Universidade de São Paulo
Instituto de Física de São Carlos

Thatyana de Faria Piola

Tracing de Aplicações Paralelas com
Informações de Alto Nível de Abstração

São Carlos
2007

Thatyana de Faria Piola

Tracing de Aplicações Paralelas com
Informações de Alto Nível de Abstração

Tese apresentada ao Instituto de Física de São Carlos
da Universidade de São Paulo para obtenção do título
de Doutor em Ciências.

Área de Concentração: Física Aplicada Computacional
Orientador: Prof. Dr. Gonzalo Travieso

SÃO CARLOS
2007

Piola, Thatyana de Faria

Tracing de aplicações paralelas com informações de
alto nível de abstração/ Thatyana de Faria Piola; orientador
Prof. Dr. Gonzalo Travieso.-- São Carlos, 2007.

106 f.:fig.

Tese (Doutorado - Programa de Pós Graduação em Física.
Área de Concentração: Física Aplicada Computacional)--- Instituto de
Física de São Carlos da Universidade de São Paulo.

1. Programação Paralela; 2. Análise de Desempenho; 3. Trace.
I.Título.

Aos meus pais Oswaldo e Ana Isabel e ao Enzo
por todo amor dedicado a mim.

Agradecimentos

Ao Senhor Deus que a mim concedeu toda graça, inspiração para a realização deste trabalho e por me acompanhar durante todos esses anos, especialmente durante as longas viagens.

A meu orientador, Prof. Dr. Gonzalo, agradeço a oportunidade, o apoio, a confiança e a competente orientação concedidas a mim durante todos esses anos e por seu constante otimismo mesmo nas horas mais difíceis.

A meu marido e companheiro Enzo o qual me faltam palavras para agradecer todo amor, carinho e pelo apoio nas horas mais difíceis.

Aos meus pais Oswaldo e Ana Isabel, meus irmãos Thaísa e Thiago, à Luciana e à Livia pelo amor, carinho e compreensão por passar tanto tempo longe, e por sermos uma grande família.

Aos grandes amigos Maria Camila e Humberto pela grande amizade, pelas longas conversas e por sempre me receberem com tanto carinho.

Aos colegas de grupo Elô, Francisco, Leonardo e especialmente ao Paulino pela grande ajuda e amizade.

Agradeço em especial à Wladerez por toda atenção e competência durante esses anos. Ao Silvio, a todas as secretárias e funcionários do setor de Pós Graduação e da biblioteca do IFSC.

Agradeço ao CNPQ pelo apoio financeiro.

Sumário

Resumo	viii
<i>Abstract</i>	ix
1 Introdução	1
1.1 Motivação	1
1.2 Níveis Hierárquicos	3
1.3 Objetivo do Trabalho	4
1.4 Organização do Texto	5
2 Processo de Análise de Desempenho	7
2.1 Coleta dos Dados	7
2.1.1 Métodos de Instrumentação	10
2.2 Transformação dos Dados	12
2.3 Visualização dos Dados	16
2.4 Data Warehousing	18
2.4.1 OLAP	20
2.5 Conclusões do Capítulo	22
3 Ferramentas	25
3.1 Toolkits	25
3.1.1 TAU	25
3.1.2 AIMS	27
3.1.3 HPCToolkit	28
3.1.4 PerfSuite	29
3.2 Profiles	30
3.2.1 VProf	31

3.2.2	Dynaprof	31
3.2.3	prof e gprof	31
3.3	Trace	32
3.3.1	Mpitrace	32
3.3.2	SiGMA	32
3.4	Visualizadores e Outros Analisadores	33
3.4.1	MPE	33
3.4.2	ParaGraph	34
3.4.3	Pablo	35
3.4.4	Paradyn	36
3.4.5	Paraver	37
3.4.6	Upshot	39
3.4.7	Vampir	41
3.5	Conclusões do Capítulo	42
4	Ferramenta de Análise Hierárquica	43
4.1	Módulo de Coleta dos Dados	44
4.1.1	HieraCollector para MPI	46
4.1.2	HieraCollector para OOPS	51
4.2	Módulo de Transformação dos Dados	57
4.2.1	Experimentos do HieraTransform para MPI	65
4.2.2	Experimentos do HieraTransform para OOPS	70
4.3	Conclusões do Capítulo	73
5	Ferramenta de Análise HieraOLAP	77
5.1	Módulo de Acesso	78
5.2	Módulo de Transformação	79
5.3	Módulo Base Multidimensional	81
5.4	Distribuição usando OLAP	85
5.4.1	Experimento para Visualização	86
5.4.2	Experimentos para Visualização das Operações <i>Drill Down</i> e <i>Roll Up</i>	87
5.4.3	Experimentos para visualização da operação <i>Rotate</i>	90
5.5	Conclusões do Capítulo	90
6	Conclusões e Trabalhos Futuros	93
6.1	Principais Contribuições	93

