC e LANL.

Tabela 4.3: Cargas de trabalho utilizadas

Carga de Trabalho	Plataforma	EPs	Período de Tempo	Número de Aplicações
SDSC95	Intel Paragon	416	1995	76.872
SDSC96	Intel Paragon	416	1996	38.719
SDSC2000	IBM SP2	128	05/1998 a 04/2000	67.667
CTC	IBM SP2	512	07/1996 a 05/1997	79.302
LANL	Conection Machine CM-5	1024	11/1994 a 09/1996	201.387

Os *logs* SDSC95 e SDSC96 têm sua origem no *San-Diego Supercomputer Center (SDSC)*. A plataforma *Paragon* consiste em uma malha com processadores da arquitetura Intel i860. Essa máquina paralela contém 416 EPs, organizados do seguinte modo: 48 EPs dedicados para aplicações interativas, 352 EPs dedicados para aplicações paralelas, 6 EPs dedicados para serviço de *login* e 10 EPs dedicados para realizar entrada e saída em arquivos. O escalonador empregado é o NQS, configurado para atender aplicações paralelas de acordo com um conjunto pré-estabelecido de filas (Wan *et al.*, 1996). Tais filas são organizadas de acordo com a quantidade de EPs que são solicitadas pela aplicação, tempo máximo de execução e de memória (Feitelson & Rudolph, 1998; Downey & Feitelson, 1999; Feitelson & Naaman, 1999; Krevat *et al.*, 2002). A carga de trabalho SDSC2000 também tem sua origem nesse centro de computação. Nessa carga de trabalho, a plataforma utilizada é um computador IBM SP2, com o algoritmo de escalonamento EASY *backfilling* (Lifka, 1995; Mu'alem & Feitelson, 2001; Krevat *et al.*, 2002; Srinivasan *et al.*, 2002; Lawson & Smirni, 2002).

O *log* CTC possui sua origem no centro de computação *Cornell Theory Center*, localizado na Universidade *Cornell*. A plataforma utilizada é uma maquina paralela IBM SP2 com 512 EPs. Um número de 430 elementos de processamento são utilizados para aplicações paralelas. Tal máquina é considerada como heterogênea, pois possui diferentes quantidades de memória para grupos diferentes de elementos de processamento. Duas partições são consideradas: *Thin* e *Wide*. A partição *Thin* possui 352 EPs com 128 MB de memória e 30 EPs com 256 MB de memória. A partição *Wide* possui 22 EPs com 256 MB de memória, 21 EPs com 512 MB de memória, 4 EPs com 1024 MB de memória e 1 EP com 2048 MB de memória. O *EASY* é o software de escalonamento empregado (Hotovy, 1996; Downey, 1997; Smith *et al.*, 1998b; Schwiegelshohn & Yahyapour, 1998; Downey & Feitelson, 1999; Squillante *et al.*, 1999; Mu'alem & Feitelson, 2001; Feitelson, 2001; Cirne & Berman, 2001; Lawson & Smirni, 2002).

O *log* LANL tem sua origem no *Los Alamos National Laboratory*. A plataforma é uma máquina paralela *Connection Machine CM-5* que tem 1024 EPs e o software de escalonamento DJM é empregado (Feitelson, 1997; Mu'alem & Feitelson, 2001; Feitelson, 2001).

Esses *logs* de execução estão de acordo com um formato padronizado para o armazenamento de carga de trabalho de aplicações paralelas. Esse formato é discutido a seguir.

4.3.1 Formato padrão para cargas de trabalho

O formato padrão para cargas de trabalho foi definido com o objetivo de facilitar a utilização das cargas para a análise e definição de modelos (Chapin *et al.*, 1999). Através desse formato, cargas de trabalho são disponibilizadas na Internet [3] e programas computacionais podem executar análises e simular sistemas de escalonamento utilizando como entrada um arquivo em formato texto padrão. Tal formato é ilustrado na Tabela 4.4 e segue os seguintes princípios:

- os arquivos são portáveis e facilmente tratados computacionalmente;
- cada aplicação paralela é representada por um linha do arquivo. Cada linha contém um número pré-definido de campos, em sua maioria do tipo `inteiro`, separados por espaços em branco. Campos que são irrelevantes ou inexistentes para uma carga de trabalho específica são marcados com o valor -1;
- o mesmo formato é usado para modelos sintéticos e *logs*. Assim, dependendo do contexto, certos campos podem ser redundantes.
- comentários são permitidos e devem ser identificados através do caracter ";". É esperado que as cargas de trabalho possuam um cabeçalho que apresente as características do ambiente de execução ou do modelo;
- o formato é definido sem possibilidade de ser estendido.

4.4 Análise das cargas de trabalho

A Tabela 4.5 ilustra, para cada carga de trabalho, o percentual de aplicações não finalizadas naturalmente e o percentual de aplicações seqüenciais. De maneira geral, as aplicações que são submetidas ao sistema não são finalizadas naturalmente por dois motivos principais. O primeiro motivo é a ocorrência de uma falha na execução da aplicação, devido algum erro de programação ou do sistema. O segundo motivo é a atividade do software de escalonamento, que em determinadas situações finaliza a execução de aplicações que mostram-se incompatíveis com

[3] www.cs.huji.ac.il/labs/parallel/workload/

Tabela 4.4: Formato padrão para carga de trabalho de aplicações paralelas

Atributo	**Descrição**
job number	Contador, iniciado em 1. Tal campo serve como chave primária para as tabelas.
submit time	Instante de tempo de submissão da aplicação, em segundos. As linhas do arquivo de *log* são ordenadas ascendentemente por tal campo, e as aplicações são numeradas nessa ordem.
wait time	Diferença expressa em segundos entre o `submit time` e o tempo em que a aplicação iniciou sua execução. Tal campo é relevante apenas para *logs* reais, e não para modelos sintéticos.
run time	Tempo de execução da aplicação, descontando o tempo em que a aplicação experimentou em espera pelos recursos.
number of allocated processors	Número de elementos de processamento utilizados pela aplicação.
average cpu time used	Tempo médio de unidade central de processamento em modo usuário e sistema, expresso em segundos. A média é calculada considerando os elementos de processamento utilizados pela aplicação.
used memory	Utilização média de memória da aplicação em kilobytes.
requested number of processors	Número de elementos de processamento requisitados pela aplicação.
requested time	Estimativa do tempo de execução da aplicação fornecida pelo usuário.
requested memory	Estimativa fornecida pelo usuário da utilização média de memória
completed	Esse campo tem o seu valor igual a 1 caso a aplicação paralela foi finalizada naturalmente e igual a 0 caso a aplicação tenha sido finalizada por algum erro, ou por alguma restrição do software de escalonamento.
user id	Um número natural entre 1 e o número total de usuários existente na carga de trabalho
group id	Um número natural entre 1 e o número total de grupos de usuários existente na carga de trabalho. Em alguns sistemas, apenas esse campo é identificado ao invés de identificadores individuais para os usuários.
executable number	Um número natural entre 1 e o número total de aplicações existentes na carga de trabalho
queue number	Identificador da fila específica do software de escalonamento, quando existe uma organização dos recursos através de fila de espera diferentes.
partition number	Caso o ambiente seja organizado em partições, esse campo indica a partição na qual a aplicação foi executada.
preceding job number	Tal campo armazena o identificador da aplicação previamente executada pelo sistema.
think time from preceding job	Diferença em segundos entre o o término da aplicação anterior e o início da aplicação corrente.

a descrição informada pelo usuário. Um caso comum é a finalização de aplicações que requisitam um tempo de execução menor que o seu tempo de execução real. Dentre as cargas de trabalho, a CTC apresenta a maior quantidade de aplicações seqüenciais (36, 23%), enquanto a LANL não possui aplicações seqüenciais.

Tabela 4.5: Aplicações paralelas não finalizadas

Carga de trabalho	**Aplicações não finalizadas (%)**	**Aplicações seqüenciais (%)**
SDSC95	17, 33	20, 91
SDSC96	02, 13	21, 02
SDSC2000	16, 51	34, 26
CTC	21, 02	36, 23
LANL	10, 19	–

As Figuras 4.1, 4.2, 4.3, 4.4, 4.5 ilustram os histogramas do tempo de execução das cargas de trabalho consideradas. Através desses histogramas, pode-se observar que grande parte das aplicações são executadas por pequenos intervalos de tempo. A maioria das aplicações da carga de trabalho SDSC95 é executada por no máximo 30 segundos. Esse valor é igual a 5 minutos para as cargas SDSC96 e LANL, e igual a 30 minutos para as cargas SDSC2000 e CTC.

As Figuras 4.1, 4.7, 4.8, 4.9, 4.10 ilustram os histogramas da quantidade de EPs requisitada das cargas de trabalho consideradas. A quantidade de EPs empregada depende dos requisitos inerentes da aplicação, que tem origem na forma em que o problema foi particionado, e da organização imposta pelo software de escalonamento. Freqüentemente o software de escalonamento emprega um número determinado de filas para organizar o acesso aos EPs disponíveis. Assim, a quantidade de EPs que podem ser utilizados pelas aplicações submetidas a uma determinada fila é limitada ao tamanho do subconjunto de recursos associados a esta fila.

A maioria das aplicações das cargas de trabalho SDSC95 e SDSC96 requisitam uma quantidade de EPs não superior a 8; da mesma forma, esse valor é igual a 4 para a carga SDSC2000 e igual a 2 para a carga de trabalho CTC. A maioria das aplicações da carga de trabalho LANL empregam um número de EPsnão maior que 32, concentrando a maioria das aplicações na faixa de 17 a 32 EPs. Aplicações sendo executadas por um período de tempo pequeno e requisitando uma quantidade pequena de EPs sugerem que existe uma correlação entre a quantidade de recursos requisitados pela aplicação e seu tempo de execução.

A Tabela 4.6 ilustra o coeficiente de correlação de *Pearson* obtido, utilizando como variáveis em questão o número de *EPs* requisitados e o tempo de execução das aplicações. O coeficiente de correlação de Pearson r indica a intensidade da associação entre duas variáveis (Shefler, 1988). Esse coeficiente varia dentro do intervalo $[-1, +1]$. Quanto mais próximo de 1 é o valor de r, mais forte é a correlação entre as duas variáveis. Para o cálculo do valor

de r, apenas as aplicações que finalizaram naturalmente sua execução são consideradas (campo *completed* = 1).

As cargas de trabalho SDSC95, SDSC96, SDSC2000 e LANL apresentam coeficientes de correlação positivos. Isso indica que as aplicações desses ambientes de produção, que requisitam uma maior quantidade de EPs, tendem a resolver problemas maiores, o que reflete no tempo de execução mais elevado.

A exceção é a carga de trabalho CTC, que apresenta correlação negativa ($r = -0,03368$). A correlação negativa na carga de trabalho CTC é devida à quantidade expressiva (36, 23%) de aplicações seqüenciais, i.e., que requisitam apenas 1 EP em sua execução. O coeficiente de correlação para essa carga de trabalho, excluindo da análise as aplicações seqüenciais, é igual a $r = 0,04887$. Mesmo assim, essa correlação é mais fraca que a correlação observada para as demais cargas de trabalho.

O teste de hipóteses T com um nível de confiança de 95% revela que não existe uma correlação forte entre a quantidade de EPs requisitados e o tempo de execução das aplicações para as cargas de trabalhos analisadas. Isso permite concluir que embora positiva, a correlação entre tais variáveis não é forte o suficiente para ser conclusivo, devido à grande variabilidade de aplicações existentes nas cargas de trabalho.

Tabela 4.6: Correlação entre a quantidade de EPs e o tempo de execução

Carga de trabalho	**Coeficiente de Correlação (r)**
SDSC95	0,30981
SDSC96	0,39465
SDSC2000	0,14539
CTC	−0,03368
LANL	0,22249

4.5 Considerações finais

Este capítulo apresentou os detalhes da carga de trabalho utilizada nesta tese. Um número de 10 aplicações e 5 traços de execução foram selecionados. O conjunto de aplicações, composto por aplicações compactas, *benchmarks* e uma aplicação sintética, representa programas paralelos comumente utilizados para a solução de problemas que demandam grande potência computacional. As aplicações possuem origens diversas, são implementadas em diferentes linguagens de programação e utilizam diferentes ambientes de passagem de mensagens.

A análise dos *logs* de execução permite observar características comuns dos centros de computação considerados. Tais características incluem a grande variabilidade no tempo de execução das aplicações, sem um padrão comum entre ambientes de produção diferentes.

Embora os logs tenham sua origem da carga de trabalho de aplicações paralelas, observa-se um número expressivo de aplicações seqüenciais nas cargas de trabalho de origem dos laboratórios de pesquisa SDSC e CTC. A correlação entre a quantidade de elementos de processamento utilizados e o tempo de execução das aplicações mostra-se positiva, mas não forte o suficiente para caracterizar uma correlação comum para todas as cargas de trabalho.

A análise e descrição da carga de trabalho apresentadas neste capítulo servem como base para a construção e validação dos modelos de aquisição de conhecimento. Além disso, por terem sido estudados em trabalhos prévios, a utilização dessas aplicações e *logs* permite que os resultados obtidos nesta tese sejam comparados com as pesquisas existentes na área.

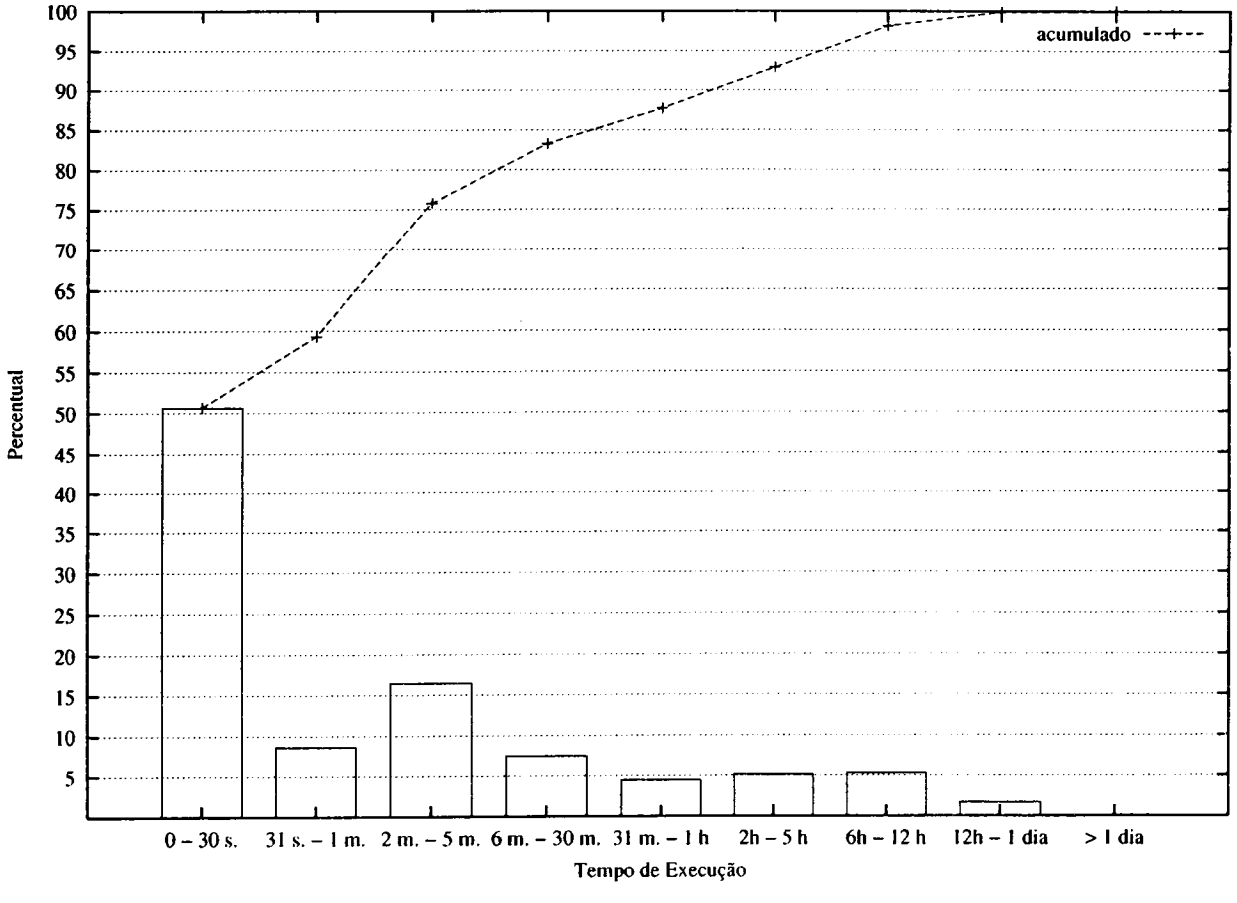


Figura 4.1: Histograma do tempo de execução das aplicações (SDSC95)

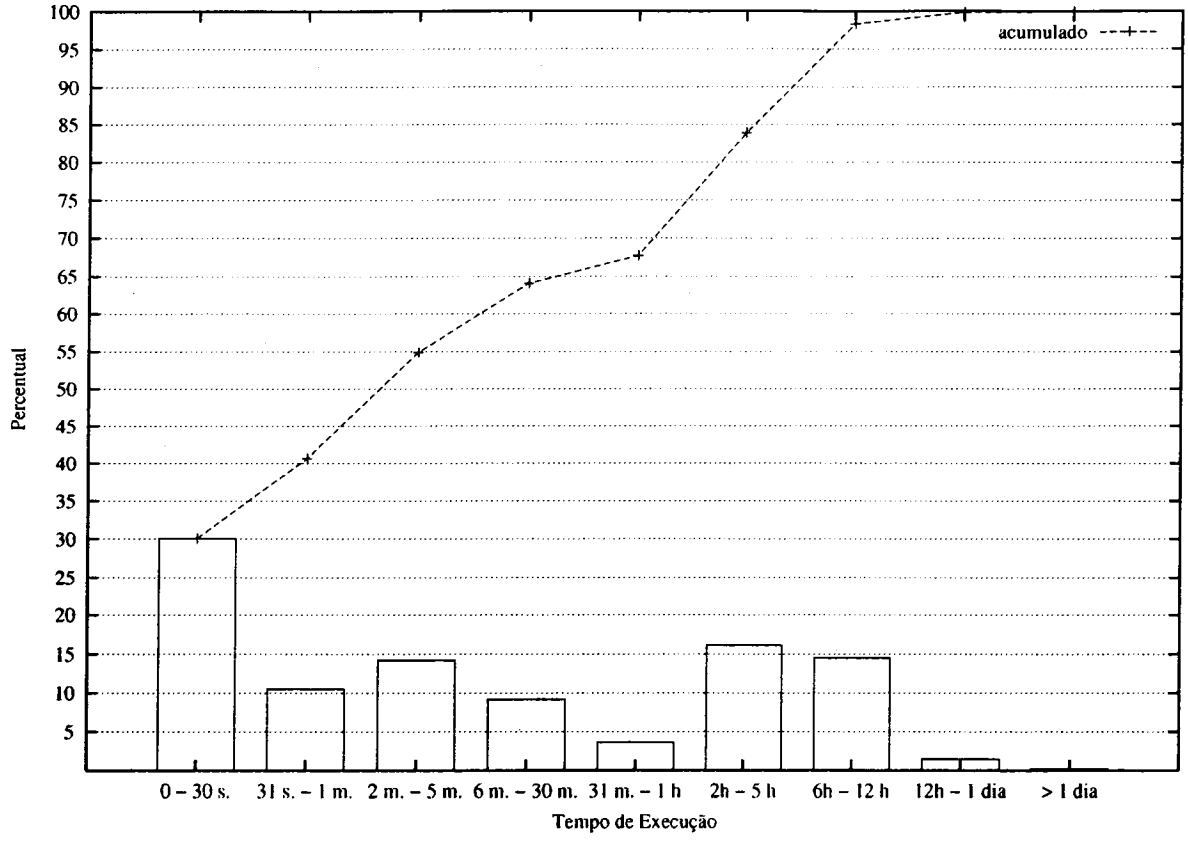


Figura 4.2: Histograma do tempo de execução das aplicações (SDSC96)

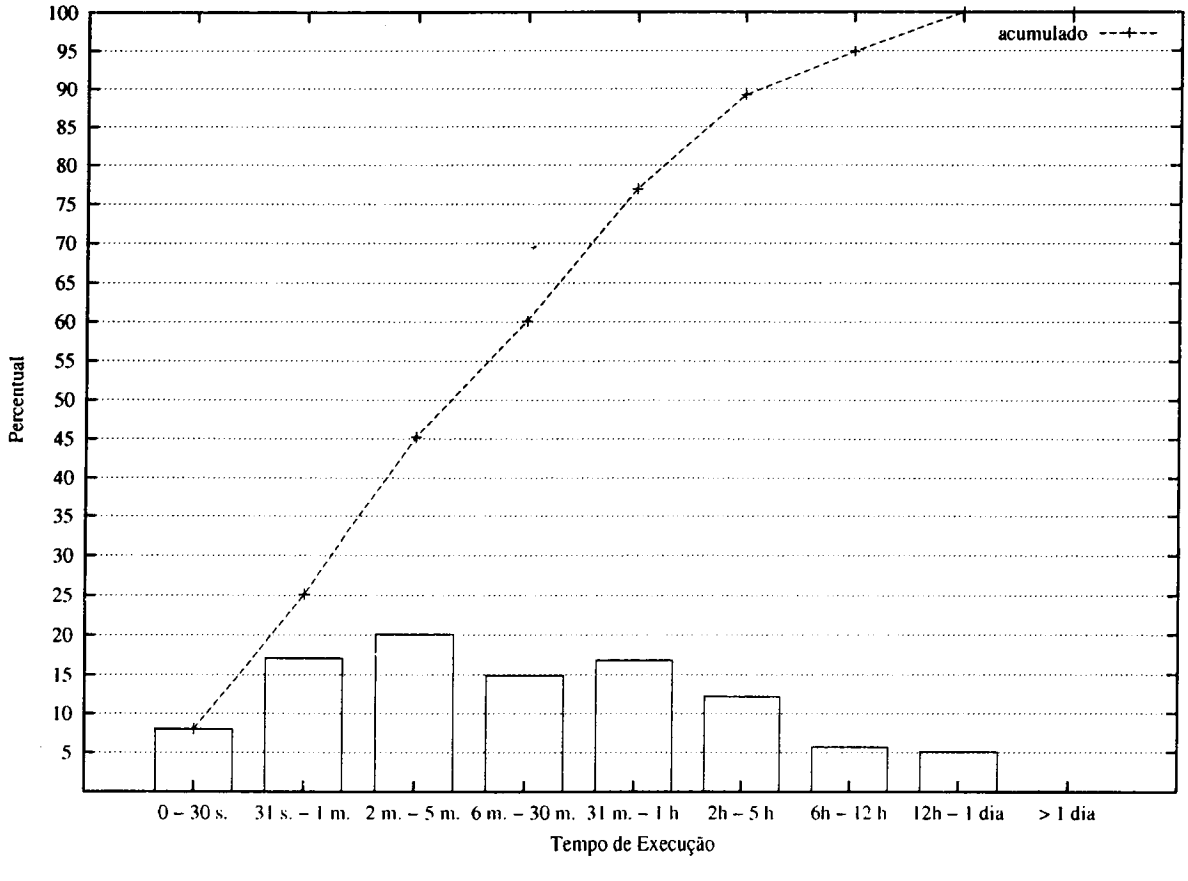


Figura 4.3: Histograma do tempo de execução das aplicações (SDSC2000)

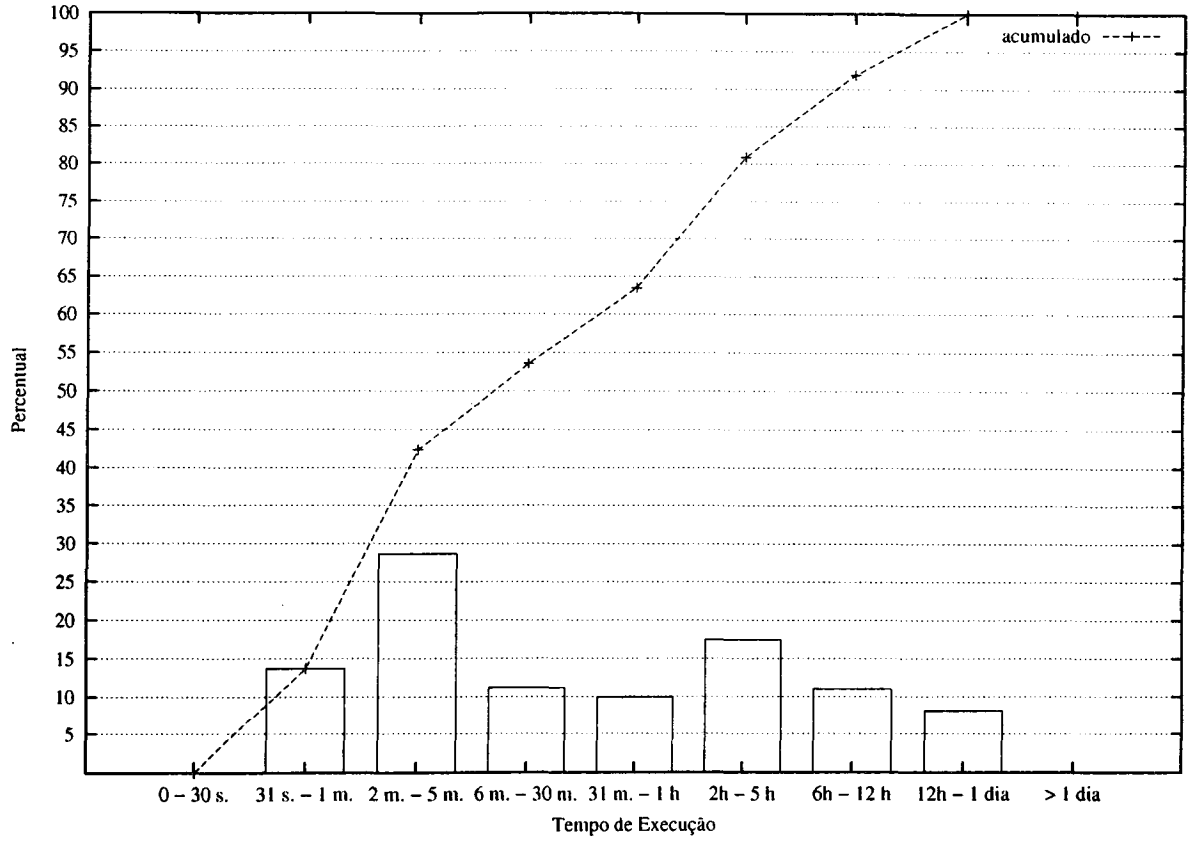


Figura 4.4: Histograma do tempo de execução das aplicações (CTC)

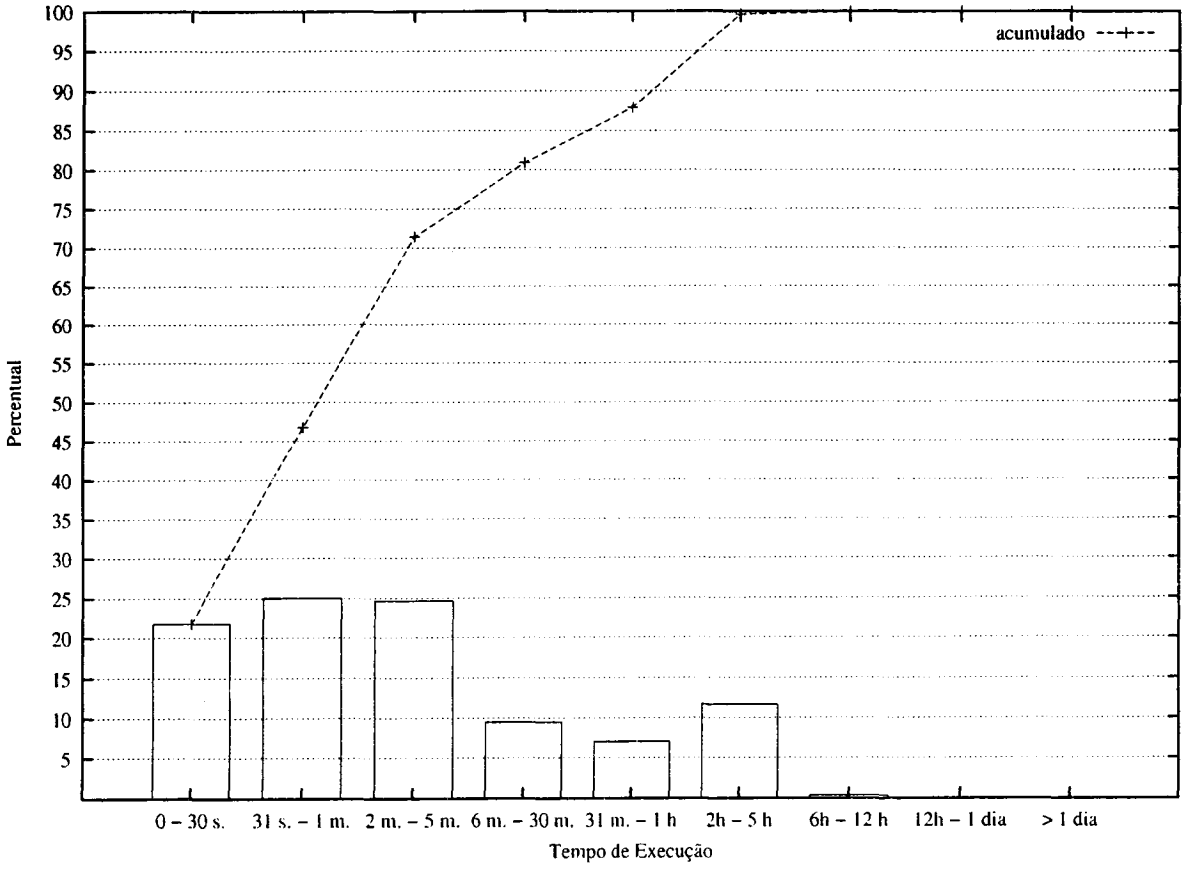


Figura 4.5: Histograma do tempo de execução das aplicações (LANL)

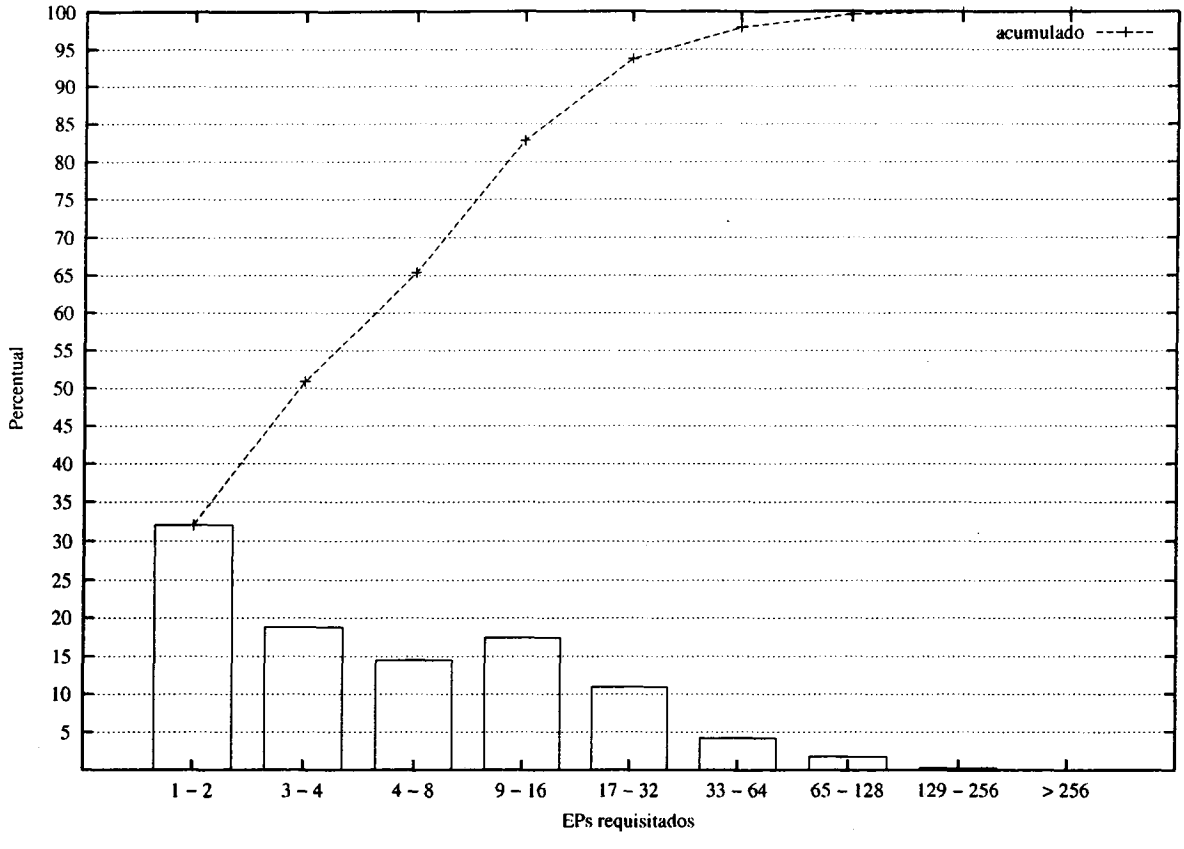


Figura 4.6: Histograma da quantidade de EPs requisitados (SDSC95)

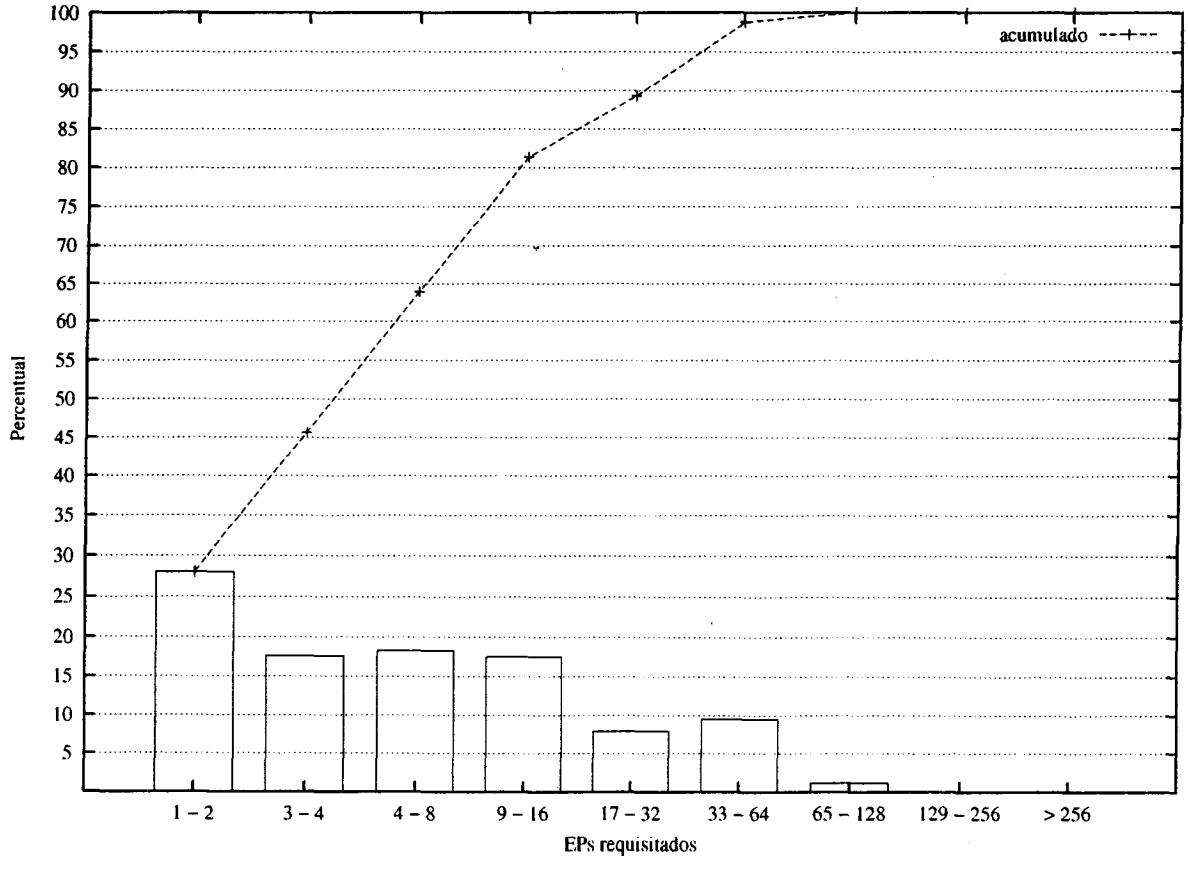


Figura 4.7: Histograma da quantidade de EPs requisitados (SDSC96)

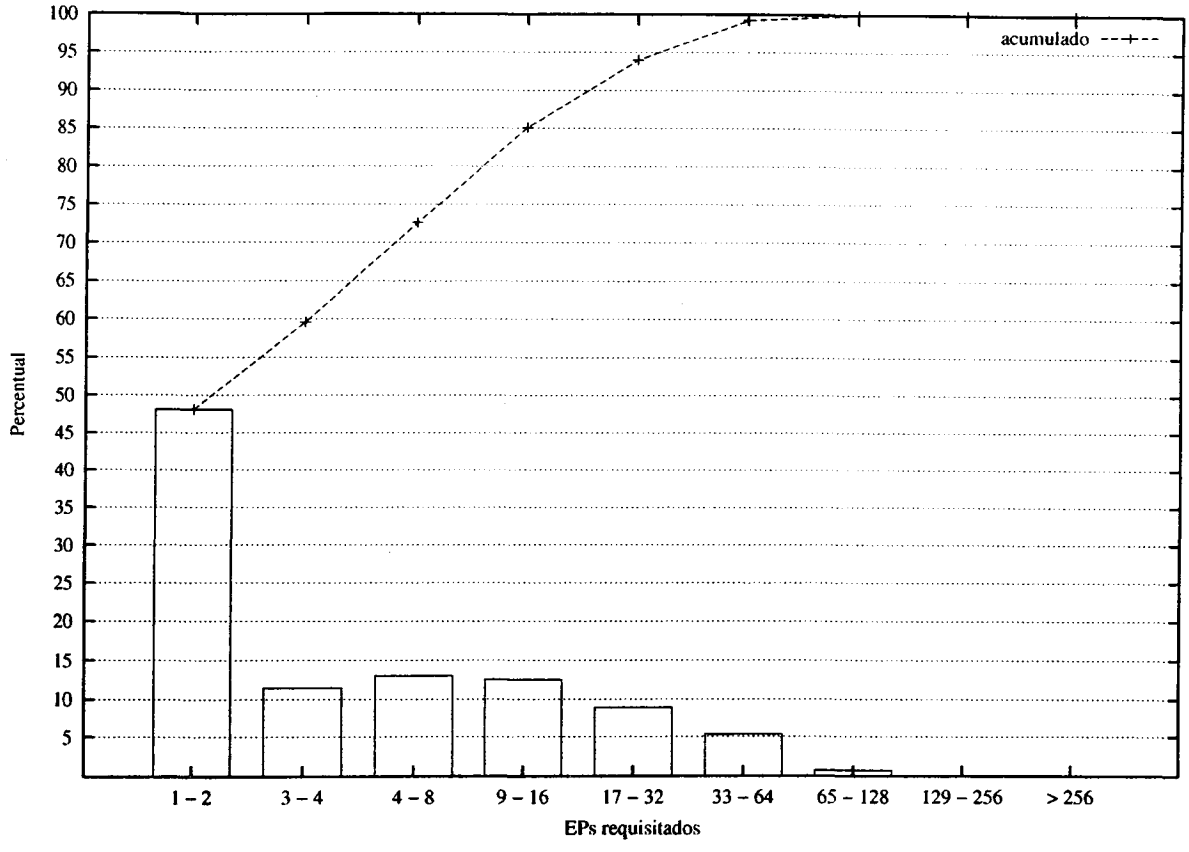


Figura 4.8: Histograma da quantidade de EPs requisitados (SDSC2000)

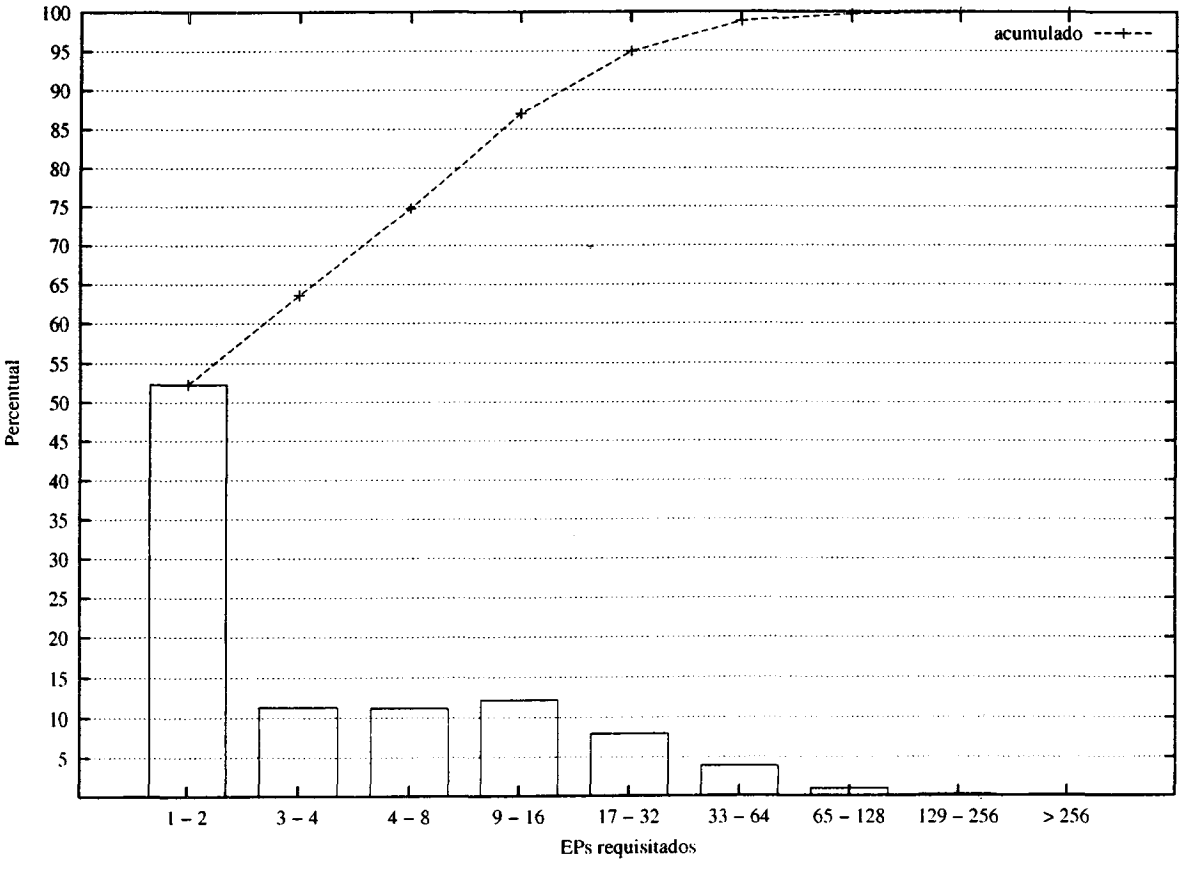


Figura 4.9: Histograma da quantidade de EPs requisitados (CTC)

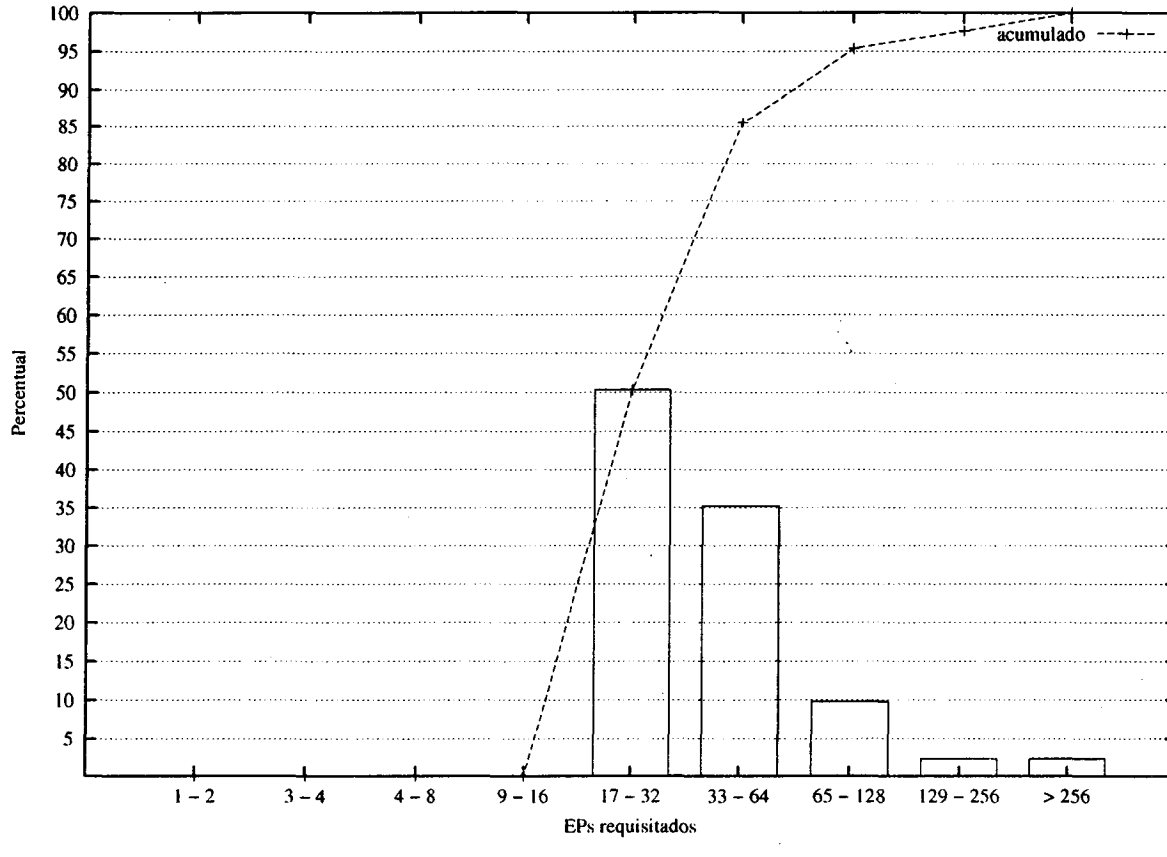


Figura 4.10: Histograma da quantidade de EPs requisitados (LANL)

CAPÍTULO

5

Exploração de Características de Execução de Aplicações Paralelas

5.1 Considerações iniciais

Trabalhos de avaliação de desempenho do escalonamento de processos têm demonstrado que o conhecimento sobre o comportamento das aplicações paralelas permite melhorar a utilização do sistema computacional e o tempo de resposta experimentado pelos usuários (Sevcik, 1989; Anastasiadis & Sevcik, 1997; Apon *et al.*, 1999; Smith *et al.*, 1998b). Através da caracterização das aplicações quanto ao seu comportamento na utilização de recursos, o software escalonador pode selecionar as políticas e localizar os recursos computacionais que melhor atendem aos requisitos das aplicações. O conhecimento sobre o comportamento das aplicações na utilização de recursos apresenta-se assim como uma ferramenta importante de apoio ao escalonamento em sistemas computacionais distribuídos.