6.2 Trabalhos Futuros	95
Referências Bibliográficas	97

Lista de Figuras

1.1	Níveis de Abstração	3
2.1	Processo de Análise de Desempenho	7
2.2	Exemplo do Modelo Estrela para Vendas	20
3.1	Arquitetura de TAU	26
3.2	Estrutura do AIMS	27
3.3	Componentes do HPCToolkit	28
3.4	Interface Pablo	35
3.5	Tabela de Visualização Paradyne	37
3.6	Estrutura da ferramenta Paraver	38
3.7	Visualização em <i>Upshot</i>	39
3.8	Exibição da ferramenta <i>Nupshot</i>	40
3.9	Exibição <i>Jumpshot</i>	40
3.10	Interface da ferramenta Vampir	41
4.1	Arquitetura <i>Hierarchical Analyses</i>	44
4.2	Exemplos de Topologias	51
4.3	Distribuições	52
4.4	Estêncil	55
4.5	Ligações do Grafo	57
4.6	Percurso por Operações	58
4.7	Percurso entre Operações Parceiras	58
4.8	Diagrama de classes UML <i>HieraTransform</i>	59
4.9	Nó criado na classe <i>htHieraNode</i>	60
4.10	Diagrama UML da classe <i>htMPINode</i>	61
4.11	Nó criado na classe <i>htMPINode</i>	61

4.12	Diagrama UML da classe htPointToPoint	61
4.13	Diagrama UML da classe htCollective	62
4.14	Diagrama UML da classe htComm	62
4.15	Diagrama de Classes do htOOPSNode	63
4.16	Nó criado na classe htOOPSNode	63
5.1	Tela principal da Ferramenta HieraOLAP	78
5.2	Importação do Arquivo XML	79
5.3	Modelo Estrela para MPI	82
5.4	Modelo Estrela para OOPS	83
5.5	Projeto de Visualização de uma face do Modelo Multidimensional	85
5.6	Dados da Aplicação do cavalo	86
5.7	Operação <i>Drill Down</i>	88
5.8	Operação <i>Roll Up</i>	89
5.9	Operação <i>Rotate</i>	91

Lista de Tabelas

4.1	Contagem do número de operações por nível da hierarquia	65
4.2	Contagem do número de operações e tipo de coleta	66
4.3	Contagem do número de operações de comunicação realizadas	66
4.4	Tempo total de execução e comunicação	67
4.5	Comparação do tempo total de execução com e sem <i>trace</i>	67
4.6	Contagem do número de operações em cada nível da hierarquia	68
4.7	Contagem do número de operações realizadas por linha e o tipo de coleta . . .	69
4.8	Contagem usando percurso por operações agrupando somente as operações de comunicação	69
4.9	Tempo total de execução e comunicação	70
4.10	Comparação do tempo total de execução com e sem <i>trace</i>	70
4.11	Contagem do número de operações em cada nível da hierarquia	70
4.12	Contagem do número de operações realizadas por linha, arquivo e o tipo de coleta	71
4.13	Contagem usando percurso por operações agrupando somente as operações de comunicação	72
4.14	Comparação do tempo total de execução com e sem <i>trace</i>	72
4.15	Contagem do número de operações em cada nível da hierarquia	73
4.16	Contagem do número de operações realizadas por linha e o tipo de coleta . . .	74
4.17	Contagem usando percurso por operações agrupando somente as operações de comunicação	75
4.18	Comparação do tempo total de execução com e sem <i>trace</i>	75

PIOLA, T. F. Tracing de aplicações paralelas com informações de alto nível de abstração, 2007. 106 f. Tese (Doutorado) - Instituto de Física, Universidade de São Paulo, São Carlos, 2007.

Resumo

A computação paralela tem se estabelecido como uma ferramenta indispensável para conseguir o desempenho esperado em aplicações de muitas áreas científicas. É importante avaliar os fatores que limitam o desempenho de uma aplicação paralela. Este trabalho vem apresentar o desenvolvimento e a implementação de uma ferramenta chamada *Hierarchical Analyses* que permite o levantamento de dados para análise de fatores de desempenho em programas paralelos de forma hierárquica, ou seja, permite coletar as informações acompanhando o nível de abstração usado pelo programador. Essa ferramenta é composta pelos módulos de coleta e transformação dos dados. O módulo de coleta chamado *HieraCollector* é responsável por coletar e armazenar os dados em arquivos no formato XML, sendo que o usuário não precisa alterar o código fonte de sua aplicação. O módulo de transformação chamado *HieraTransform* é responsável por transformar os dados coletados extraíndo medidas que permitam a realização da análise do programa paralelo. Para validação dos módulos de coleta e transformação foi utilizada a biblioteca MPI e o *framework* OOPS que utiliza orientação a objetos. Outra contribuição deste trabalho, foi o desenvolvimento da ferramenta visual chamada *HieraOLAP* que auxilia o usuário na análise de desempenho de programas paralelos.

Palavras-chave: programação paralela, análise de desempenho, trace.

PIOLA, T. F. Tracing of parallel applications with information of high level abstraction, 2007. 106 f. Thesis (Doctoral) - Instituto de Física, Universidade de São Paulo, São Carlos, 2007.

Abstract

Parallel computing has become an essential tool to achieve the performance needed by many scientific applications. The evaluation of performance factors of parallel applications is of utmost significant. This work presents the developement and implementation of a tool called Hierarchical Analyses which facilitates data collection for performance analysis of parallel programs with hierarchical information, i.e. the information is collected in the various abstraction levels used in the application program. The tool consists of a collection and a transformation modules. The collection module (HieraCollector) collects the data and stores it in XML format. The transformation module (HieraTransform) processes the collected data computing measurements to be used in the analysis of parallel code. To validate the tool, implementations adapted to MPI and the OOPS framework have been developed. Another contribution of this work is the development of a visual tool called HieraOLAP to help the user in the analysis of parallel program performance.

Keywords: parallel programming, performance analyses, trace.

CAPÍTULO
1
Introdução

1.1 Motivação

Nas últimas décadas, o desempenho e a capacidade dos microprocessadores têm crescido rapidamente, contudo as exigências das aplicações cresceram ainda mais do que a sua velocidade de operação. Apesar de toda a capacidade computacional disponível atualmente, o uso de apenas um computador não tem sido suficiente para muitas aplicações, especialmente nas áreas científica, tecnológica e industrial. Uma maneira viável e de baixo custo é conectar vários computadores e coordenar seus esforços, o que resulta em um sistema conhecido como computador paralelo que permite a execução de programas concorrentemente. Alguns exemplos desse sistema são os *clusters* de computadores do tipo Beowulf [1] e *Grid* computacional [2].

A presença de sistemas paralelos tem se tornado cada vez mais forte, não apenas em sistemas tradicionalmente de alto desempenho, onde dominam, mas também cada vez mais em sistemas comerciais. Dessa forma, a computação paralela tem emergido como uma ferramenta indispensável para a resolução de problemas em muitas áreas científicas, sendo o paralelismo, em muitos casos, o único caminho para conseguir o desempenho desejado.

Em termos de *hardware*, o processamento paralelo já atingiu um bom nível de maturidade que permite seu amplo uso em diversas aplicações. Entretanto, a maioria dos sistemas paralelos tem sido usada para aumento do *throughput* ou redução do tempo de resposta a múltiplos usuários. A razão para isto é clara: a programação de sistemas paralelos é bem mais difícil do que a de máquinas sequenciais. Isso ocorre porque, na programação paralela, é necessário se preocupar com as diversas linhas de controle, executando simultânea e concorrentemente, e com as suas interações [3]. Além disso, há uma preocupação muito grande com o desempenho, pois é a única forma de se justificar a complexidade adicional do código.