Este capítulo trata da aquisição de conhecimento sobre aplicações paralelas, empregando características de execução dos processos. Inicialmente são definidos, implementados e avaliados mecanismos para a aquisição de informações sobre a execução de processos. Tais informações são empregadas por um modelo de aquisição de conhecimento, inspirado no paradigma conexionista e na arquitetura de redes neurais auto-organizáveis ART-2A. O desempenho de classificação do modelo é avaliado através de sua utilização para a aquisição de conhecimento sobre aplicações paralelas sintéticas, reais e *benchmarks*. Esse modelo é implementado junto ao núcleo do sistema operacional Linux e seu desempenho computacional é avaliado através da execução de *benchmarks*.

Este capítulo está organizado através de 4 seções principais. Na Seção 5.2, é apresentada a forma pela qual as informações sobre a execução dos processos em aplicações paralelas são

adquiridas. Nesse sentido, são definidos mecanismos para a instrumentação do sistema e para a monitoração das aplicações, visando coletar informações sobre os processos durante a utilização dos recursos computacionais. O desempenho computacional desses mecanismos é avaliado através da execução de *benchmarks*.

O modelo de aquisição de conhecimento é apresentada na Seção 5.3. Nesta seção, estão descritos o algoritmo de rotulação desenvolvido, detalhes da implementação do algoritmo de treinamento da rede ART-2A e as medidas de desempenho definidas para observar o desempenho da categorização da rede em relação ao parâmetro de vigilância. Essa seção apresenta a avaliação do modelo de aquisição de conhecimento, através de experimentos com um conjunto de aplicações paralelas.

A Seção 5.4 apresenta a implementação do modelo de aquisição de conhecimento junto ao núcleo do sistema operacional Linux, visando a classificação em tempo de execução de processos. Essa seção descreve as modificações necessárias que são aplicadas na implementação do modelo de aquisição de conhecimento para a utilização durante a execução dos processos. A avaliação de desempenho da implementação é apresentada no final desse seção. Essa avaliação permite avaliar a impacto da utilização do modelo no desempenho do sistema, considerando a execução de *benchmarks* e diferentes configurações para o modelo de aquisição.

A Seção 5.5 apresenta as considerações finais sobre os assuntos abordados neste capítulo.

5.2 Aquisição de informações sobre processos

A preocupação com a coleta de informações sobre processos tem sido observada nas implementações de sistemas operacionais e em trabalhos de monitoração do comportamento de processos. Sistemas operacionais da família UNIX, por exemplo o Linux e o FreeBSD, possuem a chamada de sistema `getrusage()`. Essa chamada permite coletar os valores acumulados de tempo de modo usuário, tempo em modo sistema, memória ocupada, operações de leitura e escrita em arquivos, mensagens enviadas por *sockets* e sinais recebidos de processos. Para que essa chamada de sistema torne-se disponível, é necessária a compilação do núcleo com a diretiva `CONFIG_BSD_PROCESS_ACCT` ativada. Detalhes mais específicos, como a quantidade de informação envolvida em cada operação do núcleo, não são coletados por essa chamada. Tal restrição não permite sua utilização para a coleta de informações mais detalhadas sobre operações em arquivos e sobre a utilização da rede de comunicação pela aplicação.

O Linux possui a ferramenta `gprof` (Graham *et al.*, 1982, 1983), que permite o monitoramento e produz o perfil de execução de programas compilados em diversas linguagens de programação. As rotinas do programa em execução são analisadas, de forma que a quantidade de chamadas e o tempo gasto em cada rotina é coletado. Essa ferramenta é útil para melhorar a implementação de um conjunto de rotinas de um programa, visando identificar rotinas implemen-

tadas de maneira errônea ou que apresentam desempenho ruim. Apesar de prover informações valiosas sobre o desempenho das rotinas de um programa, tal ferramenta não informa detalhes sobre a utilização de recursos do sistema pelo processo, e sim o tempo de execução gasto por rotina do programa e o grafo de transição das rotinas. Além disso, para que um programa possa ser monitorado, este deve ser obrigatoriamente compilado com as bibliotecas que compõem o software. Outra restrição do `gprof` é a dificuldade para a análise de programas recursivos.

Uma outra abordagem para a monitoração é a instrumentação do código fonte das aplicações paralelas e a utilização de bibliotecas de geração de perfis de execução. A ferramenta *Pablo* (Reed *et al.*, 1993, 1996; Madhyastha & Reed, 2002) captura informações detalhadas sobre as operações de E/S de aplicações, como tempo, duração e tamanho. Essas informações podem ser analisadas de maneira *off-line*, ao término da execução da aplicação paralela, através de software específicos para a visualização. Na mesma linha, a ferramenta *Charisma* (Nieuwejaar *et al.*, 1996) permite a análise de E/S de cargas de trabalho paralelas. Tais ferramentas não contemplam operações na rede de comunicação.

Através da análise dos trabalhos correlatos, observa-se a necessidade de definição de mecanismos adicionais que permitam a aquisição de informações mais detalhadas sobre a execução dos processos, de maneira dinâmica e sem a necessidade de recompilação das aplicações paralelas. A forma de aquisição de informações utilizada neste trabalho é implementada através da instrumentação das rotinas internas do sistema operacional e da monitoração da execução dos processos de aplicações paralelas. A monitoração é realizada através da coleta de informações obtidas com a utilização de chamadas de sistema adicionadas ao sistema operacional.

O Linux (Bovet & Cesati, 2000) é o sistema operacional adotado, por ser vastamente utilizado em computadores pessoais e estações de trabalho direcionadas para o processamento paralelo. Além disso, o núcleo (*kernel*) do sistema operacional Linux apresenta bom desempenho e têm o seu código fonte distribuído livremente [1]. Maiores detalhes sobre o núcleo desse sistema são apresentados a seguir.

5.2.1 O núcleo do sistema operacional Linux

O núcleo do sistema Linux é responsável pela criação de um máquina virtual para os processos de usuário, de forma que os processos possam ser escritos sem necessitar de nenhum conhecimento específico sobre os componentes físicos instalados no computador. Além disso, permite que vários processos possam ser executados concorrentemente e é responsável por mediar o acesso aos dispositivos de maneira segura (Bovet & Cesati, 2000).

O núcleo é composto conceitualmente de cinco módulos principais: escalonamento de processos, gerência de memória, sistema de arquivos virtual, interface com a rede e comunicação entre processos. O módulo de escalonamento de processos é responsável por controlar o acesso

[1] http://www.kernel.org

dos processos à UCP, buscando justiça na distribuição do tempo de processamento. O módulo de gerência de memória permite que vários processos compartilhem com segurança a memória principal do computador, além de implementar mecanismos de memória virtual para permitir que processos utilizem mais memória que a quantidade de memória principal existente no sistema. O módulo de sistema de arquivos virtual esconde os detalhes dos dispositivos de hardware através de uma interface comum para todos os dispositivos, além de permitir a utilização de vários tipos de sistemas de arquivos que são compatíveis com outros sistemas operacionais. O módulo de comunicação entre processos implementa vários mecanismos para a comunicação e sincronização entre processos, através de memória compartilhada, *pipes*, *named pipes* e semáforos.

A distribuição do núcleo do sistema operacional é composta por um número aproximado de 15.000 arquivos e diretórios. A organização dos diretórios é ilustrada na Tabela 5.1.

Tabela 5.1: Estrutura de diretórios do código fonte do núcleo do Linux

Diretório	**Descrição**
`arch`	código específico para a arquitetura do computador
`Documentation`	documentação
`drivers`	*drivers* de dispositivos
`fs`	sistemas de arquivos locais e de rede
`include`	protótipos de funções e constantes
`init`	inicialização do sistema
`ipc`	comunicação entre processos
`kernel`	tratamento de sinais, interrupções e escalonamento de processos
`lib`	bibliotecas auxiliares
`mm`	gerência de memória
`net`	interface de *sockets* e protocolos
`scripts`	utilitários para configuração do núcleo do sistema

No diretório `arch`, encontram-se os arquivos fontes responsáveis por detalhes de nível mais próximo à arquitetura na qual o sistema operacional será executado e por fornecer uma interface virtual para as demais partes do código independentes de arquitetura. O código fonte nesse diretório é organizado de acordo com a arquitetura alvo: existe um subdiretório para cada uma das arquiteturas suportadas pelo núcleo. Apesar de boa parte desse código estar implementado na linguagem C, é comum encontrar muitas rotinas implementadas diretamente na linguagem *Assembly*. As rotinas implementadas estão envolvidas com a carregamento inicial do sistema a partir do setor de *boot*, leitura e tratamento da BIOS do sistema, tratamento de sinais e interrupções e definição dos arquivos de *stubs* das chamadas do sistema.

A documentação do núcleo é descrita no diretório `Documentation`. Nesse diretório podem ser encontrados manuais e textos explicativos que visam auxiliar o usuário na configuração de dispositivos de entrada/saída, detalhes do processador, sistemas de arquivos e configuração de protocolos de rede.

O código fonte dos *drivers* para dispositivos do sistema estão organizados no diretório `drivers`. O código abrange *drivers* para unidades de armazenamento, interfaces de rede e dispositivos de entrada.

Na diretório `fs`, está localizado o código fonte para os sistemas de arquivo local e de rede, por exemplo o NFS (*Network File System*). O suporte para diferentes tipos de partições, assim como as rotinas e chamadas ao sistema para leitura e escrita de arquivos, estão também definidos nesse diretório.

Todo o código fonte do núcleo compartilha os protótipos de funções, macros e constantes definidos no diretório `include`. Tal diretório, assim como no diretório `arch`, é organizado em subdiretórios específicos para as arquiteturas que têm suporte no Linux, além de subdiretórios com protótipos independentes da arquitetura, como é o exemplo de protótipos para os *drivers* de dispositivos.

O código fonte que define o ponto de entrada no carregamento do sistema é definido no diretório `init`. Nesse diretório encontra-se o arquivo fonte `main.c`, programa principal do núcleo, que realiza a configuração inicial dos dispositivos do sistema e torna o sistema apto para a utilização em modo usuário.

As primitivas de comunicação entre processos estão definidas no diretório `ipc`. Exemplos de modos de comunicação que o núcleo fornece ao processos de usuário são os *pipes*, *named pipes*, memória compartilhada e semáforos.

O diretório `kernel` contém o código envolvido com temporizadores do sistema, criação e escalonamento de processos e tratamento de sinais. Nesse diretório encontra-se o arquivo fonte `sched.c`, responsável pela implementação das disciplinas de escalonamento de processos existentes no núcleo do sistema Linux. Bibliotecas de software auxiliares do código fonte do núcleo, por exemplo manipulação de estruturas de dados e cadeias de caracteres, estão localizadas no diretório `lib`.

O diretório `mm` contém o código que trata da gerência de memória no sistema Linux. Nesse diretório são implementados os mecanismos de gerência de memória independentes de dispositivo, como memória virtual e métodos de substituição de páginas. Uma interface para processos de usuário é implementada, para prover acesso restrito à memória para os processos de usuário. Essa interface permite que processos de usuário realizem alocação e liberação de memória e entrada/saída em arquivos através de mapeamento de memória.

O código fonte responsável pela gerência de protocolos de comunicação, por exemplo o conjunto de protocolos TCP/IP, encontra-se no diretório `net`. Os objetos de rede são representados pela interface de *sockets*, que são pontos de acesso aos protocolos de comunicação que são implementados no núcleo.

A parte de configuração do núcleo está definida na pasta `scripts`. Essa pasta contém arquivos interpretados que permitem a configuração do sistema em modo texto ou com as facilidades do ambiente gráfico.

5.2.2 Instrumentação do núcleo

A instrumentação é realizada através da adição de novos campos na tabela de processos do sistema operacional. No sistema Linux, a tabela de processos é definida no arquivo fonte `include/linux/sched.h`, através da estrutura de dados `task_struct`. Tal estrutura possui campos para o armazenamento de informações envolvidas com os processos, por exemplo estado, utilização de memória, credenciais (usuário e grupo), limites de utilização de recursos, semáforos, sistema de arquivos utilizado e arquivos abertos,

Os novos campos adicionados à estrutura `task_struct` representam contadores associados a cada processo processo. Tais contadores armazenam a quantidade de operações de leitura ou escrita em arquivos ou na rede de comunicação, assim como a quantidade de informação envolvida em cada operação. Em operações de rede, existem contadores específicos para os protocolos UCP e UDP, da pilha de protocolos TCP/IP. Esses protocolos são monitorados pelo fato de que ambientes de passagens de mensagens como PVM (Beguelin *et al.*, 1994) e MPI (MPI Forum, 1997) os empregam na comunicação entre processos que compõem uma aplicação paralela. Os contadores criados estão ilustrados na Tabela 5.2.

Tabela 5.2: Contadores adicionados a tabela de processos

Campo	**Informação armazenada**
`io_calls_read`	operações de leitura em arquivos
`io_bytes_read`	bytes transferidos na leitura de arquivos
`io_calls_write`	operações de escrita em arquivos
`io_bytes_write`	bytes transferidos na escrita em arquivos
`tcp_calls_read`	operações de leitura em *sockets* TCP
`tcp_calls_write`	operações de escrita em *sockets* TCP
`tcp_bytes_read`	bytes transferidos (leituras) em *sockets* TCP
`tcp_bytes_write`	bytes transferidos (escritas) em *sockets* TCP
`udp_calls_read`	operações de leitura em *sockets* UDP
`udp_calls_write`	operações de escrita em *sockets* UDP
`udp_bytes_read`	bytes transferidos (leituras) em *sockets* UDP
`udp_bytes_write`	bytes transferidos (escritas) em *sockets* UDP

A atualização desse contadores é feita sempre que um processo executa alguma das funções que são monitoradas. Por exemplo, se um processo executa um leitura em um arquivo, os contadores `io_calls_read` e `io_bytes_read`, são atualizados. Para a coleta do valor desses contadores, são definidas novas chamadas de sistema (*system calls*). Uma chamada de sistema é um método tradicional em sistemas operacionais para que um processo solicite algum

serviço do núcleo do sistema ou para prover para processos que executam em modo usuário alguma informação interna ao núcleo (Tanenbaum, 1995). As novas chamadas de sistema são implementadas de forma que um processo em modo usuário atue como um monitor e possa coletar informações sobre o processo que está sendo monitorado.

A Tabela 5.3 ilustra os arquivos fonte do núcleo que foram modificados para realizar a instrumentação do sistema e acomodar as novas chamadas de sistema.

Tabela 5.3: Arquivos fonte do núcleo modificados para a instrumentação

Arquivo fonte	**Descrição**
`include/linux/sched.h`	estrutura de dados da tabela de processos
`kernel/fork.c`	criação de processos
`mm/filemap.c`	leitura e escrita em descritores de arquivo
`net/socket.c`	leitura e escrita em *sockets*
`arch/i386/kernel/entry.S`	registro das interrupções de software
`include/asm/unistd.h`	*stubs* para cada chamada de sistema
`include/linux/sys.h`	definição do número de chamadas de sistema
`fs/read_write.c`	rotinas que manipulam as novas interrupções

5.2.3 Instrumentação das rotinas do núcleo

Os contadores têm seus valores inicializados em zero quando o processo é criado. No sistema Linux, as chamadas de sistema `clone` e `fork` são empregadas para a criação de processos. Tais chamadas criam uma cópia (processo filho) exatamente igual ao processo original (processo pai), de forma que todos os processos no sistema são organizados como uma estrutura em árvore (Tanenbaum, 1992). Internamente, tais chamadas invocam a rotina do núcleo `do_fork()`. Essa rotina é diretamente responsável pela criação de um processo, copiando o segmento de dados do processo pai e acertando os valores dos registradores. A rotina `do_fork()` está localizada no arquivo fonte `kernel/fork.c`. Essa rotina é alterada para marcar como zero os valores dos contadores na criação de processos. Essa inicialização é importante para que leituras subseqüentes desses contadores não obtenham valores incoerentes.

No núcleo do sistema, o ponteiro global ao núcleo `current` referencia a função `get_current` conseqüentemente, o processo que está requisitando serviços do núcleo. A função `get_current()` é localizada no arquivo `include/asm/current.h` e retorna um ponteiro para a estrutura `task_struct` do processo. Na instrumentação do núcleo do sistema, esse ponteiro é freqüentemente referenciado para a atualização e coleta dos valores dos contadores adicionados na tabela de processos.

As rotinas de leituras em arquivos são instrumentadas através da alteração no arquivo fonte `mm/filemap.c`. Nesse arquivo, são realizadas as seguintes modificações:

- a função `generic_file_read()` é modificada para registrar nos contadores `io_calls_read` e `io_bytes_read` as operações de leitura em arquivos;
- a função `generic_file_write()` é modificada para registrar nos contadores `io_calls_write` e `io_bytes_write` as operações de escrita em arquivos.

Os pontos de acesso aos protocolos TCP/IP são implementados no código armazenado no arquivo `net/socket.c`. Nesse arquivo, são realizadas as seguintes modificações:

- a função `sock_read()` é modificada para registrar nos contadores `tcp_calls_read` e `tcp_bytes_read` as operações de leitura em *sockets* TCP;
- a função `sock_write()` é modificada para registrar nos contadores `tcp_calls_write` e `tcp_bytes_write` as operações de escrita em *sockets* TCP;
- a função `sys_recvfrom()` é modificada para registrar nos contadores `udp_calls_read` e `udp_bytes_read` as operações de leitura em *sockets* UDP;
- a função `sys_sendto()` é modificada para registrar nos contadores `udp_calls_read` e `udp_bytes_read` as operações de leitura em *sockets* UDP;
- a função `sock_readv_writev()` é modificada para registrar as leituras e os bytes transferidos em sockets TCP, quando as chamadas de sistema `readv()` e `writev()` são invocadas por processos Linux.

As demais modificações executadas no núcleo do sistema estão envolvidas diretamente com a implementação das novas chamadas de sistema para a coleta dos valores dos contadores. Tais modificações são descritas na próxima seção.

5.2.4 Implementação das novas chamadas de sistema

Através da instrumentação das rotinas do núcleo podem-se obter informações confiáveis sobre a utilização de recursos de processos. Para coletar tais informações é necessário que um processo monitor obtenha acesso aos contadores do processo, protegidos pelo núcleo do sistema. Para realizar a coleta, são definidas novas chamadas de sistema.

Uma chamada de sistema não pode ser invocada diretamente de um processo de usuário, e sim indiretamente através de uma interrupção de software. O núcleo do sistema gerencia as interrupções de software empregando uma tabela, que realiza o mapeamento do número da interrupção para a rotina apropriada do núcleo. A tabela é implementada através da linguagem *Assembly* no arquivo fonte (`arch/i386/kernel/entry.S`) e define quais são as interrupções de software associadas a cada chamada de sistema. Cada chamada de sistema recebe um

índice, que varia de zero até o número de chamadas de sistema existentes. As chamadas de sistema adicionadas ao núcleo são registradas com os números de interrupção de 253 a 264.

Além da alteração nessa tabela, é necessário gerar um *stub* para que um programa em modo usuário possa invocar as novas chamadas de sistema. Isso é feito através da modificação do arquivo `include/asm/unistd.h`. Essa etapa é necessária para a compilação de programas que utilizem tais chamadas seja facilitada. Outra modificação importante é a alteração do valor da constante `NR_syscalls`, que é definida no arquivo fonte `include/linux/sys.h`. O valor dessa constante foi alterado para refletir o aumento de chamadas de sistema.

Além do registro das chamadas de sistema, é necessário a implementação do código responsável por manipular a interrupção de software. Para isso são criadas 12 novas rotinas, uma para cada contador adicionado à tabela de processos do sistema. Todas as rotinas recebem como parâmetro de entrada o identificador (`pid`) do processo e retornam o valor do contador associado. O identificador passado como entrada é utilizado para pesquisar entre os processos postos em execução qual é a estrutura `task_struct` do núcleo do sistema que representa o processo. Para essa pesquisa, é utilizada a rotina `find_task_by_pid()`, definida no arquivo `include/linux/sched.h`. Essa rotina efetua uma busca rápida empregando a estrutura de dados do núcleo responsável por implementar uma lista de processos postos em execução pelo sistema.

Além de retornar o valor do contador, as rotinas marcam o valor do contador como zero no final de seu código. Isso permite que duas leituras consecutivas ao contador retornem a informação correspondente ao espaço de tempo entre a execução das duas chamadas. As novas rotinas foram inseridas no arquivo fonte `fs/read_write.c`. Isso permite que não seja necessária a alteração dos arquivos de definição de compilação (*makefiles*) do núcleo do sistema.

Tabela 5.4: Novas chamadas de sistema

Chamada de sistema	Interrupção	Contador associado
sys_IO_calls_read	253	io_calls_read
sys_IO_bytes_read	254	io_bytes_read
sys_IO_calls_write	255	io_calls_write
sys_IO_bytes_write	256	io_bytes_write
sys_TCP_calls_read	257	tcp_calls_read
sys_TCP_calls_write	258	tcp_calls_write
sys_TCP_bytes_read	259	tcp_bytes_read
sys_TCP_bytes_write	260	tcp_bytes_write
sys_UDP_calls_read	261	udp_calls_read
sys_UDP_calls_write	262	udp_calls_write
sys_UDP_bytes_read	263	udp_bytes_read
sys_UDP_bytes_write	264	udp_bytes_write

Duas informações adicionais são obtidas pela monitoração dos processos: o tempo de UCP em modo usuário e a utilização de memória. Tais características foram obtidas através de leituras no sistema de arquivos de processos Linux `procfs` (*process filesystem*) (Mouw, 2001). O `procfs` é um sistema de arquivos virtual que implementa uma interface para a tabela de processos do núcleo, através das abstrações comuns de tratamento de arquivos. Esse sistema de arquivos é chamado virtual por estar localizado apenas na memória principal do sistema, não sendo armazenado em unidades de armazenamento não-voláteis como sistemas de arquivo tradicionais. O sistema operacional é responsável por criar uma entrada para cada processo existente no sistema no diretório `proc`, sendo que a entrada de cada processo é um subdiretório nessa pasta cujo nome corresponde ao seu identificador (`pid`). O núcleo do sistema é responsável pela coleta e atualização dos pseudo-arquivos existentes neste diretório.

As abstrações comuns fornecem uma interface para extrair informações sobre os processos, seu ambiente de execução e sua utilização de recursos. Para extrair informações sobre um processo, é utilizado o arquivo `/proc/<pid>/stat`, onde `<pid>` corresponde ao identificador do processo monitorado. O tempo de UCP é descrito em *jiffies*, que é a unidade de temporização utilizada no núcleo do sistema Linux. Um segundo equivale a 100 *jiffies*. Tal valor tem sua origem na taxa do controlador programável de interrupção da arquitetura IBM-PC.

5.2.5 Monitoração de processos

A monitoração de processos é realizada através da leitura no sistema de arquivos `procfs` e da utilização das novas chamadas de sistema. Tal monitoração resulta em um conjunto de 14 variáveis, descritas na Tabela 5.5.

Tabela 5.5: Informações obtidas pela monitoração

Informação	**Fonte**
Tempo de UCP em modo usuário	`procfs`
Utilização de memória	`procfs`
Número de leituras em arquivos	`sys_IOCallsRead()`
Bytes lidos em arquivos	`sys_IOBytesRead()`
Número de escritas em arquivos	`sys_IOCallsWrite()`
Bytes escritos em arquivos	`sys_IOBytesWrite()`
Número de leituras em *sockets* TCP	`sys_TCPCallsRead()`
Número de escritas em *sockets* TCP	`sys_TCPCallsWrite()`
Bytes lidos em *sockets* TCP	`sys_TCPBytesRead()`
Bytes escritos em *sockets* TCP	`sys_TCPBytesWrite()`
Número de leituras em *sockets* UDP	`sys_UDPCallsRead()`
Número de escritas em *sockets* UDP	`sys_UDPCallsWrite()`
Bytes lidos em *sockets* UDP	`sys_UDPBytesRead()`
Bytes escritos em *sockets* UDP	`sys_UDPBytesWrite()`

Essas informações são coletadas através de um programa denominado `monitor`. Tal programa executa como um processo servidor, atendendo requisições que chegam através de um *named pipe* do sistema. Quando uma requisição é enviada ao `monitor`, um processo filho é criado para atender a requisição, de forma que o monitor pode atender outras requisições, se necessário. O `monitor` recebe como entrada o identificador do processo (`pid`) a ser monitorado e a taxa de amostragem, em microssegundos. A saída do programa são as 14 variáveis coletadas, amostradas de acordo com o intervalo de tempo especificado. Tal saída é gravada em um arquivo, cujo nome é uma seqüência de caracteres formada pelo nome do processo que está sendo monitorado, identificador do processo e intervalo de amostragem.

O desempenho do `monitor` e sua intrusão no desempenho do sistema são considerados na implementação. O arquivo onde as informações coletadas são gravadas é aberto no início da monitoração e fechado no término da monitoração. Isso permite que o sistema de *cache* de arquivos do sistema operacional seja aproveitado, de forma a não descarregar imediatamente em disco as informações coletadas. A amostragem periódica é realizada utilizando a chamada de sistema `usleep`. Isso permite que o processo abdique de seu tempo de UCP após realizar uma amostra de informações sobre o processo monitorado.

A Figura 5.1 ilustra um subconjunto das informações coletadas sobre a execução de um dos processos da aplicação sintética A. O intervalo de amostragem igual a 500000 microssegundos. O gráfico apresenta no eixo das abscissas o tempo de execução e no eixo das ordenadas os valores coletados dos contadores. A monitoração descreve a execução da aplicação em relação às suas fases: comunicação, processamento, operações de E/S e comunicação.

5.2.6 Avaliação da intrusão da monitoração no sistema

A análise das modificações incluídas no núcleo do sistema indicam dois fatores que podem influenciar o desempenho: a instrumentação e a monitoração. Em relação à instrumentação do núcleo, uma modificação sensível para o desempenho do sistema é a utilização da rotina `find_task_by_pid`. Essa rotina é sempre ativada quando os processos executam as operações instrumentadas. Como é necessária a busca nas estruturas de dados do núcleo, tal modificação poderia depreciar o desempenho do sistema. Além disso, a monitoração pode incluir uma sobrecarga adicional para o sistema, por efetuar chamadas de sistema e leitura no sistema de arquivos `procfs` periódicas.

Para a avaliação da intrusão da monitoração do sistema, considera-se como medida de desempenho o tempo de resposta de aplicações do pacote NAS (vide Capítulo 4). Três configurações de software são consideradas para a execução dessas aplicações:

- máquina executando o núcleo do sistema não modificado;

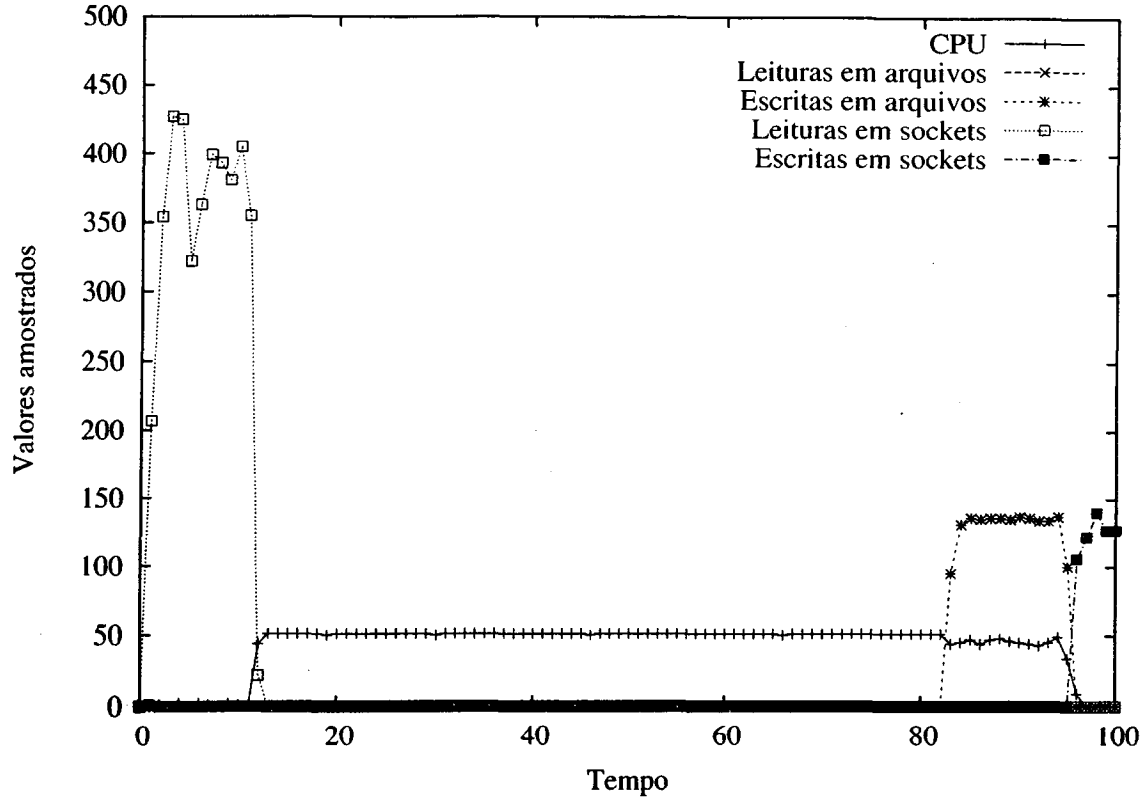


Figura 5.1: Informação obtida através da monitoração da aplicação **A**.

- máquina executando um núcleo do sistema modificado, que inclui as novas chamadas de sistema;

- máquina executando um núcleo do sistema modificado e o processo monitor, coletando informações sobre a aplicação em execução.

O intervalo de amostragem escolhido para os experimentos é igual a 250000 microssegundos, que corresponde a uma amostra a cada $1/4$ de segundo. Os núcleos não modificado e modificado compartilham os mesmos arquivos de configuração para a compilação. Assim, a imagem do sistema operacional compilada é semelhante para os dois casos, incluindo os mesmos componentes e *drivers*, sendo os arquivos modificados pela instrumentação a única diferença existente (Tabela 5.3). As aplicações são executadas em duas máquinas de configurações diferentes, EP_7 e EP_6, como uma forma de observar o impacto da monitoração em máquinas com diferentes potências computacionais.

A Tabela 5.6 ilustra os resultados obtidos para 50 execuções de cada aplicação empregando o EP_7, para um núcleo não modificado. Tais resultados são comparados com duas outras configurações de software:

- núcleo modificado (Tabela 5.7). Essa configuração permite verificar o impacto isolado das modificações realizadas no núcleo;

- núcleo modificado com o processo monitor sendo executado, com intervalo de amostragem igual a 250000 microssegundos (Tabela 5.8). Essa configuração permite verificar o impacto agregado das modificações do núcleo e da intrusão do processo monitor no desempenho do sistema.

Através dos resultados, pode-se observar a baixa variabilidade no tempo de resposta das aplicações, considerando uma amostra igual a 50 execuções. Utilizando o teste Z com um nível de confiança de 95%, pode-se concluir que as modificações no núcleo do sistema não afetam o desempenho, em relação ao tempo de resposta, de todos as aplicações executadas no EP_7. Da mesma forma, conclui-se que o núcleo modificado com a monitoração empregando um intervalo de 250000 microssegundos não afeta o desempenho das aplicações consideradas.

Tabela 5.6: Tempos médios de resposta das aplicações do pacote NAS no EP_7 (tamanho da amostra igual a 50), com núcleo não modificado

Benchmark	**BT**	**CG**	**EP**	**IS**	**LU**	**MG**	**SP**
Média	37,185	3,319	8,745	0,250	74,151	2,735	85,570
Desvio padrão	0,037	0,007	0,007	0,003	0,053	0,015	0,066

Tabela 5.7: Tempos médios de resposta das aplicações do pacote NAS no EP_7 (tamanho da amostra igual a 50), com núcleo instrumentado

Benchmark	**BT**	**CG**	**EP**	**IS**	**LU**	**MG**	**SP**
Média	37,198	3,320	8,747	0,250	74,143	2,736	85,575
Desvio padrão	0,040	0,007	0,013	0,001	0,048	0,013	0,075

Tabela 5.8: Tempos médios de resposta das aplicações NAS no EP_7 (tamanho da amostra igual a 50), com núcleo instrumentado e processo monitor com intervalo de amostragem igual a 250000 microssegundos

Benchmark	**BT**	**CG**	**EP**	**IS**	**LU**	**MG**	**SP**
Média	37,281	3,331	8,785	0,262	74,394	2,735	85,524
Desvio Padrão	0,048	0,008	0,005	0,006	0,214	0,013	0,138

Nas Tabelas 5.9, 5.10 e 5.11 estão descritos os resultados obtidos para o EP_6, para 15 execuções de cada aplicação. O tamanho de amostra igual a 15 e utilizado devido à baixa variabilidade obtida nos resultados considerando o EP_7. Nota-se a diferença de tempo de resposta para cada aplicação, em relação aos resultados do EP_7, causada pela diferença de potência computacional dos dois elementos de processamento.

Utilizando o teste T com um nível de confiança de 95%, pode-se concluir que as modificações no núcleo do sistema não afetam o tempo de resposta das aplicações. Tal afirmação é válida também para o núcleo modificado com a monitoração, empregando um intervalo

de 250000 microssegundos. O teste T é usado nesse caso, pois o tamanho da amostra está abaixo de 30 (Shefler, 1988).

Através dos resultados obtidos com a avaliação de desempenho, pode-se observar que as modificações no núcleo e a monitoração da execução dos processos não depreciam o desempenho do sistema. Assim, pode-se concluir que a implementação dos mecanismos de aquisição de informações sobre processos são adequados para serem utilizados para a exploração de características de execução de aplicações paralelas. A exploração dessas características, através da definição de um modelo de aquisição de conhecimento sobre aplicações paralelas, é descrita a seguir.

Tabela 5.9: Tempos médios de resposta das aplicações NAS no EP_6 (tamanho da amostra igual a 15), com núcleo não modificado

Benchmark	**BT**	**CG**	**EP**	**IS**	**LU**	**MG**	**SP**
Média	392,008	91,795	65,412	8,531	816,924	43,372	1073,179
Desvio padrão	5,034	0,175	0,343	0,959	14,734	0,579	31,697

Tabela 5.10: Tempos médios de resposta das aplicações do pacote NAS no EP_6 (tamanho da amostra igual a 15), com núcleo instrumentado

Benchmark	**BT**	**CG**	**EP**	**IS**	**LU**	**MG**	**SP**
Média	393,565	91,950	65,545	8,850	816,811	42,673	1069,017
Desvio padrão	3,103	0,272	0,128	0,692	35,085	1,495	16,797

Tabela 5.11: Tempos médios de resposta das aplicações do pacote NAS no EP_6 (tamanho da amostra igual a 15), com núcleo instrumentado e processo monitor com intervalo de amostragem igual a 250000 microssegundos

Benchmark	**BT**	**CG**	**EP**	**IS**	**LU**	**MG**	**SP**
Média	392,433	91,892	65,547	8,771	822,660	42,979	1070,305
Desvio padrão	3,579	0,378	0,241	1,272	21,077	5,664	28,519

5.3 Aquisição de conhecimento sobre as características de execução de aplicações paralelas

A pesquisa na área de escalonamento de processos em aplicações paralelas têm investigado formas de explorar as características de execução dos processos, visando melhorar as decisões do software de escalonamento. Para a identificação dos estados de utilização de recursos de processos, Devarakonda & Iyer (1989) propõem uma técnica da área de reconhecimento de padrões, que tem como o objetivo final a predição do tempo de UCP, operações de entrada/saída e utilização de memória de um programa no início de sua execução, dada a identidade desse programa. Essa técnica emprega uma forma de categorização estatística, através do algoritmo

k-means, para a identificação de estados de utilização de recursos de todos os processos executados em um sistema *UNIX* uniprocessado. A partir da identificação desses estados, é criado um diagrama de transição de estados, que serve para predizer o tempo de execução de um processo, levando em conta informações da última execução desse processo e o diagrama de estados de toda a carga de trabalho.

Essa técnica é também explorada, de maneira similar, por outros autores (Dimpsey & Year, 1991, 1995). Nesses trabalhos, é considerada uma carga de trabalho composta de processos seqüenciais e não são consideradas as operações de comunicação entre processos. Além disso, esses trabalhos realizam a análise após o término da execução dos processos, e não são feitas considerações sobre a utilização dessa técnica durante a execução dos processos.

Arpaci-Dusseau *et al.* (1998) e Feitelson & Rudolph (1995a) empregam informações coletadas durante a execução das aplicações para auxiliar as decisões de um software de escalonamento cooperativo implícito. Nesses trabalho, os autores avaliam a quantidade de mensagens recebidas e enviadas pelas tarefas das aplicações para escolher o tempo em que a tarefa ficará em espera ocupada ou por bloquear a execução da tarefa, visando implicitamente coordenar a execução das aplicações. Apesar disso, os autores não definem como obter tais informações para a parametrização do escalonador.

Nesse sentido, Silva & Scherson (2000) investigam o uso de medidas coletadas em tempo de execução das aplicações, empregando estimadores bayesianos e lógica nebulosa para construir um modelo que visa classificar as aplicações e ajustar os mecanismos do algoritmo de escalonamento. A idéia explorada é classificar o comportamento dos processos, em tempo de execução, em classes como orientada ao processamento, à entrada/saída e à comunicação. Nos trabalhos de Arpaci-Dusseau *et al.* (1998), Feitelson & Rudolph (1995a) e (Silva & Scherson, 2000) os esquemas de aquisição de informações e classificação não permitem que o conhecimento sobre uma determinada aplicação seja salvo e utilizado em uma nova execução da aplicação. Além disso, os esquemas definidos pelos autores estão relacionados apenas com algoritmos de escalonamento cooperativos.

Madhyastha & Reed (2002) empregam algoritmos de aprendizado de máquina para classificar o padrão de acesso de aplicações paralelas em arquivos. Os esquemas propostos utilizam como algoritmos redes neurais baseadas em perceptron multi-camadas e modelos de Markov ocultos. Tais algoritmos são treinados com cargas de trabalho sintéticas, e os algoritmos de aprendizado utilizados não permitem a atualização do conhecimento a partir da observação de novas informações. Esse trabalho tem como objetivo verificar os padrões de comportamento de acesso ao disco, i.e. a forma de acesso aos arquivos, de maneira isolada, visando melhor parametrizar o sistema de arquivos, no que diz respeito principalmente a utilização de cache e não tem o objetivo de melhor o escalonamento das aplicações no sistema.

Pela análise dos trabalhos correlatos, pode-se observar a falta de trabalhos que permitam adquirir de maneira dinâmica e incremental o conhecimento sobre o comportamento dos processos na utilização dos recursos, explorando as características de execução e permitindo a classificação das aplicações paralelas.

O modelo de aquisição de conhecimento descrito nesta seção tem o objetivo principal de classificar o comportamento das aplicações paralelas na utilização de recursos, através da identificação, em tempo de execução, das seguintes classes (Senger *et al.*, 2003, 2004b):

- Orientada ao processamento (*CPU Bound*): tal classe representa o estado de execução da aplicação paralela responsável pela realização do processamento de dados, especificamente operações com matrizes e processamento matemático em geral;
- Orientada às operações de entrada e saída (*I/O Bound*): tal classe representa os estados da aplicação paralela responsáveis pela realização de entrada e saída em dispositivos de armazenamento, por exemplo escrita no sistema de arquivos;
- Orientada à comunicação (*Communication Intensive*): tal classe representa os estados da aplicação paralela responsáveis por realizar a comunicação entre as tarefas que compõem a aplicação paralela, através de operações de envio e recebimento de mensagens pela rede de comunicação.

O esquema de classificação considera que a execução da aplicação paralela é descrita pela identificação dos estados de utilização de recursos e a partir da freqüência de visita em cada estado, obtém-se os estados mais visitados pela aplicação. Para atingir o seu objetivo, o modelo emprega informações coletadas durante a execução das aplicações, visando identificar quais são os estados de utilização de recursos visitados pelas aplicações. Essas informações são obtidas através da monitoração descrita na Seção 5.2. Assim, observações sobre a execução da aplicação paralela são realizadas a cada intervalo de tempo T_o. Para cada observação é criada uma amostra S^i, sendo que i corresponde ao índice da amostra. Cada amostra possui um tamanho n e é representada por um vetor no espaço euclidiano $X \in \Re^n$:

$$S^i = [x_1^i \quad x_2^i \quad x_3^i \quad \ldots \quad x_n^i]^T \tag{5.1}$$

Todas as amostras constituem um conjunto T:

$$T = \{S^1, S^2, \ldots, S^n\} \tag{5.2}$$

Esse conjunto corresponde ao traço de execução completo, com cada amostra em intervalos de tempo T_o

Como o objetivo do modelo é a identificação e classificação dos estados de utilização de recursos visitados pela aplicação paralela em sua execução, as informações obtidas através

desse traço de execução são classificadas através de um conjunto de categorias. Cada categoria identifica uma classe de operações na qual a aplicação está envolvida, por exemplo o estado orientado ao processamento. A incorporação de um algoritmo de categorização (*clustering*) ao modelo de classificação permite a identificação automática de padrões de utilização de recursos, tendo como origem as informações obtidas pela monitoração.

Um aspecto importante para os objetivos do modelo é a forma de categorização dos vetores de entrada do conjunto T. Tal categorização pode ser realizada de maneira distintas, empregando técnicas estatísticas ou através de algoritmo direcionados para esse fim, por exemplo o algoritmo *k-means* (Devarakonda & Iyer, 1989). Essas técnicas necessitam de que todos os vetores do conjunto T estejam disponíveis para realizar a categorização e apresentam um tempo de execução potencialmente elevado, além da necessidade da definição prévia da quantidade máxima de categorias a serem criadas. Além disso, a atualização do conhecimento obtido não é facilmente realizada.

Desde que é desejado um esquema que possa trabalhar com os dados de maneira *on-line*, é adotada como solução a utilização da arquitetura de redes neurais ART (vide Capítulo 3). Redes neurais da família ART podem aprender a partir de exemplos e generalizar o conhecimento obtido, sem um mecanismo supervisionado para orientar o aprendizado. O aprendizado é não-supervisionado, ao contrário de redes neurais supervisionadas, que necessitam de um rótulo para identificar a classe do padrão de entrada que é usado para ajustar as conexões entre os neurônios. Durante as apresentações dos padrões, a rede identifica os agrupamentos existentes na entrada e cria um conjunto de categorias para representar os dados de entrada.

Inicialmente, a versão ART-2 é utilizada por permitir a aquisição de conhecimento a partir de vetores compostos por valores contínuos (Senger *et al.*, 2002). Após um estudo detalhado sobre o comportamento dessa rede para a classificação de aplicações, optou-se pela utilização do algoritmo da rede ART-2A. Essa rede apresenta o mesmo desempenho de classificação que a rede ART-2, com melhor desempenho computacional. A incorporação da rede ART-2A adiciona as seguintes vantagens ao modelo:

- Estabilidade: para a mesma aplicação paralela, o conhecimento obtido é armazenado nos pesos da rede neural de maneira confiável, sem perda de conhecimento em possíveis atualizações;

- Plasticidade: além de não perder o conhecimento obtido, tal conhecimento pode ser estendido para adequar-se às possíveis modificações que a aplicação paralela apresente em seu comportamento;

- Desempenho: a rápida convergência do algoritmo de aprendizado da rede ART-2A, já observada previamente na literatura (Carpenter *et al.*, 1991a), permite que o modelo de

aquisição de conhecimento responda rapidamente na sua classificação, com baixa intrusão no desempenho do sistema;

- Flexibilidade: através do controle dos parâmetros do modelo e da rede ART-2A, pode-se adequar o modelo de aquisição de conhecimento de acordo com as necessidades dos algoritmos de escalonamento.

A flexibilidade da rede ART-2A permite que o usuário estabeleça qual será a sensibilidade da rede em relação a mudança de dados. Tal sensibilidade reflete-se na quantidade de categorias obtidas para um conjunto de treinamento S. O parâmetro vigilância, ρ, permite o controle da similaridade dos padrões que são agrupados em uma determinada categoria. Valores de ρ próximos a 1 fazem com que a rede ART-2A seja mais sensível às diferenças dos padrões, enquanto que valores próximos a 0 fazem com que a tolerância da rede às diferenças aumente. Uma característica importante da rede ART-2A é que a rede pode aprender sem uma etapa inicial de treinamento. Essa etapa é muitas vezes necessária em outras arquiteturas de aprendizado e torna-se custosa computacionalmente, além de inviável em determinadas situações.

Ao fim do aprendizado, a rede ART-2A retorna um conjunto de neurônios comprometidos com os padrões de entrada, representados em sua camada de saída F_2. Cada neurônio comprometido da camada F_2 representa uma categoria identificada a partir dos dados de entrada. A interpretação desses neurônios e seus pesos permite identificar os atributos principais do padrão de entrada para cada categoria criada na camada F_2. Essa interpretação é realizada através de um algoritmo de rotulação, definido dentro do escopo da família de redes ART-2A e do problema de classificação do comportamento de aplicações na utilização de recursos.

5.3.1 Rotulação dos agrupamentos

A utilização da rede neural ART-2A permite a identificação automática de um conjunto de categorias nos traços de execução dos processos em aplicações paralelas. Cada categoria representa um estado de utilização de recursos visitado pelo processo, e o conjunto de categorias representa o comportamento do processo na utilização de recursos, o que permite a classificação das aplicações paralelas.