Usuários, administradores e projetistas de sistemas de computadores estão interessados na análise de desempenho para que seja possível obter o desempenho mais alto a um custo mais baixo. A análise contínua de sistemas com desempenho mais alto a um custo baixo conduziu à multiplicação de *workstations* e computadores pessoais, muitos dos quais com o desempenho melhor do que os antigos supercomputadores [4].

A avaliação de desempenho é necessária em todo o estágio do ciclo de vida de um sistema de computador, incluindo seu projeto, fabricação, compra/venda, uso, *upgrade*, etc. Uma avaliação de desempenho é necessária quando o projetista de sistema de computador quer comparar um número de projetos alternativos e encontrar o melhor projeto e decidir qual sistema é melhor para um dado conjunto de aplicações. Mesmo se não existem alternativas, a avaliação de desempenho de sistemas atuais ajuda a determinar o quão bom é o desempenho e se qualquer melhoramento precisa ser feito. Infelizmente, os tipos de aplicações são tão numerosas que não é possível ter uma medida, um ambiente ou uma técnica padrão de desempenho para todos os casos. O primeiro passo na avaliação de desempenho é selecionar as medidas, os ambientes e as técnicas de avaliação de desempenho a serem utilizadas.

Selecionar uma técnica e uma métrica de avaliação são dois passos chaves em todos os projetos de avaliação de desempenho. As três técnicas para a avaliação de desempenho são a modelagem analítica, simulação e medição. O ponto chave em decidir a técnica de avaliação é o estágio do ciclo de vida em que o sistema está. Modelagem analítica e simulação são baseadas em modelos abstratos em vez de sistemas reais e medição é possível somente se algo similar ao sistema proposto já existe.

Modelagem analítica e simulação usam modelos, o que requer simplificações e suposições e são apenas uma representação aproximada da realidade. A simulação permite modelos mais detalhados e complexos, resultando assim em soluções mais precisas. Como medição é realizada diretamente no sistema real, ela oferece resultados mais precisos dentre as três técnicas [5].

A técnica básica de medição é o uso de instrumentação, que a torna o método com o custo mais alto em relação a modelagem analítica e simulação. Uma forma para utilizar a instrumentação na avaliação de desempenho é usar o processo proposto por Foster em [6] que possui três passos básicos que serão utilizados no desenvolvimento deste trabalho que são:

- **Coleta de dados:** é o processo pelo qual os dados sobre o desempenho do programa são obtidos da execução do programa. Esses dados são, em geral, coletados em arquivos, durante ou após a execução, podendo também ser apresentados ao usuário em tempo real. Existem três técnicas para coleta de dados: *profile* que possibilita obter dados mais ge-

rais do programa, como por exemplo, a quantidade de tempo gasto em diferentes partes o programa; *counters* possibilitam obter, por exemplo, o número de chamadas a procedimentos, e a quantidade total de mensagens e *trace* que é uma forma mais detalhada para a coleta de dados, incluindo informações como o nome do procedimento, processo remetente e destinatário de mensagens, entre outras informações sobre a execução do programa. Essas três técnicas serão discutidas no capítulo 2.

- **Transformação dos dados:** é aplicada com o objetivo de reduzir o volume total de dados. Pode ser usada para determinar valores médios ou outras estatísticas ou através de um arquivo de *trace* extrair dados referentes a *profile* e *counter* desse arquivo. Por exemplo, um *profile*, registrando o tempo gasto a chamadas de subrotinas de um determinado processador, pode ser transformado para determinar o tempo médio gasto nas chamadas de subrotinas de cada processador e o desvio padrão desse tempo médio. De maneira semelhante, um *trace* pode ser processado para gerar um histograma dando a distribuição do tempo de execução de rotinas.
- **Visualização dos dados:** é utilizada para produzir imagens para auxiliar na compreensão dos resultados pelo programador.

1.2 Níveis Hierárquicos

Hoje em dia, é comum o uso de abstração de mais alto nível pelos usuários que usam bibliotecas MPI, por exemplo. Uma vez que um programador tenha escolhido um conjunto de abstrações para a implementação de seu programa, se ele precisar coletar informações sobre a execução desse programa, essas informações deverão corresponder às informações sobre o desempenho no mesmo nível de abstração utilizada, para que possam ser compreensíveis e úteis ao programador.

Figura 1.1: Níveis de Abstração

Como exemplo, considerando um programa que realiza diagonalização de matriz em paralelo, utilizando a biblioteca POOLALi (*Parallel Object Oriented Linear Algebra Library*) [7] que é uma biblioteca paralela orientada a objetos que lida com problemas de autovalores e autovetores. A biblioteca POOLALi, mostrada na figura 1.1, utiliza diversos recursos da biblioteca ScaLAPACK [8, 9, 10] que por sua vez utiliza suas bibliotecas internas como o BLACS (*Basic Linear Algebra Communication Subprogram*) [11, 12] e o PBLAS (*Parallel BLAS*) [13, 14]. As rotinas da biblioteca BLACS utilizam bibliotecas de passagem de mensagens como PVM [15] ou as baseadas no padrão MPI [16, 17], tais como MPICH [18, 19] e LAM [20].

A biblioteca ScaLAPACK é uma biblioteca de álgebra linear desenvolvida para computadores de memória distribuída e que usa passagem de mensagens na comunicação entre os processos. O BLACS, uma biblioteca de comunicação especialmente desenvolvida para álgebra linear, considera que os processos estão distribuídos de acordo com uma grade de duas dimensões permitindo que os processos troquem mensagens entre si de forma síncrona. O PBLAS é a versão paralela do BLAS e é constituído de rotinas de álgebra linear básicas, tais como as oferecidas pelo BLAS (rotinas básicas de álgebra linear, como operações vetor-vetor, matriz-vetor e matriz-matriz), com a diferença que suas operações são realizadas em paralelo com o auxílio da biblioteca BLACS, que faz a passagem de mensagens, e do BLAS, usado nas computações locais.

No exemplo apresentado na figura 1.1, a coleta hierárquica se torna muito importante devido à existência de vários níveis, ou seja, quando várias bibliotecas são usadas em uma mesma aplicação. Nesse caso é importante que possam ser coletados os dados em todos os níveis e não apenas no nível do MPI (o nível mais baixo apresentado na figura 1.1), pois pode ser que o problema de desempenho apresentado no programa não esteja no nível do MPI e sim nos níveis intermediários como no nível das bibliotecas BLACS e PBLAS ou mesmo no nível mais alto que é o da biblioteca POOLALi.

No caso da figura 1.1, o usuário pode estar interessado nos dados coletados no nível mais baixo que é o caso do MPI, ou no nível mais alto que é o da biblioteca POOLALi, ou ainda o usuário pode estar interessado nos dados coletados no nível intermediário (PBLAS ou BLACS). Nesse caso a coleta deverá acompanhar o nível de abstração utilizado no programa.

1.3 Objetivo do Trabalho

O trabalho aqui apresentado tem por objetivo a definição e implementação de uma ferramenta que permita o levantamento de dados importantes nos vários níveis hierárquicos para análise de fatores que afetam o desempenho de programas paralelos. Essa ferramenta deve ser capaz de acompanhar o nível de bibliotecas utilizado pelo usuário em seu programa, permitindo assim,

que o usuário possa realizar a análise dos dados coletados em todos os níveis, ou apenas nos níveis escolhidos.

Assim, é proposto que seja desenvolvida uma ferramenta que permita a coleta dos dados ao nível hierárquico utilizado, permitindo realizar a transformação desses dados para facilitar a análise.

Para exemplificar a utilização da ferramenta serão utilizadas a biblioteca MPI e um *framework* chamado OOPS (*Object-Oriented Parallel System*) [21], que utiliza orientação a objetos para apoiar a implementação de programas científicos concorrentes para a execução paralela fornecendo abstrações de alto nível. Outras aplicações serão utilizadas para a validação da biblioteca.

Algumas características da ferramenta proposta são:

- Utilização da ferramenta para a realização da coleta dos dados sem necessidade de alteração de código da aplicação do usuário final (através de chaves de compilação ou ligação com bibliotecas especiais). Para o desenvolvedor de biblioteca será necessário a alteração do código.
- A coleta feita de forma hierárquica permite a coleta total e seleção de parte dos dados coletados durante a análise. Todos os dados da aplicação são coletados e armazenados em arquivos estruturados.
- Aplicação de medidas sobre os dados coletados dos vários níveis hierárquicos para a realização da análise de desempenho do programa paralelo.