Para que a classificação obtida com a rede ART-2A seja interpretada corretamente, um rótulo é associado para cada neurônio da camada F_2 comprometido com um conjunto de padrões de entrada. O algoritmo de rotulação é construído baseado na observação de que os vetores de peso da rede ART-2A reconstróem os padrões de entrada que são apresentados à rede neural. Tais pesos são também chamados de protótipos, pois se interpretados geometricamente, definem a direção do agrupamento dos dados para cada categoria obtida.

Durante a fase de aprendizagem, a rede ART-2A realiza a adaptação dos vetores de pesos, combinando os valores prévios dos pesos com o padrão de entrada que está sendo aprendido.

Nesses pesos, é aplicada a normalização euclidiana, que faz com que o tamanho do vetor seja reduzido à unidade, sendo apenas preservado o ângulo formado entre os protótipos. Assim, para cada categoria criada pela rede ART-2A para classificar os padrões de entrada, pode-se observar a influência que cada atributo teve para a categoria, observando o seu componente direto no vetor de protótipos.

Uma matriz de significância SM é empregada para realizar a rotulação. Essa matriz é composta por um conjunto de valores de significância SV_{ij}, que são obtidos diretamente a partir dos protótipos da rede ART-2A. Nessa matriz, o número de colunas é igual ao número de neurônios da camada F_2, que representam os agrupamentos, e o número de linhas é igual ao tamanho do vetor de entrada (n). Uma matriz de significância $SM = (SV_{ij})^{7\times4}$, por exemplo, descreve a matriz obtida a partir de uma rede neural que criou, até o instante de tempo em questão, 4 categorias para representar uma aplicação descrita por um conjunto de padrões de tamanho $n = 7$.

O algoritmo de rotulação é ilustrado pelo algoritmo 5.1. A fim de detectar os atributos que são mais importantes para descrever a categoria, os valores de significância dos atributos são normalizados em relação a soma total dos valores de significância da categoria, ou seja, a soma dos elementos da coluna. Após tal normalização, os valores da coluna são ordenados de modo decrescente e é calculada a freqüência acumulada dos valores de significância.

Algoritmo 5.1 Rotulação das categorias obtidas pela rede ART-2A

1: defina um valor para o limiar χ (e.g. $\chi = 51\%$) e
   para n (tamanho do vetor de entrada)
2: crie a matriz de significância, uma coluna para cada categoria criada pela rede ART-2A
3: **para** cada coluna da matriz de significância **faça**
4: some os valores de significância de cada atributo
5: normalize os valores de significância de acordo com a soma
6: calcule a distribuição de freqüência acumulada
7: ordene decrescentemente os atributos da coluna
8: $soma := 0$
9: $i := 1$
10: $C := \emptyset$
11: **enquanto** $(soma \leq \chi)$ **e** $(i \leq n)$ **faça**
12: adicione o atributo x_i ao conjunto C
13: adicione a freqüência acumulada do atributo x_i à variável $soma$
14: $i := i + 1$
15: **fim enquanto**
16: rotule a categoria de acordo com os atributos contidos no conjunto C
17: **fim para**

Para a rotulação da categoria, escolhe-se uma quantidade de atributos até que a soma da freqüência acumulada não exceda a um certo limiar. Um exemplo de valor de limiar empregado no modelo é igual a 51%. À medida que aumenta-se o valor de limiar, maior é a quantidade de atributos que são considerados como significantes para representar a categoria (Ultsch, 1993).

Como resultado final do algoritmo, obtém-se para cada categoria um conjunto C composto dos atributos mais significantes. Tais atributos são empregados para rotular a respectiva categoria.

5.3.2 Implementação do algoritmo ART-2A

O algoritmo de aprendizado da rede ART-2A é implementado [2] através de um conjunto de classes C++. As seguintes classes principais são definidas:

- `TArt2`: classe construtora da rede ART-2A. Essa classe implementa o aprendizado da rede e os métodos `Save()` e `Restore()`, que permitem que o conhecimento obtido pela rede possa ser respectivamente armazenado e restaurado. O conjunto de protótipos obtidos durante o treinamento são empregados pelo algoritmo de rotulação;

- `TInput`: essa classe implementa facilidades para a leitura dos padrões de entrada que são apresentados a rede;

- `TCluster`: essa classe implementa a representação de uma categoria. Essa classe utiliza uma lista de objetos `TNode`, que implementam os padrões associados a uma determinada categoria. A classe `TCluster` tem como métodos principais o método `GetCentroid()`, que retorna o elemento centróide [3] da categoria e o método `GetCounter()`, que retorna o número de padrões que são armazenados em uma categoria. O número de objetos `TCluster` é igual ao número de neurônios comprometidos da camada F_2 da rede ART-2A.

- `TOutPut`: essa classe utiliza uma lista encadeada de objetos do tipo `TCluster`, visando representar a saída obtida pela classificação. Nessa classe, também estão implementados mecanismos para obter medidas de avaliação de desempenho da classificação realizada pela rede.

A implementação é validada através da utilização de dois conjuntos de dados: `Spanning Tree` e `Iris`. O conjunto `Spanning tree` é composto de dados não-rotulados que podem ser mostrados graficamente em uma estrutura similar a uma árvore (Blake & Merz, 1998). Esse conjunto serve para analisar o comportamento do algoritmo de aprendizado à medida que o valor da vigilância ρ varia (Fausett, 1994). O conjunto de dados `Iris` é composto de 150 instâncias rotuladas através de três classes. Os rótulos permitem observar o erro de classificação da rede ART-2A, comparando as categorias criadas pela rede com o rótulo dos elementos obtidos diretamente do conjunto de dados.

[2] Essa implementação do algoritmo de treinamento da rede ART-2A é distribuído livremente através da licença GPL no sítio da Internet `http://www.ljsenger.net`

[3] Centróide corresponde a um ponto no espaço cujas coordenadas são médias das coordenadas dos pontos armazenados em uma categoria

Valores maiores de ρ fazem com que a rede neural gere um maior número de categorias. Como o objetivo da rede neural é categorizar padrões semelhantes em uma mesma categoria, torna-se indesejável casos extremos onde há apenas um padrão por categoria. Surge, então, a questão de como definir o melhor valor para ρ. Para definir o valor ideal de ρ, são propostas duas medidas de desempenho, que relacionam as distâncias entre os padrões contidos em cada categoria e a distância entre centróides de diferentes categorias (He *et al.*, 2003a; Gokcay & Principe, 2000).

A primeira medida proposta é a distância *intra* definida pela Equação 5.3. Essa medida, baseada em (He *et al.*, 2003a), permite quantificar a distância média entre os padrões contidos em uma categoria i e seu centróide c_i. Essa medida calcula a diferença média dos ângulos formados entre os vetores x_{ij}, que representam os padrões contidos em uma categoria c_i, e o centróide dessa categoria, conforme a lei dos cossenos (Pappas, 1989):

$$Intra = \frac{\sum_{i=1}^{m} \sum_{j=1}^{n} ||x_{i,j}|| * ||c_i||}{m} \tag{5.3}$$

onde m é o número de categorias formadas pela rede.

A segunda medida, chamada de *inter*, é baseada no trabalho de Gokcay & Principe (2000). Essa medida define a distância entre os centróides de cada categoria criada pela rede neural e é descrita pela Equação 5.4. Essa equação calcula a distância média entre centróides das categorias geradas pela rede neural e, assim como a Equação 5.3, também utiliza as leis dos cossenos para calcular a diferença entre os ângulos dos vetores. Empregando essas medidas, os resultados podem ser plotados sobre a mesma escala, permitindo definir um ponto no plano R^2 onde ambas as funções se igualam, isto é, onde a distância *intra* é minimizada enquanto a distância *inter* é maximizada. Esse ponto de encontro entre as duas funções define o ρ ideal e, conseqüentemente, o número de categorias e padrões por categorias gerados pela rede neural.

$$Inter = \frac{\sum_{i=1}^{m} \sum_{j=1}^{m} ||c_i|| * ||c_j||}{m} \tag{5.4}$$

As Figuras 5.2 e 5.3 apresentam os resultados das equações de distância *intra* e *inter* para os conjuntos de dados utilizados, considerando $\rho \in [0, 1]$. Através desses resultados, obtém-se o valor ideal de $\rho = 0,99$ para o conjunto de dados `Iris` e o valor de $\rho = 0,95$ para o conjunto de dados `Spanning tree`. Os valores de ρ obtidos para esses conjuntos correspondem respectivamente àqueles obtidos por Vicentini (2002) e Fausett (1994). O erro de classificação para o conjunto de dados `Iris` é igual a 92%, valor similar ao obtido por Vicentini (2002). Os resultados desses experimentos permitem validar as medidas de distância, que são utilizadas para orientar a escolha do valor da vigilância. Essas medidas são utilizadas para definir melhores valores de vigilância que devem ser empregados nos experimentos apresentados a seguir.

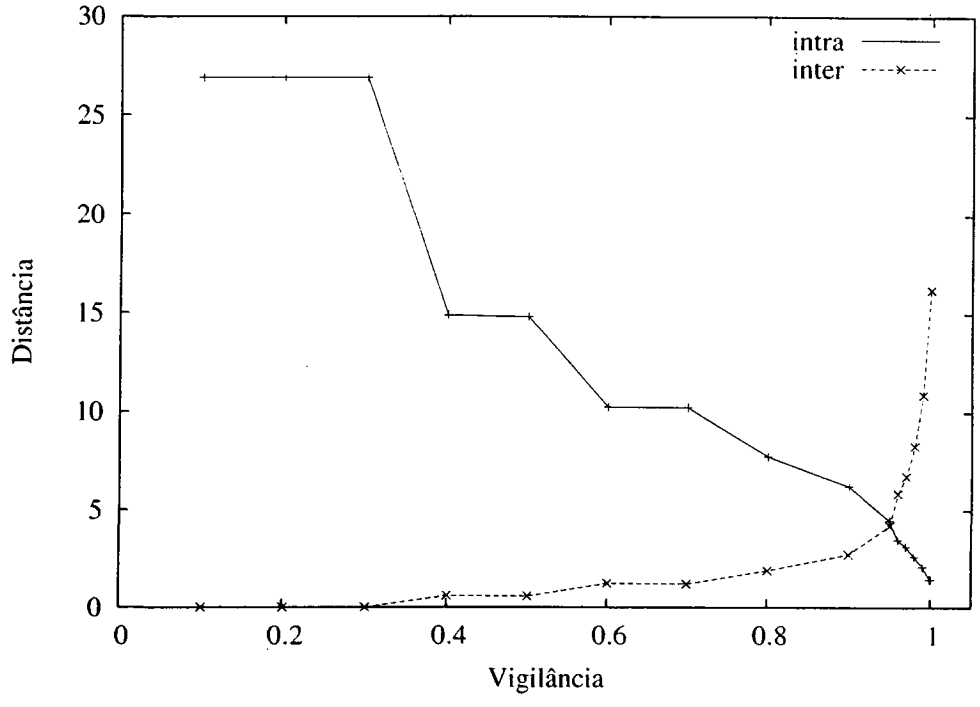


Figura 5.2: Variação das distâncias *intra* e *inter* para o conjunto de dados `Spanning Tree`

5.3.3 Avaliação da aquisição de conhecimento

O esquema de classificação é avaliado utilizando informações coletadas sobre execuções das aplicações A, CV, PSTSWM e SP (vide Capítulo 4). Nos experimentos, é utilizado um vetor de entrada de tamanho $n = 7$ (Tabela 5.12). O vetor de entrada contém informações relacionadas com a execução da aplicação: tempo de UCP em modo usuário, operações de leitura e escrita em arquivos e operações de envio e recebimento pela rede de comunicação. Tais informações são obtidas através da monitoração de processos, conforme descrito na Seção 5.2. Durante a execução dos experimentos, variou-se o valor do intervalo de amostragem T_o entre 200, 500 e 1000 milissegundos.

Tabela 5.12: Descrição do vetor de entrada

Atributo	Descrição
1	Tempo de UCP em modo usuário
2	Operação de E/S (Kbytes lidos)
3	Operação de E/S (Kbytes escritos)
4	Comunicação (Kbytes lidos em *sockets* TCP)
5	Comunicação (Kbytes escritos em *sockets* TCP)
6	Comunicação (Kbytes lidos em *sockets* UDP)
7	Comunicação (Kbytes escritos em *sockets* UDP)

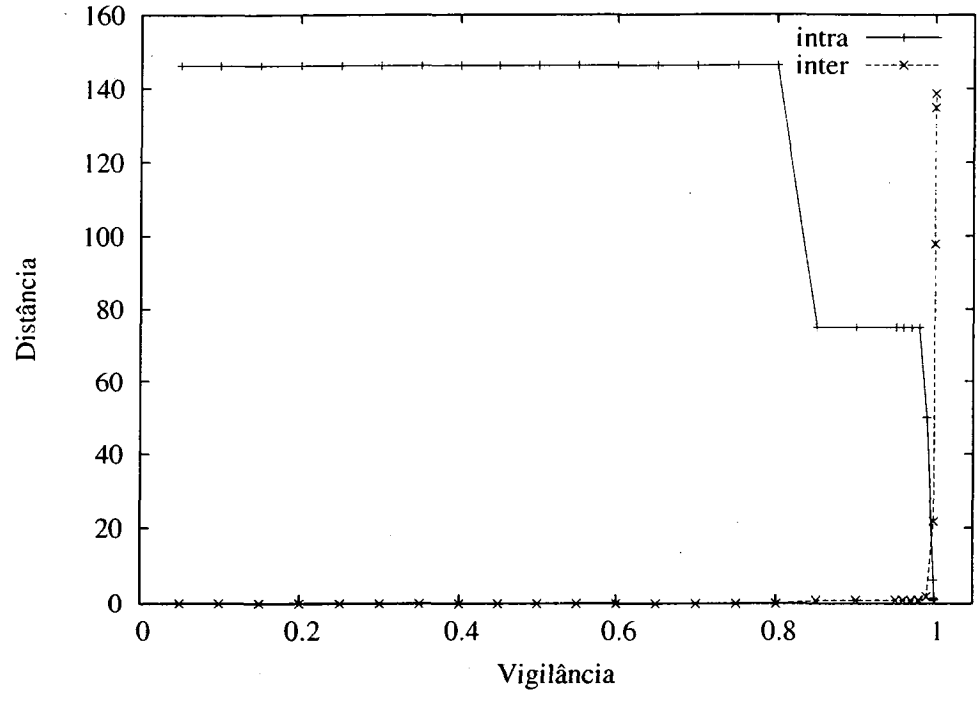


Figura 5.3: Variação das distâncias *intra* e *inter* para o conjunto de dados `Iris`

Para o atributo 1, a seguinte função de normalização é empregada ($h_1(x)$):

$$h_1(x) = \frac{x}{T_o} \tag{5.5}$$

onde x é o valor obtido para o atributo e T_o o intervalo de amostragem empregado. Tal normalização delimita o valor do atributo no intervalo $0 \leq x \leq 1$. Para os demais atributos, a função $h_2(x)$ é utilizada:

$$h_2(x) = \begin{cases} 0 & \text{se } x = 0 \\ log(x) & \text{caso contrário} \end{cases} \tag{5.6}$$

sendo x o valor a ser normalizado e $log(x)$ o logaritmo natural de x. Essas operações de normalização são necessárias para reduzir proporcionalmente a magnitude dos componentes do vetor de entrada S, que podem depreciar o desempenho de classificação do modelo (Nicklayev, 1996).

Os experimentos de classificação das aplicações são realizados da maneira descrita a seguir. As características da execução da aplicação paralela, coletadas ao longo de sua execução, compõem um conjunto de treinamento T, que é apresentado ao algoritmo de treinamento da rede ART-2A. Esse algoritmo é executado com um número de ciclos de treinamento é igual a 10 e a rede ART-2A é utilizada em seu modo de classificação. Os valores dos demais parâmetros desse algoritmo são definidos de acordo com a Tabela 3.5.

A monitoração e coleta das características de execução das aplicações são realizadas com o software `monitor` (vide Seção 5.2.5). Para todas as aplicações consideradas, as medidas de avaliação descritas pelas equações 5.3 e 5.4, são empregadas. Assim, pode-se obter valores de ρ

ideais para cada aplicação paralela considerada. Através dessas medidas, obtém-se o valor de ρ no intervalo entre 0,90 e 0.95.

Aplicação A

Os resultados obtidos com a classificação da aplicação A e sua variante A' estão ilustrados nas Tabelas 5.13, 5.14 e 5.15. Para essa aplicação, obtém-se um número de categorias igual a 4, que corresponde ao número de fases de utilização de recursos sinteticamente definidos para essa aplicação.

Através da Tabela 5.13, pode-se observar os vetores de pesos obtidos após o treinamento da rede. Os valores dos pesos representam o conhecimento extraído a partir dos dados e definem protótipos sobre o comportamento da aplicação paralela. Os valores de significância, calculados utilizando tais protótipos como entrada para o algoritmo de rotulação, são também ilustrados. A partir dos valores de significância, o algoritmo de rotulação cria uma lista ordenada (Tabela 5.15) que é empregada para selecionar os atributos mais relevantes para cada categoria. Os atributos mais significantes para esse aplicação estão ilustrados na Tabela 5.14, assim como a quantidade de vezes (freqüência) em que a aplicação passou em cada estado em sua execução. Assim, pode-se observar que a aplicação A visitou em aproximadamente 86% do traço de sua execução a categoria 2, que representa as operações orientadas ao processamento. As operações de acesso em arquivos são contabilizadas pela categoria 3 e representam 11% do tempo de execução da aplicação. O acesso a rede de comunicação é representado pelas categorias 1 e 4.

A execução da aplicação A' permite observar a plasticidade da rede ART-2A . Os dados obtidos com a execução dessa aplicação são apresentados à rede ART-2A já treinada previamente com a execução da aplicação A. Isso permite que a rede classifique a aplicação com o conhecimento obtido previamente, e, se necessário, adapte-se para refletir a situação atual do comportamento da aplicação. Com esse experimento, pode-se observar a estabilidade e plasticidade da rede ART-2A, pois as categorias previamente criadas para a aplicação A são mantidas e atualizadas. Nesse caso, a freqüência interna para cada categoria é modificada para refletir o novo comportamento da aplicação. Um mudança na freqüência relativa do categoria 3 ocorre (aproximadamente 74%), devido à característica de redução da fase orientada ao processamento.

Aplicação CV

A Tabela 5.16 apresenta os resultados obtidos para a aplicação CV. Nesse caso, a matrizes de significância e os protótipos não são mostrados devido à observação de que apenas uma categoria é identificada pela rede. Tal aplicação é orientada ao processamento e realiza pouca troca de informações entre as tarefas. Essa troca ocorre apenas no início e no término da execução. Uma categoria é gerada pela rede ART-2A para classificar a aplicação como orientada

Tabela 5.13: Protótipos para as aplicações A e A'

	Categorias				Matriz de significância (%)			
Atributo	1	2	3	4				
1	0.00	1.00	0.05	0,00	0.00	100.00	4.76	0.00
2	0.00	0,00	0,00	0,00	0,00	0,00	0,00	0.00
3	0.00	0.00	0,95	0,00	0,00	0,00	95,23	0.00
4	1.00	0.00	0.00	0,00	100,00	0,00	0,00	0.00
5	0,00	0,00	0,00	1,00	0,00	0,00	0,00	100,00
6	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
7	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00

Tabela 5.14: Criação da lista ordenada de relevância dos atributos para cada categoria das aplicações **A e A'**

1		2		3		4	
Atributo	Freq. (%)	Atributo	Freq. (%)	Atributo	Freq. (%)	Atributo	Freq. (%)
4	100,00	1	100,00	3	95,23	5	100,00
1	100,00	2	100,00	1	100,00	1	100,00
2	100,00	3	100,00	2	100,00	2	100,00
3	100,00	4	100,00	4	100,00	3	100,00
5	100,00	5	100,00	5	100,00	4	100,00
6	100,00	6	100,00	6	100,00	6	100,00
7	100,00	7	100,00	7	100,00	7	100,00

ao processamento. Nessa aplicação paralela, a troca de informação pela rede de comunicação, como é muito pequena, é filtrada pelo algoritmo da rede ART-2A.

Aplicação PSTSWM

A aplicação PSTSWM é mais pesada computacionalmente que as aplicações A e CV. Os resultados obtidos para essa aplicação são descritos nas Tabelas 5.17, 5.18 e 5.19. Através dessas tabelas pode-se observar que grande parte da execução da aplicação (aproximada-

Tabela 5.15: Atributos selecionados para rotular as categorias obtidas para as aplicações A e A'

Categoria	Atributo	Descrição	Freqüência A (%)	Freqüência A (%)
1	4	orientada à comunicação	01.76	11,11
2	1	orientada ao processamento	85,58	05,55
3	3	orientada à E/S	11,47	74,07
4	5	orientada à comunicação	01,17	09,25

Tabela 5.16: Classificação da aplicação CV

Categoria	Freqüência	(%)
1	811	100

mente 56%) é classificada como orientada a comunicação (categoria 1). As demais fases da aplicação são classificadas respectivamente como orientada a comunicação (categorias 2 e 4) e orientada à UCP (categoria 3).

Tabela 5.17: Protótipos ART-2A para a aplicação PSTSWM

	Categorias				**Matriz de significância (%)**			
Atributo	1	2	3	4				
1	0,00	0,12	1,00	0,09	0,30	10,94	100,00	6,26
2	0,00	0,00	0,00	0,00	0.00	0,00	0,00	0,00
3	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
4	1,00	0,00	0,00	0,70	99,70	0,00	0,00	46,87
5	0,00	0,99	0,00	0,70	0,00	89,06	0,00	46,87
6	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
7	0,00	0.00	0,00	0,00	0.00	0,00	0,00	0.00

Tabela 5.18: Criação da lista ordenada de relevância dos atributos para cada categoria na aplicação PSTSWM

1		2		3		4	
Atributo	Freq. (%)	Atributo	Freq. (%)	Atributo	Freq. (%)	Atributo	Freq. (%)
4	99,70	5	89,06	1	100,00	4	46,87
1	100,00	1	100,00	2	100,00	5	93,74
2	100,00	2	100,00	3	100,00	1	100,00
3	100,00	3	100,00	4	100,00	2	100,00
5	100,00	4	100,00	5	100,00	3	100,00
6	100,00	6	100,00	6	100,00	6	100,00
7	100,00	7	100,00	7	100,00	7	100,00

Aplicação SP

A aplicação SP é executada considerando três cargas de trabalho: W, A e B. A diferença entre essas cargas de trabalho é o tamanho do problema considerado, que é igual a 36^3 para o problema W, 64^3 para o problema A e 102^3 para o problema B. As Tabelas 5.20, 5.22 e 5.24 ilustram os resultados obtidos com a classificação. Inicialmente, devido ao tamanho reduzido do problema, apenas uma categoria é criada pelo esquema de classificação (Tabela 5.20).

Tabela 5.19: Atributos selecionados para rotular as categorias obtidas para a aplicação PSTSWM

Categoria	Atributos	Descrição	Freqüência (%)
1	4	orientada à comunicação	56,07
2	5	orientada à comunicação	16,58
3	1	orientada à processamento	14,79
4	4;5	orientada à comunicação	12,56

Empregando os tamanhos de problema A e B, um número de categorias igual a 4 é observado (Tabelas 5.22 e 5.24). Esse aumento no número de categorias indica que o comportamento da aplicação é diferente em relação ao tamanho de problema W, devido ao fato de a aplicação visitar com mais freqüência o estado orientado ao processamento.

Tabela 5.20: Protótipos ART-2A para a aplicação SP com tamanho de problema W

	Categoria(k)	Matriz de significância (%)
Atributo	1	
1	0,0470	3.24
2	0,0000	0.00
3	0,0000	0.00
4	0,7801	53.72
5	0,6250	43.04
6	0,0000	0.00
7	0,0000	0.00

Tabela 5.21: Atributos selecionados para rotular as categorias obtidas para a aplicação SP (carga W)

Categoria	Atributos	Descrição	Freqüência (%)
1	4,5	orientada à comunicação	100,00

5.3.4 Diagrama de transição de estados

Além da classificação dos estados visitados pela execução das aplicações, pode-se obter o diagrama de transição entre esses estados. Nesse diagrama, o valor p_{ij} descreve a probabilidade de transição de um estado i a um estado j. Esse valor é obtido através do número de visitas em cada estado observado na execução da aplicação, empregando a seguinte equação, de acordo com Devarakonda & Iyer (1989):

$$p_{ij} = \frac{\text{número de transições observadas a partir de um estado } i \text{ ao estado } j}{\text{número de transições a partir do estado } i} \tag{5.7}$$

Tabela 5.22: Protótipos ART-2A para a aplicação SP com tamanho de problema A

	Categorias				Matriz de significância (%)			
Atributo	1	2	3	4				
1	0,047	0,066	1,000	0,117	3,24	6,20	100,000	10,541
2	0,000	0,000	0,000	0,000	0,00	0,00	0,000	0,000
3	0,000	0,000	0,000	0,000	0,00	0,00	0,000	0,000
4	0,780	0,998	0,000	0,000	53,72	93,80	0,000	0,000
5	0,624	0,000	0,000	0,993	43,04	0,00	0,000	89,459
6	0,000	0,000	0,000	0,000	0,00	0,00	0,000	0,000
7	0,000	0,000	0,000	0,000	0,00	0,00	0,000	0,000

Tabela 5.23: Atributos selecionados para rotular as categorias obtidas para a aplicação SP (tamanho de problema A)

Categoria	Atributos	Descrição	Freqüência (%)
1	4;5	orientada à comunicação	87,33
2	4	orientada à comunicação	4,22
3	1	orientada à comunicação	6,89
4	5	orientada à comunicação	1,56

A seguir, são descritos os diagramas de transição de estados para a aplicação SP. Tal aplicação é empregada como exemplo por permitir que seja modificado o tamanho do problema a ser resolvido e, conseqüentemente, a relação processamento/comunicação. Os diagramas de estado para a aplicação SP são ilustradas pelas Figuras 5.4, 5.5 e 5.6. Cada estado é representado por um círculo e as transições entre os estados são indicadas através de setas direcionadas. Os círculos representam os neurônios (categorias) comprometidos da camada F_2 da rede neural ART-2A e são rotulados internamente de acordo com os atributos selecionados pelo algoritmo

Tabela 5.24: Protótipos ART-2A para a aplicação SP com tamanho de problema B

	Categorias(k)				Matriz de significância (%)			
Atributo	1	2	3	4				
1	0,047	0,066	1,000	0,117	3,24	6,20	100,000	10,541
2	0,000	0,000	0,000	0,000	0,00	0,00	0,000	0,000
3	0,000	0,000	0,000	0,000	0,00	0,00	0,000	0,000
4	0,780	0,998	0,000	0,000	53,72	93,80	0,000	0,000
5	0,624	0,000	0,000	0,993	43,04	0,00	0,000	89,459
6	0,000	0,000	0,000	0,000	0,00	0,00	0,000	0,000
7	0,000	0,000	0,000	0,000	0,00	0,00	0,000	0,000

Tabela 5.25: Atributos selecionados para rotular as categorias obtidas para a aplicação SP (tamanho de problema B)

Categoria	Atributos	Descrição	Freqüência (%)
1	4, 5	orientada à comunicação	55, 87
2	4	orientada à comunicação	2, 31
3	1	orientada ao processamento	36, 75
4	5	orientada à comunicação	5, 07

de rotulação. As setas direcionados possuem um valor percentual que indica a probabilidade de transição entre os estados.

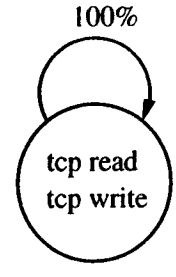


Figura 5.4: Diagrama de estados para a aplicação SP (tamanho de problema W)

É interessante observar o comportamento do modelo à medida que o tamanho do problema da aplicação SP varia. Inicialmente (Figura 5.4), devido ao problema ser mais reduzido, o modelo captura apenas um estado de execução para a aplicação e o rotula como orientado à comunicação. Apesar da aplicação realizar processamento nesse estado, o valor de limiar do algoritmo de rotulação seleciona apenas os atributos 4 e 5 para rotular a categoria. Com o tamanho de problema A (Figura 5.5), já pode ser observada a separação dos estados, pois o maior tamanho de problema indica um maior tempo de processamento e a criação de novos estados para acomodar tal modificação no comportamento da aplicação.Como a transferência de dados torna-se mais custosa, estados específicos para os atributos 4 e 5, que indicam operações em rede, são criados. Empregando o tamanho de problema B, observa-se uma atualização na probabilidade de transição entre os estados (Figura 5.6). Um mudança importante que pode ser notada é a probabilidade de transição interna para o estado de utilização de UCP, aproximadamente igual a 59% (Figura 5.6), indicando um maior custo de operações matemáticas para esse tamanho de problema.

5.3.5 Influência da carga e heterogeneidade do sistema

A aplicação A é executada em elementos de processamento carregados para observar a estabilidade do modelo na presença de carga externa induzida. A carga externa é gerada através de um programa chamado de `parasita`, que é composto por um conjunto de operações orientadas ao processamento. Essas operações elevam a utilização da UCP a 100% em

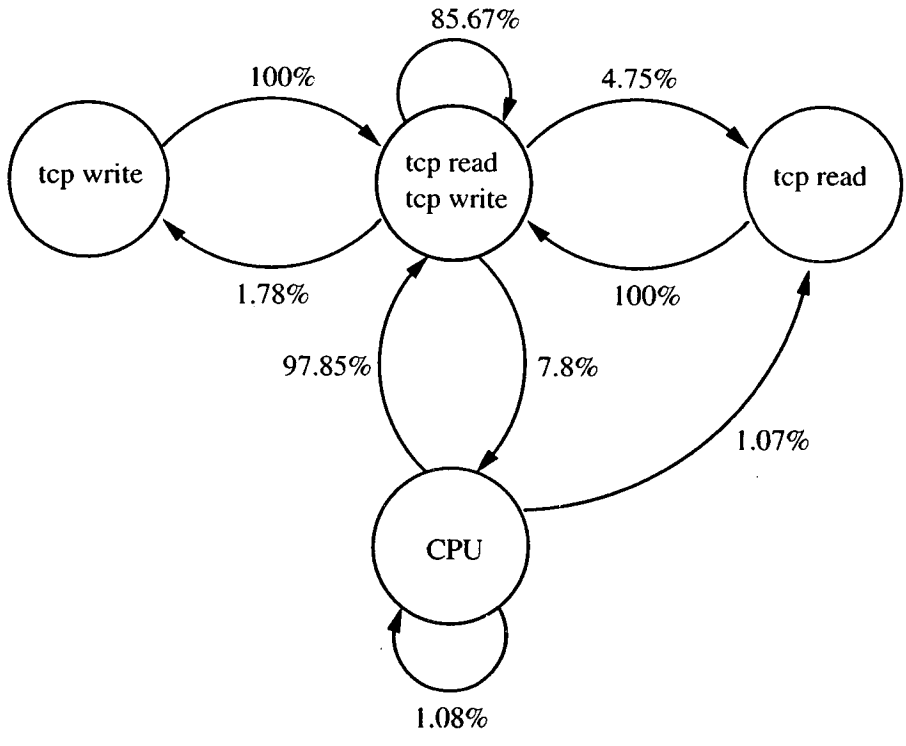


Figura 5.5: Diagrama de estados para a aplicação SP (tamanho de problema *A*)

um pequeno intervalo de tempo. As Tabelas 5.26 e 5.27 ilustram os resultados empregando respectivamente 1 e 3 programas parasitas. Observa-se que as mudanças na carga de trabalho e no tamanho do intervalo de amostragem não depreciaram a classificação do modelo para a aplicação considerada. A classificação obtida é semelhante ao resultado sem carga externa (Tabela 5.13), com a diferença existindo apenas na freqüência interna de cada grupo.

A Tabela 5.28 apresenta os resultados da execução da aplicação A, sendo executada em quatro elementos de processamento heterogêneos. Tais computadores possuem a mesma arquitetura, porém apresentam diferenças em sua velocidade de relógio e capacidade de memória (vide Capítulo 4, Tabela 4.2).

Tabela 5.26: Influência da carga de trabalho externa para a aplicação A (1 *parasita* e $T_o = 200ms$)

Categoria	Freqüência (%)
1	01,75
2	84,47
3	12,77
4	01,01

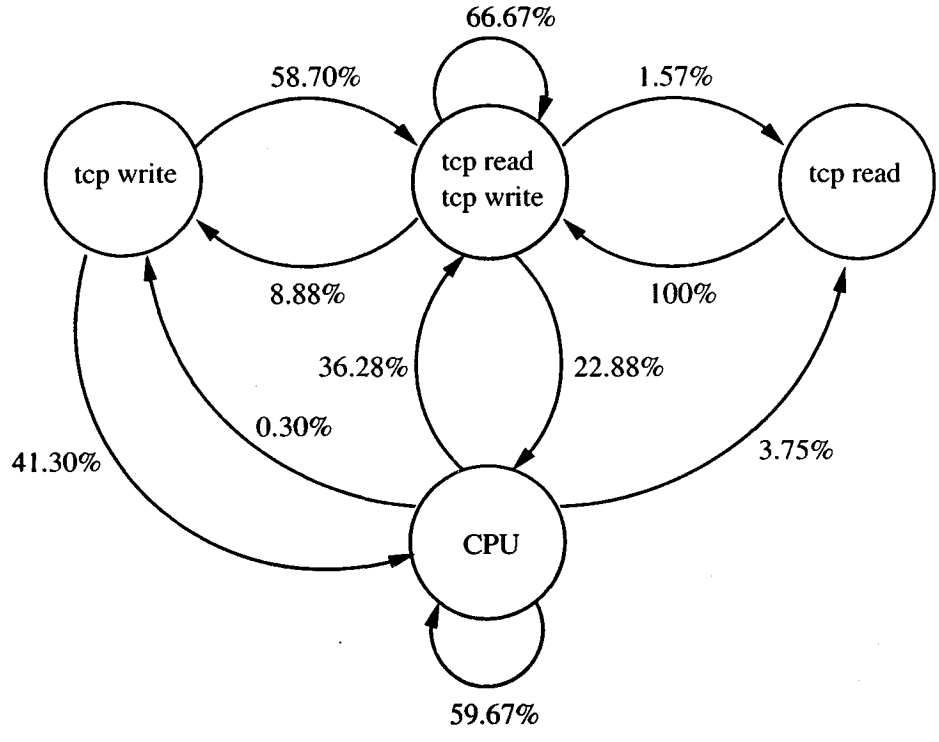


Figura 5.6: Diagrama de estados para a aplicação SP (tamanho de problema B)

Tabela 5.27: Influência da Carga de trabalho externa para a aplicação A (3 *parasitas* e $T_o = 500ms$)

Categoria	Freqüência (%)
1	0,90
2	85,92
3	12,81
4	0,36

5.4 Classificação de processos Linux

A implementação do algoritmo de aprendizado é realizada de forma que o código de aquisição de conhecimento e de classificação é inserido no núcleo do sistema operacional Linux. Isso permite que o desempenho computacional do algoritmo e sua intrusão no desempenho do sistema sejam avaliados e, através dessa implementação, pode-se verificar o comportamento do modelo para a aquisição de conhecimento em tempo de execução dos processos (Senger *et al.*, 2005).

Duas estruturas de dados principais são criadas e armazenadas na tabela de processos. A primeira estrutura, chamada de `accounting_struct`, tem os mesmos contadores definidos previamente para a aquisição de informações sobre processos (vide Seção 5.2). Tais contadores englobam, por exemplo, informações como tempo de UCP empregado e quantidade de bytes

Tabela 5.28: Aplicação A em quatro elementos de processamento

EPs	1	2	3	4
Categorias	Freqüência(%)			
1	01,76	01,96	01,54	01,65
2	85,62	84,68	84,21	85,62
3	12,74	12,02	11,94	11,66
4	01,86	01,32	01,28	01,80

transferidos em disco e na rede. Essa estrutura é criada visando melhorar a organização do código, encapsulando os contadores relacionados ao processo referenciado pela tabela.

A segunda estrutura de dados, chamada de `TArt2`, é incluída na tabela de processos para armazenar as variáveis que representam o estado atual da rede neural ART-2A. Essas variáveis englobam, por exemplo, os pesos que interligam as camadas F_1 e F_2 e os valores de freqüência de visitas em cada categoria criada pela rede neural. A estrutura de dados `TArt2` é armazenada dinamicamente na tabela de processos, i.e., a tabela de processos armazena inicialmente apenas um ponteiro para essa estrutura. A memória necessária para o armazenamento das variáveis dessa estrutura é somente alocada se for requisitada a classificação do processo. Isso permite que memória principal não seja desperdiçada em processos do sistema ou para os quais a classificação não foi requisitada. Com essa implementação, busca-se reduzir o impacto da classificação no desempenho do sistema.

A estrutura de dados `TArt2` é utilizada pelo algoritmo de classificação. Esse algoritmo, previamente implementado na linguagem C++, é modificado e novamente implementado utilizando a linguagem C, uma vez que a parte independente do código do núcleo do sistema operacional Linux é implementada com a linguagem C. Busca-se, assim, uma uniformização da linguagem de programação adotada.

A Tabela 5.29 descreve os arquivos do núcleo do sistema Linux modificados para acomodar a implementação do algoritmo de aprendizado.

Tabela 5.29: Arquivos fonte do núcleo modificados para a implementação da rede ART-2A

Arquivo fonte	**Descrição**
`include/linux/tart.h`	definição das rotinas da rede ART-2A e de escrita no `procfs`
`fs/proc/array.c`	implementação das rotinas da rede e escrita no `procfs`
`kernel/exit.c`	término do processo

No arquivo fonte `/include/linux/tart.h`, são definidos a estrutura `struct TArt2` e os protótipos das funções que manipulam tal estrutura. O algoritmo de aprendizado é im-

plementado no arquivo fonte `fs/proc/array.c`. Nesse arquivo, as seguintes rotinas são definidas:

- `inline struct TArt2 *CreateArt2(int m, int n, float rho)`: essa rotina é responsável por criar uma nova estrutura `TArt2`, empregando as variáveis `m` e `n`, que definem respectivamente a quantidade de neurônios na camada F_1 e a quantidade máxima de categorias que podem ser criados na camada F_2. O parâmetro de entrada `rho` define o valor do parâmetro de vigilância ρ;

- `inline struct TArt2 *DestroyArt2(struct TArt2 *art)`: essa rotina implementa o código responsável por eliminar a estrutura `TArt2` referenciada pelo ponteiro `struct TArt2 *art`. Tal rotina é chamada quando um processo é finalizado e existe uma estrutura `TArt2` referenciada para esse processo;

- `inline void SetS(float *vector, struct TArt2 *art)`: responsável por apresentar um conjunto de valores de entrada para o algoritmo de aprendizado;

- `inline void UpdateF1(struct TArt2 *art)`: essa rotina realiza as operações de normalização e supressão de ruído ART-2A;

- `inline void SendSignaltoF2(struct TArt2 *art)`: calcula a ativação dos neurônios da camada F_1;

- `inline float f(float value, struct TArt2 *art)`: filtragem de ruído para componentes do vetor de entrada;

- `inline void ResetF2Activations(struct TArt2 *art)`: faz com que as ativações da camada F_2 sejam marcadas com o valor igual a zero;

- `inline void Train(struct TArt2 *art)`: realiza o treinamento da rede para o padrão corrente. Essa rotina emprega as rotinas `ResetF2Activations()`, `UpdateF1()` e `SendSignaltoF2()`. Além disso, implementa o laço de controle responsável por verificar o sinal de *reset* da rede.

- `inline void Resonance(struct TArt2 *art)`: realiza o aprendizado do padrão corrente. Esse rotina efetua a atualização dos pesos da rede neural para que o padrão seja aprendido.

- `inline void N(float *vector, struct TArt2 *art)`: realiza a normalização euclidiana do vetor `vector`;

- `inline void ChooseWinner(struct TArt2 *art)`: encontra o neurônio com maior ativação da camada F_2. O neurônio com maior ativação (vencedor) é armazenado na variável `J`;

- `inline char *Save(struct TArt2 *art, char *buffer)`: copia para a variável `buffer` os pesos da rede.
- `inline float ComputeNorm(float *vector, struct TArt2 *art)`: calcula a norma do vetor `vector`;
- `inline int IsCommitted(int f2_neuron, struct TArt2 *art)`: retorna verdadeiro caso o neurônio `f2_neuron` da camada F_2 é comprometido com algum padrão;
- `inline void SetCommited(int f2_neuron, int value, struct TArt2 *art)`: marca o neurônio `f2_neuron` como comprometido;
- `inline char* SaveMatrix(char *buffer, struct task_struct *p)`: copia as informações sobre a rede neural para o arquivo do `procfs` correspondente;
- `inline char *label(float *wt, int size, float threshold, char *buffer)`: implementa o algoritmo de rotulação.

Dentro da estrutura `TArt2` é criado um espaço de armazenamento temporário de tamanho finito (*buffer*), que representa uma memória recente das operações realizadas pelo processo. Esse `buffer` é atualizado sempre que é requisitada a classificação do processo e é definido como uma estrutura circular que garante que os padrões de entrada sejam apresentados mais de uma vez para rede, permitindo que a rede possa estabilizar o seu aprendizado em resposta aos padrões de entrada. O padrão de comportamento mais recente é inserido na última posição do *buffer* para garantir que o último neurônio vencedor da camada F_2 represente o estado atual do processo.

A requisição de classificação de processo e a disponibilização das informações obtidas após a classificação são feitas através do arquivo `/proc/pid/status`, que é localizado no `procfs`. Por ser criado no `procfs`, esse arquivo corresponde a uma interface de comunicação ou ponto de acesso entre o núcleo do sistema operacional e o espaço de endereçamento em nível de usuário. Assim, define-se que um processo do sistema será classificado quando o seu arquivo de `status` for acessado por qualquer processo em nível de usuário. A cada vez que é realizado um acesso a esse arquivo, as seguintes ações são tomadas pelo código inserido no núcleo:

- os valores dos contadores da estrutura `accouting_struct` são copiados para a próxima posição disponível no *buffer* do processo monitorado e os contadores são marcados com valores iguais a zero. Quando o *buffer* é totalmente preenchido, as informações mais antigas são removidas, disponibilizando espaço para a informação recente;
- os padrões armazenados no *buffer* são apresentados ao algoritmo de aprendizado, através da rotina `SetS()` e o treinamento é realizado através da rotina `Train()`. A apresentação do *buffer* à rede neural é repetida um número pré-definido de vezes. A constante

`EPOCHS` define o número de vezes, ou ciclos, que os padrões armazenados no *buffer* são apresentados para a rede neural, e a constante `BSIZE` define o tamanho máximo do *buffer*. O número de ciclos de treinamento utilizado é igual a 3, e o tamanho máximo do *buffer* é igual a 30;

- algoritmo de rotulação, também implementado no núcleo, é chamado após o término do treinamento da rede. Tal algoritmo associa um rótulo ao neurônio vencedor, de forma que o estado atual do processo em relação a utilização de recursos possa ser identificado;

- a classificação obtida pelo algoritmo de aprendizado é disponibilizada através do arquivo `/proc/pid/status`, onde `pid` corresponde ao identificador do processo monitorado.

Uma região crítica é definida para cada processo, para que essas ações sejam executadas corretamente, protegidas de acessos concorrentes. Para isso, é definido o semáforo `acc_sema`, que controla o acesso à execução do algoritmo de aprendizado. Esse semáforo bloqueia requisições ao algoritmo de aprendizado enquanto esteja em andamento uma requisição prévia de classificação do processo.

5.4.1 Avaliação da intrusão no desempenho do sistema

Para a avaliação do impacto do modelo de classificação no desempenho do sistema, são consideradas execuções de um conjunto de benchmarks do pacote NAS (vide Capítulo 4). Inicialmente, os benchmarks são executados nos computadores EP_6 e EP_7 sem empregar o modelo de classificação. As médias do tempo de resposta dessas execuções servem como referência para observar o impacto do modelo de classificação no tempo de resposta dos processos. Após, são realizadas execuções dos benchmarks utilizando o modelo de classificação, para diferentes intervalos de amostragem T_o iguais a 250 ms, 500 ms e 1 segundo. Um intervalo de amostragem igual a 250 milissegundos, por exemplo, indica que a cada segundo do tempo de execução da aplicação serão enviados 4 amostras de seu comportamento ao algoritmo de aprendizado. Intervalos de amostragem menores permitem analisar com uma menor granulação o comportamento dos processos, mas impõem maiores atrasos no tempo de resposta dos processos.

Assim, obtém-se para cada benchmark 4 médias de tempo de resposta para cada computador empregado, que correspondem à média do tempo de execução em um sistema livre (i.e. sem a classificação), um sistema com classificação e $T_o = 250$ ms, $T_o = 500$ ms e $T_o = 1$ segundo. Os tamanhos das amostras para a composição dessas médias são iguais a 15 para o EP_6 e 30 para o EP_7. As médias são comparadas com o teste T, considerando um nível de confiança igual a 95% (vide Apêndice A, tabelas A.28, A.29,A.30,A.31,A.32 e A.33).

A medida de desempenho utilizada é o *slowdown*. Essa medida corresponde ao tempo de execução de uma aplicação em um sistema carregado T_b normalizado pelo tempo de execução

em um sistema livre T_f:

$$Slowdown = \frac{T_b}{T_f} \tag{5.8}$$

Considera-se o sistema carregado como sendo o núcleo do sistema operacional com a implementação do algoritmo de aprendizado e o sistema livre como sendo o caso em que o núcleo do sistema operacional não foi modificado. Assim, um valor de *slowdown* igual 1.05 indica que a rede ART-2A impõe um atraso igual a 5% no tempo de resposta.

5.4.2 Resultados obtidos

Os resultados da análise da intrusão da implementação da rede ART-2A no desempenho do sistema para o elemento de processamento EP_6 estão descritos na Tabela 5.30. Cada linha dessa tabela apresenta o *slowdown* observado para cada benchmark. Como previsto, o *slowdown* diminui à medida que o intervalo de amostragem T_o torna-se maior. Para intervalos de amostragem menores, mais informações são enviadas para o algoritmo de classificação, e, conseqüentemente, tem-se maior utilização do tempo de UCP, que reflete-se no valor de *slowdown* obtido.