1.4 Organização do Texto

O texto está organizado da seguinte maneira: o capítulo 2 apresenta técnicas usadas para a coleta, transformação e visualização dos dados em sistemas paralelos. Ferramentas utilizadas para a coleta, análise e visualização dos dados são apresentadas no capítulo 3. O capítulo 4 apresenta as características gerais e detalhes da ferramenta desenvolvida. Uma ferramenta visual para auxiliar na análise dos programas paralelos é descrita no capítulo 5. As conclusões e trabalhos futuros são apresentados no capítulo 6.

CAPÍTULO 2 Processo de Análise de Desempenho

O processo de análise de desempenho é composto por três passos básicos: a coleta, a transformação e a visualização dos dados; como esquematizado na figura 2.1. Algumas técnicas de coleta de dados usadas para obter informações sobre o desempenho de programas paralelos são discutidas na seção 2.1. A seção 2.2 mostra como os dados coletados são transformados para a facilitar a análise de desempenho. Técnicas de visualização são apresentadas na seção 2.3. A seção 2.4 apresenta o conceito de *Data Warehousing* [22] e a tecnologia OLAP (*OnLine Analytical Processing*) [23] também utilizadas no desenvolvimento deste trabalho. A seção 2.5 descreve o algumas conclusões deste capítulo e as técnicas que serão utilizadas no desenvolvimento deste trabalho.

Figura 2.1: Processo de Análise de Desempenho

2.1 Coleta dos Dados

Como mencionando anteriormente, a coleta de dados é o processo pelo qual os dados sobre o desempenho de um programa são obtidos. Os dados são normalmente coletados em arquivos, durante ou após a execução, embora em algumas situações eles sejam apresentados em tempo real.

A instrumentação pode ser inserida de duas formas:

- **Estática:** é feita durante o desenvolvimento ou compilação do código. Uma desvantagem desse método é a falta de flexibilidade, pois a instrumentação não pode ser modificada durante a execução do programa. Assim, qualquer mudança nos tipos de dados coletados requer reinstrumentação, recompilação e uma nova execução. Exemplos de sistemas que fazem instrumentação estática são: gprof [24], AIMS (*Automated Instrumentation and Monitoring System*) [25], EEL (*Executable Editing Library*) [26] e Pablo [27].
- **Dinâmica:** é feita durante a execução. Ao contrário da instrumentação estática, a dinâmica permite aos usuários mudar pontos de instrumentação, focalizando regiões do código onde problemas de desempenho são detectados. A desvantagem aqui é que a aplicação deve executar por um tempo suficiente para que os usuários encontrem problemas de desempenho modificando a instrumentação. O Paradyne [28] é um exemplo de sistema que realiza a instrumentação dinâmica.

Os níveis mais comuns em que um sistema pode ser instrumentado são [29, 30]:

Instrumentação por Hardware: a abordagem em *hardware* consiste em associar instruções ao sistema de *hardware* para observar o comportamento do programa sem perturbá-lo. O monitoramento de *hardware* tem a vantagem de não ser intrusivo. Tem sido usado com sucesso para monitoramento de redes de comunicação [31], onde muitas informações de monitoramento tem que ser geradas e processadas rapidamente. As desvantagens do monitoramento de *hardware* são: requer *hardware* adicional, o que o torna mais caro; fornece dados de nível muito baixo e não satisfaz as exigências do programador da aplicação em ambientes paralelos; forma a classe menos portátil de mecanismos de monitoramento.

Instrumentação por Software: adiciona um conjunto de instruções na aplicação para habilitar a coleta em tempo de execução de toda a informação desejada sobre o comportamento do sistema. Fornece informação em alto nível de abstração, diretamente empregada por projetistas ou ferramentas de exibição gráfica. Monitoramento por *software* em geral compartilha os recursos necessários com o sistema monitorado. O programa é instrumentado inserindo instruções no código para detectar eventos. O uso de monitoramento por *software* apresenta as seguintes vantagens [31]:

- a informação monitorada é apresentada de uma maneira fácil de entender e usar, comparada à informação de baixo nível gerada pelo monitoramento de *hardware*;

- monitoramento por *software* é portátil e pode facilmente ser reproduzido;
- comparado ao monitoramento por *hardware* é mais fácil de projetar e construir e é mais flexível;
- nenhum recurso de *hardware* dedicado é requerido. Isto o faz mais barato do que o monitoramento por *hardware*.