Tabela 5.30: Valores de *slowdown* observados para os benchmarks sendo executados no EP_6

T_o	BT	CG	EP	IS	LU	MG	SP	Média geral
$250ms$	1,05764	1,03772	1,03110	1,09784	1,07171	1,02987	1,05695	1,05469
$500ms$	1,03732	1,02089	1,02810	1,09830	1,04889	1,02419	1,00412	1,03740
$1s$	1,00153	1,02036	1,01924	1,03180	1,03965	1,02275	1,00162	1,01957

Os resultados para o computador EP_7 são apresentados através da Tabela 5.31. Nesse caso, o comportamento é similar ao observado previamente no computador EP_6, com o atraso imposto no tempo de resposta sendo inversamente proporcional ao tamanho do intervalo de amostragem utilizado.

Tabela 5.31: Valores de *slowdown* observados para os benchmarks sendo executados no EP_7

T_o	BT	CG	EP	IS	LU	MG	SP	Média geral
$250ms$	1,00641	1,01813	1,00543	1,07795	1,01821	1,00604	1,00833	1,01496
$500ms$	1,00439	1,00408	1,00513	1,07131	1,00098	1,00335	1,00241	1,01209
$1s$	1,00342	1,00337	1,00417	1,02877	1,00516	0,99946	1,00396	1,00598

5.5 Considerações finais

Este capítulo descreveu os detalhes da implementação da monitoração de processos empregada neste trabalho, o modelo de aquisição de conhecimento e de classificação de processos de aplicações paralelas e a implementação desse modelo junto ao núcleo do sistema operacional Linux.

A implementação da monitoração de processos é direcionada para um conjunto de computadores que executam o sistema operacional Linux. Contadores inseridos em rotinas instrumentadas do núcleo registram as operações realizadas por processos e chamadas de sistema adicionais são definidas e implementadas, visando a coleta dessas informações. O sistema de arquivos `procfs` é utilizado pela solução, que emprega a aplicação `monitor`. A aplicação `monitor` utiliza as chamadas de sistema e realiza leituras no `procfs` para registrar operações realizadas por processos em intervalos de tempo previamente definidos.

A instrumentação do núcleo do sistema, associada à utilização do sistema de arquivos `procfs`, permite coletar informações sobre a execução de processos, no que tange a utilização de recursos do sistema. A vantagem principal dessa abordagem é que as aplicações paralelas não necessitam ser recompiladas para realizar a sua monitoração. Outra vantagem é que a aplicação paralela monitorada pode ser desenvolvida em qualquer linguagem de programação e utilizando qualquer ambiente de passagem de mensagens (e.g. implementações MPI e PVM), sem a necessidade de bibliotecas de software adicionais.

O mecanismo de monitoração descrito pode ser empregado em outros domínios, por exemplo em aplicações servidoras e de acesso a base dados, como uma forma de identificar possíveis gargalos de desempenho. A monitoração pode revelar informações úteis ao desenvolvedor e ao administrador do sistema sobre o comportamento das aplicações.

Através da avaliação de desempenho, que empregou um subconjunto representativo de aplicações paralelas do pacote NAS, pôde-se observar que a instrumentação do núcleo e a monitoração não são intrusivas ao ponto de depreciar o tempo de resposta das aplicações. Essa característica é muito importante em sistemas paralelos, onde geralmente busca-se como objetivo principal o melhor desempenho na computação.

A compilação do núcleo do sistema operacional reduz a portabilidade da solução da monitoração. Uma alternativa para melhorar a portabilidade da solução é a implementação através de módulos de núcleo. Os módulos de núcleo foram definidos visando tornar o sistema operacional Linux mais flexível, de forma que *drivers* ou outros componentes de software pudessem ser incluídos junto ao núcleo. A interface de programação LKM (*Linux kernel modules*), permite que módulos sejam implementados e sejam inseridos em tempo de execução no núcleo do sistema operacional, sem a necessidade da recompilação do núcleo.

A implementação da monitoração através de módulos de núcleo pode ser realizada interceptando as chamadas de sistema da aplicação monitorada, e registrando os valores para uma utilização futura. A figura do monitor continuaria sendo necessária, para a comunicação com tal módulo. Tal solução removeria a necessidade da compilação do núcleo em elementos de processamento utilizados para o processamento paralelo. Estudos podem ser conduzidos para verificar a viabilidade dessa alternativa, como uma forma de melhor a portabilidade do mecanismo de monitoração. A abordagem de monitoração proposta necessita da existência do sistema operacional Linux nos elementos de processamento da máquina paralela virtual. Observa-se, entretanto, a preferência na utilização do sistema operacional Linux em máquinas paralelas virtuais de memória distribuída, principalmente em arranjos computacionais geograficamente não distribuídos.

Dessa forma, conclui-se que a monitoração de processos proposta neste capítulo é adequada para os objetivos desta tese, pela confiabilidade das informações obtidas e pelo desempenho demonstrado pela sua implementação. Assim, considera-se essa monitoração como ferramenta para a aquisição de conhecimento sobre o comportamento de aplicações paralelas na utilização de recursos.

O modelo de aquisição de conhecimento a partir das informações coletadas durante a execução dos processos que compõem as aplicações paralelas tem como objetivo de classificar os padrões de utilização de recursos das aplicações. Inicialmente, são coletadas informações sobre a execução da aplicação, ou traços de execução, em intervalos de tempo pré-definidos. Tais informações alimentam um modelo de classificação, que emprega o paradigma de redes neurais para identificar padrões e realizar a categorização a partir das informações. Após a categorização, é empregado um algoritmo que permite identificar quais são os atributos mais relevantes para cada categoria criada. Esse algoritmo atribui automaticamente rótulos às categorias criadas. O resultado final é a aquisição do conhecimento sobre o comportamento da aplicação, identificando os estados de utilização de recursos, a freqüência na qual cada estado é visitado e a média de utilização de recursos (representada pelo centróide de cada categoria).

O modelo apresenta vantagens quando comparado com outras abordagens descritas na literatura. O conhecimento obtido pelo modelo pode ser estendido à medida que as aplicações são executadas novamente, sem perder o conhecimento obtido previamente. O modelo captura as modificações possíveis na aplicação, permitindo a atualização do conhecimento previamente obtido. A classificação pode ser executada após ou durante a execução da aplicação, dependendo dos requisitos do software de escalonamento. Isso permite que o modelo proposto seja empregado em diferentes cenários de escalonamento de aplicações e onde busca-se o balanceamento de carga no sistema.

Os parâmetros principais do modelo são o intervalo de amostragem T_o e o parâmetro de vigilância ρ da rede neural ART-2A. À medida que o valor do parâmetro T_o aumenta, menor é a granulosidade das informações obtidas e à medida que o valor desse parâmetro diminui, maior é

o impacto no desempenho do sistema. Considerando o ambiente de execução utilizado e a carga de trabalho analisada, observa-se que o valor de T_o igual a 500 milissegundos é adequado. Em relação ao parâmetro ρ, observa-se que valores contidos entre 0,90 e 0,95 são adequados para as aplicações analisadas.

Os valores ideais de ρ são obtidos através da análise das medidas de distância definidas (*intra* e *inter*). Essas medidas de distância têm inspiração no funcionamento da rede, por trabalhar com a diferença entre vetores através do ângulo formado por eles e podem ser aplicadas em qualquer situação onde busca-se definir o valor ideal para a vigilância da rede ART-2A. Assim, essas medidas têm grande utilidade durante a realização de experimentos com a rede ART-2A e podem ser empregadas em outras áreas, que necessitem realizar tarefas de aquisição de conhecimento e reconhecimento de padrões.

O algoritmo de rotulação proposto permite automatizar a interpretação das categorias obtidas pela rede ART-2A. Esse algoritmo utiliza os protótipos criados pela rede neural durante o aprendizado para identificar os atributos principais que orientam a criação das categorias. As categorias são rotuladas empregando os atributos principais e um valor de limiar. O valor de limiar permite definir a quantidade de atributos que são escolhidos para rotular uma dada categoria.

Através do estudo das arquiteturas ART-2A, é definido e implementado o algoritmo de treinamento empregando um conjunto de classes C++. Essa implementação pode ser empregada não somente na classificação de aplicações paralelas, mas em qualquer outro problema que possua como possível solução a utilização da rede neural ART-2A. Além do algoritmo de treinamento da rede, são definidos e implementados também os mecanismos para tratar a categorização dos dados, juntamente com as medidas de avaliação de desempenho da classificação.

Através dos resultados obtidos pelos experimentos, conclui-se que o modelo de aquisição de conhecimento é adequado para classificar aplicações paralelas, podendo ser empregado em diferentes algoritmos de escalonamento e suprindo a necessidade freqüente de um conhecimento mais detalhado da carga de trabalho submetida ao sistema computacional distribuído. Devido às suas características, o modelo pode ser empregado ao término da execução das aplicações, como nos resultados descritos neste capítulo, ou de maneira *on-line*, durante a execução da aplicação.

A implementação do modelo de classificação de processos junto ao núcleo do sistema operacional Linux permite observar o comportamento do modelo para a classificação em tempo de execução dos processos e permite avaliar a intrusão do modelo no desempenho do sistema.

A implementação do algoritmo de classificação, baseado na arquitetura de rede ART-2A, foi modificada, trocando o modelo de orientação à objetos pelo modelo orientado à procedimentos. Além dessa mudança, foi definida uma nova estrutura de dados (*buffer*), que permite que as informações colhidas em um espaço de tempo anterior sejam inseridas juntamente com as informações mais recentes. Tal estrutura é importante para que a rede neural estabilize e

consiga codificar os valores de entrada adequadamente. Optou-se por utilizar um tamanho de *buffer* igual a 30, e um número de ciclos igual a 3. Pôde-se verificar, através dos resultados obtidos pela classificação, que tais valores permitem uma boa qualidade na classificação do comportamento dos processos. A qualidade da classificação é diretamente proporcional a esses valores, ao contrário da intrusão no desempenho do sistema, que aumenta à medida que mais informações são enviadas à rede ART-2A.

Uma melhoria que pode ser realizada na implementação é a criação de um novo ponto de acesso no `procfs`, dedicado apenas a realizar uma interface com a implementação do algoritmo de aprendizado. O arquivo `/proc/pid/status` é utilizado por diferentes ferramentas do sistema operacional, por exemplo o programa computacional `ps`, que exibe informações sobre os processos em execução. Um ponto de acesso definido como uma entrada nova no `procfs` isolaria a interface de comunicação com a implementação do algoritmo de aprendizado para programas computacionais específicos que realizam a monitoração.

Através da execução de benchmarks em dois computadores com potências computacionais diferentes, pôde-se observar o impacto que a implementação do algoritmo de classificação impõe no desempenho do sistema. Para o computador EP_6, os valores de atraso são iguais a $0,5\%$, $0,3\%$ e $0,1\%$, considerando respectivamente valores de intervalos de amostragem iguais a 250 ms, 500 ms e 1 segundo; para o computador EP_7, os valores são iguais a 1%, 1% e $0,5\%$. Os resultados obtidos indicam que a implementação do algoritmo de classificação e sua utilização em tempo de execução produzem pequenos atrasos no tempo de execução dos processos. Tal característica é bastante atrativa para a utilização do algoritmo em sistemas paralelos reais. Pelos resultados obtidos, conclui-se que o algoritmo de classificação proposto é adequado para a aquisição de conhecimento mais detalhado sobre as aplicações paralelas, por apresentar bons desempenhos de classificação e computacional.

A troca de informações pelo arquivo `proc/pid/status` permite que qualquer processo em modo usuário possa obter informações sobre o comportamento de um determinado processo, desde que sejam observadas as condições de segurança de acesso a recursos definidas pelo sistema operacional Linux. As informações são obtidas em tempo de execução do processo monitorado, de maneira *on-line*. Assim, a utilização do programa computacional `monitor` e das chamadas de sistema previamente definidas (Seção 5.2) é indicada para situações onde seja necessária a aquisição de conhecimento sobre a aplicação após o término de sua execução, de maneira *off-line*.

CAPÍTULO

6

Exploração de Características da Carga de Trabalho

6.1 Considerações iniciais

Diversos trabalhos têm demonstrado que as informações gravadas em traços de execução de aplicações paralelas podem ser empregadas na predição de características de execução de aplicações paralelas (Downey, 1997; Smith *et al.*, 1998b; Krishnaswamy *et al.*, 2004). Nesse contexto, informações sobre a submissão de aplicações paralelas ao sistema compõem uma base de experiências prévias, na qual o conhecimento pode ser extraído. Assim, sendo uma aplicação paralela definida como um ponto de consulta x_q, busca-se encontrar um conjunto de aplicações similares a x_q e, através dessas aplicações, gerar uma predição sobre uma característica da aplicação x_q previamente ou durante a sua execução. Exemplos de características que podem ser preditas são a utilização de memória, utilização de UCP e tempo de execução.

As características das aplicações paralelas submetidas ao sistema são chamadas de atributos e a similaridade entre x_q e as demais aplicações da base é calculada através desses atributos. Exemplos de atributos de uma aplicação são o nome do arquivo executável, usuário que submete a aplicação ao sistema e a quantidade de EPs requisitados. Quando utilizados para calcular a similaridade, tais atributos são chamados de atributos de entrada. Atributos de saída são os atributos não utilizados para calcular a similaridade e que o algoritmo de aprendizado busca predizer o seu valor.

Aplicações da base de experiências podem ser consideradas similares à aplicação x_q por terem sido submetidas ao sistema pelo mesmo usuário, submetidas no mesmo espaço de tempo e com os mesmos argumentos da aplicação x_q. Definições mais complexas envolvem a combinação

desses atributos e a forma pela qual cada atributo é levado em consideração no cálculo do atributo de saída.

Dentre os trabalhos relacionados com a definição de similaridade em aplicações paralelas e com a predição de características de execução de aplicações paralelas, destacam-se os trabalhos de Downey (1997), Harchol-Balter & Downey (1997), Gibbons (1997), Smith *et al.* (1999), Krishnaswamy *et al.* (2004), Iverson *et al.* (1999) e Kapadia *et al.* (1999).

Downey (1997) categoriza todas as aplicações previamente executadas através do identificador da fila do escalonador e cria modelos através de distribuições de probabilidade. Esses modelos são empregados para realizar a predição do tempo de execução. Downey (1997) observa que a distribuição dos tempos de execução das aplicações pode ser modelada através de um função logarítmica $\beta_0 + \beta_1\ ln\ t$. Os parâmetros β_0 e β_1 são obtidos para cada uma das categorias utilizadas: aplicações seqüenciais, aplicações pequenas, aplicações médias e aplicações longas. Cada uma dessas categorias corresponde à natureza das aplicações, que é explicitamente identificada quando o usuário submete aplicações às filas do escalonador. O modelo é definido omitindo uma quantidade de 10% das aplicações de tempo de vida mais curto e as 10% aplicações de tempo de vida mais longo das cargas de trabalho SDSC95 e CTC e com essas restrições, apresenta um coeficiente de determinação[1] igual a $R^2 = 0,99$. Através desse modelo, duas técnicas são utilizadas para realizar a predição. A primeira técnica prediz o tempo de vida médio de uma aplicação, tendo como entrada o tempo de vida atual da aplicação:

$$\sqrt{\alpha \times e^{\frac{1-\beta_0}{\beta_1}}} \tag{6.1}$$

Downey (1997) verifica que esse modelo é um bom preditor para aplicações cujo tempo de vida está no intervalo entre 6 a 12 horas. Para tempos de vida acima desse intervalo, o modelo não produz boas predições. A segunda técnica emprega o tempo de vida médio condicional:

$$\frac{t_{max} - \alpha}{log t_{max} - log \alpha} \tag{6.2}$$

sendo $t_{max} = e^{(1-\beta_0)/\beta_1}$. Downey (1997) observa que para cada carga de trabalho, os parâmetros da distribuição necessitam ser novamente calculados, sendo que para cada carga de trabalho um número considerável de aplicações deve ser desconsiderado nos traços de execução para que o modelo encaixe-se com maior precisão nos dados reais. Além disso, o modelo de predição é mais adequado para realizar a predição do tempo de execução de aplicações que já foram executadas por um instante de tempo α.

Harchol-Balter & Downey (1997) modelam a distribuição de probabilidade para o tempo de vida de aplicações seqüenciais. Nesse trabalho, traços de execução de diferentes estações de trabalho são analisados e os autores observam distribuições de probabilidade que definem que o

[1] O coeficiente de determinação, representado por R^2, determina a qualidade (*goodness*) dos resultados obtidos com a correlação linear entre duas variáveis, através da análise da variância do erro de predição obtido pelo modelo linear. Quanto mais próximo de 1 for o valor desse coeficiente, melhor é a qualidade obtida com a correlação linear.

tempo de vida restante das aplicações é proporcional à quantidade de tempo de execução atual da aplicação. Os autores exploram essas distribuições e propõem um algoritmo de balanceamento de carga, com migração de aplicações, para sistemas distribuídos. O algoritmo de carga proposto busca realizar a migração de tarefas cujo tempo de vida restante esperado seja suficiente grande para amortizar os custo totais envolvidos com a migração das aplicações.

Gibbons (1997) usa as informações de execuções prévias de aplicações paralelas para predizer o tempo de execução de aplicações paralelas, através de um proposta de organização de um histórico de perfis de execução das aplicações. Nesse trabalho, são introduzidos os conceitos de gabaritos (*templates*) e categorias aplicados à predição do tempo de execução de aplicações paralelas. Um gabarito é formado por um conjunto de atributos, que aplicado ao conjunto de execuções prévias, identifica um conjunto de categorias. Empregando essa idéia, os modelos de Downey (1997) podem ser considerados como sendo da classe gabarito fixo, pois o gabarito empregado (identificador da fila do escalonador) não varia. Por exemplo, um gabarito () indica todas as aplicações previamente executadas; da mesma forma, um gabarito (u, e) corresponde as categorias formadas pelas aplicações que possuem o mesmo usuário e o mesmo identificador do arquivo executável. Gabaritos aplicados ao traço de execuções prévias correspondem a um definição fixa de similaridade. Nesse trabalho, os gabaritos são aplicados na carga de trabalho para encontrar aplicações similares.

Os gabaritos utilizados por Gibbons (1997) estão descritos na Tabela 6.1, onde u corresponde ao identificador do usuário, e é o identificador do arquivo executável, n é a quantidade de EPs requisitados e t é o tempo de vida atual da aplicação no instante que a predição é efetuada. Esses gabaritos são selecionados pela análise do coeficiente de variação dos tempos de predição dentro das categorias. Após agrupar as aplicações nas categorias identificadas pelos gabaritos, duas técnicas são empregadas para gerar a predição do tempo de execução: média e regressão linear. Gibbons (1997) não relata uma etapa inicial de filtragem dos dados.

Tabela 6.1: Gabaritos definidos por Gibbons (1997)

Gabarito	**Técnica de predição**
(u, e, n, t)	média
(u, e)	regressão linear
(e, n, t)	média
(e)	regressão linear
(n, t)	média
$()$	regressão linear

Smith *et al.* (1999) estendem a idéia de Gibbons (1997), apresentando algoritmos específicos para encontrar um conjunto de gabaritos mais adequados para cada carga de trabalho, ao invés de trabalhar com gabaritos fixos e independentes das características da carga de trabalho. Dois algoritmos de busca são empregados: um algoritmo guloso (*greedy search*) e um algoritmo

genético. Para realizar a predição, são empregados regressão linear e média. Smith *et al.* (1999) concluem que o algoritmo genético apresenta o melhor desempenho para definir aplicações mais similares juntamente com o preditor sendo a média dos pontos da categoria. Outras conclusões importantes obtidas através desse trabalho (Smith *et al.*, 1998a, 1999) são:

1. a qualidade da predição é diretamente proporcional à quantidade de atributos utilizada nos algoritmos;
2. para realizar predições, a localidade temporal na base de experiências é importante: assim, deve-se empregar como base de experiências as aplicações que foram executadas mais recentemente;
3. a média é um melhor preditor que a regressão linear;
4. os atributos mais importantes para definir aplicações similares são: identificador do arquivo executável da aplicação, argumentos da aplicação, usuário, grupo e quantidade de EPs requisitados.

Assim como no trabalho de Gibbons (1997), não é relatada uma etapa de filtragem inicial das informações obtidas pelas execuções prévias. Outra característica comum nesses trabalhos é a utilização do atributo n (quantidade de EPs utilizados) de maneira discreta. A quantidade de EPs requisitada pelas aplicações é discretizada utilizando números em potência de 2, por exemplo $n = 2$, $n = 4$ e $n = 128$. Essa estratégia de busca pelos melhores gabaritos para cada carga de trabalho é também explorada no trabalho de Krishnaswamy *et al.* (2004), que emprega uma técnica baseada em *rough sets* para definir os melhores gabaritos para cada categoria.

Iverson *et al.* (1999) propõem a utilização de um algoritmo estatístico para a predição do tempo de execução de aplicações paralelas, com o objetivo de melhorar as decisões do escalonamento estático em um cenário heterogêneo. Nesse sentido, os autores utilizam a técnica dos vizinhos mais próximos para encontrar execuções prévias da mesma aplicação correspondente ao ponto de consulta. Os atributos usados são específicos ao problema para o qual as aplicações paralelas são implementadas, como tamanho do problema. Iverson *et al.* (1999) não consideram atributos como usuário, grupo e identificador da aplicação. Um contribuição desse trabalho é a idéia de utilizar características do ambiente de execução (e.g. capacidade de processamento), medidas através de benchmarks específicos, juntamente com atributos intrínsecos às aplicações paralelas no cálculo de similaridade.

Kapadia *et al.* (1999) avaliam os algoritmos dos vizinhos mais próximos, vizinhos mais próximos ponderados pela distância e regressão localmente ponderada para a predição do tempo de execução de ferramentas de simulação. Os algoritmos são aplicados considerando apenas as execuções prévias de 3 ferramentas, e não são feitas considerações sobre a utilização das técnicas de predição considerando outras aplicações e outros atributos, além dos parâmetros de execução das aplicações paralelas observadas. Os autores concluem que o algoritmo dos vizinhos mais

próximos produz os melhores resultados para a predição do tempo de execução das ferramentas de simulação consideradas.

Este capítulo apresenta a utilização do algoritmo IBL (vide Capítulo 3) para adquirir conhecimento a partir de bases de experiências, visando predizer características de aplicações paralelas. Ao invés de explorar as características de execução das aplicações, assunto abordado no Capítulo 5, este capítulo tem como objetivo a utilização desse algoritmo de aprendizado para a exploração de características da carga de trabalho relacionadas com a submissão das aplicações paralelas ao sistema, por exemplo usuário, identificador da fila e nome do arquivo executável. Nesse sentido, são descritos os detalhes da implementação e da utilização desse algoritmo em traços de execução reais para a predição do tempo de execução de aplicações paralelas.

O restante deste capítulo está organizado em 3 seções. O modelo de aquisição de conhecimento e sua implementação são apresentados na Seção 6.2. Através da Seção 6.3, são apresentados os resultados obtidos com a utilização desse modelo de aquisição de conhecimento em traços de execução reais. Embora qualquer característica de execução possa ser predita através do modelo, essa seção apresenta os resultados obtidos com a predição do tempo de execução das aplicações. Nessa seção, os resultados previamente descritos na literatura são comparados com os resultados obtidos com o algoritmo IBL. Finalizando este capítulo, a Seção 6.4 apresenta as considerações finais.

6.2 Aquisição de conhecimento

O aprendizado baseado em instâncias traz várias vantagens para a predição do tempo de execução de aplicações paralelas. A vantagem principal do algoritmo é a sua característica incremental para aquisição de conhecimento. À medida que novas aplicações são executadas e seus traços de execução são incluídos na base de experiências, o conhecimento, que é extraído apenas no momento da consulta, pode ser adquirido automaticamente. Uma segunda vantagem é a possibilidade de trabalhar com vários atributos das aplicações ao mesmo tempo, ponderando a contribuição de cada atributo para gerar a estimativa do atributo de saída. Outras vantagens são o bom desempenho computacional e a boa tolerância ao ruído na base de experiências.

O algoritmo IBL, definido no Capítulo 3, é implementado utilizando o paradigma orientado a objetos, através da linguagem C++. Essa implementação do algoritmo comunica-se com uma base de dados, que contém um conjunto de tabelas, onde cada tabela corresponde a uma das cargas de trabalho utilizadas. Assim, o algoritmo de aprendizado é implementado como uma aplicação cliente, que requisita informações para um servidor, responsável por enviar as informações sobre os traços de execução. A vantagem principal dessa implementação é a separação do algoritmo e da base de experiências, de forma que a base pode ser executada em um outro computador, diferente daquele que executa o algoritmo de aprendizado. As bases de

experiências, uma base para cada traço de execução, são implementadas como tabelas de uma base de dados. O software utilizando como gerenciador de banco de dados é o MySQL [2], distribuído livremente pela licença GPL [3]. Esse gerenciador foi escolhido por ser rápido, confiável e por distribuir livremente uma interface de programação para acesso ao banco de dados. A implementação do algoritmo IBL utiliza essa interface de programação para realizar o acesso aos traços de execução.

A quantidade de atributos (ou colunas) das tabelas é igual ao formato padrão para traços de execução de aplicações paralelas definido por Feitelson (Tabela 4.4) e um campo adicional (`d`), do tipo `real`, responsável por armazenar temporariamente a distância entre a instância na base de conhecimento e o ponto de consulta.

Índices de consulta do banco de dados são definidos e utilizados nessas tabelas para melhorar o desempenho da implementação do algoritmo de aprendizado. Observa-se uma queda considerável no desempenho computacional do algoritmo de aprendizado quando esses índices não são empregados. Os índices são definidos empregando como chave os atributos `job_number` e `d`. O índice `job_number` é utilizado nas consultas que são limitadas a uma faixa estabelecida de aplicações gravadas na base. Dessa forma, o cálculo de distância entre o ponto de consulta e as aplicações fica limitado apenas a esse faixa, empregando tal índice, não sendo necessário calcular a distância para todas as experiências existentes nas tabelas. O índice `d` está relacionado com o parâmetro vizinhança (k). Através desse índice, pode-se recuperar eficientemente os k vizinhos mais próximos encontrados após o cálculo da distância.

O Algoritmo 6.1 apresenta um esqueleto de um programa computacional simplificado para avaliar a predição do tempo de execução de um conjunto de aplicações paralelas, através do conjunto de classes definido. Inicialmente, um objeto do tipo `TIbl` é criado. A seguir, o traço de execução que servirá como base de experiências é selecionado (passos 2 e 3) e os parâmetros principais do algoritmo de aprendizado, que são o tamanho da vizinhança e o valor de t, são instanciados (passos 4 e 5). Após, é especificado qual é o atributo de saída (passo 6) e qual é a coluna referente ao campo `d`, que serve para armazenar temporariamente as distâncias (passo 7). Nesse caso, define-se a base de experiências como sendo os primeiros 640 registros do traço de execução e selecionam-se as 384 aplicações subseqüentes, numeradas pelos registros 641 até 1024, para que seus tempos de execução sejam preditos.

Nesse algoritmo, os pesos dos atributos são definidos através do método `Setz()`, dentro do primeiro laço de repetição (passo 12). Como a implementação define que todos atributos são considerados inicialmente como numéricos, o método `SetAttrType()` é utilizado para definir quais atributos são nominais (e.g. identificador do usuário, do grupo e da fila do escalonador), o que ocorre no segundo laço de repetição (passo 15). Após o programa conectar-se ao banco de

[2] http://www.mysql.org
[3] http://www.gnu.org

dados (passo 18), os valores mínimos e máximos dos atributos, que são utilizados na normalização dos atributos, são calculados (passo 21).

O laço de repetição, no qual são realizadas as 384 predições. é iniciado na seqüência (passo 22). Como o experimento realiza predições do tempo de execução das aplicações também gravadas no traço de execução, o ponto de consulta xq é montado a partir do próximo registro do conjunto de treinamento (linha 23).

O método `ComputeSimilarity()` realiza o cálculo da distância do ponto x_q em relação a todos os registros da base de experiências (registros 1 a 640) e armazena os valores de distância no registro `d`. Na seqüência, o método `GetNeighborhood()` realiza uma consulta na base de experiências para obter os k vizinhos (nesse caso $k = 5$) mais próximos ao ponto de consulta (passo 25).

Algoritmo 6.1 Exemplo da utilização das classes C++ do algoritmo IBL

```
 1: TIbl ibl = new TIbl(18);
 2: ibl->SetDatabaseName("workload");
 3: ibl->SetTableName("SDSC95");
 4: ibl->SetNeighborhood(5);
 5: ibl->SetSigma(0.125);
 6: ibl->SetDistanceField(18);
 7: ibl->SetOutputField(3);
 8: ibl->SetTrainStartJob(1);
 9: ibl->SetTrainEndJob (640);
10: ibl->SetTestStartJob (641);
11: ibl->SetTestEndJob (1024);
12: para cada atributo que define as aplicações faça
13:    use o método ibl->Setz() para definir o peso do atributo
14: fim para
15: para cada atributo que define as aplicações faça
16:    use o método ibl->SetAttrType() para definir o tipo do atributo
17: fim para
18: ibl->MySQLConnect();
19: job_number = ibl->GetTestStartJob();
20: job_end = ibl->GetTestEndJob();
21: ibl->ComputeMinMax();
22: enquanto job_number <= job_end faça
23:    ibl->GetQueryPointFromTable(job_number)
24:    ibl->ComputeSimilarity();
25:    ibl->ComputeNeighborhood();
26:     E = ibl->wNN();
27:    job_number++;
28: fim enquanto
29: ibl->ComputeErrors();
30: ibl->PrintErrors();
31: delete ibl;
```

Como último conjunto de instruções do laço, o método `wNN()` realiza a predição do tempo de execução da ponto de consulta, através do método dos vizinhos mais próximos ponderados pela distância (passo 26). O resultado da predição para a aplicação corrente é armazenado na variável `E`. Os erros de classificação obtidos são calculados através do método `ComputeErrors()` e através do método `PrintErrors()`, o erro médio de classificação pode ser obtido (passos 29 e 30).

6.3 Avaliação do modelo

Os traços de execução SDSC95, SDSC96, SDSC2000, CTC e LANL são empregados para a avaliação do modelo de predição. Inicialmente, os dados das cargas de trabalho são filtrados visando remover registros inconsistentes e principalmente aplicações não finalizadas naturalmente. A Tabela 6.2 apresenta a quantidade total de aplicações em cada traço de execução e a quantidade de aplicações não utilizadas em cada carga de trabalho.

Tabela 6.2: Número de aplicações utilizadas e desprezadas por carga de trabalho

Carga de Trabalho	Número de aplicações	Não utilizadas
SDSC95 (Intel Paragon)	76.872	13.328
SDSC96 (Intel Paragon)	38.719	825
SDSC2000 (IBM SP2)	67.667	11.175
CTC (IBM SP2)	79.302	16.672
LANL (Conection Machine CM-5)	201.387	20.537

Após a filtragem dos dados, experimentos são conduzidos para observar os desempenhos computacional e de predição do algoritmo IBL. A metodologia utilizada nesses experimentos consiste em dividir o número total de aplicações gravadas nos traços de execução, sem considerar aquelas não finalizadas corretamente, em um número determinado de conjuntos. Nesses conjuntos, reserva-se uma fração de $2/3$ para a base de experiências e $1/3$ para o conjunto de teste. Tal estratégia é escolhida visando manter a localidade temporal dos dados considerados.

Assim, as execuções mais recentes, em relação ao ponto de consulta, são empregadas para realizar as predições. O desempenho computacional do algoritmo de aprendizado é inversamente proporcional ao tamanho da base de experiências: quanto maior é tamanho da base de experiências, maior é o tempo para realizar uma consulta na base para encontrar os k vizinhos mais próximos. Na Tabela 6.3, estão relacionados os tamanhos das bases de experiências utilizados, assim como os tamanhos dos conjuntos de teste e a quantidade de amostras obtida considerando os tamanhos da base do conjunto de treinamento em relação a quantidade total de aplicações existente no traço de execução.

Tabela 6.3: Detalhes dos experimentos com o algoritmo IBL

Carga de Trabalho	Tamanho da base de experiências	Tamanho do conjunto de teste	Quantidade de amostras
SDSC95	716	369	70
SDSC96	421	210	60
SDSC2000	621	320	60
CTC	688	355	60
LANL	700	300	60

Dentre os atributos existentes no formato padrão de cargas de trabalho, apenas os mais relevantes são escolhidos, compreendendo o nome do usuário, nome do grupo que o usuário pertence, identificador da fila do escalonador na qual a aplicação foi submetida, nome do arquivo executável, argumentos do arquivo executável e quantidade de EPs requisitados (Smith *et al.*, 1998b). Esses atributos são listados na Tabela 6.4. Os atributos identificador do usuário e identificador de grupo estão gravados na cargas de trabalho SDSC95 e SDSC96 e o atributos identificador da fila e identificador do usuário não são gravados na carga de trabalho CTC.

Tabela 6.4: Atributos de entrada utilizados para cada carga de trabalho

Atributo	SDSC95	SDSC96	SDSC2000	CTC	LANL
número de EPs utilizados	•	•	•	•	•
identificador do usuário	•	•	•	•	•
identificador de grupo			•		•
nome do arquivo executável			•	•	•
identificador da fila	•	•	•		•

Para a avaliação da qualidade da predição do algoritmo IBL, duas medidas de desempenho são utilizadas: o erro médio absoluto e o erro médio percentual. O erro absoluto é igual ao módulo da diferença, expresso em minutos, entre valor obtido com a predição e o valor real gravado na base de experiências. O erro percentual é igual ao erro absoluto divido pelo valor máximo entre a predição e o valor real. Por exemplo, dado um ponto de consulta cujo valor do atributo de saída é igual a 100 e o algoritmo de aprendizado retorna um valor igual a 90, tem-se um erro absoluto igual a 10 e um erro percentual igual a 10%; para uma instância cujo o valor do atributo de saída seja igual a 1000 e o algoritmo IBL retorna um valor igual a 900, tem-se um erro absoluto igual a 100 e um erro percentual igual a 10%.

Como os traços de execução apresentam grande variabilidade no tempo de execução das aplicações, considera-se principalmente o erro percentual médio para a avaliação do algoritmo IBL, pois essa medida permite obter uma melhor estimativa da qualidade da classificação do algoritmo. Os resultados de erro médio são armazenados em uma tabela específica para a

validação da qualidade da predição do algoritmo de aprendizado. Cada registro dessa tabela corresponde ao erro obtido durante o teste do algoritmo.

Variam-se os parâmetros principais do algoritmo IBL, que são a vizinhança k e a largura da função gaussiana t^2, visando identificar a parametrização mais adequada para cada carga de trabalho. Os valores de k experimentados são 5; 10; 25 e 50, e os valores de t^2 experimentados são $0,125; 0,250; 0,500; 1,000$ e $2,000$. O valor de k define a quantidade de instâncias mais próximas ao ponto de consulta que serão empregadas para a predição e o valor de t^2 define o peso que cada vizinho terá no cálculo da estimativa, de acordo com sua distância ao ponto de consulta. A combinação desses parâmetros constitui 20 grupos. Os erros de classificação para cada grupo são analisados através de análise de variância *ANOVA* e, quando necessário, as médias são comparadas através do teste estendido de Tukey (Shefler, 1988).

6.3.1 Resultados obtidos

O resultados obtidos com a utilização do algoritmo, avaliados através da medida erro médio percentual, são apresentados no Apêndice A, Tabelas A.9, A.12, A.14, A.17, A.20. A análise estatística dos resultados obtidos com a utilização do algoritmo IBL revela que o melhor valor de k, para todas as cargas de trabalho, é igual a 5, e que não existe diferença estatística entre os valores de t^2 utilizados (vide Apêndice A, Tabelas A.9, A.12, A.14, A.17, A.20). Apesar disso, observa-se que o erro médio nas predições aumenta à medida que são experimentados valores maiores de t^2. Isso indica que para a predição do tempo de execução de aplicações paralelas, é melhor trabalhar com um função de ponderação para os vizinhos que seja mais seletiva, distribuindo pesos menores para vizinhos mais distantes ao ponto de consulta.

Os resultados para a melhor combinação dos parâmetros estão ilustrados graficamente na Figura 6.1. Pode-se observar o comportamento do algoritmo de predição para cada carga de trabalho analisada. Cada ponto nesses gráficos, identificado pelo símbolo "+", corresponde a um média de predições do algoritmo IBL, onde a quantidade de amostras por média é igual aos valores descritos na terceira coluna da Tabela 6.3.

O algoritmo IBL apresenta resultados de predição com erro no intervalo entre 38% e 57%, dependendo da carga de trabalho utilizada (Tabela 6.5). Os melhor desempenho de classificação é observado para a carga de trabalho LANL, com um erro de predição igual a $38,05\%$. Isso deve-se ao fato de que, a carga de trabalho LANL, assim como a carga de trabalho SDSC2000, têm uma quantidade maior de atributos relevantes gravados em seu traços de execução.

Os resultados obtidos nas demais cargas de trabalho são semelhantes. O algoritmo IBL apresenta erros de predição iguais a $57,46\%$ e $55,40\%$, respectivamente, para a cargas de trabalho SDSC95 e SDSC96. Isso deve-se a falta de atributos relevantes nesses traços, por exemplo identificador do usuário e identificador do arquivo executável. O algoritmo IBL precisa

Tabela 6.5: Resultados do algoritmo IBL

Carga de trabalho	Erro médio percentual (%)
SDSC95	57,46
SDSC96	55,40
SDSC2000	50,05
CTC	50,45
LANL	38,05

de um bom número de atributos para diferenciar as aplicações e melhor definir a vizinhança do ponto de consulta.

Influência da quantidade de atributos utilizada

Um aspecto observado com a utilização das cargas de trabalho no formato padrão (SWF) é que algumas informações, presentes nos traços de execução originais, são desprezadas na conversão para o formato SWF. Tais informações podem ser relevantes para a definição de similaridade entre as aplicações e podem levar a uma melhor qualidade no algoritmo de predição.

Nesse sentido, experimentos são executados para observar a qualidade de predição do algoritmo IBL na carga de trabalho CTC original. Esses experimentos têm o objetivo de verificar qual o desempenho de classificação obtido para essa carga de trabalho, quando considerados os atributos suprimidos na conversão ao formato SWF. Assim, os atributos utilizados nesses experimentos são: classe da aplicação, número de processadores requisitados, tipo dos processadores, tipo da aplicação, identificador do usuário e identificador da aplicação. O traço de execução original prevê outros atributos relevantes, como memória utilizada e tipo dos elementos de processamento, mas tais atributos possuem o mesmo valor para todas as aplicações gravadas no traço de execução, o que indica que o software de escalonamento não realizou a gravação desses atributos adequadamente.

Assim como nos experimentos anteriores, executa-se uma filtragem nos dados gravados no traço de execução e as 20 combinações de parâmetros são experimentadas. Como a quantidade de aplicações é idêntica à quantidade descrita na Tabela 6.2, os tamanhos considerados para a base de experiências e conjunto de testes permanecem iguais àqueles descritos na Tabela 6.3.

Assim como na carga de trabalho gravada no formato padrão, a combinação de parâmetros que apresenta melhores resultados é $k = 5$ e $t^2 = 0,125$ (vide Apêndice A, Tabelas A.48, A.49 e A.50). O erro médio percentual observado nas predições é igual a $48,61\%$, que corresponde a uma melhoria de $3,64\%$ na qualidade da predição do algoritmo.

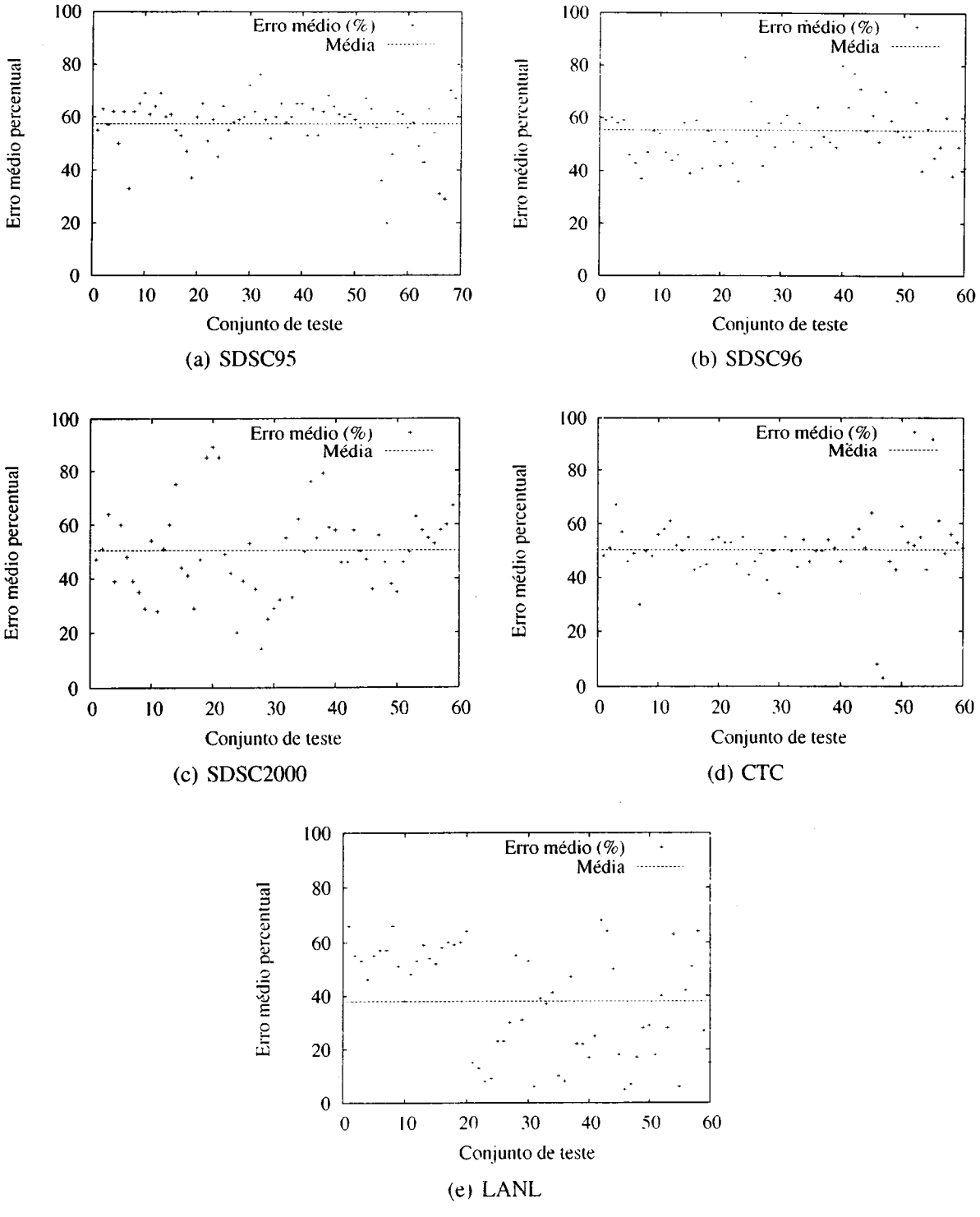


Figura 6.1: Erro de classificação obtido para cada conjunto de teste

Qualidade de predição do algoritmo com valores de t^2 variáveis

Um possibilidade na utilização do aprendizado baseado em instâncias é, ao invés de trabalhar com o valor de t^2 fixo, empregar valores diferentes para t^2 em cada consulta (Atkeson & Schaal, 1995). Isso permite que os pesos para os pontos vizinhos ao ponto de consulta sejam ponderados de maneira diferente, de acordo com as distâncias observadas entre o ponto x_q e seus vizinhos no momento da consulta. Experimentos são realizados para observar a qualidade do algoritmo IBL utilizando uma largura da função variável. Assim, o valor de t na equação:

$$w_i(x) = e^{\frac{-E(x_q,x)}{2\times t^2}} \tag{6.3}$$

varia para cada consulta e é igual a $t = E(x_q, x_k)$, que corresponde à distância entre o ponto de consulta (x_q) e o k-ésimo vizinho considerado (x_k). Assim como nos experimentos previamente descritos, os valores de k utilizados são iguais a 5, 15, 25 e 50. Os resultados obtidos são analisados através da ANOVA, com 4 grupos por carga de trabalho.

Os valores de erro médio percentual obtido para cada combinação de parâmetros são apresentados através nas Tabelas A.35, A.38, A.40, A.42 e A.45. Nesses experimentos, os valores de erro médio percentual obtidos são diretamente proporcionais aos valores de k experimentados. A análise estatística com ANOVA (vide Apêndice A, Tabelas A.36, A.39,A.41,A.43 e A.46), juntamente com o teste de Tukey (Apêndice A,Tabelas A.37 e A.44) permite concluir, com um nível de significância de 95%, que o melhor valor de vizinhança para todas as cargas de trabalho é igual a $k = 5$, de maneira semelhante aos resultados obtidos com valores fixos de t.

A Tabela 6.6 ilustra os erros médios percentuais obtidos com a utilização desses parâmetros, considerando a mesma configuração para tamanhos de base de experiências e de conjunto de teste ilustrados na Tabela 6.3, comparados aos valores obtidos com a utilização de t com valores fixos.

Tabela 6.6: Resultados do algoritmo IBL com valor de t variável

Carga de trabalho	**Erro médio percentual (%)**	
	t fixo	t variável
SDSC95	57,46	57,38
SDSC96	55,40	54,60
SDSC2000	50,05	66,95
CTC	50,45	54,63
LANL	38,05	38,29

Através do teste Z, os valores de erro médio percentual obtidos com t fixo e variável são comparados, com um nível de significância de 95% (vide Apêndice A, Tabela A.47). Os resultados indicam que a qualidade da classificação, expressa pelo erro médio percentual, é estatisticamente igual para as cargas de trabalho SDSC95, SDSC96, CTC e LANL. Para a carga

de trabalho SDSC2000, os valores de t variáveis apresentam qualidade de classificação piores que a qualidade da classificação obtida com valores de t fixos.

Desempenho computacional do algoritmo IBL

A Figura 6.2 mostra o gráfico do tempo de resposta do algoritmo IBL, expresso em milissegundos, empregando $k = 5$ e variando o tamanho da base de experiências com os valores entre 5 a 1024 para a carga de trabalho CTC. Nesse gráfico, cada ponto corresponde à média do tempo de resposta para 50 predições, utilizando o EP_7 para armazenar a base de experiências e executar o algoritmo IBL (os valores observados e utilizados para compor essas médias estão ilustrados no Apêndice A, na Tabela A.34). Pode-se observar que o tempo de resposta aumenta linearmente, à medida que aumenta o tamanho da base de experiências. Por exemplo, o tempo médio de resposta é igual a 1.15 milissegundos utilizando um número de 5 instâncias na base de experiências e igual a 126 milissegundos quando utilizadas 1024 instâncias. Assim, pode-se obter um modelo linear para a predição do tempo de resposta do algoritmo IBL, através de regressão linear:

$$tr = 0,2732 * tb - 0,3480 \tag{6.4}$$

onde tr é o tempo de resposta do algoritmo, expresso em milissegundos, e tb é o tamanho da base de experiências, expresso pelo número de instâncias empregadas. O valor observado de R^2 para esse modelo de desempenho é igual $0,998$.

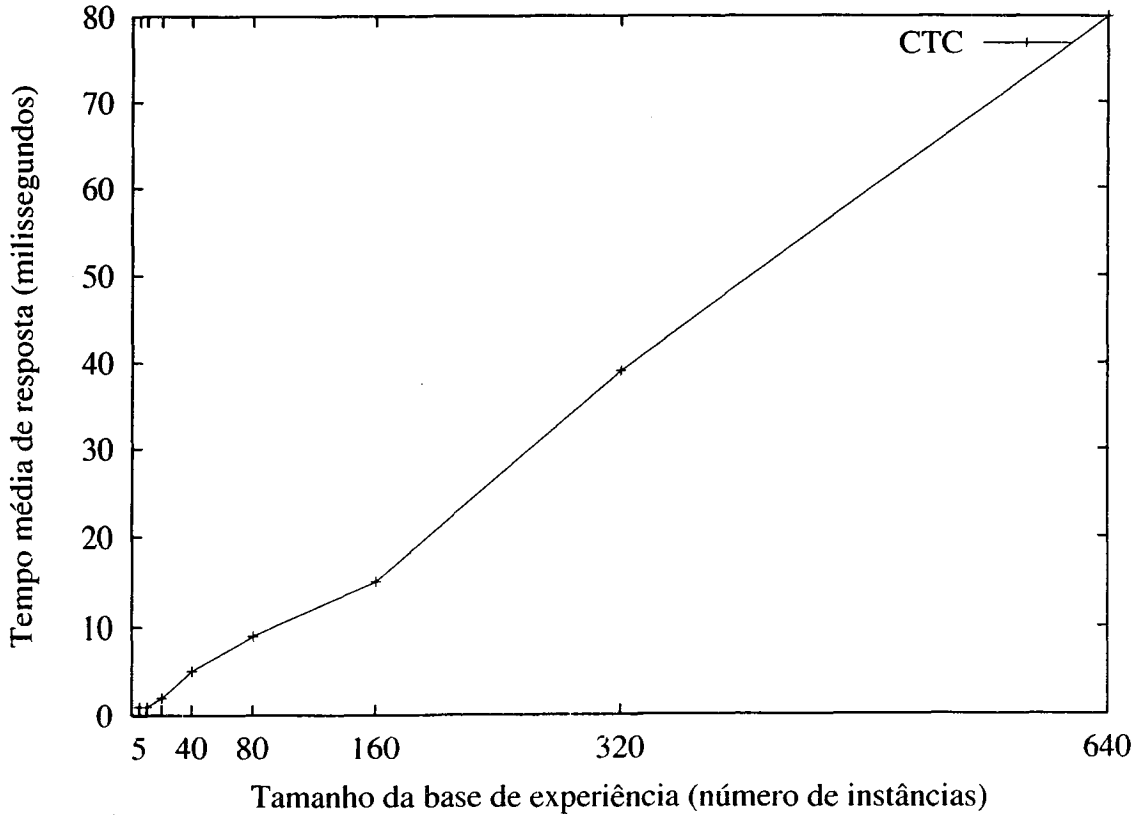


Figura 6.2: Tempo de resposta do algoritmo IBL

6.3.2 Qualidade da predição do algoritmo IBL comparada com trabalhos relacionados

A Tabela 6.7 apresenta os melhores resultados da predição obtida pelo algoritmo IBL, expressos pelo erro médio absoluto, comparados com 6 abordagens prévias descritas na literatura. Para a comparação, apenas os trabalhos relacionados que utilizam as mesmas cargas de trabalho são consideradas, pois os resultados do algoritmo estão fortemente relacionados com as características da carga de trabalho. Além disso, as informações utilizadas em trabalhos não considerados na comparação não se encontram disponíveis, tornando-se inviável a comparação da qualidade da classificação.

Os resultados das técnicas de Downey (1997) e Gibbons (1997) foram extraídos do trabalho de Smith *et al.* (1999), que utilizou o código fonte provido por esses autores para predizer o tempo de execução, utilizando as cargas de trabalho SDSC95, SDSC96 e CTC. O algoritmo IBL alcança resultados que são respectivamente 43,03% e 21,78% melhores que os melhores resultados obtidos previamente pelos outros pesquisadores para as cargas de trabalho SDSC95 e SDSC96, e qualidade de predição similar para carga de trabalho CTC.

Tabela 6.7: Resultados obtidos com o algoritmo IBL comparados com trabalhos relacionados

	Erro médio absoluto (minutos)		
Técnica	**SDSC95**	**SDSC96**	**CTC**
Downey (tempo médio de vida)	82,44	102,04	179,46
Downey (tempo médio condicional)	171,00	168,24	201,34
Gibbons (gabaritos fixos)	74,05	122,55	124,06
Smith (algoritmo guloso)	67,63	76,20	118,05
Smith (algoritmo genético)	59,65	74,56	106,73
Krishnaswamy (*rough sets*)	61,46	110,44	--
Algoritmo IBL	**33,98**	**58,32**	**106,07**

6.4 Considerações finais

Este capítulo apresentou o modelo de aquisição de conhecimento que tem como objetivo realizar predições de características de execução de aplicações paralelas. Esse modelo utiliza o algoritmo IBL, que realiza o aprendizado baseado em instâncias para definir aplicações similares a um ponto de consulta e, através do método dos vizinhos mais próximos ponderados pela distância, gerar um estimativa de um atributo da aplicação paralela. Experimentos foram realizados com diferentes cargas de trabalho empregando como atributo de saída o tempo de execução das aplicações paralelas. Através desses experimentos, pôde-se verificar a influência dos parâmetros do algoritmo na qualidade das predições.

Diferentes combinações dos parâmetros principais do algoritmo IBL, que são a vizinhança (k) e a largura da função gaussiana (t^2), foram avaliados. Conclui-se que a melhor combinação de parâmetros, quando o valor de t^2 é fixo nas predições, é $k = 5$ e $t^2 = 0,125$. Pôde-se verificar que à medida que o valor desses parâmetros aumenta, maior é o erro obtido com a predição. Para a maioria das cargas de trabalho, os resultados obtidos indicaram que valores de t^2 variáveis, utilizando a distância entre o ponto de consulta e o k-ésimo vizinho utilizado com largura da função gaussiana, não trouxeram benefícios para o algoritmo de predição; os erros médios obtidos para as cargas de trabalho CTC, SDSC95, SDSC96 e LANL são estatisticamente iguais quando utilizados valores de t^2 fixos e variáveis. A exceção ocorre com a carga de trabalho SDSC2000, na qual o valor de t^2 variável apresentou erros de predição estatisticamente maiores, quando comparados com os erros obtidos com a utilização de valores de t^2 fixos. Nesse contexto, conclui-se que é interessante utilizar valores de vizinhança e de largura da função gaussiana respectivamente iguais a 5 e $0,125$, e que valores de t^2 variáveis não trazem benefícios para a predição do tempo de execução de aplicações paralelas.

A implementação do algoritmo, realizada através de um conjunto de classes C++ e o gerenciador de banco de dados MySQL, mostrou-se estável e eficiente. A implementação permite a separação da base de experiências e do algoritmo de aprendizado, tornando possível que esses dois componentes do algoritmo possam estar localizados em computadores distintos. O tempo de resposta do algoritmo mostrou-se adequado, permitindo gerar predições com um bom desempenho computacional.

Os resultados obtidos com algoritmo IBL demonstraram que o seu desempenho de predição é superior aos 5 trabalhos prévios relacionados na literatura, que empregam cargas de trabalho idênticas. Esse fato, aliado à característica incremental na atualização de conhecimento, torna o algoritmo IBL bastante atrativo para a predição de características de execução das aplicações paralelas.

Devido à ausência de cargas de trabalho de cenários heterogêneos, a avaliação restringiu-se à utilização de traços de execução de ambientes paralelos homogêneos. Isso permitiu que os resultados obtidos fossem comparados com trabalhos prévios, mas não permitiu que o desempenho de predição do algoritmo fosse avaliado em cenários heterogêneos. A utilização do algoritmo IBL em ambientes heterogêneos pode ser realizada através de duas abordagens.

A primeira abordagem é a inclusão de características do ambiente de execução nos atributos de entrada, que são utilizados para definir o conjunto de aplicações similares ao ponto de consulta. Tal abordagem, já empregada no trabalho de Iverson *et al.* (1999), permite que execuções prévias das aplicações sejam selecionadas de acordo com o cenário na qual tais aplicações foram executadas. Atributos de entrada adicionais podem ser adicionados em cada instância, representando a potência computacional de cada EP.

Como proposto por Iverson *et al.* (1999), esses atributos adicionais podem ser obtidos através da execução de benchmarks específicos em cada EP. Da mesma forma, pode-se utilizar funções de distância distintas para os atributos que representam a capacidade computacional dos elementos de processamento. Uma questão não resolvida nessa primeira abordagem é o fato de que mais de uma tarefa pode estar alocada por elemento de processamento, situação comum em máquinas paralelas virtuais. Devido à multiprogramação, os elementos de processamento podem ser compartilhados por mais de uma tarefa e a carga pode variar durante a execução das aplicações, situação que não ocorre nos cenários de máquinas paralelas com políticas de escalonamento de compartilhamento de espaço.

Um segunda abordagem, mais atrativa para cenários heterogêneos e com variação de carga, é estimar a carga que uma aplicação submete ao sistema, tendo como unidade o tempo total de UCP utilizada pela aplicação. A utilização de UCP como atributo de saída, permite que seja isolada a influência da multiprogramação no atributo de saída. Por exemplo, se uma aplicação necessita de uma quantidade de 100 milhões de instruções para ser executada, tal valor não será alterado devido à carga do sistema. Embora não seja equivalente à predição do tempo final de execução, a predição do tempo de processamento, por exemplo em milhões de instruções, é interessante para o escalonamento em sistemas distribuídos. O próximo capítulo aborda a utilização das predições do algoritmo IBL em cenários homogêneos e heterogêneos, assim como a utilização do modelo de aquisição de comportamento de aplicações paralelas (vide Capítulo 5) para melhorar as decisões do software de escalonamento.

CAPÍTULO

7

Utilização de Características sobre Aplicações Paralelas no Escalonamento

7.1 Considerações iniciais

Este capítulo apresenta os algoritmos de escalonamento desenvolvidos, que utilizam o conhecimento obtido através dos modelos de aquisição descritos nos Capítulos 5 e 6.

O primeiro algoritmo de escalonamento, chamado de `BACKFILL-IBL`, é definido e avaliado considerando máquinas de processamento paralelo como cenário de execução. Esse algoritmo segue uma política de compartilhamento de espaço, de forma que cada EP é empregado sem multiprogramação local. Esse algoritmo utiliza predições sobre o tempo de execução de aplicações paralelas, de acordo com o modelo de predição descrito no Capítulo 6, para melhor utilizar os elementos de processamento disponíveis. O desempenho desse algoritmo é avaliado através de simulação, empregando os traços de execução previamente estudados e considerando as características originais do cenário de execução da cargas de trabalho (Senger *et al.*, 2004a). O algoritmo `BACKFILL-IBL` é descrito na Seção 7.2, assim como o simulador empregado para a avaliação de desempenho deste algoritmo, a metodologia de avaliação e os resultados obtidos.

O segundo algoritmo de escalonamento, chamado de `GAS` (*genetic algorithm scheduling*), emprega o conhecimento obtido pelas aplicações paralelas na utilização de recursos, com o objetivo de melhorar a atribuição de processos em um cenário distribuído. Esse algoritmo considera um cenário de execução heterogêneo, composto por máquinas com diferentes capacidades de processamento, memória e velocidades de acesso à dispositivos de armazenamento secundários, interligadas através de redes de diferentes capacidades de transferência de dados. O algoritmo `GAS` é avaliado através de simulação, empregando o modelo de simulação proposto por Mello & Senger (2004). A avaliação de desempenho considera um modelo de sistema distribuído

parametrizado através de aplicações sintéticas que medem as capacidades dos EPs envolvidos e das redes de comunicação. O algoritmo de escalonamento GAS, o modelo de simulação e os resultados obtidos com a simulação são descritos na Seção 7.3. As considerações finais deste capítulo são apresentadas na Seção 7.4.

7.2 Algoritmo de escalonamento BACKFILL-IBL

O escalonamento de aplicações paralelas e seqüenciais em máquinas verdadeiramente paralelas e em outros sistemas de memória distribuída é usualmente realizado atribuindo para cada aplicação um partição dos recursos disponíveis. Esses recursos são atribuídos de forma que ficam alocados exclusivamente para a aplicação, até que a aplicação seja finalizada. Assim, a multiprogramação ocorre de maneira espacial, se considerados todos os EPs do sistema, não existindo execução concorrente de processos no mesmo elemento de processamento. Nesse tipo de ambiente de execução, o escalonamento realizado através da disciplina FCFS (*first-come, first-served*) está longe de ser o algoritmo ideal. Ainda que escalonar as aplicações através de sua ordem de submissão ao sistema seja justo e determinístico para o usuário, a disciplina de escalonamento FCFS cria fragmentação dos recursos e gera uma baixa utilização do sistema (Feitelson & Rudolph, 1998). Embora existam soluções que teoricamente permitam um melhor desempenho, como o escalonamento cooperativo (Zhang *et al.*, 2000) e o particionamento dinâmico (Naik *et al.*, 1993), algoritmos que realizam *backfilling* são comumente adotados em sistemas paralelos de memória distribuída.

A técnica de *backfilling*, no contexto de escalonamento em máquinas paralelas, foi primeiramente apresentada por Lifka (1995). Essa técnica implementa uma melhoria no algoritmo baseado na disciplina FCFS, que permite que aplicações que não estão no início da fila do escalonador sejam executadas antes das aplicações que foram submetidas anteriormente, visando diminuir a fragmentação dos recursos. A idéia é permitir que aplicações com tempo de execução mais reduzido saltem à frente das demais aplicações que estão na fila do escalonador, com a restrição que a execução das aplicações com tempo de execução mais reduzido não cause atrasos na execução das demais aplicações da fila.

Os algoritmos que empregam técnicas de *backfilling* são classificados como agressivos ou conservadores (Aida, 2000). No escalonamento agressivo, as aplicações com tempo de execução menores são movidas para o início da fila apenas se elas não interferem a execução da aplicação que está no início da fila. Já no escalonamento conservador, uma aplicação com menor tempo de execução é movida para o início da fila apenas se essa aplicação não atrase a execução de todas as aplicações que estão à sua frente na fila.

Algoritmos que realizam *backfilling* necessitam de estimativas sobre o tempo de execução das aplicações. Essas estimativas são usadas para determinar se uma aplicação é suficiente

pequena para saltar na fila à frente de outras aplicações e ser executada sem intervir no tempo de execução das demais aplicações existentes na fila. Estimativas do tempo de execução são comumente providas pelos usuários que submetem as aplicações paralelas ao sistema. As estimativas providas pelos usuários não são confiáveis, desde que os usuários tendem a superestimar o tempo de execução das aplicações submetidas, pois o escalonador finaliza as aplicações que usam tempo de sistema maior que a sua estimativa (Mu'alem & Feitelson, 2001).

O primeiro algoritmo de escalonamento descrito neste capítulo é o BACKFILL-IBL. Esse algoritmo emprega a técnica de *backfilling*, baseando-se nas estimativas de execução das aplicações providas pelo algoritmo IBL (vide Capítulo 3). A idéia é, ao invés de utilizar as estimativas de execução providas pelos usuários, realizar predições com o algoritmo IBL, seguindo o modelo de aquisição de conhecimento descrito no Capítulo 6, e utilizar tais predições como estimativas do tempo de execução das aplicações. Tais predições envolvem a utilização dos mesmos traços de execução obtidos previamente como base de experiências e o algoritmo IBL para obter estimativas do tempo de execução. Assim, busca-se verificar a utilidade da aquisição e utilização do conhecimento sobre o tempo de execução nas decisões do escalonador.

Experimentos são conduzidos para avaliar o desempenho do algoritmo BACKFILL-IBL, utilizando um simulador orientado a traços de execução. O simulador utilizado, definido inicialmente por Downey (1997), utiliza uma fila de escalonamento compartilhada entre os elementos de processamento do sistema e o algoritmo de escalonamento FCFS. Esse simulador é modificado e três novos algoritmos de escalonamento que empregam a técnica de *backfilling* agressivo são implementados: BACKFILL-REAL, BACKFILL-USUARIO e o algoritmo BACKFILL-IBL. O algoritmo BACKFILL-REAL utiliza os tempos de execução reais gravados nos traços de execução, enquanto os algoritmos BACKFILL-USUARIO e BACKFILL-IBL utilizam respectivamente as estimativas do tempo de execução providas pelos usuários e pelo algoritmo IBL.

Embora seja irreal em situações práticas, o algoritmo de escalonamento BACKFILL-REAL serve como um limite superior no desempenho da atividade de escalonamento, desde que o conhecimento sobre o tempo de execução das aplicações é exato. O algoritmo de escalonamento BACKFILL-USUARIO representa o escalonamento em sistemas paralelos reais onde as estimativas do usuário são empregadas, como o software de escalonamento do computador IBM-SP2 (Lifka, 1995). Assim, três níveis de conhecimento sobre aplicações paralelas são utilizados: nenhum conhecimento sobre o tempo de execução (FCFS), conhecimento imperfeito sobre o tempo de execução (BACKFILL-USUARIO e BACKFILL-IBL) e conhecimento perfeito (BACKFILL-REAL). A seguir, são descritos os detalhes do simulador aprimorado e os resultados obtidos com as simulações.

7.2.1 Modelo de simulação

O simulador utilizado, definido inicialmente em (Downey, 1997), permite modelar um sistema paralelo onde uma quantidade definida de elementos de processamento são organizados em partições, sendo que o acesso aos processadores de cada partição é controlado utilizando uma fila de escalonamento. Inicialmente, apenas o algoritmo de escalonamento baseado na disciplina de atendimento FCFS foi implementado nesse simulador por Downey (1997).

Esse simulador é modificado para que sejam incorporados outros algoritmos de escalonamento, assim como outras medidas de avaliação de desempenho do sistema.

O simulador utiliza a simulação orientada a eventos. Assim, dois eventos são definidos: a chegada e o término das aplicações. Quando o simulador está processando a chegada de uma aplicação, verifica se existem EPs disponíveis para atender a essa aplicação. Caso afirmativo, a aplicação é iniciada e um evento do tipo término é escalonado para o instante de tempo correspondente a soma do tempo atual de simulação mais o tempo real de execução da aplicação, que é obtido através do traço de execução. Caso contrário, em que não existam elementos de processamento disponíveis, a aplicação é inserida na fila de escalonamento onde aguarda a sua vez para ser executada. A forma com que essa fila de escalonamento é gerenciada depende do algoritmo de escalonamento utilizado.

Embora as modificações incluídas no simulador permitam que as simulações sejam executadas utilizando distribuições de probabilidade para o tempo de execução, quantidade de elementos de processamento requisitados e intervalos de chegada, as simulações são realizadas utilizando diretamente as informações gravadas nos traços de execução, caracterizando uma simulação orientada a traços de execução (*trace-driven*). Isso permite que os traços de execução previamente analisados sirvam como entrada para o simulador, definindo situações reais de carga de trabalho de máquinas paralelas e que as predições do tempo de execução feitas sejam utilizadas.

A Tabela 7.1 ilustra o formato do arquivo de entrada, que representa o traço de execução, sendo que cada linha de entrada corresponde a uma aplicação submetida ao sistema. Cada linha do arquivo traz informações sobre a aplicação paralela, que são o seu identificador, a quantidade de EPs requisitados, o tempo da submissão, o tempo de execução da aplicação observado previamente e as estimativas do usuário e do algoritmo IBL. Esses arquivos de entrada são gerados, um para cada carga de trabalho utilizada (SDSC95, SDSC96, SDSC2000, CTC e LANL), com as predições realizadas com o algoritmo IBL, empregando os melhores parâmetros observados na avaliação do modelo de aquisição de conhecimento proposto no Capítulo 5.

A simulação é conduzida utilizando os atributos providos no arquivo de entrada, sendo que o intervalo de chegada é obtido pelo dia e pelo tempo de chegada das aplicações. Ao fim da simulação, têm-se um arquivo de saída, cujo formato é descrito na Tabela 7.2. Através desse arquivo, podem ser obtidos, para cada aplicação, o tempo que a aplicação experimentou na

Tabela 7.1: Formato do arquivo de entrada para o simulador

Atributo	Descrição
1	identificador da aplicação
2	número de EPs requisitados
3	dia da submissão
4	tempo de chegada
5	tempo de execução da aplicação (real)
6	estimativa do usuário para o tempo de execução da aplicação
7	tempo de execução da aplicação (predição IBL)

fila, e os tempos de execução e de residência no sistema. Pode-se obter também o *slowdown* experimentado pela aplicação, além do *slowdown* limitado. O *slowdown* limitado (*bounded slowdown*) elimina a ênfase dada em aplicações com tempo de execução mais reduzido, de acordo com a seguinte equação (Equação 7.1):

$$SL = max(\frac{T_w + T_r}{max(T_r, \tau)}, 1) \tag{7.1}$$

onde SL é o *slowdown* limitado, T_r denota o tempo de execução da aplicação, T_w denota o tempo de espera da aplicação na fila e τ corresponde ao valor de limiar empregado. Os valores de limiar que são definidos no simulador estão descritos na Tabela 7.2.

Tabela 7.2: Formato do arquivo de saída do simulador

Atributo	Descrição
1	número da saída
2	identificador da aplicação
3	número de EPs
4	relógio da simulação
5	tempo que a aplicação experimentou na fila
6	tempo de execução da aplicação
7	tempo de residência da aplicação
8	*slowdown* da aplicação
9	*slowdown* da aplicação limitado em 10s
10	*slowdown* da aplicação limitado em 1m
11	*slowdown* da aplicação limitado em 10m

Além do arquivo de saída, o simulador fornece um arquivo que representa as médias das medidas de desempenho de todas aplicações que foram executadas no sistema. Os atributos desse arquivo estão descritos na Tabela 7.3. Através do arquivo de médias, podem-se analisar as médias de desempenho dos algoritmos de escalonamento. Os resultados obtidos através da utilização do simulador são descritos na próxima subseção.

Tabela 7.3: Formato do arquivo de médias do simulador

Atributo	Descrição
1	número da saída
2	tempo médio na fila
3	tempo médio de resposta
4	utilização média do sistema
5	*slowdown* médio
6	*bounded slowdown* médio (10s)
7	*bounded slowdown* médio (1m)
8	*bounded slowdown* médio (10m)

7.2.2 Resultados obtidos

Para avaliar os algoritmos de escalonamento `BACKFILL-REAL`, `BACKFILL-USUARIO`, `BACKFILL-IBL` e `FCFS`, são empregadas as cargas de trabalho previamente utilizadas, o simulador descrito anteriormente e um valor de limiar para o *slowdown* limitado igual a 10 segundos (Aida, 2000; Mu'alem & Feitelson, 2001). As médias dos valores de *slowdown* obtidos nas simulações, para cada carga de trabalho, são comparadas através do teste Z, considerando um nível de confiança de 95%.

A Figura 7.1 apresenta graficamente os valores de *slowdown* limitado obtidos e a Tabela 7.4 apresenta as médias obtidas. O algoritmo `BACKFILL-USUARIO` não é simulado para as cargas de trabalho SDSC95 e SDSC96, pois tais cargas de trabalho não possuem gravados em seus traços de execução as estimativas providas pelos usuários. Esses resultados correspondem às simulações das máquinas paralelas correspondentes ao traço de execução relacionado. A quantidade de elementos de processamento utilizados para cada carga de trabalho está de acordo com os valores descritos através da Tabela 4.3.

Através desses resultados, pode-se observar que o algoritmo `BACKFILL-REAL` produz menores valores de *slowdown* limitado para todas as cargas de trabalho. A utilização do tempo de execução real das aplicações, permite que sejam alcançados os melhores valores de desempenho. Isso indica que quanto melhor forem as estimativas do tempo de execução, melhor é o desempenho dos algoritmos de escalonamento que utilizam a técnica de backfilling.

O algoritmo de escalonamento `BACKFILL-IBL` produz melhores resultados, quando comparado com os algoritmos `FCFS` e `BACKFILL-USUARIO`, em todas as cargas de trabalho analisadas. As melhorias do algoritmo `BACKFILL-IBL`, quando comparado com o algoritmo FCFS são respectivamente iguais a $77,83\%$, $61,47\%$, $41,03\%$ e $37,26\%$. Quando comparado aos algoritmos `BACKFILL-USUARIO`, as melhorias do algoritmo `BACKFILL-IBL` no desempenho do sistema para as cargas de trabalho SDSC2000, CTC e LANL são respectivamente iguais a $22,86\%$, $7,84\%$ e $26,79\%$. A análise estatística dos resultados, utilizando o teste Z para

Tabela 7.4: Resultados das simulações expressos pelo *slowdown* limitado em $10s$

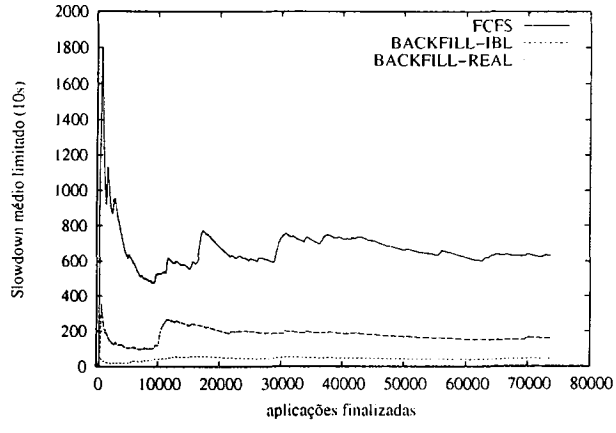
Carga de Trabalho	**FCFS**	**Algoritmos com *backfilling***		
		REAL	**USUARIO**	**IBL**
SDSC95	1486,87	88,66	–	329,58
SDSC96	683,02	32,00	–	263,14
SDSC2000	360,40	17,08	33,12	25,55
CTC	6,97	1,18	4,46	4,11
LANL	127,53	36,73	108,79	80,00

realizar a comparação entre as médias com um nível de confiança de 95% (vide Apêndice A, Tabelas A.51, A.52, A.53, A.54 e A.55), permite concluir que o algoritmo `BACKFILL-IBL` apresenta menores resultados de *slowdown* que o algoritmo `IBL-USUARIO`, para todas as cargas de trabalho consideradas. Conseqüentemente, considerando o mesmo teste estatístico, o algoritmo `BACKFILL-IBL` apresenta melhor desempenho quando comparado ao algoritmo de escalonamento `FCFS`.

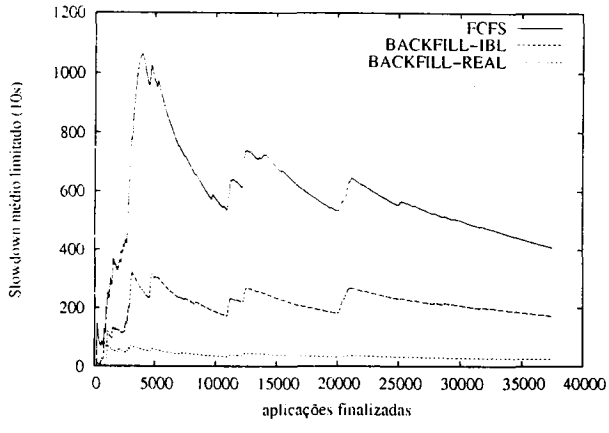
7.3 Algoritmo de escalonamento GAS

O objetivo do algoritmo de escalonamento GAS (*genetic algorithm scheduling*) é encontrar uma melhor solução para a atribuição de uma aplicação paralela em um ambiente de execução distribuído e heterogêneo. Como entradas para o algoritmo têm-se a aplicação paralela, representada pela sua quantidade de EPs solicitados, seu padrão de comportamento na utilização de recursos (o que é definido empregandó o modelo de aquisição descrito no Capítulo 5) e a predição de sua ocupação, em milhões de instruções, obtida através do algoritmo IBL (vide Capítulo 6). Além dessas informações, o algoritmo conta com as características do ambiente distribuído, como a capacidade de processamento de cada máquina, os custos de comunicação entre os EPs do ambiente distribuído e a carga atual de cada EP. O algoritmo de escalonamento emprega tais informações para buscar quais são os EPs mais adequados para atender aos requisitos das aplicações paralelas.

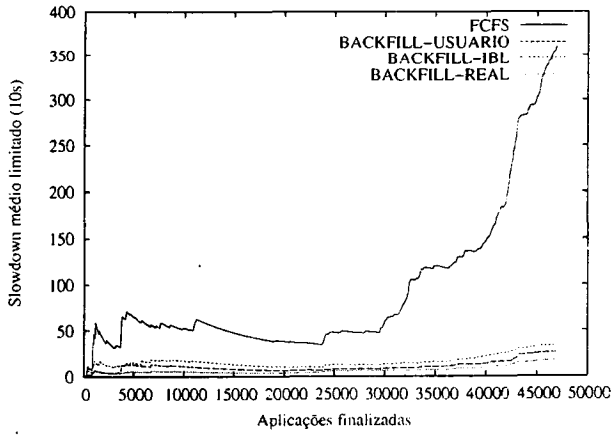
Devido à quantidade de fatores existentes, inerentes tanto ao ambiente de execução quanto às características das aplicações, o espaço de busca torna-se elevado. Assim, é comum a utilização de heurísticas para encontrar uma solução aceitável para o problema, pois muitas vezes a solução ótima não pode ser encontrada em tempo hábil aos requisitos do software de escalonamento, que deve minimizar os atrasos impostos à execução das aplicações durante suas atividades. Para a solução desse problema, a estratégia de busca utilizada emprega técnicas de algoritmos genéticos (de P. Braga *et al.*, 2000).



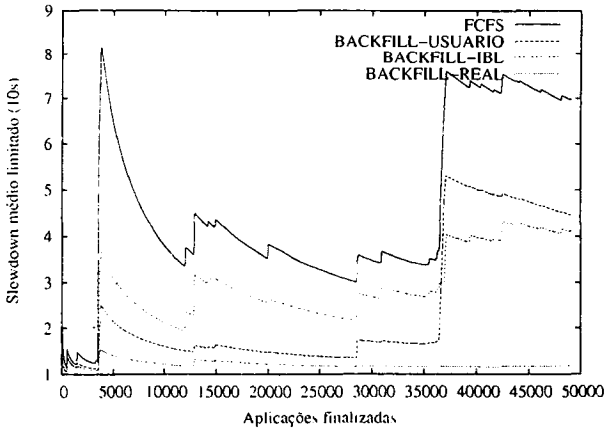
(a) SDSC95



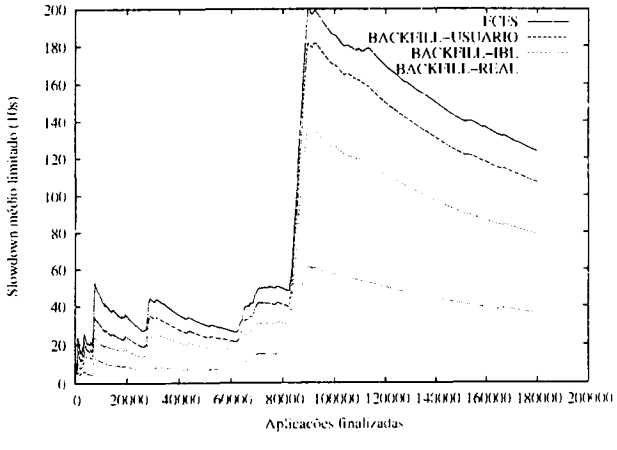
(b) SDSC96



(c) SDSC2000



(d) CTC



(e) LANL

Figura 7.1: Desempenho dos algoritmos de escalonamento avaliados

Algoritmos genéticos são baseados em técnicas de busca probabilísticas usadas para a exploração de grandes espaços de busca (Cortes, 2004). Esses algoritmos são um caso particular de algoritmos evolutivos, que têm inspiração na teoria de Darwin. Essa teoria propõe que indivíduos mais fortes de uma população interagem entre si propagando seus genes e gerando descendentes mais aptos às condições de vida. Essas técnicas exploram os conceitos da biologia para melhorar a eficiência da busca aleatória.

Algoritmos genéticos definem inicialmente um conjunto de soluções aleatórias para o problema, sendo que cada solução é chamada de indivíduo e esse conjunto de soluções é chamado de população. O algoritmo evolui através de gerações, sendo que cada geração consiste na avaliação da qualidade de cada indivíduo da população para representar a solução para o problema. A aptidão de cada indivíduo é calculada através de uma função de aptidão (*fitness*). A função de aptidão define quais indivíduos sobreviverão e serão combinados, emulando o processo de cruzamento e reprodução entre seres vivos, para produzir uma nova geração de indivíduos. É possível também realizar mutações nesses indivíduos visando melhor representar o processo evolutivo.

As informações principais sobre as aplicações e sobre o ambiente de execução utilizadas pelo algoritmo de escalonamento GAS são:

- cap_i: vetor com a capacidade de processamento dos EPs em milhões de instruções por segundo (MIPS);
- l_i: vetor de carga das máquinas;
- c_i: ocupação de um processo, expresso em milhões de instruções. Ao invés da ocupação da aplicação, utiliza-se o valor de ocupação de cada processo da aplicação paralela;
- r_{ij}: vetor bidimensional dos custos de comunicação entre os EPs;
- $atrasoMemoria(i, u)$: função que define o atraso imposto pela utilização de memória para um elemento de processamento i e para uma memória usada igual a u KBytes. Essa função define o *slowdown* imposto às aplicações à medida que a memória principal do sistema é preenchida e permite verificar o atraso imposto às aplicações pelo uso de memória de troca.

As informações cap_i, r_{ij} e $atrasoMemoria(i, u)$ são obtidas respectivamente através da execução de programas computacionais específicos, descritos na próxima seção.

O cálculo do índice de carga realizado pelo algoritmo GAS emprega a predição da ocupação das aplicações. Considera-se o algoritmo IBL como preditor da ocupação de um processo e adiciona-se a essa predição o erro obtido nos experimentos descritos no Capítulo 6, que, em média, corresponde a 50%. O erro de predição do algoritmo IBL é modelado como uma

variável aleatória, cujo valor é obtido através de uma função de distribuição normal. A utilização da predição da ocupação de uma aplicação permite ponderar a contribuição de cada aplicação na carga do elemento de processamento. A carga de um elemento de processamento i, que possui n processos em execução, é definida de acordo com a seguinte equação:

$$l_i = \frac{\sum_j^n c_j}{cap_i} * atrasoMemoria(i.u) \tag{7.2}$$

A solução para o problema, representada por um indivíduo, corresponde a um vetor t, cujo tamanho é igual à quantidade de EPs solicitados pela aplicação. Cada elemento desse vetor corresponde a um número índice inteiro e representa um EP que pode ser utilizado. Como o sistema é heterogêneo e mais de um processo pode ser alocado por EP, o vetor admite repetição de valores. Por exemplo, um vetor $t = \{1, 1, 2, 2\}$ corresponde a uma possível solução para uma aplicação que tem 4 processos. Nesse caso, dois processos da aplicação serão escalonadas para o EP identificado pelo índice 1 e os demais processos serão escalonadas para o EP índice 2.

A função de aptidão dos indivíduos é composta por dois termos. O primeiro termo denota a estimativa do custo computacional de atribuir um processo t ao elemento de processamento i. Para isso emprega-se o vetor de carga l_i. À medida que a função de aptidão percorre os elementos do vetor t, que indicam quais os computadores são empregados pela solução corrente, o custo de processamento da solução é atualizado com a carga da aplicação que será escalonada. Sendo c_j a carga que uma aplicação j adiciona a um elemento de processamento j, o custo em relação ao processamento $C_{processamento}$ para um vetor de tamanho n é igual a (Equação 7.3):

$$C_{processamento}(t) = \sum_i^n (\frac{c_j}{cap_{t_i}} + l_{t_i}) \tag{7.3}$$

onde cap_{t_i} é a capacidade de processamento da máquina, medida através da aplicação `cpu`. Nessa equação, o valor de n corresponde ao número de EPs solicitados pela aplicação a ser escalonada. Durante o processamento de cada elemento do vetor t, o vetor de cargas é atualizado da seguinte maneira:

$$l_{t_i} = l_{t_i} + \frac{c_j}{cap_{t_i}} \tag{7.4}$$

Isso permite que o custo de atribuir mais de um processo por elemento de processamento seja calculado também.

O segundo termo considera o custo que será adicionado ao realizar a comunicação entre os processos. Sendo $r_{i,j}$ a estimativa de comunicação entre os computadores, o custo de rede C_{rede} para uma solução representada por um vetor t de tamanho n é igual a:

$$C_{rede}(t) = \sum_{i=0}^{n} \sum_{j=i+1}^{n} r_{t_i,t_j} \tag{7.5}$$

Tais custos são ponderados de acordo com o padrão de comportamento da aplicação na utilização de recursos, obtido através do modelo de aquisição de conhecimento descrito no

Capítulo 5. Define-se duas variáveis, α e β, como sendo respectivamente o percentual de tempo em que a aplicação visita estados de processamento e comunicação. Por exemplo, uma aplicação identificada com $\alpha = 0,1$ e $\beta = 0.9$ encontra-se realizando processamento 10% do tempo total de sua execução e os demais 90% realizando comunicação entre seus processos. Assim, a função de aptidão f_t para um vetor t é igual:

$$f(t) = \alpha \times \frac{1}{(C_{processamento}(t) + e)} + \beta \times \frac{1}{(C_{rede}(t) + e)} \tag{7.6}$$

sendo e um parâmetro fixo que previne a divisão por zero.

O custo computacional do algoritmo GAS é proporcional à quantidade de EPs solicitados pela aplicação. A quantidade de EPs requisitados pela aplicação define o tamanho do vetor t, que serve como entrada para o algoritmo genético. À medida que o tamanho do vetor cresce, cresce também o tempo computacional para a geração de indivíduos no algoritmo genético e para a aplicação da função de aptidão.

7.3.1 Modelo de simulação

O desempenho do algoritmo de escalonamento GAS é avaliado através de experimentos de simulação. O modelo de simulação adotado é o UniMPP (*Unified modeling for predicting performance*), definido por Mello & Senger (2004). Esse modelo permite avaliar ambientes distribuídos heterogêneos e o tempo de resposta de aplicações paralelas. Esse modelo une as características de consumo de UCP dos modelos de (Amir *et al.*, 2003) e (Mello, 2002), do tempo utilizado na transmissão de mensagens modelado em Culler *et al.* (1993) e Sivasubramaniam *et al.* (1994), o volume de mensagens e as distribuições de probabilidades de geração de mensagens de Chodnekar (1997) e Vetter & Mueller (2003).

Nesse modelo, todo elemento de processamento PE_i é composto pela sêxtupla $\{pc_i, mm_i, vm_i, dr_i, dw_i, lo_i\}$, onde: pc_i é a capacidade computacional total de cada computador, medida em instruções por unidade de tempo, mm_i é a capacidade total de memória principal, vm_i é a capacidade total da memória virtual, dr_i é o *throughput* de leitura do disco rígido, dw_i é o *throughput* de escrita no disco rígido e lo_i é o tempo consumido no envio e recebimento de mensagens. Da mesma forma, os processos das aplicações paralelas são representadas pela sêxtupla $\{mp_j, sm_j, pdfdm_j, pdfdr_j, pdfdw_j, pdfnet_j\}$, onde: mp_j representa o consumo de processamento, sm_j é a quantidade de memória estática alocada pelo processo, $pdfdm_j$ é a distribuição de probabilidades para a ocupação dinâmica de memória, $pdfdr_j$ é a distribuição de probabilidades para leitura em arquivos, $pdfdw_j$ é a distribuição de probabilidades para escrita em arquivos, $pdfnet_j$ é a distribuição de probabilidades para envio e recebimento de mensagens

O simulador para esse modelo[1] é implementado na linguagem Java, que agrega recursos para a implementação de diferentes algoritmos de escalonamento. Juntamente com o simulador,

[1] SchedSim - disponível no site http://www.icmc.usp.br/~mello/outr.html.

é definido um conjunto de aplicações sintéticas, que permitem parametrizar as características do ambiente distribuído. Essas *aplicações* são:

- `cpu`: mede a capacidade de processamento do EP, em milhões de instruções por segundo (MIPS);
- `disc`: mede a velocidade de escrita e leitura em disco;
- `memo-and-swap`: mede o atraso (*slowdown*) imposto ao sistema à medida que a memória é utilizada;
- `net`: mede o desempenho da comunicação entre duas máquinas.

O algoritmo `GAS` é implementado no simulador como um nova classe da linguagem Java. O simulador permite a definição e implementação de diferentes algoritmos de escalonamento, empregando classes abstratas. O algoritmo `GAS` utiliza o pacote de classes `GA`, desenvolvido por Jeff Smith[2]. O pacote `GA` implementa as classes `GAFloat` e `GAString` que definem respectivamente a codificação de indivíduos através de números reais e seqüências de caracteres. O algoritmo `GAS` é implementado através da classe `GAString`.

Os parâmetros de controle do algoritmo genético são: população inicial de 100 indivíduos; probabilidade de *crossover* igual a $0,7$; e probabilidade de mutação igual a $0,01$. Esses parâmetros são selecionados através de experimentação prática e da análise dos resultados, visando maximizar a precisão das soluções e minimizar o tempo gasto com o algoritmo genético.

7.3.2 Resultados obtidos

Experimentos são conduzidos para avaliar o algoritmo de escalonamento GAS, considerando características de um ambiente distribuído heterogêneo real, composto por 10 elementos de processamento interligados através de redes distintas. Os elementos de processamento são representados por computadores pessoais, com a mesma arquitetura, mas com diferenças em suas capacidades de processamento. As capacidades de processamento de cada computador são obtidas através da execução das aplicações sintéticas `cpu`, `disc`, `memo-and-swap` e `net`.

Os resultados obtidos com a utilização dessas aplicações no ambiente de execução estão descritos na Tabela 7.5. Pode-se notar a heterogeneidade dos recursos pelos valores obtidos para a capacidade de processamento, memória e velocidade de escrita e leitura em disco.

Tal heterogeneidade existe também no meio de comunicação entre os computadores. Os computadores estão situados em 4 sub-redes, nomeadas como $R1$, $R2$, $R3$ e $R4$. Os EPs 1, 2 e 3 situam-se em uma rede Ethernet de 1 GBit/s $R1$; os EPs 4, 5 e 6 são interligados através de uma *switch* de 100 MBits/s na rede $R2$, assim como os EPs 7 e 8, situados na rede $R3$, e os EPs

[2] jeff@SoftTechDesign.com

9 e 10, que são situados na rede $R4$. A rede $R1$ emprega como roteador o computador 1 para conectar-se à rede $R2$; a rede $R2$ conecta-se à rede $R3$ através da rede interna do ICMC/USP. Essas redes são interligadas à rede $R4$ através da Internet. Os resultados obtidos com a execução da aplicação `net` considerando essa configuração das redes de comunicação são descritos na Tabela 7.6. Esse valores correspondem ao atraso para envio de uma mensagem de tamanho igual a 32 KBytes pela rede de comunicação.

Tabela 7.5: Cenário distribuído

EP	**MIPS**	**Memória** (MB)	**Troca** (*swap*)	**Disco** (MBytes/s)	
				Leitura	**Escrita**
1	1135,31	1024	1024	76,33	66,85
2	1145,86	1024	1024	76,28	65,55
3	1148,65	1024	1024	75,21	66,56
4	187,54	256	512	17,82	16,60
5	313,38	256	512	10,25	6,70
6	151,50	64	512	7,30,	7,21
7	1052,87	256	512	14,42	15,74
8	1052,87	256	512	14,42	15,74
9	350,00	512	512	10,25	6,70
10	350,00	512	512	10,25	6,70

Tabela 7.6: Custo de comunicação em segundos

	1	2	3	4	5	6	7	8	9	10
1	–	0,000040	0,000040	0,000087	0,000074	0,000136	0,000404	0,000404	0,063291	0,063291
2		–	0,000040	0,000081	0,000074	0,000173	0,000440	0,000440	0,060240	0,060240
3			–	0,000081	0,000074	0,000173	0,000440	0,000440	0,060240	0,060240
4				–	0,000161	0,000161	0,000442	0,000442	0,064020	0,064020
5					–	0,000145	0,000420	0,000420	0,064000	0,064000
6						–	0,000492	0,000492	0,064000	0,064000
7							–	0,000092	0,260563	0,260563
8								–	0,260563	0,000092
9									–	0,000092
10										–

O modelo de Feitelson é empregado como carga de trabalho (Feitelson & Jette, 1997; Feitelson & Rudolph, 1998; Feitelson & Weil, 1998; Lo *et al.*, 1998; Ghare & Leutenegger, 1999; Talby *et al.*, 1999; Aida, 2000; Mu'alem & Feitelson, 2001; Feitelson, 2001). Esse modelo é baseado na análise de seis traços de execução dos seguintes ambientes de produção:

- 128-node iPSC/860 at NASA Ames;

- 128-node IBM SP1 at Argonne;
- 400-node Paragon at SDSC;
- 126-node Butterfly at LLNL;
- 512-node IBM SP2 at CTC;
- 96-node Paragon at ETH, Zurich.

Desse modelo, são obtidas distribuições de probabilidade para o intervalo de chegada das aplicações paralelas, da quantidade de elementos de processamento requisitados por aplicação e do tempo de execução das aplicações. Além de sintetizar a carga imposta pelas aplicações, esse modelo define o intervalo de chegada através de uma distribuição exponencial de média igual a 1.500 segundos. A vantagem da utilização desse modelo é a possibilidade de variar a quantidade de elementos de processamento requisitados por aplicação. A carga de trabalho é gerada com aplicações paralelas requisitando um número máximo de 8 EPs.

Os resultados obtidos com o algoritmo GAS são comparados com 7 algoritmos de escalonamento e balanceamento de carga: DPWP, DPWP-modificado, Random, Disted, Lowest, Global e TLBA. Esses algoritmos são descritos a seguir.

O algoritmo DPWP (*Dynamic Policy Without Preemption*) realiza o escalonamento de processos em aplicações paralelas considerando um cenário de execução distribuído e heterogêneo (Araújo *et al.*, 1999a,b). Esse algoritmo permite o escalonamento de aplicações desenvolvidas em PVM, MPI e CORBA, de maneira que os detalhes envolvidos com a atribuição de processos são supervisionados pelo software de escalonamento AMIGO (Souza, 2000). O índice de carga utilizado nesse algoritmo é o tamanho da fila de cada EP. Através desse índice, define-se a carga de um EP de acordo com a razão entre a quantidade de processos em execução e a capacidade computacional do EP. A capacidade computacional do EP é medida através de *benchmarks* específicos (Souza, 2000; Santos, 2001). O algoritmo de escalonamento DPWP utiliza os índices de cargas para a criação de uma lista ordenada dos EPs. Os processos da aplicação paralela são atribuídas aos EPs dessa lista, de maneira circular.

Uma versão modificada do algoritmo DPWP, chamado de DPWP-modificado, é também definido e implementado no simulador. Essa versão emprega como índice de carga, ao invés do tamanho da fila de processos do EP, a ocupação dos processos para encontrar os EPs menos carregados do ambiente (Equação 7.3). Assim, o índice de carga da versão modificada é igual a razão entre a soma da ocupação dos processos em execução e a capacidade computacional do EP, de forma que a ocupação dos processos é obtida através do algoritmo IBL.

Os algoritmos Disted, Global, Lowest e Random são definidos no trabalho de Zhou & Ferrari (1987). Nesse trabalho, esses algoritmos têm como objetivo o balanceamento de cargas e são definidos através de dois componentes principais: o LIM (*Load information*

manager) e o LBM (*Load balance manager*). O primeiro é responsável pela política de informação e monitora a cargas dos computadores com o objetivo de calcular os índices de carga. O último define a política de localização, que tem como objetivo localizar o melhor computador do ambiente para qual os processos sejam atribuídos. A forma pela qual tais componentes realizam suas tarefas permite definir os diferentes algoritmos. Esses algoritmos diferem dos algoritmos de escalonamento pois estão orientados para realizar o balanceamento de carga, de forma que não existe um software escalonador global para o qual as aplicações são submetidas. De fato, cada EP deve gerenciar localmente os processos das aplicações que chegam a ele, iniciando-as localmente ou definindo a forma pela qual um outro EP será localizado para executar processos. Esses algoritmos estão apresentados nos próximos parágrafos, de acordo com a descrição provida por de Mello (2003).

No algoritmo `Disted`, os EPs calculam periodicamente seus índices de carga e caso o índice de carga atual seja diferente do último índice de carga calculado, o índice mais atual é enviado para todos os LIMs dos computadores do sistema. Quando um novo processo é submetido ao sistema, o LBM emprega as informações do LIM local ao EP para encontrar um computador mais apto a receber esse processo, que é aquele que possui o menor índice de carga. Se a carga de todos os EPs estiver acima de um limite pré-estabelecido, o processo é executado no EP que recebeu o processo.

O algoritmo `Global` define um LIM mestre que recebe todos os índices de carga do sistema, enquanto os demais computadores executam LIMs escravos. Esse escravos calculam os índices de carga localmente e enviam esses índices para o mestre. Tal ação ocorre quando os escravos percebem mudanças significativas na carga local. Quando um processo é submetido ao ambiente, o LBM centraliza as decisões, analisando os índices de carga e definindo em qual EP o processo deve ser iniciado.

Um LBM e um LIM mestre são definidos no algoritmo `Central` e centralizam as decisões de escalonamento. O LIM mestre recebe as informações providas pelos LIMs locais aos EPs do ambiente e o LBM mestre recebe as submissões dos processos para alocá-los nos EPs.

O algoritmo `Lowest` tem como objetivo realizar o balanceamento de carga minimizando a quantidade de mensagens entre os componentes do algoritmo. Quando um processo é submetido ao ambiente, o LIM do EP que recebeu a requisição define um conjunto limitado de LIMs remotos. A carga dos EPs desse conjunto é recebida e o EP mais ocioso é eleito para receber o processo.

O algoritmo `Random` não emprega informações sobre a carga do sistema para a tomada de decisões. Quando um processo é submetido ao ambiente de execução, o algoritmo seleciona um EP aleatoriamente. O índices de carga utilizados pelos algoritmos `Lowest`, `Disted` e `Global` é calculado de acordo com o número de processos na fila de execução. Zhou & Ferrari

(1987) observam que os algoritmos Lowest, Disted e Global apresentam desempenho similares e o algoritmo Random apresenta o pior resultado, e indicam o algoritmo Lowest para cenários distribuídos.

O algoritmo TLBA (*Tree Load Balancing Algorithm*) visa o balanceamento de cargas em sistemas distribuídos heterogêneos e com possibilidade de crescimento em escala (Mello, 2003; Mello *et al.*, 2003). Esse algoritmo cria uma topologia de interconexão lógica entre os EPs em formato de árvore e realiza migração de processos para melhorar o balanceamento de cargas no sistema.

A Figura 7.2 ilustra os desempenhos dos algoritmos de escalonamento GAS, Random, Disted, Global, Lowest, DPWP e DPWP-modificado para um conjunto de aplicações variando entre 50 e 1000. Pode-se observar que o algoritmo de escalonamento GAS apresenta o melhor desempenho, expresso pelos menores tempos de resposta, para os números de aplicações experimentados. Isso ressalta a importância de considerar o conhecimento sobre as aplicações na atribuição de processos no ambiente distribuído.

O pior desempenho é observado para o algoritmo Random. Isso deve-se ao fato de que esse algoritmo não emprega informações sobre a carga dos EPs e sobre as aplicações paralelas submetidas. Os algoritmos DPWP, Lowest, Disted, Global e TLBA, que dizem respeito apenas às características de carga do ambiente, produzem tempos médios de resposta maiores que o algoritmo Random e piores que o algoritmo GAS. O algoritmo DPWP-modificado apresenta resultados piores que o algoritmo DPWP. Isso indica que, embora o índice de carga do algoritmo DPWP-modificado empregue o conhecimento sobre o tempo de execução das aplicações, a estratégia de atribuição de processos sem distinção dos recursos mais apropriados penaliza os tempos de resposta experimentados pelas aplicações.

7.4 Considerações Finais

Este capítulo apresentou dois algoritmos de escalonamento, denominados BACKFILL-IBL e GAS, que utilizam conhecimento sobre as aplicações paralelas em suas decisões. Esses algoritmos foram avaliados através de simulação e os resultados obtidos, considerando diferentes cenários de execução, foram descritos.

O primeiro cenário de execução considera máquinas com processamento paralelo homogêneas, que podem ser formadas por uma máquina verdadeiramente paralela ou por um arranjo (*cluster*) de estações de trabalho e computadores pessoais, gerenciados exclusivamente por um escalonador com política de compartilhamento de espaço. Nesse cenário, define-se o algoritmo BACKFILL-IBL, que corresponde a uma melhoria no algoritmo BACKFILL-USUARIO, comumente adotado em cenários de execução similares. Através de simulação, mostrou-se que as predição sobre o tempo de execução de aplicações paralelas e sua utilização com o algoritmo

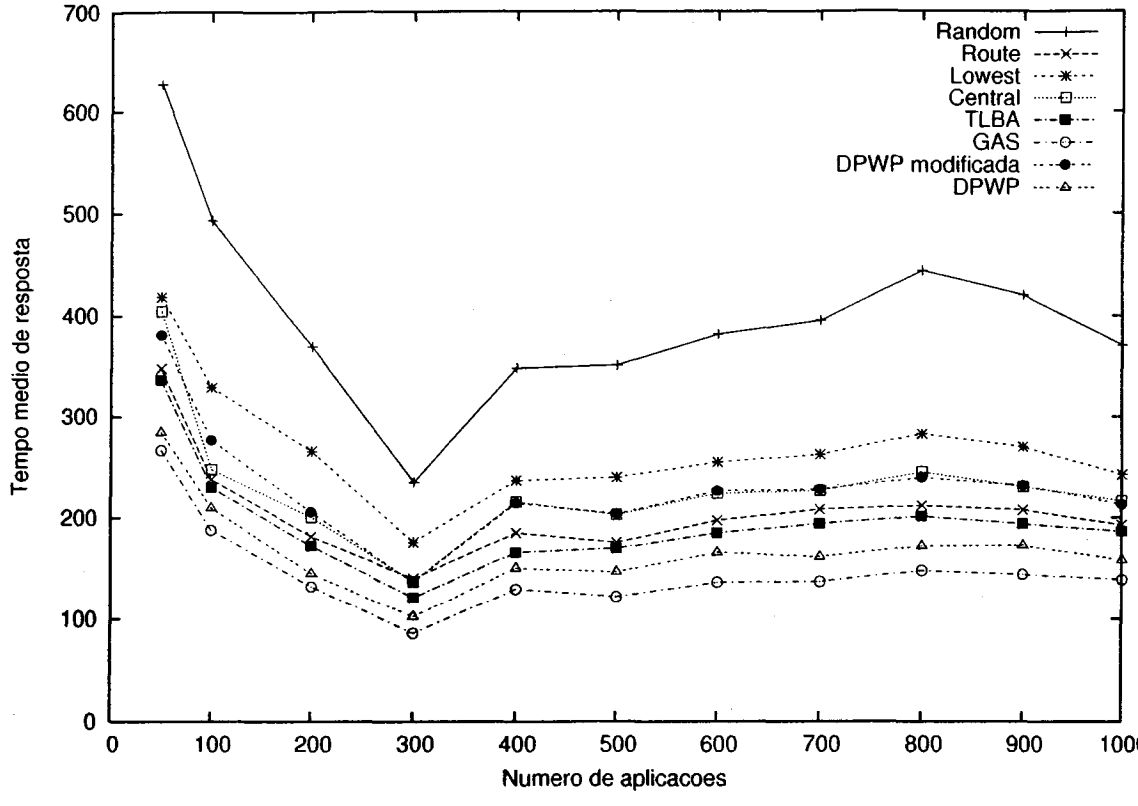


Figura 7.2: Desempenho dos algoritmos de escalonamento

BACKFILL-IBL são úteis para melhorar as decisões do software de escalonamento. Essa utilidade pode ser verificada pelos valores de *slowdown* impostos às aplicações quando o algoritmo BACKFILL-IBL é empregado, que são menores que os valores de *slowdown* observados para os algoritmos de escalonamento FCFS e BACKFILL-USUARIO. Além disso, os bons resultados obtidos como o algoritmo BACKFILL-REAL indicam que o desempenho do sistema melhora à medida que são utilizados algoritmos de escalonamento que realizam *backfilling* e que empregam estimativas sobre o tempo de execução mais corretas.

As extensões implementadas no simulador de políticas de compartilhamento de espaço permitem a avaliação de diferentes algoritmos de escalonamento. Essas extensões definem uma ferramenta útil para a avaliação de algoritmos de escalonamento. Através desse simulador, pode-se também utilizar diferentes algoritmos de escalonamento durante a simulação.

Uma característica não modelada na avaliação do algoritmo BACKFILL-IBL é o término das aplicações em virtude de estimativas abaixo dos tempos reais de execução das aplicações. Em sistemas reais, o software de escalonamento finaliza aplicações que encontram-se em execução por um tempo maior que a estimativa provida pelo usuário. Isso oferece um garantia que usuários não informem estimativas abaixo do tempo real de execução de suas aplicações, tentando assim forçar o escalonador a começar a executar suas aplicações mais rapidamente. Como o algoritmo IBL pode gerar estimativas de tempo de execução abaixo dos valores reais, se

o software de escalonamento trabalhar dessa forma, pode acontecer a finalização das aplicações pelo escalonador.

Existem duas soluções para esse problema. A primeira é utilizar um fator multiplicador para as estimativas, o que possibilita que um número menor de aplicações se encontre na situação de término pelo escalonador. Alguns trabalhos (Mu'alem & Feitelson, 2001) também observam que multiplicar estimativas podem adicionar uma certa flexibilidade ao software de escalonamento. Uma segunda solução é remover a possibilidade de término das aplicações quando o algoritmo de aprendizado IBL é utilizado como preditor, encarando assim esses algoritmo como sendo parte integrante do software de escalonamento.

Através dos resultados obtidos dos experimentos de simulação com o algoritmo `GAS`, pôde-se observar a importância da utilização do conhecimento sobre o comportamento das aplicações paralelas na utilização de recursos. Esse algoritmo realiza uma busca heurística, utilizando a técnica de algoritmos genéticos para encontrar um melhor conjunto de recursos, considerando as características de configuração e de carga, para o escalonamento de aplicações paralelas. Os resultados obtidos mostram que o algoritmo `GAS` apresenta melhor desempenho, quando comparado com os algoritmos de escalonamento `DPWP`, `DPWP-modificada`, `Lowest`, `Disted`, `Global`, `Random` e `TLBA`.

O custo computacional do algoritmo `GAS` está relacionado com a quantidade de EPs solicitados pelas aplicações paralelas e a quantidade de EPs que podem ser empregados. Outras estratégias de busca podem ser utilizadas, visando melhorar a possibilidade de crescimento em escala do algoritmo de escalonamento.

Através dos resultados obtidos, conclui-se que algoritmos de escalonamento que utilizam conhecimento sobre as aplicações paralelas que são submetidas melhoram o tempo de resposta do sistema. Esse fato indica a necessidade de utilização desse conhecimento no escalonamento, considerando cenários de execução homogêneos e heterogêneos.

CAPÍTULO

8

Conclusões

8.1 Considerações iniciais

Os resultados das pesquisas na área de escalonamento de processos em sistemas computacionais distribuídos têm demonstrado a importância do conhecimento sobre a carga de trabalho nas decisões do software de escalonamento. Isso reforça a relevância do estudo de técnicas que permitam a aquisição desse conhecimento, de forma que possa ser empregado com bom desempenho na tomada de decisões dos escalonadores. Além disso, torna-se necessário que os mecanismos de aquisição e utilização desse conhecimento possibilitem a atualização do conhecimento obtido e que esses mecanismos apresentem baixa intrusão no desempenho do sistema.

Esta tese abordou a exploração das características das aplicações paralelas visando a utilização do conhecimento obtido no escalonamento de processos. Nesse sentido, foram definidos e avaliados dois modelos de aquisição de conhecimento, um relacionado com o comportamento das aplicações paralelas na utilização de recursos, e outro relacionado com predições sobre características de execução das aplicações. Esses modelos, embora direcionados para aplicações paralelas, englobam também aplicações seqüenciais. Através de experimentos de simulação, demonstrou-se como o conhecimento obtido pode ser empregado em sistemas paralelos e distribuídos, considerando dois algoritmos de escalonamento e dois cenários de execução. Esse estudo foi conduzido através da definição de mecanismos de aquisição de informações sobre as aplicações, de modelos de aquisição de conhecimento e de algoritmos de escalonamento, acompanhados de suas utilizações, análises e avaliações de desempenho.

O restante deste capítulo está organizado através de 4 seções. A Seção 8.2 apresenta as conclusões principais obtidas com este trabalho. A Seção 8.3 apresenta as contribuições relevantes deste trabalho para a pesquisa em escalonamento de processos em ambientes distribuídos.

A Seção 8.4 apresenta sugestões para trabalhos futuros. Finalizando este capítulo, a Seção 8.5 apresenta a lista de trabalhos publicados durante o desenvolvimento deste trabalho.

8.2 Conclusões

Visando atingir os seus objetivos, o desenvolvimento deste trabalho abrangeu as etapas de levantamento e análise de aplicações paralelas reais e da carga de trabalho, exploração de características de execução das aplicações, exploração de características de submissão das aplicações paralelas e a utilização do conhecimento obtido em algoritmos de escalonamento.

Dentre as aplicações analisadas nesta tese, todas são implementadas através ambientes de passagem de mensagens e predomina o uso do Fortran como linguagem de programação, devido às facilidades que essa linguagem fornece para o tratamento de operações matemáticas, vetores e matrizes de dados. Outra alternativa que tem sido verificada no desenvolvimento de aplicações paralelas é a utilização de bibliotecas de núcleo, por exemplo as bibliotecas representadas pelas aplicações de núcleo do pacote NAS (vide Capítulo 4). Bibliotecas de núcleo também utilizam ambientes de passagem de mensagens.

Pôde-se observar que as aplicações analisadas nesta tese divergem muito em seu comportamento na utilização de recursos. Tal divergência tem origem na natureza do problema e na forma pela qual a aplicação paralela foi projetada e implementada. Dessa forma, a caracterização global, mesmo do conjunto de aplicações paralelas utilizadas nesta tese, é difícil. Isso ressalta ainda mais a importância de se analisar o comportamento de cada aplicação de maneira isolada, visando adquirir conhecimento sobre as características intrínsecas de suas execuções.

A maioria das aplicações analisadas segue o paradigma de desenvolvimento de paralelismo de dados, onde um código semelhante é replicado entre os elementos de processamento, sendo que a execução difere apenas pelas informações processadas pelas tarefas. Isso deve-se ao fato de que tal paradigma apresenta uma maior facilidade para o projeto e implementação, além de permitir uma depuração mais fácil que o paradigma de desenvolvimento funcional, onde as tarefas têm códigos diferentes. Tal afirmação corrobora as conclusões obtidas por outros autores (Corbalan *et al.*, 2001).

Os traços de execução dos centros de computação de alto desempenho analisados nesta tese apresentaram, além das aplicações paralelas, um conjunto expressivo de aplicações seqüenciais. Observou-se também que diferentes centros de computação apresentaram diferentes distribuições para o tempo de execução e quantidade de elementos de processamento requisitados, devido às diferentes características das aplicações desenvolvidas nesses ambientes. Essas características divergentes das aplicações acarretam dificuldade em estabelecer modelos que sirvam para diferentes ambientes de produção, pois cada ambiente apresentou características específicas em seus traços. Sobre esse aspecto, a conclusão obtida é que modelos de aquisição

de conhecimento devem utilizar características específicas da carga de trabalho do sistema para o qual os modelos são desenvolvidos.

Os mecanismos propostos para a aquisição de informações de processos, apresentados no Capítulo 5, foram implementados através da instrumentação do sistema operacional Linux e da monitoração da execução das tarefas. Esses mecanismos mostraram-se estáveis e permitiram a aquisição de informações mais detalhadas sobre a execução de tarefas que compõem aplicações paralelas. A avaliação de desempenho conduzida, que considerou a execução de um conjunto de *benchmarks* representativos de cargas paralelas, permitiu concluir que tais mecanismos apresentam baixa intrusão no desempenho do sistema. Assim, pode-se depreender que tais mecanismos são adequados para definir uma fonte de informações para modelos de aquisição de conhecimento.

O modelo de aquisição de conhecimento sobre o comportamento de aplicações paralelas mostrou-se adequado para representar as características dessas aplicações durante a utilização de recursos do sistema. Tal adeqüabilidade foi verificada através de resultados experimentais, que compreenderam a execução, monitoração e aquisição de conhecimento, considerando um conjunto representativo de aplicações paralelas seqüencias e paralelas reais e sintéticas. A incorporação da arquitetura de redes neurais ART-2A permitiu que o modelo fosse empregado de maneira *on-line* e que o conhecimento fosse obtido de maneira automática, com a possibilidade de atualização. Assim, informações de execuções mais recentes das aplicações podem ser utilizadas para a atualização do conhecimento intrínseco às aplicações, representando o seu comportamento através de um conjunto de categorias, identificadas e rotuladas automaticamente pelo modelo.

O algoritmo de rotulação desenvolvido permitiu a identificação da significância dos atributos de execução utilizados na categorização realizada pela rede ART-2A. Esse algoritmo mostrou-se estável e eficiente, e, juntamente com a implementação desenvolvida para a rede neural ART-2A, constitui um modelo que pode ser empregado para a aquisição de conhecimento em sistemas paralelos reais.

Através da integração desse modelo de aquisição de conhecimento junto ao núcleo do sistema operacional Linux, pôde-se verificar o desempenho e a adeqüabilidade da solução proposta para a utilização em sistemas computacionais reais. Os desempenhos computacional e de classificação obtidos são adequados, e o impacto no desempenho do sistema é baixo. O desempenho observado deve-se à implementação do modelo proposto, que emprega o modelo de aquisição com o processamento inicial das informações através de um *buffer* de tamanho limitado e realimentado ao modelo, a cada requisição ao ponto de acesso ao serviço de classificação dos processos.

O algoritmo de aprendizado desenvolvido, inspirado no modelo de aprendizado baseado em instâncias, apresentou bom desempenho de classificação e permitiu a obtenção de valores de

erro de predição menores quando comparado com outras técnicas propostas na literatura. Além disso, o modelo de predição apresentou vantagens tais como o bom desempenho computacional e a possibilidade de atualização automática do conhecimento sobre as aplicações paralelas e seqüenciais. A implementação do modelo de predição, através de classes C++ e do gerenciador de banco de dados MySQL, apresentou bom desempenho computacional, o que permite que o modelo de aquisição de conhecimento seja implementado juntamente ao software de escalonamento sem incluir grandes atrasos na tomada de decisões de gerência de aplicações e recursos.

Os algoritmos de escalonamento que empregam o conhecimento sobre as aplicações apresentaram desempenhos melhores em relação aos algoritmos que não empregam tal conhecimento. Isso pôde ser observado através dos resultados obtidos com os algoritmos propostos BACKFILL-IBL e GAS, que utilizam o conhecimento sobre as aplicações em suas decisões para a atribuição das tarefas no ambiente distribuído.

No cenário composto por máquinas paralelas homogêneas, o desempenho do algoritmo BACKFILL-IBL, que utiliza as predições providas pelo algoritmo IBL, superou os desempenhos do algoritmo FCFS, que não utiliza estimativas, e do algoritmo BACKFILL-USUARIO, que emprega as estimativas providas pelos usuários. O bom desempenho observado para o algoritmo BACKFILL-REAL indica a importância do conhecimento mais preciso sobre o tempo de execução das aplicações. Assim, acredita-se que melhores resultados podem ser observados à medida que a qualidade de predição do algoritmo IBL seja incrementada.

O algoritmo de escalonamento GAS foi proposto para realizar o escalonamento em um cenário de execução composto por computadores heterogêneos e interligados por redes de diferentes capacidades de transmissão de dados. O algoritmo de escalonamento GAS apresentou bons resultados quando comparado com os algoritmos Random, DPWP, DPWP-modificado, Global, Disted, Lowest e TLBA. Através dos resultados obtidos na simulação desses algoritmos, pôde-se observar as vantagens da utilização do conhecimento sobre as aplicações para o escalonamento em um cenário de execução composto por um sistema distribuído, dinâmico e heterogêneo.

Pelos resultados obtidos com a exploração de características de aplicações paralelas, através de algoritmos de aprendizado de máquina incrementais, e a utilização e avaliação do conhecimento obtido em algoritmos de escalonamento, conclui-se que os objetivos desta tese foram alcançados.

Os resultados obtidos com este trabalho, suas contribuições e as possibilidades criadas para a realização de trabalhos futuros, reforçam a linha de pesquisa atual do grupo de sistemas distribuídos e programação concorrente, na área de escalonamento de processos em ambientes virtuais distribuídos e avaliação de desempenho de sistemas computacionais. A seguir, são apresentadas as contribuições principais desta tese, as sugestões para trabalhos futuros e a lista de trabalhos publicados durante o desenvolvimento desta pesquisa.

8.3 Contribuições

Através deste trabalho, pôde-se contribuir com o estado da arte da área de escalonamento de processos, empregando-se técnicas da área de aprendizado de máquina para a exploração das características da carga de trabalho de sistemas paralelos e distribuídos. Assim, este trabalho apresenta contribuições relevantes para o desenvolvimento de técnicas eficientes de escalonamento de processos em aplicações paralelas, considerando diferentes cenários de execução.

Este trabalho apresenta as seguintes contribuições principais para a área de escalonamento de processos em aplicações paralelas:

- Estudo amplo e objetivo sobre a exploração de características de aplicações paralelas, levando à apresentação de algoritmos de aprendizado de máquina como uma solução para o problema de aquisição de conhecimento sobre as aplicações e geração de informações relevantes sobre a carga de trabalho do sistema. As informações geradas pelos modelos mostram-se importantes para a compreensão do comportamento dos processos de aplicações paralelas na utilização dos recursos computacionais. Assim, os modelos e algoritmos definidos através desse estudo têm aplicação direta em ambientes paralelos reais, pois permitem incrementar as decisões do software de escalonamento com o uso de conhecimento mais detalhado das aplicações que são submetidas ao sistema computacional;

- Definição, implementação e avaliação de modelos de aquisição de conhecimento sobre aplicações paralelas, explorando características de execução e de submissão da carga de trabalho. Comparados aos trabalhos publicados na literatura, os modelos desenvolvidos permitem a atualização do conhecimento sobre as aplicações e apresentam bom desempenho computacional e de classificação. Tais características são importantes, pois permitem a utilização desses modelos em diferentes cenários paralelos e para definição de diferentes algoritmos de escalonamento. Além disso, as implementações dos modelos de aquisição de conhecimento constituem uma ferramenta valiosa para o desenvolvimento de outros algoritmos de escalonamento, pois estabelecem um conjunto de mecanismos importantes para a exploração das características e aquisição de conhecimento sobre a carga de trabalho;

- Definição e avaliação de algoritmos de escalonamento que utilizam o conhecimento sobre as características das aplicações em suas decisões de atribuição de processos e gerência dos recursos computacionais. Os algoritmos apresentados têm forte ligação com os modelos de aquisição de conhecimento e dessa forma permitem observar o ganho que pode ser obtido com a utilização desses modelos para auxiliar as decisões do software de escalonamento. Esses algoritmos podem ser utilizados em ambientes paralelos reais, através da incorporação desse algoritmos junto ao ambiente de escalonamento ou ao sistema operacional.

8.4 Trabalhos futuros

Através da realização deste trabalho, contempla-se a possibilidade da realização dos seguintes trabalhos futuros:

- Estudo de diferentes configurações dos parâmetros do algoritmo IBL e sua validação através de simulação das políticas de escalonamento;

- Avaliação do algoritmo `BACKFILL-IBL` com limitação na busca de aplicações paralelas cujo tempo de execução é mais reduzido. As simulações apresentadas nesta tese para o algoritmo de *backfilling* foram realizadas considerando todas as aplicações da fila como espaço de busca. Pesquisas podem ser realizadas para avaliar os benefícios em utilizar um espaço de busca mais reduzido, definido de maneira estática ou dinâmica;

- Integração das técnicas propostas para a aquisição de conhecimento junto ao software de escalonamento. A implementação dos modelos de aquisição de conhecimento e as avaliações de desempenho conduzidas permitiram observar que as técnicas propostas podem ser integradas junto ao software de escalonamento;

- Aplicação do modelo de aquisição de conhecimento, descrito no Capítulo 5, em outros domínios. O modelo de aquisição sobre o comportamento das aplicações paralelas pode ser empregado na solução de problemas de outras áreas de conhecimento, que necessitem da categorização e rotulação automática e incremental das informações;

- Permitir que a implementação do modelo de aquisição de conhecimento junto ao núcleo do sistema operacional Linux armazene o conhecimento sobre os processos após o término de execução. Da forma atual, a implementação permite a classificação dos processos durante a execução desses, mas não salva a configuração dos pesos da rede ART-2A . Estudos podem ser conduzidos para definir um forma de implementação que permita que os resultados da classificação estejam disponíveis em futuras execuções das aplicações;

- Estudar a possibilidade de utilização dos modelos de aquisição de conhecimento desenvolvidos em ambientes de memória compartilhada distribuída;

- Investigar a utilização de outras heurísticas de busca, ao invés de algoritmos genéticos, para encontrar a melhor atribuição para as tarefas. Apesar do algoritmo genético apresentar bons resultados, seu tempo computacional aumenta à medida que aumenta o tamanho do problema, expresso pela quantidade de EPs disponíveis e pela quantidade de EPs requisitados pelas aplicações. Estudos podem ser conduzidos para definir algoritmos que realizem a busca pela solução em um tempo computacional mais reduzido, visando a possibilidade de crescimento em escala da solução seja mais adequada.

8.5 Trabalhos publicados

Os artigos científicos e o capítulo de livro publicados durante a realização deste trabalho estão listados abaixo:

1. Senger, L. J.; Santana, M. J.; Santana, R. H. C. (2002). Uma nova abordagem para a aquisição de conhecimento sobre o comportamento de aplicações paralelas na utilização de recursos. *Proceedings of WORKCOMP*, p. 79–85, São José dos Campos, São Paulo, Brasil.

2. Senger, L. J.; Santana, M. J.; Santana, R. H. C. (2003). A new approach for acquiring knowledge of resource usage in parallel applications. *Proceedings of International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'2003)*, p. 607–614, Montreal, Canadá.

3. Senger, L. J.; Santana, M. J.; Santana, R. H. C. (2004). Using runtime measurements and historical traces for acquiring knowledge in parallel applications. *International Conference on Computational Science (ICCS)*, p. 661–665, Springer, LNCS 3036, Junho, Krakow, Polônia.

4. Senger, L. J.; Santana, M. J.; Santana, R. H. C. (2004a). An instance-based learning approach for predicting parallel applications execution times. *Proceedings of 3rd International Information and Telecommunication Technologies Symposium (I2TS'2004)*, p. 9–15, São Carlos Federal University - UFSCar, São Carlos, SP, Brasil.

5. Senger, L. J.; Mello, R. F. de; Santana, M. J.; Santana, R. H. C. (2005). An on-line approach for classifying and extracting application behavior on linux. Capítulo do Livro *High Performance Computing: Paradigm and Infrastructure (a ser publicado)*. John Wiley and Sons Inc.

Referências

Aha, D. W. (1998). Feature weighting for lazy learning algorithms. *Feature Extraction, Construction and Selection: A Data Mining Perspective*. Norwell MA: Kluwer.

Aha, D. W.; Kibler, D.; Albert, M. K. (1991). Instance-based learning algorithms. *Machine Learning*, v.1, n.6, p.37–66.

Aida, K. (2000). Effect of job size characteristics on job scheduling performance. *In Job Scheduling Strategies for Parallel Processing*, p. 1–17. LNCS 1911.

Almasi, G.; Gottlieb, A. (1994). *Highly Parallel Computing*. Benjamin/Cummings Publishing Company Inc.

Amir, Y.; Awerbuch, B.; Barak, A.; Borgstrom, R. S.; Keren, A. (2003). An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Transactions on Parallel and Distributed Systems*, v.14, n.1, p.39–50.

Anastasiadis, S. V.; Sevcik, K. C. (1997). Parallel application scheduling on networks of workstations. *Journal of Parallel and Distributed Computing*, v.43, n.2, p.109–124.

Anderson, T.; Culler, D.; Patterson, D. (1995). A Case for NOW (Networks of Workstations). *IEEE Micro*, v.15, n.1, p.54–64.

Apon, A. W.; Wagner, T. D.; Dowdy, L. W. (1999). A Learning Approach to Processor Allocation in Parallel Systems. *Proceedings of the 1999 ACM CIKM International Conference on Information and Knowledge Management*, p. 531–537.

Araújo, A. P. F.; Santana, M. J.; Santana, R. H. C.; Souza, P. S. L. (1999a). DPWP: A new load balancing algorithm. *5th International Conference on Information Systems Analysis and Synthesis - ISAS'99*, Orlando, U.S.A.

Araújo, A. P. F.; Santana, M. J.; Santana, R. H. C.; Souza, P. S. L. (1999b). A new dynamical scheduling algorithm, international conference on parallel and distributed processing techniques and applications. *Proceedings of PDPTA'99*, Las Vegas,Nevada, U.S.A.

Arpaci, R. H.; Dusseau, A. C.; Vahdat, A. M.; Liu, L. T.; Anderson, T. E.; Patterson, D. A. (1995). The Interaction of Parallel and Sequential Workloads on a Network of Workstations. *Proceedings of ACM SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, p. 267–278.

Arpaci-Dusseau, A. C. (2001). Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, v.19, n.3, p.283–331.

Arpaci-Dusseau, A. C.; Culler, D. E.; Mainwaring, M. (1998). Scheduling with Implicit Information in Distributed Systems. *Proceedings of ACM SIGMETRICS'98*, p. 233–248.

Atkeson, C.; Schaal, S. (1995). Memory-based neural networks for robot learning. *Neurocomputing*, v.9, p.243–269.

Bachelder, I.; Waxman, A.; Seibert, M. (1993). A neural system for mobile robot visual place learning and recognition. *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, v. LNCS 807, p. 198–212, Berlin.

Bailey, D. H.; Barszcz, E.; Barton, J. T.; Browning, D. S.; Carter, R. L.; Dagum, D.; Fatoohi, R. A.; Frederickson, P. O.; Lasinski, T. A.; Schreiber, R. S.; Simon, H. D.; Venkatakrishnan, V.; Weeratunga, S. K. (1994). The NAS Parallel Benchmarks. Relatório Técnico RNR-94-007, Nasa Ames Research Center.

Barak, A.; La'adan, O. (1998). The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, v.4-5, n.13, p.361–372.

Baraldi, A.; Parmiggiani, F. (1995). A neural network for unsupervised categorization of multivalued input patterns: an application to satellite image clustering. *IEEE transactions on geoscience and remote sensing*, v.33, n.2.

Barszcz, E.; Fatoohi, R.; Venkatakrishnan, V.; Weeratunga, S. (1993). Solution of regular sparse triangular linear systems on vector and distributed memory multiprocessors. Relatório Técnico RNR-93-07, NASA Ames Research Center, Moffett Field, CA 94035.

Barton, J. M.; Bitar, N. (1995). A scalable multi-discipline, multiple-processor scheduling framework for IRIX. *Job Scheduling Strategies for Parallel Processing*, p. 45–69. LNCS 949.

Beaugendre, P.; Priol, T.; René, C. (1997). Cobra: a CORBA-compliant programming environment for high-performance computing. Relatório Técnico 1141, Institut Bational de Recherche en Informatique et en Automatique.

Beguelin, A.; Gueist, A.; Dongarra, J.; Jiang, W.; Manchek, R.; Sunderam, V. (1994). *PVM: Parallel Virtual Machine: User's Guide and tutorial for Networked Parallel Computing*. MIT Press.

Benítez-Pérez, H.; García-Nocetti, F. (2002). Fault classification using time variable ART2-A networks. *Proceedings of the 2002 IEEE international conference on control applications*, p. 832–837, Glasgow, Scotland, U.K.

Berry, M.; Chen, D.; Koss, P.; Kuck, D.; Lo, S.; Pang, Y.; Pointer, L.; Roloff, R.; Sameh, A.; Clementi, E.; Chin, S.; Scheider, D.; Fox, G.; Messina, P.; Walker, D.; Hsiung, C.; Schwarzmeier, J.; Lue, K.; Orszag, S.; Seidl, F.; Johnson, O.; Goodrum, R.; Martin, J. (1989). The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, v.3, n.3, p.5–40.

Blake, C.; Merz, C. (1998). UCI repository of machine learning databases.

Bode, B.; Halstead, D. M.; Kendall, R.; Lei, Z.; Jackson, D. (2000). The portable batch scheduler and maui scheduler on linux clusters. *Usenix conference*, Atlanta, GA, USA.

Bovet, D. P.; Cesati, M. (2000). *Understanding the Linux Kernel*. O'Reilly.

Brecht, T.; Guha, K. (1996). Using Parallel Program Characteristics in Dynamic Processor Allocation Policies. *Performance Evaluation*, v.28, n.4, p.519–539.

Brehm, J.; Worley, P. H.; Madhukar, M. (1998). Performance modeling for spmd message-passing programs. *Concurrency: Practice and Experience*, v.5, n.10, p.333–357.

Bricker, A.; Litzkow, M.; Livny, M. (1991). Condor technical summary. Relatório técnico, (TR 1069), Department of Computer Science, University of Wisconsin.

Carpenter, G. A.; Gjaja, M. N.; Gopal, S.; Woodcock, C. E. (1997). ART neural networks for remote sensing: Vegetation classification from lansat TM and terrain data. *IEEE Transactions on Geoscience and Remote Sensing*, v.35, n.2.

Carpenter, G. A.; Grossberg, S. (1988). The ART of adaptive pattern recognition by a self-organizing neural network. *Computer*, v.21, n.3, p.77–88.

Carpenter, G. A.; Grossberg, S. (1989a). ART 2: Self-organization of Stable Category Recognition Codes for Analog Input Patterns. *Applied Optics*, v.26, n.23, p.4919–4930.

Carpenter, G. A.; Grossberg, S. (1989b). ART 2: Self-organization of Stable Category Recognition Codes for Analog Input Patterns. *Applied Optics*, v.26, n.23, p.4919–4930.

Carpenter, G. A.; Grossberg, S.; Markuzon, N.; Reynolds, J. H.; Rosen, D. B. (1992). Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps. *IEEE Transactions on Neural Networks*, v.3, n.5, p.698–712.

Carpenter, G. A.; Grossberg, S.; Rosen, D. B. (1991a). ART 2-A: An Adaptive Resonance Algorithm for Rapid Category Learning and Recognition. *Neural Networks*, v.4, p.494–504.

Carpenter, G. A.; Grossberg, S.; Rosen, D. B. (1991b). Fuzzy ART: Fast stable learning and categorization of analog patterns by an adaptive resonance system. *Neural Networks*, v.4, p.759–771.

Casavant, T. L.; Kuhl, J. G. (1988). A Taxonomy of Scheduling in General-purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, v.14, n.2, p.141–154.

Chamberlain, B.; Deitz, S.; Snyder, L. (2000). A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures. *Proceedings of the ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC'00), Dallas, Texas, USA*. ACM Press and IEEE Computer Society Press.

Chapin, S. J.; Cirne, W.; Feitelson, D. G.; Jones, J. P.; Leutenegger, S. T.; Schwiegelshohn, U.; Smith, W.; Talby, D. (1999). Benchmarks and standards for the evaluation of parallel job schedulers. *In Job Scheduling Strategies for Parallel Processing*, p. 66–89. LNCS 1659.

Chiola, G.; Ciaccio, G. (2000). Efficient parallel processing on low-cost clusters with GAMMA active ports. *Parallel Computing*, v.26, n.2, p.333–354.

Chodnekar, S. (1997). Towards a communication characterization methodology for parallel applications. *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA '97)*, p. 310. IEEE Computer Society.

Cirne, W.; Berman, F. (2001). A comprehensive model of the supercomputer workload. *4th Workshop on Workload Characterization.*

Corbalan, J.; Martorell, X.; Labarta, J. (2001). Improving Gang Scheduling through Job Performance Analysis and Malleability. *International Conference on Supercomputing*, p. 303–311, Sorrento, Italy.

Cortes, O. A. C. (1999). Desenvolvimento e avaliação de algortimos numéricos paralelos. Dissertação (Mestrado), ICMC-USP, São Carlos, São Paulo, Brasil.

Cortes, O. A. C. (2004). *Um sistema nebuloso-evolutivo para determinar a portabilidade de benchmarks paralelos*. Tese (Doutorado), ICMC-USP, São Carlos, São Paulo, Brasil.

Coulouris, G.; Dollimore, J.; Kindberg, T. (1994). *Distributed Systems: Concepts and Design.* Addison Wisley.

Culler, D. E.; Karp, R. M.; Patterson, D. A.; Sahay, A.; Schauser, K. E.; Santos, E.; Subramonian, R.; von Eicken, T. (1993). LogP: Towards a realistic model of parallel computation. *Principles Practice of Parallel Programming*, p. 1–12.

de P. Braga, A.; Ludermir, T. B.; Carvalho, A. C. P. L. F. (2000). *Redes Neurais Artificiais: Teoria e Aplicações*. LTC.

Devarakonda, M. V.; Iyer, R. K. (1989). Predictability of Process Resource Usage: A Measurement-based Study on UNIX. *IEEE Transactions on Software Engineering*, v.15, n.12, p.1579–1586.

Dimpsey, R. T.; Year, R. K. (1991). Performance prediction and tuning on a multiprocessor. *Proc. International Symposium on Computer Architecture*, p. 190–199.

Dimpsey, R. T.; Year, R. K. (1995). A measurement-based model to predict the performance impact of system modifications: A case study. *IEEE Transactions on Parallel and Distributed Systems*, v.6, n.1.

Douglis, F.; Ousterhout, J. K. (1991). Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, v.21, n.8, p.757–785.

Downey, A. B. (1997). Predicting queue times on space-sharing parallel computers. *11th International Parallel Processing Symposium*. Also available as University of California technical report number CSD-96-906.

Downey, A. B.; Feitelson, D. G. (1999). The elusive goal of workload characterization. *Performance Evalution Review*, v.26, n.4, p.14–29.

Drake, J. B.; Hammond, S.; James, R.; Worley, P. H. (1999). Performance tuning and evaluation of a parallel community climate model. *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC99)*, Portland, OR.

Duke, D. W.; Green, T. P.; Pasko, J. L. (1994). Research toward a heterogeneous networked computing cluster: The distributed queueing system version 3.0. Relatório técnico, Florida State University.

Ekmecic, I.; Tartalja, I.; Milutinovic, V. (1996). A survey of heterogeneous computing: Concepts and systems. *Proceedings of the IEEE*, v.84, n.8, p.1127–1144.

Emilio, G. H. (1996). *A methodology for design of parallel benchmarks*. Tese (Doutorado), University of Southampton, Inglaterra.

Etsion, Y.; Tsafrir, D.; Feitelson, D. G. (2004). Desktop scheduling: How can we know what the user wants? *14th Workshop on Network and Operating System Support for Digital Audio and Video*, p. 110–115.

Fatoohi, R. (1992). Adapting the INS3D-LU code to the CM2 and iPSC/860. Relatório Técnico RNR-92-024, NASA Ames.

Fatoohi, R. A. (1989). Multitasking a Navier-Stokes algorithm on the CRAY-2. *Journal of Supercomputing*, v.3, n.2, p.109–124.

Fausett, L. (1994). *Fundamentals of Neural Networks*. Prentice Hall.

Feitelson, D. G. (1997). Memory usage in the LANL CM-5 workload. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 78–94. LNCS 1291.

Feitelson, D. G. (2001). Metrics for parallel job scheduling and their convergence. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 188–205. LNCS 2221.

Feitelson, D. G.; Jette, M. A. (1997). Improved utilization and responsiveness with gang scheduling. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 238–261. LNCS 1291.

Feitelson, D. G.; Naaman, M. (1999). Self-tuning systems. *IEEE Software*, v.16, n.2, p.52–60.

Feitelson, D. G.; Nitzberg (1995). Job characteristics of a production parallel scientific workload on the NASA ames iPSC/860. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, v. 949, p. 337–360. LNCS 949.

Feitelson, D. G.; Rudolph, L. (1995a). Coscheduling Based on Run-Time Identification of Activity Working Sets. *International Journal of Parallel Programming*, v.23, n.2, p.136–160.

Feitelson, D. G.; Rudolph, L. (1995b). Parallel job scheduling: Issues and approaches. *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, p. 1–18. LNCS 949.

Feitelson, D. G.; Rudolph, L. (1998). Metrics and benchmarking for parallel job scheduling. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 1–24. LNCS 1459.

Feitelson, D. G.; Rudolph, L.; Schwiegelshohn, U.; Sevcik, K. C.; Wong, P. (1997). Theory and Practice in Parallel Job Scheduling. *Job Scheduling Strategies for Parallel Processing*, v. 1291, p. 1–34. LNCS 1291.

Feitelson, D. G.; Weil, A. M. (1998). Utilization and predictability in scheduling the ibm sp2 with backfilling. *12th Intl. Parallel Processing Symposium*, p. 542–546.

Ferschweiler, K.; Harrah, S.; Keon, D.; Calzarossa, M.; Tessera, D.; Pancake, C. (2002). The tracefile testbed: A community repository for identifying and retreving HPC data. *Proceedings of the International Conference on Parallel Processing (ICPP'02)*, p. 40–47.

Filippidis, A.; Jain, L. C.; Lozo, P. (1999). Degree of familiarity ART2 in knowledge-based landmine detection. *IEEE Transactions on Neural Networks*, v.10, n.1.

Flynn, M. J.; Rudd, K. W. (1996). Parallel architectures. *ACM Computing Surveys*, v.28, n.1, p.67–70.

Foster, I.; Kesselman, C. (1997). Globus: A metacomputing infrastructure toolkit. *International journal of supercomputer applications*, v.11, n.2, p.115–128.

Foster, T.; Toonen, B.; Worley, P. H. (1996). Performance of parallel computers for spectral atmospheric models. *Journal Atm. Oceanic Tech.*, v.5, n.13, p.1031–1045.

Foster, Y. (1995). *Designing and Bulding Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Publishing Company.

Frank, T.; Kraiss, K.; Kuhlen, T. (1998). Comparative analysis of fuzzy art and art-2a network clustering performance. *IEEE transactions on neural networks*, v.9, n.3, p.544–559.

Gan, K.; Lua, K. (1992). Chinese character classification using adaptive resonance network. *Pattern Recognition*, v.25, n.8, p.877–888.

Ghare, G.; Leutenegger, S. T. (1999). The effect of correlating quantum allocation and job size for gang scheduling. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 91–110. LNCS 1659.

Gibbons, R. (1997). A Historical Application Profiler for Use by Parallel Schedulers. *Job Scheduling Strategies for Parallel Processing*, p. 58–77. LNCS.

Gokcay, E.; Principe, J. (2000). A new clustering evaluation function using renyi's information potential. *Proceedings of ICASSP 2000*, Istanbul, Turkey.

Graham, S.; Kessler, P.; McKusick, M. (1982). gprof: A call graph execution profiler. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, p. 120–126.

Graham, S.; Kessler, P.; McKusick, M. (1983). An execution profiler for modular programs. *Software - Practice and Experience*, v.13, p.671–685.

Harchol-Balter, M.; Downey, A. B. (1997). Exploiting Process Lifetimes Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, v.15, n.3, p.253–285.

He, J.; Tan, A.-H.; Tan, C.-L. (2002). ART-C: A neural architecture for self-organization under constraints. *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, p. 2550–2555, Hawaii, USA.

He, J.; Tan, A.-H.; Tan, C.-L. (2003a). Modified art 2a growing network capable of generating a fixed number of nodes. *IEEE Transactions on Neural Networks*, v.3, n.15, p.728–737.

He, J.; Tan, A.-H.; Tan, C.-L. (2003b). On machine learning methods for chinese documents classification. *Applied Intelligence*, v.18, p.311–322.

Hey, T.; Lancaster, D. (2000). The development of parkbench and performance prediction. *The International Journal of High Performance Computing Applications*, v.14, n.3, p.205–215.

Hockney, R.; Berry, M. (1994). Public international benchmarks for parallel computers. Relatório Técnico 1, Parkbench comitee.

Hotovy, S. (1996). Workload evolution on the Cornell Theory Center IBM SP2. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 27–40. LNCS 1162.

IBM (1996). *Using and Administering LoadLeveler Release 3.0.*

Iverson, M. A.; ner, F.; Follen, G. J. (1999). Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Transactions on computers*, v.48, n.12, p.1374–1379.

Jain, R. (1991). *The Art of computer systems performance analysis: Techniques for experimental design, measurement, simulation and modeling*. John Wiley & Sons, Inc.

Kapadia, N. H.; Fortes, J. A. B.; Brodley, C. E. (1999). Predictive application-performance modeling in a computational grid environment. *IEEE international symposium on High performance computing*, p. 47–54.

Kaplan, J.; Nelson, M. (1994). A Comparison of Queueing, Cluster, and Distributed Computing Systems. Relatório técnico, NASA 109025 (NASA LaRC).

Keahey, K.; Gannon, D. (1997). PARDIS: A parallel approach to corba. *Proceedings of the 6th IEEE Internation Symposium on High Performance Distributed Computing*, p. 303–311.

Keyvan, S.; Rabelo, L. C. (1992). Sensor signal analysis by neural networks for surveillance in nuclear reactors. *IEEE Transactions on nuclear science*, v.39, n.2.

Krevat, E.; nos, J. G. C.; Moreira, J. E. (2002). Job scheduling for the BlueGene/L system. Feitelson, D. G.; Rudolph, L.; Schwiegelshohn, U., editores, *Job Scheduling Strategies for Parallel Processing*, p. 38–54. LNCS 2537.

Krishnaswamy, S.; Zaslavsky, A.; Loke, S. W. (2004). Estimating computation times to support scheduling of data intensive applications. *IEEE Distributed Systems Online (Special issue on Data Management, I/O Techniques and Storage Systems for Large-scale Data Intensive Applications)*, v.5, n.4.

Lamport, L. (1990). Concurrent reading and writing of clocks. *ACM Transactions on Computer Systems*, v.8, n.4, p.305–310.

Lauria, M.; Chien, A. (1997). MPI-FM: High performance mpi on workstation clusters. *Journal of Parallel and Distributed Computing*, v.1, n.40, p.4–18.

Lawson, B. G.; Smirni, E. (2002). Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. Feitelson, D. G.; Rudolph, L.; Schwiegelshohn, U., editores, *Job Scheduling Strategies for Parallel Processing*, p. 72–87. LNCS 2537.

Leffler, S.; McKusick, M.; Karels, M.; Quarterman, J. (1989). *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company.

Lifka, D. A. (1995). The ANL/IBM scheduling system. *In Job Scheduling Strategies for Parallel Processing*, p. 17–42. LNCS 1659.

Litzkow, M. J.; Livny, M.; Mutka, M. W. (1988). Condor - a hunter of idle workstations. *IEEE VII International conference on distributed computing systems*, p. 104–111.

Lo, V.; Mache, J.; Windisch, K. (1998). A comparative study of real workload traces and synthetic workload models for parallel job scheduling. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 25–46. LNCS 1459.

Madhyastha, T. M.; Reed, D. A. (2002). Learning to classify parallel input/output access patterns. *IEEE Transactions on Parallel and Distributed Systems*, v.13, n.8.

McBryan, O. A. (1994). An overview of message passing environments. *Parallel Computing*, v.20, n.4, p.417–444.

Mello, R. F. (2002). Analysis on the significant information to update the tables on occupation of resources by using a peer-to-peer protocol. *16th Annual International Symposium on High Performance Computing Systems and Applications*, Moncton, New-Brunswick, Canada.

Mello, R. F. (2003). *Proposta e avaliação de desempenho de um algoritmo de balanceamento de carga para ambientes distribuídos heterogêneos e escaláveis*. Tese (Doutorado), Escola de engenharia de São Carlos (EESC), São Carlos, São Paulo, Brasil.

Mello, R. F.; Senger, L. J. (2004). A new migration model based on the evaluation of processes load and lifetime on heterogeneous computing environments. *16th Symposium on Computer Architecture and High Performance Computing (SBAC'2004)*, p. 222–227, Foz do Iguaçu, PR, Brazil.

Mello, R. F.; Trevelin, L. C.; Paiva, M. S.; Yang, L. T. (2003). Comparative study of the server-initiated lowest algorithm using a load balancing index based on the process behavior

for heterogeneous environment. *Networks, Software Tools and Application, ISSN 1386-7857*. Kluwer.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Morimoto, K.; Matsumoto, T.; Hiraki, K. (1999). Performance evaluation of MPI/MBCF with the NAS parallel benchmarks. *Proceedings of 6th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'99)*, p. 19–26. LNCS 1697.

Mouw, E. (2001). Linux kernel procfs guide. Relatório técnico, Delfty University of Technology an Systems.

MPI Forum (1997). *MPI-2: Extensions to the Message Passing Interface*. Message Passing Interface Forum.

Mu'alem, A. W.; Feitelson, D. G. (2001). Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, v.12, n.6, p.529–543.

Naik, V. K. (1993). Performance issues in implementing nas parallel benchmark applications on ibm sp-1. Relatório técnico, T.J. Watson Research Center, IBM.

Naik, V. K. (1995). A scalable implementation of the NAS parallel benchmark bt on distributed memory systems. *IBM systems journal*, v.34, n.2, p.273–291.

Naik, V. K.; Setia, S. K.; Squillante, M. S. (1993). Performance analysis of job scheduling policies in parallel supercomputing environments. *Supercomputing '93*, p. 824–833.

Naik, V. K.; Setia, S. K.; Squillante, M. S. (1997). Processor Allocation in Multiprogrammed Distributed-memory Parallel Computer Systems. *Journal of Parallel and Distributed Computing*, v.47, n.1, p.28–47.

Nicklayev, O. (1996). Performance data reduction using dynamic statistical clustering. Dissertação (Mestrado), Departament of Computer Science, University of Illimois at Urbana-Champaign.

Nieuwejaar, N.; Kotz, D.; Purakayastha, A.; Ellis, C. S.; Best, M. L. (1996). File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, v.7, n.10.

Pappas, T. (1989). *The Joy of Mathematics*. Wide World Publishing.

Parsons, E. W.; Sevcik, K. C. (1997). Implementing multiprocessor scheduling disciplines. *Lecture Notes in Computer Science*, v.1291, p.166–176.

Peper, F.; Zhang, B.; Noda, H. (1993). A comparative study of ART2-A and the self organizing map. *Proceedings of 1993 International Joint Conference on Neural Networks*, p. 1425–1429.

Pirsig, R. (1993). Perfect: Performance evaluation for cost-effective transformations. Relatório Técnico 964, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

Quinn, M. J. (1994). *Parallel Computing: Theory and Practice*. MCGraw Hill.

Reed, D. A.; Elford, C. L.; Madhyastha, T.; Aydt, W. H.; Smirni, E. (1993). Scalable performance analysis: The pablo performance analysis environment. *Scalable Parallel Libraries Conference*, p. 104–113.

Reed, D. A.; Elford, C. L.; Madhyastha, T.; Scullin, W. H.; Aydt, R. A.; Smirni, E. (1996). I/o, performance analysis and performance data immersion. *MASCOTS'96*, p. 5–16.

Ritchie, D. M.; Thompsom, K. (1974). The UNIX time-sharing system. *Communications of the ACM*, v.17, n.7, p.365–375.

Santos, R. R. (2001). Escalonamento de aplicações paralelas: Interface AMIGO-CORBA. Dissertação (Mestrado), Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo.

Santos, R. R.; Souza, P. S. L.; Santana, M. J.; Santana, R. H. C. (2001). Performance evaluation of distributed applications development tools from interprocess communication point of view. *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA.

Schnor, B.; Petri, S.; Oleyniczak, R.; Langendörfer, H. (1996). Scheduling of parallel applications on heterogeneous workstation clusters. Yetongnon, K.; Hariri, S., editores, *Proceedings of the ISCA 9th International Conference on Parallel and Distributed Computing Systems*, v. 1, p. 330–337, Dijon. ISCA.

Schwiegelshohn, U.; Yahyapour, R. (1998). Improving first-come-first-serve job scheduling by gang scheduling. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 180–198. LNCS 1459.

Senger, L. J. (1997). Avaliação de Desempenho do PVMW95. Dissertação (Mestrado), Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo.

Senger, L. J.; Caldas Jr., J. (2001). Análise de Risco de Crédito Utilizando Redes Neurais Artificiais. *Revista do CCEI*, v.5, n.8, p.19–26.

Senger, L. J.; de Gouveia, L. T. (2000). A computação paralela distribuída com a utilização do PVM 3.4. *Revista do CCEI*, v.4, n.6.

Senger, L. J.; Mello, R.; Santana, M. J.; Santana, R. H. C. (2005). An on-line approach for classifying and extracting application behavior on linux. *High Performance Computing: Paradigm and Infrastructure*. John Wiley and Sons Inc. (a ser publicado).

Senger, L. J.; Santana, M. J.; Santana, R. H. C. (2002). Uma nova abordagem para a aquisição de conhecimento sobre o comportamento de aplicações paralelas na utilização de recursos. *Proceedings of WORKCOMP*, p. 79–85.

Senger, L. J.; Santana, M. J.; Santana, R. H. C. (2003). A new approach for acquiring knowledge of resource usage in parallel applications. *Proceedings of International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'2003)*, p. 607–614.

Senger, L. J.; Santana, M. J.; Santana, R. H. C. (2004a). An instance-based learning approach for predicting parallel applications execution times. *Proceedings of 3rd International Information and Telecommunication Technologies Symposium*, p. 9–15, São Carlos Federal State University - UFSCar, São Carlos, SP, Brasil.

Senger, L. J.; Santana, M. J.; Santana, R. H. C. (2004b). Using runtime measurements and historical traces for acquiring knowledge in parallel applications. *International Conference on Computational Science (ICCS'2004)*, v. LNCS 3036, p. 661–665. Springer.

Setia, S.; Tripathi, S. (1993). A comparative analysis of static processor partitioning policies for parallel computers. *International Workshop on Modeling and Simulation of Computer and Telecommunication Systems (MASCOTS)*, p. 283–286.

Sevcik, K. C. (1989). Characterizations of Parallelism in Applications and their use in Scheduling. *Performance Evaluation Review*, v.17, n.1, p.171–180.

Shefler, W. C. (1988). *Statistics: Concepts and Applications*. The Benjamin/Cummings.

Shivaratri, N. G.; Krueger, P.; Singhal, M. (1992). Load distributing for locally distributed systems. *IEEE Computer*, v.25, n.12, p.33–44.

Silva, F. A. B. D.; Scherson, I. D. (2000). Improving Parallel Job Scheduling Using Runtime Measurements. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 18–38. LNCS 1911.

Sivasubramaniam, A.; Singla, A.; Ramachandran, U.; Venkateswaran, H. (1994). An approach to scalability study of shared memory parallel systems. *Measurement and Modeling of Computer Systems*, p. 171–180.

Slobodcicov, I. (2003). Implementação em paralelo do método de elementos finitos para equações de águas rasas. Dissertação (Mestrado), Universidade Federal do Rio de Janeiro (COPPE/UFRJ).

Smith, W.; Foster, I.; Taylor, V. (1998a). Predicting application run times using historical information. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 122–142. LNCS 1459.

Smith, W.; Foster, I. T.; Taylor, V. E. (1998b). Predicting Application Run Times Using Historical Information. *JSSPP*, p. 122–142.

Smith, W.; Taylor, V.; Foster, I. (1999). Using run-time predictions to estimate queue wait times and improve scheduler performance. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 202–219. LNCS 1659.

Souza, P. S. L. (2000). *AMIGO: Uma Contribuição para a Convergência na Área de Escalonamento de Processos*. Tese (Doutorado), IFSC-USP.

Souza, P. S. L.; Santana, M. J.; Santana, R. H. C. (1999). A new scheduling environment for near-optimal performance. *International Conference on Parallel and Distributed Processing Techniques and Applications -PDPTA'99*, Las Vegas, Nevada, U.S.A.

Souza, P. S. L.; Santana, M. J.; Santana, R. H. C.; Senger, L. J.; Picinato, R. (1997). O impacto do protocolo TCP/IP na computação paralela distribuída no ambiente windows95. *15o Simpósio Brasileiro de Redes de Computadores (SBRC97)*, São Carlos.

Squillante, M. S.; Yao, D. D.; Zhang, L. (1999). The impact of job arrival patterns on parallel scheduling. *Perf. Eval. Rev.*, v.26, n.4, p.52–59.

Srinivasan, S.; Kettimuthu, R.; Subramani, V.; Sadayappan, P. (2002). Selective reservation strategies for backfill job scheduling. Feitelson, D. G.; Rudolph, L.; Schwiegelshohn, U., editores, *Job Scheduling Strategies for Parallel Processing*, p. 55–71. LNCS 2537.

Sunderam, V. S.; Geist, G. A.; Dongarra, J.; Manchek, R. (1994). The PVM concurrent computing system: Evolution, experiences and trends. *Parallel Computing*, v.20, p.531–545.

Talby, D.; Feitelson, D. G.; Raveh, A. (1999). Comparing logs and models of parallel workloads using the co-plot method. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 43–66. Springer Verlag. LNCS 1659.

Tanenbaum, A. S. (1992). *Modern Operating Systems*. Prentice Hall, New Jersey.

Tanenbaum, A. S. (1995). *Sistemas Operacionais Modernos*. Prentice-Hall do Brasil, Rio de Janeiro.

Terada, R. (1991). *Desenvolvimento de Algoritmos e Estruturas de Dados*. Makron Books, Rio de Janeiro.

Thain, D.; Tannenbaum, T.; Livny, M. (2004). Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*. Wiley Interscience, (a ser publicado).

Theimer, M. M.; Lantz, K. A.; Cheriton, D. R. (1985). Preemptable remote execution facilities for the V-System. *Proceedings of the Tenth Symposium on Operating System Priciples*, p. 2–12.

Ultsch, A. (1993). Self-organising neural networks for monitoring and knowledge acquisition of a chemical process. *Proceedings of ICANN-93*, p. 864–867.

Vetter, J. S.; Mueller, F. (2003). Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput.*, v.63, n.9, p.853–865.

Vicentini, J. F. (2002). Indexação e recuperação de informações utilizando redes neurais da família ART. Dissertação (Mestrado), Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo.

Vlajic, N.; Card, H. C. (2001). Vector quantization of images using modified adaptive resonance algorithm for hierarchical clustering. *IEEE Transactions on Neural Network*, v.12, n.5.

von Eicken, T.; Culler, D. E.; Goldstein, S. C.; Schauser, K. E. (1992). Active messages: A mechanism for integrated communication and computation. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Goald Cost, Australia.

Wan, M.; Moore, R.; Kremenek, G.; Steube, K. (1996). A batch scheduler for the Intel Paragon with a non-contiguous node allocation algorithm. Feitelson, D. G.; Rudolph, L., editores, *Job Scheduling Strategies for Parallel Processing*, p. 48–64. LNCS 1162.

White, S.; Alund, A.; Sunderam, V. S. (1994). Performance of the NAS parallel benchmarks on PVM based networks. Relatório Técnico RNR-94-008, Department of Mathematics and Computer Science, Emory University, Atlanta, GA, USA.

Whiteley, J. R.; Davis, J. F. (1993). Qualitative interpretation of sensor patterns. *IEEE Expert*, v.8, p.54–63.

Whiteley, J. R.; Davis, J. F. (1996). Observations and problems applying ART2 for dynamic sensor pattern interpretation. *IEEE Transactions on Systems, Man and Cybernetics-Part A: Systems and Humans*, v.26, n.4, p.423–437.

Wilkinson, B.; Allen, M. (2004). *Parallel Programming: Techniques and Applications using networked workstations and parallel computers*. Pearson/ Prentice Hall.

Wilson, D. R.; Martinez, T. R. (1997a). Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research*, v.6, p.1–34.

Wilson, D. R.; Martinez, T. R. (1997b). Instance pruning techniques. *Proc. 14th International Conference on Machine Learning*, p. 403–411. Morgan Kaufmann.

Wilson, D. R.; Martinez, T. R. (2000). Reduction techniques for instance-based learning algorithms. *Machine Learning*, v.38, n.3, p.257–286.

Witten, I. H.; Frank, E. (1999). *Data mining: pratical machine learning tools and techniques with java implementations*. Morgan Kaufmann.

Worley, P. H. (1998). Impact of communication protocol on performance. *Proceeedings of the Second International Workshop on Software Engineering and Code Design in Parallel Meteorological and Oceanographic Applications*, Scottsdale, AZ.

Worley, P. H. (2000). Performance evaluation of the ibm sp and the compaq alphaserver sc. *Proceedings of the ACM International Conference of Supercomputing*, Santa Fe, New Mexico.

Worley, P. H. (2002). Scaling the unscalable: A case study on the alphaserver sc. *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing*, Baltimore, MD.

Worley, P. H.; Jr., T. H. D.; Fahey, M. R.; III, J. B. W.; Bland, A. S. (2002). Early evaluation of the ibm p690. *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing*, Baltimore, MD.

Worley, P. H.; Robinson, A. C.; Mackay, D. R.; Barragy, E. J. (1998). A study of application sensitivity to variation in message passing latency and bandwidth. *Concurrency: Practice and Experience*, v.5, n.10, p.387–406.

Worley, P. H.; Toonen, B. (1995). A users' guide to PSTSWM. Relatório técnico, ORNL Technical Report ORNL/TM-12779.

Xu, C.; He, B. (2000). Pcb: A distributed computing system in corba. *Journal of Parallel and Distributed Computing*, v.60, n.10, p.1293–1310.

Zaki, M. J.; Li, W.; Cierniak, M. (1995). Performance impact of processor and memory heterogeneity in a network of machines. Relatório Técnico 574, Department of Computer Science, University of Rochester.

Zhang, Y.; Sivasubramaniam, A.; Moreira, J.; Franke, H. (2000). A simulation based study of scheduling mechanisms for a dynamic cluster environment. *Proceedings of the 2000 international conference on Supercomputing*, p. 100–109, Santa Fe, New Mexico, USA. ACM.

Zhou, S.; Ferrari, D. (1987). An experimental study of load balancing performance. Relatório Técnico UCB/CSD 87/336, Universidade da Califórnia, Berkeley.

Ziv, A.; Bruck, J. (1996). Checkpointing in Parallel and Distributed Systems. Zomaya, A. Y. H., editor, *Parallel and Distributed Computing Handbook*, p. 274–301. McGraw-Hill.

APÊNDICE

A

Tabelas de Resultados

Este Apêndice apresenta os resultados obtidos através dos experimentos realizados nesta tese. Os resultados acompanham suas estatísticas descritivas, como os valores de média, variância e desvio padrão, assim como os resultados das inferências estatísticas empregadas. A análise estatística da variância, ANOVA, e os testes de comparação de médias Z, T e $Tukey$ são descritos em detalhes em Shefler (1988).

Tabela A.1: Resultados NAS para núcleo não modificado (EP_1)

	BT	CG	EP	FT	IS	LU	MG	SP
	37,13	3,33	8,75	1,62	0,25	74,15	2,74	85,63
	37,10	3,31	8,75	1,63	0,27	74,12	2,73	85,47
	37,14	3,32	8,74	1,62	0,25	74,14	2,75	85,54
	37,11	3,33	8,76	1,63	0,25	74,09	2,73	85,55
	37,14	3,32	8,75	1,62	0,25	74,18	2,76	85,60
	37,11	3,31	8,74	1,62	0,25	74,20	2,75	85,59
	37,16	3,32	8,74	1,63	0,25	74,08	2,73	85,40
	37,14	3,32	8,75	1,62	0,25	74,15	2,72	85,47
	37,14	3,31	8,74	1,63	0,25	74,16	2,72	85,63
	37,20	3,32	8,74	1,62	0,25	74,25	2,74	85,62
	37,17	3,32	8,75	1,62	0,25	74,12	2,74	85,57
	37,16	3,32	8,74	1,63	0,25	74,20	2,72	85,60
	37,20	3,31	8,74	1,62	0,25	74,15	2,74	85,57
	37,17	3,33	8,75	1,63	0,25	74,22	2,77	85,62
	37,18	3,32	8,74	1,62	0,25	74,18	2,73	85,57
	37,24	3,32	8,74	1,62	0,25	74,11	2,78	85,61
	37,20	3,31	8,75	1,63	0,25	74,09	2,72	85,47
	37,20	3,33	8,74	1,62	0,25	74,24	2,73	85,57
	37,14	3,32	8,74	1,62	0,25	74,11	2,71	85,67
	37,25	3,32	8,75	1,62	0,25	74,06	2,77	85,48
	37,22	3,31	8,74	1,62	0,25	74,16	2,72	85,65
	37,16	3,33	8,74	1,63	0,25	74,16	2,74	85,58
	37,18	3,32	8,75	1,62	0,25	74,13	2,74	85,43
	37,20	3,32	8,74	1,62	0,25	74,14	2,74	85,61
	37,21	3,32	8,74	1,62	0,26	74,07	2,75	85,62
	37,18	3,33	8,76	1,63	0,25	74,15	2,72	85,68
	37,17	3,33	8,74	1,63	0,25	74,10	2,76	85,53
	37,20	3,31	8,74	1,62	0,25	74,23	2,72	85,63
	37,22	3,32	8,75	1,62	0,25	74,13	2,72	85,57
	37,18	3,32	8,74	1,63	0,25	74,26	2,75	85,53
	37,17	3,33	8,75	1,62	0,26	74,07	2,72	85,46
	37,19	3,32	8,74	1,63	0,25	74,20	2,72	85,65
	37,18	3,32	8,77	1,62	0,25	74,10	2,72	85,59
	37,20	3,31	8,74	1,62	0,25	74,22	2,74	85,63
	37,27	3,33	8,74	1,63	0,25	74,15	2,73	85,57
	37,18	3,31	8,77	1,62	0,25	74,26	2,72	85,68
	37,18	3,33	8,76	1,63	0,25	74,17	2,74	85,61
	37,28	3,32	8,74	1,62	0,25	74,16	2,73	85,63
	37,18	3,31	8,75	1,63	0,25	74,16	2,74	85,66
	37,20	3,31	8,74	1,63	0,25	74,20	2,73	85,61
	37,24	3,33	8,74	1,62	0,25	74,08	2,73	85,50
	37,20	3,32	8,75	1,63	0,25	74,15	2,72	85,51
	37,18	3,31	8,74	1,62	0,25	74,20	2,75	85,60
	37,19	3,32	8,74	1,62	0,25	74,19	2,73	85,51
	37,25	3,33	8,75	1,62	0,25	74,15	2,74	85,48
	37,19	3,32	8,74	1,61	0,25	74,17	2,73	85,54
	37,17	3,31	8,74	1,64	0,25	74,07	2,77	85,61
	37,21	3,32	8,75	1,62	0,25	74,13	2,72	85,59
	37,20	3,33	8,74	1,62	0,25	74,15	2,74	85,54
	37,20	3,32	8,74	1,63	0,25	74,17	2,72	85,57
	37,19	3,31	8,75	1,62	0,25	74,04	2,73	85,51
Média	37,18	3,31	8,74	1,62	0,25	74,15	2,73	85,57
Variância	0,00144	0,00005	0,00006	0,00003	0,00001	0,00288	0,00025	0,0044
Desvio padrão	0,03791	0,00735	0,00783	0,00564	0,00337	0,05363	0,01591	0,06654

Tabela A.2: Resultados NAS para núcleo instrumentado (EP_1)

	BT	CG	EP	FT	IS	LU	MG	SP
	37,22	3,32	8,75	1,63	0,25	74,16	2,74	85,66
	37,20	3,33	8,75	1,62	0,25	74,14	2,75	85,51
	37,13	3,32	8,78	1,63	0,25	74,14	2,73	85,60
	37,16	3,32	8,74	1,62	0,25	74,14	2,74	85,45
	37,16	3,32	8,75	1,62	0,25	74,14	2,74	85,68
	37,17	3,33	8,74	1,63	0,25	74,11	2,72	85,56
	37,17	3,31	8,74	1,62	0,25	74,13	2,73	85,49
	37,15	3,33	8,76	1,63	0,25	74,08	2,73	85,38
	37,19	3,32	8,74	1,63	0,25	74,16	2,72	85,64
	37,16	3,31	8,74	1,62	0,25	74,12	2,78	85,50
	37,24	3,32	8,75	1,63	0,25	74,07	2,73	85,52
	37,16	3,32	8,74	1,62	0,25	74,12	2,73	85,68
	37,21	3,31	8,74	1,63	0,25	74,11	2,73	85,61
	37,20	3,32	8,75	1,62	0,25	74,19	2,74	85,54
	37,16	3,33	8,75	1,62	0,25	74,10	2,74	85,49
	37,18	3,32	8,74	1,63	0,25	74,13	2,74	85,42
	37,21	3,31	8,75	1,62	0,26	74,14	2,74	85,46
	37,25	3,32	8,74	1,62	0,25	74,21	2,73	85,61
	37,18	3,33	8,74	1,57	0,25	74,19	2,75	85,58
	37,19	3,32	8,75	1,62	0,25	74,17	2,72	85,52
	37,19	3,32	8,74	1,63	0,25	74,21	2,74	85,62
	37,15	3,32	8,74	1,62	0,25	74,16	2,74	85,67
	37,19	3,33	8,75	1,62	0,25	74,13	2,71	85,61
	37,30	3,32	8,74	1,62	0,25	74,09	2,75	85,61
	37,18	3,33	8,74	1,63	0,25	74,13	2,73	85,60
	37,19	3,32	8,75	1,63	0,25	74,10	2,74	85,62
	37,15	3,33	8,74	1,62	0,25	74,15	2,75	85,65
	37,21	3,32	8,74	1,62	0,25	74,32	2,72	85,67
	37,26	3,31	8,75	1,63	0,25	74,14	2,72	85,65
	37,19	3,31	8,74	1,61	0,25	74,07	2,74	85,65
	37,28	3,33	8,74	1,63	0,25	74,11	2,73	85,66
	37,17	3,33	8,74	1,62	0,25	74,15	2,74	85,50
	37,19	3,31	8,80	1,62	0,25	74,22	2,73	85,54
	37,31	3,32	8,74	1,63	0,25	74,12	2,76	85,59
	37,20	3,33	8,78	1,62	0,25	74,20	2,75	85,54
	37,18	3,32	8,75	1,63	0,25	74,07	2,72	85,65
	37,20	3,31	8,74	1,62	0,25	74,08	2,75	85,53
	37,22	3,32	8,79	1,62	0,25	74,12	2,74	85,54
	37,19	3,33	8,75	1,63	0,25	74,11	2,73	85,70
	37,19	3,31	8,74	1,62	0,26	74,14	2,72	85,65
	37,29	3,32	8,74	1,63	0,25	74,20	2,76	85,63
	37,18	3,33	8,75	1,62	0,25	74,15	2,73	85,57
	37,18	3,32	8,74	1,62	0,25	74,14	2,74	85,59
	37,29	3,31	8,74	1,63	0,25	74,09	2,73	85,50
	37,20	3,32	8,75	1,62	0,25	74,23	2,75	85,57
	37,19	3,33	8,74	1,62	0,25	74,08	2,74	85,67
	37,19	3,32	8,74	1,62	0,25	74,18	2,77	85,52
	37,22	3,31	8,75	1,62	0,25	74,19	2,74	85,60
	37,18	3,33	8,74	1,63	0,25	74,16	2,72	85,53
	37,19	3,32	8,74	1,62	0,25	74,15	2,72	85,56
	37,18	3,32	8,75	1,62	0,25	74,19	2,72	85,47
	37,19	3,32	8,74	1,62	0,25	74,14	2,73	85,57
Média	0,00163	0,00005	0,00017	0,00008	0,00000	0,00234	0,00019	0,00564
Variância	0,04032	0,00717	0,01309	0,00913	0,00196	0,04837	0,01383	0,0750
Desvio padrão	0,04032	0,00717	0,01309	0,00913	0,00196	0,04837	0,01383	0,07508

Tabela A.3: Resultados NAS com monitor (250000 microssegundos, EP_1)

	BT	CG	EP	FT	IS	LU	MG	SP
	37,32	3,34	8,78	1,59	0,26	74,19	2,73	85,49
	37,27	3,33	8,78	1,63	0,28	74,22	2,75	85,65
	37,25	3,31	8,78	1,63	0,27	74,29	2,76	85,69
	37,28	3,32	8,78	1,64	0,27	74,11	2,75	85,45
	37,29	3,34	8,78	1,59	0,27	74,23	2,74	85,76
	37,30	3,34	8,78	1,62	0,27	74,27	2,77	85,88
	37,28	3,32	8,78	1,56	0,27	74,40	2,75	85,64
	37,31	3,33	8,78	1,54	0,26	74,27	2,75	85,40
	37,28	3,33	8,78	1,63	0,26	74,10	2,74	85,42
	37,17	3,33	8,78	1,60	0,26	74,27	2,71	85,30
	37,26	3,32	8,78	1,63	0,26	74,17	2,74	85,55
	37,31	3,34	8,78	1,50	0,27	74,09	2,74	85,63
	37,26	3,34	8,78	1,58	0,26	73,96	2,71	85,55
	37,29	3,33	8,78	1,61	0,26	74,02	2,75	85,59
	37,25	3,33	8,78	1,60	0,26	74,11	2,74	85,29
	37,27	3,32	8,78	1,63	0,26	74,09	2,73	85,52
	37,27	3,33	8,78	1,64	0,26	74,22	2,71	85,54
	37,00	3,33	8,78	1,59	0,26	74,28	2,75	85,62
	37,27	3,33	8,78	1,58	0,26	74,04	2,74	85,62
	37,26	3,32	8,78	1,62	0,26	74,37	2,72	85,24
	37,33	3,34	8,78	1,60	0,26	74,30	2,73	85,33
	37,31	3,34	8,79	1,61	0,26	74,42	2,74	85,61
	37,27	3,33	8,79	1,63	0,26	74,32	2,75	85,55
	37,30	3,31	8,78	1,64	0,26	74,35	2,72	85,60
	37,29	3,34	8,78	1,60	0,26	74,26	2,72	85,55
	37,27	3,34	8,79	1,63	0,26	74,35	2,75	85,60
	37,29	3,32	8,78	1,63	0,28	74,40	2,74	85,45
	37,27	3,32	8,78	1,64	0,26	74,35	2,72	85,46
	37,28	3,33	8,79	1,63	0,26	74,61	2,75	85,43
	37,29	3,34	8,79	1,63	0,26	74,71	2,75	85,52
	37,29	3,32	8,79	1,64	0,26	74,63	2,73	85,63
	37,32	3,33	8,79	1,63	0,26	74,61	2,74	85,36
	37,28	3,34	8,79	1,62	0,26	74,61	2,74	85,55
	37,28	3,33	8,79	1,64	0,26	74,60	2,72	85,60
	37,27	3,33	8,79	1,62	0,26	74,69	2,74	85,42
	37,29	3,33	8,79	1,63	0,26	74,44	2,73	85,39
	37,28	3,33	8,79	1,64	0,26	74,47	2,73	85,22
	37,30	3,34	8,79	1,64	0,26	74,59	2,73	85,51
	37,32	3,34	8,79	1,62	0,26	74,54	2,74	85,48
	37,30	3,32	8,79	1,59	0,28	74,53	2,72	85,47
	37,32	3,34	8,79	1,63	0,26	74,45	2,72	85,71
	37,31	3,34	8,80	1,63	0,26	74,39	2,74	85,70
	37,28	3,33	8,79	1,63	0,28	74,66	2,73	85,55
	37,32	3,34	8,79	1,64	0,26	74,57	2,73	85,59
	37,25	3,35	8,79	1,62	0,26	74,74	2,75	85,48
	37,32	3,33	8,79	1,63	0,25	74,66	2,73	85,29
	37,29	3,33	8,79	1,64	0,26	74,67	2,73	85,58
	37,33	3,33	8,80	1,62	0,26	74,61	2,74	85,72
	37,31	3,32	8,79	1,63	0,26	74,76	2,71	85,70
	37,31	3,34	8,79	1,63	0,26	74,57	2,74	85,44
	37,31	3,34	8,79	1,63	0,26	74,57	2,74	85,44
Média	37,28	3,33	8,78	1,61	0,26	74,39	2,73	85,52
Variância	0,00237	0,00008	0,00003	0,00077	0,00004	0,04601	0,00018	0,01909
Desvio padrão	0,04865	0,00887	0,00577	0,02769	0,00627	0,21450	0,01344	0,1381

Tabela A.4: Teste Z para EP_1

Núcleo não modificado comparado ao núcleo instrumentado Valor crítico= 1,96								
	BT	CG	EP	FT	IS	LU	MG	SP
Estimativa do erro	0,00783	0,00145	0,00216	0,00152	0,00055	0,01021	0,00298	0,01419
Valor de Z	-1,67851	-0,67546	-0,90904	0,77502	0,71088	0,74871	-0,32879	-0,34551
Teste de Hipóteses	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0
Núcleo não modificado comparado ao núcleo instrumentado e monitor com intervalo de amostragem igual a 250000 microssegundos Valor crítico= 1,96								
Estimativa do erro	0,00872	0,00163	0,00137	0,00400	0,00101	0,03127	0,00295	0,02169
Valor de Z	-1,50611	-0,60214	-1,42619	0,29439	0,38927	0,24456	-0,33278	-0,22603
Teste de Hipóteses	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0

Tabela A.5: Resultados NAS para núcleo não modificado (EP_2)

	BT	CG	EP	IS	LU	MG	SP
	396,01	91,96	65,65	9,35	829,93	43,88	1083,26
	395,94	91,94	65,64	9,35	829,85	43,88	1083,17
	395,94	91,94	65,64	9,35	829,85	43,88	1083,17
	395,99	91,96	65,64	9,35	829,96	43,89	1083,26
	395,98	91,96	65,65	9,35	829,90	43,88	1083,19
	396,00	91,96	65,64	9,35	829,96	43,89	1083,27
	396,00	91,95	65,65	9,35	829,95	43,89	1083,26
	395,94	91,94	65,64	9,35	829,85	43,88	1164,41
	391,34	91,61	65,51	6,82	805,99	42,82	1047,39
	380,13	91,59	64,73	7,21	809,23	43,00	1042,85
	384,88	91,57	65,11	8,19	799,85	42,88	1041,49
	388,69	91,68	64,97	7,96	806,39	42,54	1046,68
	388,99	91,60	65,70	7,63	797,15	42,88	1044,11
	389,11	91,68	64,86	7,93	796,86	42,52	1044,92
	389,18	91,59	65,15	7,43	799,14	42,87	1083,26
Média	392,008	91,795	65,412	8,531	816,924	43,372	1073,179
Variância	25,340	0,031	0,118	0,919	217,101	0,335	1004,727
Desvio padrão	5,034	0,175	0,343	0,959	14,734	0,579	31,69

Tabela A.6: Resultados NAS para núcleo instrumentado (EP_2)

	BT	CG	EP	IS	LU	MG	SP
	395,96	92,16	65,65	9,35	830,21	43,72	1083,17
	395,95	92,16	65,64	9,35	830,21	43,73	1083,17
	395,94	92,15	65,64	9,35	830,21	43,72	1083,17
	395,94	92,16	65,65	9,35	830,21	43,72	1083,16
	395,94	92,15	65,65	9,35	830,21	43,73	1083,16
	395,95	92,16	65,64	9,35	830,21	43,72	1083,17
	395,95	92,15	65,65	9,35	830,21	43,73	1083,17
	395,95	92,16	65,64	9,35	830,21	43,72	1083,17
	395,95	92,15	65,65	9,35	902,70	42,81	1034,54
	388,32	91,61	65,44	8,75	770,26	42,81	1046,72
	389,74	91,40	65,42	8,27	771,53	42,91	1058,96
	389,80	91,75	65,37	7,85	771,68	41,54	1055,42
	391,97	91,66	65,38	8,32	770,83	40,13	1056,85
	389,77	91,73	65,36	8,02	811,71	39,40	1055,89
	390,34	91,70	65,40	7,39	811,78	40,70	1061,54
Média	393,565	91,950	65,545	8,850	816,811	42,673	1069,017
Variância	9,626	0,074	0,016	0,478	1230,924	2,234	282,142
Desvio padrão	3,103	0,272	0,128	0,692	35,085	1,495	16,797

Tabela A.7: Resultados NAS com monitor (250000 microssegundos, EP_2)

	BT	CG	EP	IS	LU	MG	SP
	393,24	91,71	65,4	9,4	844,07	44,04	1090,28
	399,82	92,39	65,36	9,41	844,26	44,06	1090,62
	399,88	91,71	65,32	9,41	844,38	44,06	1090,88
	387,88	91,71	65,37	9,41	844,6	44,07	1091,01
	390,39	92,45	66,09	9,41	844,84	44,09	1091,27
	392,65	92,48	66,1	9,42	844,94	61,42	1144,11
	387,88	91,96	65,37	9,4	787,73	41,91	1059,43
	391,98	91,71	65,48	9,4	824,01	39,28	1063,17
	394,34	92,39	65,52	9,38	804,59	35,37	1051,25
	393,24	91,71	65,62	4,77	822,22	40,46	1054,14
	391,24	91,86	65,52	9,34	800,04	42,23	1046,09
	390,39	91,71	65,59	8,49	810,12	41,27	1047,48
	392,03	91,12	65,56	9,04	791,46	41,06	1042,84
	388,88	91,71	65,38	7,57	809,94	42,47	1045,85
	392,65	91,76	65,53	7,72	822,7	38,9	1046,16
Média	392,433	91,892	65,547	8,771	822,660	42,979	1070,305
Variância	12,806	0,143	0,058	1,617	444,230	32,078	813,337
Desvio padrão	3,579	0,378	0,241	1,272	21,077	5,664	28,519

Tabela A.8: Teste T para EP_2

Núcleo não modificado comparado ao núcleo instrumentado 28 graus de liberdade, valor crítico= 2,048							
	BT	CG	EP	IS	LU	MG	SP
Estimativa da variância	17,482	0,052	0,067	0,698	724,012	1,284	643,434
Estimativa do erro	1,526	0,083	0,094	0,305	9,825	0,413	9,26
Valor de T	-1,019	-1,853	-1,409	-1,044	0,011	1,689	0,449
Teste de Hipóteses	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0
Núcleo não modificado comparado ao núcleo instrumentado e monitor com intervalo de amostragem igual a 250000 microssegundos 28 graus de liberdade, valor crítico= 2,048							
Estimativa da variância	19,072	0,086	0,087	1,262	330,665	16,206	909,031
Estimativa do erro	1,127	0,076	0,076	0,290	4,695	1,039	7,784
Valor de T	-0,376	-1,269	-1,768	-0,825	-1,221	0,377	0,369
Teste de Hipóteses	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0	Aceita H_0

Tabela A.9: Resultados algoritmo IBL para a carga de trabalho SDSC95

Grupo	Parâmetros IBL	Tamanho	Soma	Média	Variância
A	$(t^2 = 0,125, k = 5)$	70	40,21934	0,57456	0,01132
B	$(t^2 = 0,125, k = 10)$	70	40,63904	0,58056	0,01313
C	$(t^2 = 0,125, k = 25)$	70	41,86917	0,59813	0,01156
D	$(t^2 = 0,125, k = 50)$	70	43,03173	0,61474	0,01077
E	$(t^2 = 0,250, k = 5)$	70	40,22594	0,57466	0,01142
F	$(t^2 = 0,250, k = 10)$	70	40,82623	0,58323	0,01307
G	$(t^2 = 0,250, k = 25)$	70	42,48532	0,60693	0,01167
H	$(t^2 = 0,250, k = 50)$	70	44,23021	0,63186	0,01129
I	$(t^2 = 0,500, k = 5)$	70	40,27608	0,57537	0,01150
J	$(t^2 = 0,500, k = 10)$	70	41,04209	0,58632	0,01289
K	$(t^2 = 0,500, k = 25)$	70	43,04077	0,61487	0,01136
L	$(t^2 = 0,500, k = 50)$	70	45,05125	0,64359	0,01167
M	$(t^2 = 1,000, k = 5)$	70	40,31469	0,57592	0,01158
N	$(t^2 = 1,000, k = 10)$	70	41,19935	0,58856	0,01269
O	$(t^2 = 1,000, k = 25)$	70	43,40386	0,62006	0,01126
P	$(t^2 = 1,000, k = 50)$	70	45,43587	0,64908	0,01190
Q	$(t^2 = 2,000, k = 5)$	70	40,33962	0,57628	0,01161
R	$(t^2 = 2,000, k = 10)$	70	41,30557	0,59008	0,01257
S	$(t^2 = 2,000, k = 25)$	70	43,57563	0,62251	0,01125
T	$(t^2 = 2,000, k = 50)$	70	45,60437	0,65149	0,01200

Tabela A.10: Tabela ANOVA resultados IBL carga de trabalho SDSC95

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	0,93207	19	0,04906	4,14840	1,91805
Dentro dos grupos	16,31907	1380	0,01183		
Totais	17,25115	1399			

Tabela A.11: Teste Estendido de Tukey carga de trabalho SDSC95

valor crítico de igual a 0,065																			
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
B	0,006																		
C	0,024	0,018																	
D	0,040	0,034	0,017																
E	0,000	0,006	0,023	0,040															
F	0,009	0,003	0,015	0,032	0,009														
G	0,032	0,026	0,009	0,008	0,032	0,024													
H	0,057	0,051	0,034	0,017	0,057	0,049	0,025												
I	0,001	0,005	0,023	0,039	0,001	0,008	0,032	0,056											
J	0,012	0,006	0,012	0,028	0,012	0,003	0,021	0,046	0,011										
K	0,040	0,034	0,017	0,000	0,040	0,032	0,008	0,017	0,039	0,029									
L	**0,069**	0,063	0,045	0,029	**0,069**	0,060	0,037	0,012	**0,068**	0,057	0,029								
M	0,001	0,005	0,022	0,039	0,001	0,007	0,031	0,056	0,001	0,010	0,039	**0,068**							
N	0,014	0,008	0,010	0,026	0,014	0,005	0,018	0,043	0,013	0,002	0,026	0,055	0,013						
O	0,045	0,039	0,022	0,005	0,045	0,037	0,013	0,012	0,045	0,034	0,005	0,024	0,044	0,031					
P	**0,075**	**0,069**	0,051	0,034	**0,074**	**0,066**	0,042	0,017	**0,074**	0,063	0,034	0,005	**0,073**	0,061	0,029				
Q	0,002	0,004	0,022	0,038	0,002	0,007	0,031	0,056	0,001	0,010	0,039	**0,067**	0,000	0,012	0,044	**0,073**			
R	0,016	0,010	0,008	0,025	0,015	0,007	0,017	0,042	0,015	0,004	0,025	0,054	0,014	0,002	0,030	0,059	0,014		
S	0,048	0,042	0,024	0,008	0,048	0,039	0,016	0,009	0,047	0,036	0,008	0,021	0,047	0,034	0,002	0,027	0,046	0,032	
T	**0,077**	**0,071**	0,053	0,037	**0,077**	**0,068**	0,045	0,020	**0,076**	**0,065**	0,037	0,008	**0,076**	0,063	0,031	0,002	**0,075**	0,061	0,029

Tabela A.12: Resultados algoritmo IBL para a carga de trabalho SDSC96

Grupo	Parâmetros IBL	Tamanho	Soma	Média	Variância
A	$(t^2 = 0,125, k = 5)$	60	32,62200	0,54370	0,01364
B	$(t^2 = 0,125, k = 10)$	60	32,26364	0,53773	0,01445
C	$(t^2 = 0,125, k = 25)$	60	32,77442	0,54624	0,01512
D	$(t^2 = 0,125, k = 50)$	60	33,29156	0,55486	0,01502
E	$(t^2 = 0,250, k = 5)$	60	32,72316	0,54539	0,01342
F	$(t^2 = 0,250, k = 10)$	60	32,37187	0,53953	0,01447
G	$(t^2 = 0,250, k = 25)$	60	33,08682	0,55145	0,01503
H	$(t^2 = 0,250, k = 50)$	60	33,98744	0,56646	0,01503
I	$(t^2 = 0,500, k = 5)$	60	32,74861	0,54581	0,01334
J	$(t^2 = 0,500, k = 10)$	60	32,47468	0,54124	0,01447
K	$(t^2 = 0,500, k = 25)$	60	33,36695	0,55612	0,01520
L	$(t^2 = 0,500, k = 50)$	60	34,71753	0,57863	0,01486
M	$(t^2 = 1,000, k = 5)$	60	32,77915	0,54632	0,01334
N	$(t^2 = 1,000, k = 10)$	60	32,50789	0,54180	0,01447
O	$(t^2 = 1,000, k = 25)$	60	33,58893	0,55982	0,01519
P	$(t^2 = 1,000, k = 50)$	60	35,15799	0,58597	0,01451
Q	$(t^2 = 2,000, k = 5)$	60	32,79380	0,54656	0,01334
R	$(t^2 = 2,000, k = 10)$	60	32,52524	0,54209	0,01447
S	$(t^2 = 2,000, k = 25)$	60	33,73753	0,56229	0,01509
T	$(t^2 = 2,000, k = 50)$	60	35.35449	0,58924	0,01430

Tabela A.13: Tabela ANOVA dos resultados IBL carga de trabalho SDSC96

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	0,26935	19	0,01418	0,98159	1,92030
Dentro dos grupos	17,04188	1180	0,01444		
Totais	17,31123	1199			

Tabela A.14: Resultados algoritmo IBL para a carga de trabalho SDSC2000

Grupo	Parâmetros IBL	Tamanho	Soma	Média	Variância
A	$(t^2 = 0,125, k = 5)$	60	30,03411	0,50057	0,02558
B	$(t^2 = 0,125, k = 10)$	60	30,59894	0,50998	0,02591
C	$(t^2 = 0,125, k = 25)$	60	31,17181	0,51953	0,02683
D	$(t^2 = 0,125, k = 50)$	60	31,43887	0,52398	0,02652
E	$(t^2 = 0,250, k = 5)$	60	30,36675	0,50611	0,02569
F	$(t^2 = 0,250, k = 10)$	60	31,14121	0,51902	0,02642
G	$(t^2 = 0,250, k = 25)$	60	32,36327	0,53939	0,02819
H	$(t^2 = 0,250, k = 50)$	60	33,70938	0,56182	0,02742
I	$(t^2 = 0,500, k = 5)$	60	30,56559	0,50943	0,02602
J	$(t^2 = 0,500, k = 10)$	60	31,56985	0,52616	0,02723
K	$(t^2 = 0,500, k = 25)$	60	33,34674	0,55578	0,02880
L	$(t^2 = 0,500, k = 50)$	60	35,67287	0,59455	0,02872
M	$(t^2 = 1,000, k = 5)$	60	30,69592	0,51160	0,02611
N	$(t^2 = 1,000, k = 10)$	60	31,85006	0,53083	0,02734
O	$(t^2 = 1,000, k = 25)$	60	33,90606	0,56510	0,02881
P	$(t^2 = 1,000, k = 50)$	60	36,67182	0,61120	0,02972
Q	$(t^2 = 2,000, k = 5)$	60	30,77518	0,51292	0,02612
R	$(t^2 = 2,000, k = 10)$	60	31,99744	0,53329	0,02743
S	$(t^2 = 2,000, k = 25)$	60	34,16461	0,56941	0,02884
T	$(t^2 = 2,000, k = 50)$	60	37,15744	0,61929	0,03010

Tabela A.15: Tabela ANOVA dos resultados IBL carga de trabalho SDSC2000

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	1,44284	19	0,07594	2,77229	1,92030
Dentro dos grupos	32,32277	1180	0,02739		
Totais	33,76561	1199			

Tabela A.16: Teste Estendido de Tukey carga de trabalho SDSC2000

valor crítico igual a 0, 11																			
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
B	0,01																		
C	0,02	0,01																	
D	0,02	0,01	0,00																
E	0,01	0,00	0,01	0,02															
F	0,02	0,01	0,00	0,00	0,01														
G	0,04	0,03	0,02	0,02	0,03	0,02													
H	0,06	0,05	0,04	0,04	0,06	0,04	0,02												
I	0,01	0,00	0,01	0,01	0,00	0,01	0,03	0,05											
J	0,03	0,02	0,01	0,00	0,02	0,01	0,01	0,04	0,02										
K	0,06	0,05	0,04	0,03	0,05	0,04	0,02	0,01	0,05	0,03									
L	0,09	0,08	0,08	0,07	0,09	0,08	0,06	0,03	0,09	0,07	0,04								
M	0,01	0,00	0,01	0,01	0,01	0,01	0,03	0,05	0,00	0,01	0,04	0,08							
N	0,03	0,02	0,01	0,01	0,02	0,01	0,01	0,03	0,02	0,00	0,02	0,06	0,02						
O	0,06	0,06	0,05	0,04	0,06	0,05	0,03	0,00	0,06	0,04	0,01	0,03	0,05	0,03					
P	**0,11**	0,10	0,09	0,09	**0,11**	0,09	0,07	0,05	0,10	0,09	0,06	0,02	0,10	0,08	0,05				
Q	0,01	0,00	0,01	0,01	0,01	0,01	0,03	0,05	0,00	0,01	0,04	0,08	0,00	0,02	0,05	0,10			
R	0,03	0,02	0,01	0,01	0,03	0,01	0,01	0,03	0,02	0,01	0,02	0,06	0,02	0,00	0,03	0,08	0,02		
S	0,07	0,06	0,05	0,05	0,06	0,05	0,03	0,01	0,06	0,04	0,01	0,03	0,06	0,04	0,00	0,04	0,06	0,04	
T	**0,12**	**0,11**	0,10	0,10	**0,11**	0,10	0,08	0,06	**0,11**	0,09	0,06	0,02	**0,11**	0,09	0,05	0,01	**0,11**	0,09	0,00

Tabela A.17: Resultados algoritmo IBL para a carga de trabalho CTC

Grupo	Parâmetros IBL	Tamanho	Soma	Média	Variância
A	$(t^2 = 0,125, k = 5)$	60	30,27061	0,50451	0,01670
B	$(t^2 = 0,125, k = 10)$	60	30,87838	0,51464	0,01680
C	$(t^2 = 0,125, k = 25)$	60	31,57921	0,52632	0,01663
D	$(t^2 = 0,125, k = 50)$	60	32,53252	0,54221	0,01816
E	$(t^2 = 0,250, k = 5)$	60	31,12534	0,51876	0,01686
F	$(t^2 = 0,250, k = 10)$	60	32,54506	0,54242	0,01790
G	$(t^2 = 0,250, k = 25)$	60	34,68032	0,57801	0,01828
H	$(t^2 = 0,250, k = 50)$	60	36,60622	0,61010	0,02040
I	$(t^2 = 0,500, k = 5)$	60	31,95984	0,53266	0,01713
J	$(t^2 = 0,500, k = 10)$	60	33,84624	0,56410	0,01833
K	$(t^2 = 0,500, k = 25)$	60	36,83222	0,61387	0,01947
L	$(t^2 = 0,500, k = 50)$	60	39,30439	0,65507	0,02180
M	$(t^2 = 1,000, k = 5)$	60	32,52100	0,54202	0,01726
N	$(t^2 = 1,000, k = 10)$	60	34,62083	0,57701	0,01848
O	$(t^2 = 1,000, k = 25)$	60	37,87365	0,63123	0,01995
P	$(t^2 = 1,000, k = 50)$	60	40,54574	0,67576	0,02255
Q	$(t^2 = 2,000, k = 5)$	60	32,81245	0,54687	0,01732
R	$(t^2 = 2,000, k = 10)$	60	34,98814	0,58314	0,01857
S	$(t^2 = 2,000, k = 25)$	60	38,34690	0,63912	0,02020
T	$(t^2 = 2,000, k = 50)$	60	41,07157	0,68453	0,02293

Tabela A.18: Tabela ANOVA dos resultados IBL carga de trabalho CTC

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	3,50332	19	0,18439	9,81493	1,59534
Dentro dos grupos	22,16773	1180	0,01879		
Totais	25,67105	1199			

Tabela A.19: Teste Estendido de Tukey carga de trabalho CTC

valor crítico igual a 0, 10																			
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
B	0,010																		
C	0,022	0,012																	
D	0,038	0,028	0,016																
E	0,014	0,004	0,008	0,023															
F	0,038	0,028	0,016	0,000	0,024														
G	0,073	0,063	0,052	0,036	0,059	0,036													
H	**0,106**	0,095	0,084	0,068	0,091	0,068	0,032												
I	0,028	0,018	0,006	**0,010**	0,014	**0,010**	0,045	0,077											
J	0,060	0,049	0,038	0,022	0,045	0,022	0,014	0,046	0,031										
K	**0,109**	0,099	0,088	0,072	0,095	0,071	0,036	0,004	0,081	0,050									
L	**0,151**	**0,140**	**0,129**	**0,113**	**0,136**	**0,113**	0,077	0,045	**0,122**	0,091	0,041								
M	0,038	0,027	0,016	0,000	0,023	0,000	0,036	0,068	0,009	0,022	0,072	**0,113**							
N	0,073	0,062	0,051	0,035	0,058	0,035	0,001	0,033	0,044	0,013	0,037	0,078	0,035						
O	**0,127**	**0,117**	**0,105**	0,089	**0,112**	0,089	0,053	0,021	0,099	0,067	0,017	0,024	0,089	0,054					
P	**0,171**	**0,161**	**0,149**	**0,134**	**0,157**	0,133	0,098	0,066	**0,143**	**0,112**	0,062	0,021	**0,134**	0,099	0,045				
Q	0,042	0,032	0,021	0,005	0,028	0,004	0,031	0,063	0,014	0,017	0,067	**0,108**	0,005	0,030	0,084	**0,129**			
R	0,079	0,068	0,057	0,041	0,064	0,041	0,005	0,027	0,050	0,019	0,031	0,072	0,041	0,006	0,048	0,093	0,036		
S	**0,135**	**0,124**	**0,113**	0,097	**0,120**	0,097	0,061	0,029	**0,106**	0,075	0,025	0,016	0,097	0,062	0,008	0,037	0,092	0,056	
T	**0,180**	**0,170**	**0,158**	**0,142**	**0,166**	**0,142**	**0,107**	0,074	**0,152**	**0,120**	0,071	0,029	**0,143**	**0,108**	0,053	0,009	**0,138**	**0,101**	0,045

Tabela A.20: Resultados algoritmo IBL para a carga de trabalho LANL

Grupo	Parâmetros IBL	Tamanho	Soma	Média	Variância
A	$(t^2 = 0,125, k = 5)$	60	22,82928	0,38049	0,03957
B	$(t^2 = 0,125, k = 10)$	60	22,30879	0,37181	0,03838
C	$(t^2 = 0,125, k = 25)$	60	22,13347	0,36889	0,03590
D	$(t^2 = 0,125, k = 50)$	60	22,24708	0,37078	0,03494
E	$(t^2 = 0,250, k = 5)$	60	22,90444	0,38174	0,03982
F	$(t^2 = 0,250, k = 10)$	60	22,41176	0,37353	0,03887
G	$(t^2 = 0,250, k = 25)$	60	22,26690	0,37112	0,03667
H	$(t^2 = 0,250, k = 50)$	60	22,54232	0,37571	0,03711
I	$(t^2 = 0,500, k = 5)$	60	22,92672	0,38211	0,03994
J	$(t^2 = 0,500, k = 10)$	60	22,46872	0,37448	0,03934
K	$(t^2 = 0,500, k = 25)$	60	22,60139	0,37669	0,03891
L	$(t^2 = 0,500, k = 50)$	60	23,13998	0,38567	0,04127
M	$(t^2 = 1,000, k = 5)$	60	22,95520	0,38259	0,04019
N	$(t^2 = 1,000, k = 10)$	60	22,54102	0,37568	0,03990
O	$(t^2 = 1,000, k = 25)$	60	22,87001	0,38117	0,04062
P	$(t^2 = 1,000, k = 50)$	60	23,61218	0,39354	0,04431
Q	$(t^2 = 2,000, k = 5)$	60	22,97335	0,38289	0,04034
R	$(t^2 = 2,000, k = 10)$	60	22,59712	0,37662	0,04026
S	$(t^2 = 2,000, k = 25)$	60	23,03413	0,38390	0,04165
T	$(t^2 = 2,000, k = 50)$	60	23,84575	0,39743	0,04586

Tabela A.21: Tabela ANOVA dos resultados IBL carga de trabalho LANL

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	0,06215	19	0,00327	0,08242	1,92030
Dentro dos grupos	46,83548	1180	0,03969		
Totais	46,89763	1199			

Tabela A.22: Tempos de execução para os benchmarks do pacote NAS utilizando o núcleo do sistema operacional com algoritmo de aprendizado embutido (T_o = 250 milisegundos, EP_6)

	BT	CG	EP	IS	LU	MG	SP
	414,59	95,79	67,37	9,38	874,83	44,62	1130,20
	414,90	95,14	67,39	9,38	877,00	44,67	1136,26
	414,34	95,25	67,48	9,38	874,91	44,63	1130,43
	414,13	95,12	67,39	9,38	874,86	44,72	1131,31
	413,90	95,26	67,35	9,38	874,60	44,68	1130,25
	415,34	95,28	67,48	9,38	875,40	44,71	1130,71
	414,78	95,25	67,49	9,38	875,65	44,63	1130,74
	414,05	95,29	67,49	9,38	874,59	44,73	1131,72
	415,11	95,28	67,49	9,38	874,59	44,62	1133,61
	414,53	95,29	67,49	9,38	877,15	44,70	1131,15
	414,16	95,04	67,49	9,38	874,57	44,67	1130,69
	414,28	95,16	67,40	9,38	875,81	44,62	1132,95
	415,01	95,09	67,40	9,39	876,77	44,63	1131,68
	415,06	95,28	67,48	9,38	876,77	44,71	1130,69
	414,88	95,35	67,50	9,16	875,11	44,67	1132,94
Média	414,60	95,26	67,45	9,37	875,51	44,67	1131,69
Variância	0,197854	0,029531	0,002969	0,003254	0,929707	0,001607	2,707327
Desvio padrão	0,444808	0,171847	0,054485	0,057046	0,964213	0,040083	1,645396

Tabela A.23: Tempos de execução para os benchmarks do pacote NAS utilizando o núcleo do sistema operacional com algoritmo de aprendizado embutido (T_o = 500 milisegundos, EP_6)

	BT	CG	EP	IS	LU	MG	SP
	407,16	93,75	67,38	9,37	861,14	44,39	1090,28
	408,36	93,82	67,22	9,37	860,31	44,45	1090,62
	407,42	93,81	67,23	9,37	858,25	44,34	1090,88
	407,33	93,86	67,24	9,37	860,32	44,35	1091,01
	407,43	93,82	67,25	9,37	860,29	44,34	1091,27
	406,40	93,62	67,26	9,37	860,30	44,31	1144,11
	407,42	93,79	67,24	9,37	858,66	44,34	1059,43
	407,27	93,63	67,19	9,37	858,38	44,34	1063,17
	406,54	93,81	67,25	9,37	860,30	44,89	1123,56
	407,40	93,88	67,24	9,37	858,38	44,45	1054,14
	406,43	93,65	67,22	9,37	860,33	44,69	1046,09
	406,48	93,66	67,28	9,37	860,32	44,35	1047,48
	406,51	93,64	67 26	9,37	858,57	44,35	1042,84
	406,48	93,66	67,27	9,37	860,31	44,32	1045,85
	400,92	93,30	67,22	9,37	817,10	44,41	1046,16
Média	406,64	93,71	67,25	9,37	856,86	44,42	1075,13
Variância	2,815595	0,021395	0,001814	0,000000	121,936711	0,025641	965,077383
Desvio padrão	1,677974	0,146271	0,042594	0,000000	11,042496	0,160128	31,065695

Tabela A.24: Tempos de execução para os benchmarks do pacote NAS utilizando o núcleo do sistema operacional com algoritmo de aprendizado embutido (T_o = 1 segundo, EP_6)

	BT	CG	EP	IS	LU	MG	SP
	390,40	93,67	66,74	9,31	816,94	44,38	1089,00
	391,96	93,81	66,64	9,31	861,19	44,45	1092,36
	393,56	93,79	66,71	7,91	846,07	44,34	1070,89
	394,63	93,81	66,84	8,27	871,83	44,35	1058,18
	390,71	93,82	66,72	8,85	836,94	44,32	1076,80
	391,78	93,62	66,64	8,99	864,05	44,30	1055,69
	393,74	93,69	66,73	8,75	830,65	44,34	1062,69
	393,74	93,49	66,66	8,91	859,85	44,34	1090,23
	390,56	93,50	66,73	9,20	848,05	44,23	1063,37
	394,25	93,92	66,73	8,60	864,28	44,45	1101,23
	393,00	93,65	66,64	8,77	846,20	44,55	1075,26
	392,43	93,63	66,68	9,19	834,17	44,35	1063,50
	391,79	93,62	66,65	8,83	860,09	44,35	1064,80
	391,59	93,66	66,64	8,11	841,61	44,32	1060,06
	395,00	93,28	66,31	9,04	857,86	44,31	1062,67
Média	392,61	93,66	66,67	8,80	849,32	44,36	1072,45
Variância	2,260264	0,025640	0,013035	0,180992	232,096498	0,005741	206,714255
Desvio padrão	1,503417	0,160125	0,114172	0,425432	15,234714	0,075769	14,377561

Tabela A.25: Tempos de execução para os benchmarks do pacote NAS utilizando o núcleo do sistema operacional com algoritmo de aprendizado embutido ($T_o = 250$ millisegundos, EP_7)

	BT	CG	EP	IS	LU	MG	SP
	37,79	3,38	8.90	0.29	74,51	2,75	86,51
	37,31	3,33	8.79	0.28	74.45	2.74	86,33
	37,32	3,34	8.79	0.28	74.43	2.73	86.24
	37,36	3,33	8.79	0.25	74.45	2.74	86.35
	37,40	3,35	8.79	0.28	74.44	2.76	86.36
	37,38	3,33	8.79	0,28	74.48	2.76	86.31
	37,36	3,32	8.79	0.28	74.49	2.71	86.30
	37,34	3,33	8.79	0.28	74.49	2.73	86.28
	37,39	3,33	8.79	0.28	74.48	2.76	86.19
	37,41	3,33	8.78	0.27	74.47	2.75	86.23
	37,36	3,33	8.79	0.26	74.50	2.76	86.24
	37,38	3,33	8.79	0,27	74.51	2.77	86,23
	37,38	3,34	8.79	0.27	74.50	2.77	86.21
	37,37	3,33	8.79	0.27	74.46	2.76	86.22
	37,36	3,34	8.79	0.27	74.46	2.77	86.21
	37,37	3,34	8.78	0,27	74.46	2,76	86.14
	37,37	3,33	8.79	0,27	74.49	2.75	86,10
	37,35	3,33	8,79	0,27	74.43	2,76	86.05
	37,38	3,34	8.79	0,26	74.46	2,76	86,05
	37,36	4,20	8.79	0.36	74.46	2,74	86,40
	37,42	3,37	8.79	0.25	77.61	2,73	86,28
	37,46	3,37	8.78	0.25	77.55	2.74	86,39
	37,50	3,39	8.79	0,25	77.65	2,78	86,35
	37,51	3,39	8.79	0,25	77,53	2,79	86,36
	37,52	3,37	8.79	0.27	77,81	2,75	86,36
	37,53	3,38	8.79	0,25	77,54	2,74	86,36
	37,51	3,38	8,79	0,26	77,33	2,75	86,34
	37,51	3,39	8,79	0,27	77,57	2,76	86,48
	37,50	3,39	8.80	0,26	77,55	2,74	86,34
	37,51	3,39	8,79	0,26	77,48	2,75	86,31
Média	37,42	3,38	8,79	0.27	75.50	2,75	86,28
Variância	0,009383	0,024593	0,000422	0,000417	2,201412	0,000272	0,012197
Desvio padrão	0,096864	0,156822	0,020536	0,020424	1,483716	0,016484	0,110441

Tabela A.26: Tempos de execução para os benchmarks do pacote NAS utilizando o núcleo do sistema operacional com algoritmo de aprendizado embutido ($T_o = 500$ millisegundos, EP_7)

	BT	CG	EP	IS	LU	MG	SP
	38,69	3,37	8,89	0,27	74,22	2,74	86,07
	37,29	3,34	8,79	0,28	74,22	2,75	86,02
	37,31	3,33	8,78	0,28	74,23	2,74	85,86
	37,32	3,34	8,79	0,26	74,23	2,75	85,99
	37,30	3,34	8,79	0,28	74,22	2,76	86,02
	37,30	3,32	8,79	0,28	74,23	2,75	86,00
	37,33	3,33	8,79	0,29	74,23	2,70	85,97
	37,30	3,33	8,78	0,27	74,21	2,77	85,96
	37,32	3,33	8,79	0,27	74,22	2,76	85,91
	37,31	3,34	8,78	0,29	74,24	2,77	85,93
	37,34	3,33	8,79	0,27	74,22	2,77	86,00
	37,33	3,32	8,79	0,27	74,21	2,75	85,91
	37,31	3,34	8,79	0,27	74,22	2,76	85,87
	37,32	3,33	8,78	0,27	74,26	2,77	85,82
	37,30	3,31	8,79	0,27	74,21	2,71	85,83
	37,30	3,34	8,78	0,27	74,19	2,76	85,72
	37,32	3,34	8,79	0,26	74,24	2,73	85,64
	37,30	3,34	8,78	0,28	74,24	2,71	85,61
	37,30	3,33	8,79	0,26	74,20	2,76	85,66
	37,30	3,33	8,78	0,26	74,21	2,74	85,57
	37,31	3,34	8,79	0,26	74,23	2,74	85,50
	37,30	3,32	8,79	0,26	74,23	2,74	85,62
	37,30	3,33	8,79	0,26	74,22	2,75	85,54
	37,27	3,33	8,79	0,26	74,21	2,73	85,56
	37,28	3,33	8,79	0,26	74,18	2,74	85,59
	37,30	3,34	8,79	0,26	74,23	2,75	85,62
	37,31	3,33	8,78	0,27	74,25	2,73	85,63
	37,27	3,33	8,78	0,26	74,30	2,74	85,65
	37,27	3,33	8,79	0,26	74,19	2,73	85,65
	37,26	3,34	8,79	0,26	74,24	2,74	85,60
Média	37,35	3,33	8,79	0,27	74,22	2,74	85,78
Variância	0,064529	0,000106	0,000376	0,000088	0,000515	0,000322	0,032910
Desvio padrão	0,254026	0,010283	0,019384	0,009371	0,022695	0,017953	0,181411

Tabela A.27: Tempos de execução para os benchmarks do pacote NAS utilizando o núcleo do sistema operacional com algoritmo de aprendizado embutido ($T_o = 1$ segundo, EP_7)

	BT	CG	EP	IS	LU	MG	SP
	37,40	3,33	8.79	0,27	74,47	2,77	86,09
	37,34	3,34	8,78	0,26	74,48	2,76	86,03
	37,34	3,32	8,78	0,26	74,43	2,75	86,06
	37,31	3,33	8,78	0,26	74,50	2,74	85,99
	37,34	3,34	8,78	0,27	74,52	2,75	85,84
	37,31	3,34	8,78	0,26	74,48	2,75	85,77
	37,30	3,33	8,78	0,26	74,45	2,75	85,77
	37,34	3,34	8,78	0,26	74,46	2,75	85,78
	37,35	3,33	8,78	0,26	74,44	2,73	85,69
	37,31	3,31	8,78	0,26	74,47	2,73	85,80
	37,33	3,32	8,79	0,27	74,47	2,73	85,74
	37,33	3,34	8,78	0,26	74,45	2,74	85,84
	37,30	3,33	8,79	0,26	74,52	2,74	85,87
	37,32	3,33	8,78	0,25	74,44	2,74	85,87
	37,34	3,33	8,79	0,25	74,48	2,73	85,82
	37,33	3,32	8,78	0,25	74,48	2,72	85,92
	37,28	3,34	8,78	0,27	74,52	2,72	85,98
	37,33	3,34	8,78	0,25	74,50	2,73	86,02
	37,28	3,33	8,78	0,25	74,50	2,74	85,93
	37,28	3,33	8,78	0,25	74,49	2,74	85,88
	37,30	3,32	8,78	0,25	74,55	2,73	86,00
	37,31	3,32	8,78	0,25	74,54	2,72	85,97
	37,27	3,33	8,78	0,27	74,61	2,72	85,93
	37,27	3,34	8,78	0,25	74,57	2,72	86,05
	37,31	3,33	8,79	0,25	74,70	2,73	85,95
	37,30	3,33	8,78	0,25	74,71	2,72	86,01
	37,29	3,34	8,79	0,25	74,63	2,72	86,00
	37,30	3,34	8,78	0,27	74,72	2,72	85,95
	37,28	3,33	8,78	0,27	74,64	2,71	85,89
	37,28	3,33	8,78	0,25	74,80	2,72	85,84
Média	37,31	3,33	8,78	0,26	74,53	2,73	85,91
Variância	0,000839	0,000064	0,000017	0,000065	0,009301	0,000204	0,011000
Desvio padrão	0,028969	0,008030	0,004068	0,008052	0,096440	0,014288	0,104879

Tabela A.28: Valores de *slowdown* para os benchmarks NAS considerando o núcleo não instrumentado comparado ao núcleo com o algoritmo de classificação (T_o=250ms, EP_6)

Valor crítico de T igual a 2.048 com 28 graus de liberdade							
Benchmark	**BT**	**CG**	**EP**	**IS**	**LU**	**MG**	**SP**
Núcleo não instrumentado	392.01	91.80	65.41	8.53	816.92	43.37	1070.72
Núcleo com algoritmo de classificação	414.60	95.26	67.45	9.37	875.51	44.67	1131.69
Estimativa da Variância	12.76885	0.03010	0.06032	0.46113	109.01523	0.16839	522.84154
Estimativa do erro	1.30480	0.06335	0.08968	0.24796	3.81253	0.14984	8.34938
Valor de T	-17.31754	-54.65811	-22.68014	-3.36613	-15.36601	-8.64459	-7.30258
Teste de Hipóteses	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0
Slowdown	1.05764	1.03772	1.03110	1.09784	1.07171	1.02987	1.05695
Média do slowdown				**1.05**			

Tabela A.29: Valores de *slowdown* para os benchmarks NAS considerando o núcleo não instrumentado comparado ao núcleo com o algoritmo de classificação (T_o=500ms, EP_6)

Valor crítico de T igual a 2.048 com 28 graus de liberdade							
Benchmark	**bt**	**cg**	**ep**	**is**	**lu**	**mg**	**sp**
Núcleo não instrumentado	392.01	91.80	65.41	8.53	816.92	43.37	1070.72
Núcleo com o algoritmo de classificação	406.64	93.71	67.25	9.37	856.86	44.42	1075.13
Estimativa da Variância	14.07773	0.02603	0.05974	0.45951	169.51874	0.18041	1004.02657
Estimativa do erro	1.37005	0.05892	0.08925	0.24752	4.75421	0.15510	11.57023
Valor de T	-10.67749	-32.55534	-20.59340	-3.38824	-8.40098	-6.76563	-0.38109
Teste de Hipóteses	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Aceita H0
Slowdown	1.03732	1.02089	1.02810	1.09830	1.04889	1.02419	1.00412
Média dos valores de *slowdown*				**1.04**			

Tabela A.30: Valores de *slowdown* para os benchmarks NAS considerando o núcleo não instrumentado comparado ao núcleo com o algoritmo de classificação (T_o=1 segundo, EP_6)

Valor crítico de T igual a 2.048 com 28 graus de liberdade							
Benchmark	**bt**	**cg**	**ep**	**is**	**lu**	**mg**	**sp**
Núcleo não instrumentado	392.01	91.80	65.41	8.53	816.92	43.37	1070.72
Núcleo com o algoritmo de classificação	392.61	93.66	66.67	8.80	849.32	44.36	1072.45
Estimativa da Variância	13.80006	0.02815	0.06535	0.55000	224.59863	0.17046	624.84501
Estimativa do erro	1.35647	0.06127	0.09335	0.27080	5.47234	0.15076	9.12758
Valor de T	-0.44331	-30.49907	-13.48350	-1.00196	-5.91971	-6.54461	-0.18975
Teste de Hipóteses	Aceita H0	Rejeita H0	Rejeita H0	Aceita H0	Rejeita H0	Rejeita H0	Aceita H0
Slowdown	1.00153	1.02036	1.01924	1.03180	1.03965	1.02275	1.00162
Média dos valores de *slowdown*				**1.02**			

Tabela A.31: Valores de *slowdown* para os benchmarks NAS considerando o núcleo não instrumentado comparado ao núcleo com o algoritmo de classificação (T_o=250ms , EP_7)

Valor crítico de Z igual a 1.96							
Benchmark	**bt**	**cg**	**ep**	**is**	**lu**	**mg**	**sp**
Núcleo não instrumentado	37.19	3.32	8.75	0.25	74.15	2.74	85.57
Núcleo com o algoritmo de classificação ART-2A	37.42	3.38	8.79	0.27	75.50	2.75	86.28
Estimativa do erro	0.01848	0.02865	0.00391	0.00376	0.27099	0.00376	0.02225
Valor de Z	-12.89909	-2.10105	-12.15304	2.92381	-4.98151	-4.39324	-32.05271
Teste de Hipóteses	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0
Slowdown	1.00641	1.01813	1.00543	1.07795	1.01821	1.00604	1.00833
Média dos valores de *slowdown*				**1.02007**			

Tabela A.32: Valores de *slowdown* para os benchmarks NAS considerando o núcleo não instrumentado comparado ao núcleo com o algoritmo de classificação (T_o=500ms , EP_7)

Benchmark	**bt**	**cg**	**ep**	**is**	**lu**	**mg**	**sp**
Núcleo não instrumentado	37.19	3.32	8.75	0.25	74.15	2.74	85.57
Núcleo com o algoritmo de classificação	37.35	3.33	8.79	0.27	74.22	2.74	85.78
Estimativa do erro	0.04669	0.00215	0.00371	0.00178	0.00864	0.00398	0.03443
Valor de Z	-3.49928	-6.30530	-12.09329	-10.06813	-8.44224	-2.30798	-5.99880
Teste de Hipóteses	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0
Slowdown	1.00439	1.00408	1.00513	1.07131	1.00098	1.00335	1.00241
Média do valores de *slowdown*				**1.01309**			

Tabela A.33: Valores de *slowdown* para os benchmarks NAS considerando o núcleo não instrumentado comparado ao núcleo com o algoritmo de classificação (T_o=1 segundo , EP_7)

Benchmark	**bt**	**cg**	**ep**	**is**	**lu**	**mg**	**sp**
Núcleo não instrumentado	37.19	3.32	8.75	0.25	74.15	2.74	85.57
Núcleo com o algoritmo de classificação ART-2A	37.31	3.33	8.78	0.26	74.53	2.74	85.91
Estimativa do erro	0.00753	0.00180	0.00133	0.00155	0.01917	0.00394	0.02134
Valor de Z	-16.86803	-6.23098	-27.38972	-4.66903	-19.95819	-0.21401	-15.86796
Teste de Hipóteses	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Rejeita H0	Aceita H0	Rejeita H0
Slowdown	1.00342	1.00337	1.00417	1.02877	1.00516	1.00031	1.00396
Média dos valores de *slowdown*				**1.00702**			

Tabela A.34: Tempos de resposta em microssegundos de 50 execuções do algoritmo IBL para diferentes tamanhos de base de experiências, entre 5 e 4096, considerando o EP_7

	5	10	20	40	80	160	320	640	1024	2048	4096
	1509	2029	4027	5723	10525	19370	45583	79015	129403	336488	538789
	988	2260	2816	3856	12363	13581	30605	63009	101211	329205	697417
	1235	1933	2564	4257	11816	18753	31003	61341	103543	326908	425878
	919	1874	2580	3966	12551	17545	44995	89703	100224	209764	675855
	918	1887	2607	6905	7558	18191	44668	89924	98422	332476	423584
	1222	1424	2240	3792	10749	21143	45257	60334	99046	334548	426223
	1217	1927	3318	6062	10621	19088	43832	62106	99126	209266	423996
	1200	1417	3997	6092	11150	12905	45079	88397	99447	211463	660500
	1216	1558	2239	5806	11299	12825	46198	90584	99132	223343	732230
	919	1420	2240	3812	11090	18381	44524	60273	98912	224947	699602
	1241	1425	2359	5818	7623	12830	45423	61313	98856	329653	688202
	2135	2152	2231	3865	11568	20673	43916	60817	99364	212622	692487
	934	1452	2242	3783	7527	13542	43488	60908	152688	321088	684497
	909	2047	2227	5381	13378	12960	43642	59971	162508	323746	686178
	921	2011	2259	5363	12005	17882	46102	61587	148942	316408	660496
	914	1432	2229	5187	7884	18028	46814	97193	144343	325494	679357
	918	1464	2231	5487	7509	13116	44073	96414	154914	209090	428247
	1258	1424	2224	5950	7637	13134	45046	92771	98455	330021	705240
	1238	1417	2268	6267	10515	14104	44083	61032	100933	325531	703409
	920	1428	3405	5824	10563	17615	45048	91984	159568	209373	692352
	929	1418	3390	5211	7752	12618	47285	92867	156928	208976	716668
	909	1416	3624	5017	10646	17507	30889	93631	154452	209178	699308
	910	1474	2241	5110	10660	12800	49380	96593	152383	369691	712202
	924	1416	3391	5404	7526	12574	44182	92903	152827	327033	698477
	912	1416	2517	6400	7622	12929	30208	60990	151167	325803	424442
	913	1435	2237	5241	8862	12882	31224	91689	150646	220542	429580
	1423	2082	3109	5360	7615	13554	44805	90853	98847	217868	688226
	920	2111	3087	4201	7710	12700	44112	88281	158011	329019	425253
	913	2068	3039	5173	10530	12788	32004	95906	150715	336693	423394
	920	1431	3215	5390	7780	12649	30868	99492	106116	210838	424236
	1320	2073	3388	5367	7784	18774	45930	61173	99060	322008	697257
	1282	1439	3388	5268	10622	18711	46770	87650	153013	323068	682336
	1281	1445	3285	5299	7769	12776	42323	91441	154771	208681	423360
	1156	1979	3761	3786	11301	13833	42813	94384	150237	330716	426207
	1338	1974	2965	5289	11871	12715	30764	87251	149920	334239	691417
	926	1886	3049	5270	11957	12757	31169	90214	163314	323487	429112
	910	1937	3227	5650	8339	13024	47180	89011	151023	319519	686761
	1232	2060	3021	5640	7646	17772	45878	90334	154219	331837	687100
	1215	2102	2994	5580	11320	12971	30908	91565	99393	352066	426048
	1211	2008	3113	3783	10587	18347	30892	60951	159132	344112	423835
	1510	1892	2409	6060	10956	12834	44001	93358	159314	357590	425964
	1296	1928	3006	4047	11915	13470	45181	91376	162001	345691	684591
	1240	1955	3151	6147	11380	13917	31198	86964	99324	342736	676812
	1196	1980	3144	6881	7540	17833	30475	89191	102861	322086	702393
	1201	1836	3072	3924	7577	17792	30430	88495	102941	208732	425560
	1188	1864	3048	3801	10540	12662	31206	88691	155379	332563	426175
	1238	1963	2240	3898	7517	12591	44141	60100	161169	325885	425721
	1558	1573	3040	5416	11238	12592	30788	94075	156863	329507	423542
Média	1140,22	1759,82	2873,86	5134,92	9787,8	14998,96	40008,98	82373,9	131358,04	296236,48	571732,32
Desvio padrão	238,09	282,87	515,98	880,48	1861,75	2757,76	6906,29	14454,64	27149,48	55950,10	132951,69

Tabela A.35: Resultados para a carga de trabalho SDSC95, parâmetro t variável

Grupo	Vizinhança	Amostras	Soma	Média	Variância
A	$k = 5$	70	40.17277	0.57389	0.01168
B	$k = 10$	70	41.10538	0.58721	0.01276
C	$k = 25$	70	43.44031	0.62057	0.01143
D	$k = 50$	70	45.48097	0.64972	0.01210

Tabela A.36: Tabela ANOVA para a carga de trabalho SDSC95, parâmetro t variável

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	0,24459	3	0,08153	6.79512	2.63731
Dentro dos grupos	3,31156	276	0,01200		
Totais	3,55615	279			

Tabela A.37: Teste de Tukey para a carga de trabalho SDSC95, parâmetro t variável

Grupos	Diferença entre médias	Teste de hipóteses
A,B	0,01332	Aceita H0
A,C	0,04668	Aceita H0
A,D	0,07583	Rejeita H0
B,C	0,03336	Aceita H0
B,D	0,06251	Aceita H0
C,D	0,02915	Aceita H0

Tabela A.38: Resultados do algoritmo IBL para a carga de trabalho SDSC96, parâmetro t variável

Grupo	Vizinhança	Amostras	Soma	Média	Variância
A	$k = 5$	60	32,76557	0,54609	0,01333
B	$k = 10$	60	32,51274	0,54188	0,01445
C	$k = 25$	60	33,64597	0,56077	0,01515
D	$k = 50$	60	35,25265	0.58754	0.01443

Tabela A.39: Tabela ANOVA para a carga de trabalho SDSC96, parâmetro t variável

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	0,07666	3	0,02555	1,78199	2,64285
Dentro dos grupos	3,38401	236	0,01434		
Totais	3,46066	239			

Tabela A.40: Resultados do algoritmo IBL para a carga de trabalho SDSC2000, parâmetro t variável

Grupo	Vizinhança	Amostras	Soma	Média	Variância
A	$k = 5$	60	40,17280	0.66955	0.03271
B	$k = 10$	60	41,25580	0.68760	0,03254
C	$k = 25$	60	43,27967	0.72133	0.02655
D	$k = 50$	60	43,61299	0.72688	0.02920

Tabela A.41: Tabela ANOVA para a carga de trabalho SDSC2000, parâmetro t variável

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	0.13510	3	0.04503	1.48878	2.64285
Dentro dos grupos	7.13861	236	0.03025		
Totais	7.27371	239			

Tabela A.42: Resultados do algoritmo IBL para a carga de trabalho CTC, parâmetro t variável

Grupo	Vizinhança	Amostras	Soma	Média	Variância
A	$k = 5$	60	32.78171	0.54636	0,01730
B	$k = 10$	60	34.96838	0.58281	0.01855
C	$k = 25$	60	38,35085	0.63918	0.02018
D	$k = 50$	60	41.08342	0.68472	0.02293

Tabela A.43: Tabela ANOVA para a carga de trabalho CTC, parâmetro t variável

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	0.67090	3	0.22363	11.33010	2.64285
Dentro dos grupos	4.65819	236	0.01974		
Totais	5.32909	239			

Tabela A.44: Teste de Tukey para a carga de trabalho CTC, parâmetro t variável

Grupos	Diferença entre médias	Teste de hipóteses
A,B	0,03644	Aceita H0
A,C	0,09282	Aceita H0
A,D	0,13836	Rejeita H0
B,C	0,05637	Aceita H0
B,D	0,10192	Aceita H0
C,D	0,04554	Aceita H0

Tabela A.45: Resultados do algoritmo IBL para a carga de trabalho LANL, parâmetro t variável

Grupo	Vizinhança	Amostras	Soma	Média	Variância
A	$k = 5$	60	22,97710	0,38295	0,04032
B	$k = 10$	60	22.61380	0.37690	0,04019
C	$k = 25$	60	23.08317	0.38472	0.04175
D	$k = 50$	60	23,90414	0.39840	0,04611

Tabela A.46: Tabela ANOVA para a carga de trabalho LANL, parâmetro t variável

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	0,01484	3	0,00495	0,11752	2.64285
Dentro dos grupos	9,93439	236	0,04209		
Totais	9,94923	239			

Tabela A.47: Teste Z entre os melhores resultados do algoritmo IBL com valor de t fixo e variável (valor crítico de Z igual a 1,96)

Carga de trabalho	SDSC95		SDSC96		SDSC2000		CTC		LANL	
t	fixo	variável	fixo	variável	fixo	variável	fixo	variável	fixo	variável
Média	0,57456	0,57390	0,54370	0,54609	0,50057	0,66955	0,50451	0,54636	0,38049	0,38295
Variância	0,01132	0,01169	0,01364	0,01333	0,02558	0,03271	0,01670	0,01730	0,03957	0,04032
Erro	0,01813		0,02120		0,03117		0,02380		0,03649	
Valor de Z	0,03669		-0,11288		-5,42163		-1,75817		-0,06752	
Hipóteses	Aceita H0		Aceita HO		Rejeita H0		Aceita H0		Aceita H0	

Tabela A.48: Resultados algoritmo IBL para a carga de trabalho CTC original

Grupo	Parâmetros IBL	Tamanho	Soma	Média	Variância
A	$(t^2 = 0,125, k = 5)$	60	29.17000	0.48617	0,01185
B	$(t^2 = 0,125, k = 10)$	60	29,73422	0.49557	0.01155
C	$(t^2 = 0.125, k = 25)$	60	30,65495	0.51092	0,01211
D	$(t^2 = 0.125, k = 50)$	60	31,55257	0.52588	0,01126
E	$(t^2 = 0,250, k = 5)$	60	30,13336	0,50222	0,01255
F	$(t^2 = 0,250, k = 10)$	60	31.45166	0.52419	0,01232
G	$(t^2 = 0,250, k = 25)$	60	33,56833	0.55947	0,01363
H	$(t^2 = 0,250, k = 50)$	60	35,69059	0.59484	0,01277
I	$(t^2 = 0,500, k = 5)$	60	31,28335	0.52139	0.01289
J	$(t^2 = 0,500, k = 10)$	60	33,43827	0.55730	0,01284
K	$(t^2 = 0,500, k = 25)$	60	36,69788	0.61163	0.01576
L	$(t^2 = 0,500, k = 50)$	60	39,25049	0.65417	0,01447
M	$(t^2 = 1,000, k = 5)$	60	31,57656	0.52628	0,01300
N	$(t^2 = 1,000, k = 10)$	60	33,77997	0.56300	0,01279
O	$(t^2 = 1,000, k = 25)$	60	36,90245	0.61504	0,01433
P	$(t^2 = 1,000, k = 50)$	60	39,72404	0.66207	0,01468
Q	$(t^2 = 2,000, k = 5)$	60	31,65282	0,52755	0,01287
R	$(t^2 = 2,000, k = 10)$	60	33,94509	0.56575	0,01277
S	$(t^2 = 2,000, k = 25)$	60	37,11590	0,61860	0,01440
T	$(t^2 = 2,000, k = 50)$	60	39,93745	0,66562	0,01478

Tabela A.49: Tabela ANOVA dos resultados IBL carga de trabalho CTC original

Fonte de Variação	Soma dos quadrados	G.l.	Variância Estimada	Razão F	Valor crítico de F
Entre os grupos	3,71576	19	0,19557	14,83687	1,59534
Dentro dos grupos	15,55370	1180	0,01318		
Totais	19,26946	1199			

Tabela A.50: Teste Estendido de Tukey carga de trabalho CTC original

valor crítico igual a 0, 08																			
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
B	0,01																		
C	0,02	0,02																	
D	0,04	0,03	0,01																
E	0,02	0,01	0,01	0.02															
F	0,04	0,03	0,01	0,00	0,02														
G	0,07	0,06	0,05	0,03	0,06	0,04													
H	**0,11**	**0,10**	**0,08**	0.07	**0,09**	0,07	0,04												
I	0,04	0,03	0,01	0,00	0,02	0,00	0,04	0,07											
J	0,07	0,06	0,05	0,03	0,06	0,03	0,00	0,04	0,04										
K	**0,13**	**0,12**	**0,10**	**0,09**	**0,11**	**0,09**	0,05	0,02	**0,09**	0,05									
L	**0,17**	**0,16**	**0,14**	**0,13**	**0,15**	**0,13**	**0,09**	0,06	**0,13**	**0,10**	0,04								
M	0,04	0,03	0,02	0,00	0,02	0,00	0,03	0,07	0,00	0,03	**0,09**	**0,13**							
N	**0,08**	0,07	0,05	0,04	0,06	0,04	0,00	0,03	0,04	0,01	0,05	**0,09**	0,04						
O	**0,13**	**0,12**	**0,10**	**0,09**	**0,11**	**0,09**	0,06	0,02	**0,09**	0,06	0,00	0,04	**0,09**	0,05					
P	**0,18**	**0,17**	**0,15**	**0,14**	**0,16**	**0,14**	**0,10**	0,07	**0,14**	**0,10**	0,05	0,01	**0,14**	**0,10**	0,05				
Q	0,04	0,03	0,02	0,00	0,03	0,00	0,03	0,07	0,01	0,03	**0,08**	**0,13**	0,00	0,04	**0,09**	0,13			
R	**0,08**	0,07	0,05	0,04	0,06	0,04	0,01	0,03	0,04	0,01	0,05	**0,09**	0,04	0,00	0,05	**0,10**	0,04		
S	**0,13**	**0,12**	**0,11**	**0,09**	**0,12**	**0,09**	0,06	0,02	**0,10**	0,06	0,01	0,04	**0,09**	0,06	0,00	0,04	**0,09**	0,05	
T	**0,18**	**0,17**	**0,15**	**0,14**	**0,16**	**0,14**	**0,11**	0,07	**0,14**	**0,11**	0,05	0,01	**0,14**	**0,10**	0,05	0,00	**0,14**	**0,10**	0,05

Tabela A.51: Resultados das simulações para a carga de trabalho SDSC95

Carga de trabalho	Resultados		Valores de Z (valor crítico igual a 1, 96)		
	Média do *slowdown*	Variância	FCFS	BACKFILL-IBL	BACKFILL-REAL
FCFS	1486.87086	28107429,10821			
BACKFILL-IBL	329, 58778	2530329,02843	56,79		
BACKFILL-REAL	88, 66486	422096,17843	71,10	38,08	

Tabela A.52: Resultados das simulações para a carga de trabalho SDSC96

Carga de trabalho	Resultados			Valores de Z (valor crítico igual a 1, 96)		
	Tamanho	Média do *slowdown*	Variância	FCFS	BACKFILL-IBL	BACKFILL-REAL
FCFS	683,02303	5802028,05517				
BACKFILL-IBL	263,14074	1450149,62903	30,18			
BACKFILL-REAL	32,00007	52913,81736	52,08	36,49		

Tabela A.53: Resultados das simulações para a carga de trabalho SDSC2000

Carga de trabalho	Resultados		Valores de Z (valor crítico igual a 1, 96)			
	Média do *slowdown*	Variância	FCFS	BACKFILL-REAL	BACKFILL-IBL	BACKFILL-USER
FCFS	360,40112	1363276,05506				
BACKFILL-REAL	17,08574	7808,70334	63,51			
BACKFILL-IBL	25,55488	39875,92087	61,23	8,40		
BACKFILL-USER	33,12894	26054,67244	60,14	18,88	6,38	

Tabela A.54: Resultados das simulações para a carga de trabalho CTC

Carga de trabalho	Resultados		Valores de Z (valor crítico igual a 1,96)			
	Média do *slowdown*	Variância	FCFS	BACKFILL-REAL	BACKFILL-IBL	BACKFILL-USER
FCFS	6,97080	1374,1089				
BACKFILL-REAL	1,18091	7,44398	34,52			
BACKFILL-IBL	4,11192	674,70661	14,00	24,87		
BACKFILL-USER	4,46870	797,99260	11,90	25,67	2,06	

Tabela A.55: Resultados das simulações para a carga de trabalho LANL

Carga de trabalho	Resultados		Valores de Z (valor crítico igual a 1,96)			
	Média do *slowdown*	Variância	FCFS	BACKFILL-REAL	BACKFILL-IBL	BACKFILL-USER
FCFS	127,53374	290912,07703				
BACKFILL-REAL	36,73441	24439,61963	68,62			
BACKFILL-IBL	80,00116	120982,9609	31,43	48,15		
BACKFILL-USER	108,79511	202557,80978	11,32	64,19	21,48	