A principal falha da instrumentação por *software* está relacionada com o tempo de sobrecarga introduzida pela coleta da informação. Para limitar o efeito de intrusão, a instrumentação é, geralmente, limitada aos eventos cuja observação é considerada essencial. Existem várias abordagens para instrumentação por *software*, por exemplo:

- **Instrumentação do Código Fonte:** as instruções são inseridas no código fonte do programa em pontos apropriados. Fornece facilidades de monitoramento flexível e portátil. A desvantagem é que a modificação de instruções necessita de recompilação das partes afetadas do código. Essas instruções podem ser inseridas manualmente ou automaticamente.
- **Instrumentação de Rotinas de Bibliotecas:** como uma alternativa para o método anterior, instruções podem ser inseridas no código fonte de rotinas da biblioteca. O comportamento do programa pode ser monitorado chamando as primitivas da biblioteca instrumentada. Uma vantagem deste método é que ele fornece um alto nível de portabilidade. A desvantagem é que somente aqueles eventos para os quais as instruções são inseridas nas rotinas de bibliotecas podem ser detectados e relatados.
- **Instrumentação do Código Objeto:** instruções são inseridas no código objeto ou em uma representação intermediária do programa em tempo de compilação. Isto é feito usando um compilador de instrumentação especial. Instrumentar a representação intermediária ao invés do código objeto tem a vantagem de que a instrumentação é independente da máquina e pode ser transparente ao programador. Isto resulta em menor sobrecarga do que a instrumentação do código fonte, porque o baixo nível das instruções de máquina pode ser usado ao invés do alto nível dos comandos fonte que precisam ser compilados. A desvantagem deste método é que ele requer um compilador de instrumentação.
- **Instrumentação do Kernel:** instruções são inseridas no código do *kernel* para detectar eventos do sistema. Essas instruções são executadas quando um programa faz chamadas a uma função do *kernel* (por exemplo, envio e recepção de mensagem). Esse método é similar à técnica de instrumentação de rotinas de biblioteca e

tem a vantagem de fazer a detecção do evento transparente para o programa. Uma desvantagem é que somente eventos relacionados a chamadas do *kernel* podem ser detectados e não eventos da aplicação.

Instrumentação Híbrida: é projetada para se beneficiar das vantagens da instrumentação por *hardware* e por *software*, enquanto supera suas ineficiências. É baseada na adição de poucas instruções nos programas, a fim de selecionar a informação adequada para explicar o comportamento. A informação é coletada com um dispositivo específico e transmitida para o sistema que o interpreta e exibe o resultado.

A principal vantagem da instrumentação híbrida é que introduz baixa sobrecarga no sistema monitorado quando comparada com a instrumentação por *software*. É mais barato em comparação à instrumentação por *hardware* porque compartilha seus recursos com o sistema monitorado usando assim menos recursos dedicados. Por compartilhar recursos, a instrumentação híbrida é mais intrusiva. A instrumentação híbrida é menos portátil do que a instrumentação por *software*, por causa do uso de *hardware* dedicado.

2.1.1 Métodos de Instrumentação

Independente do nível em que a instrumentação é feita, existem três técnicas para capturar dados de desempenho: *profile*, *counter* e *trace*. Cada uma delas reflete um diferente balanceamento entre volume de dados, a sobrecarga da instrumentação, a complexidade de implementação, e cada uma pode ser implementada de diferentes maneiras [32].

A técnica de *profile* consiste em monitorar um sistema durante sua execução e analisar os dados obtidos. Existem duas maneiras de coletar dados de *profile* [33]: através de eventos de *tracing* obtidos da instrumentação do código ou através de amostragem estatística. Eventos de *trace* podem ser mais intrusivos, mas permitem ao *profile* registrar todos os eventos de interesse. Amostragem estatística pode ser menos intrusiva para a execução do sistema, mas não pode fornecer informações completas da execução.

Essas técnicas são utilizadas para análise de desempenho, o que possibilita identificar componentes de um sistema que limitam seu desempenho; medir o uso de recursos utilizados; saber quanto tempo é gasto em chamadas às rotinas; o número de vezes que cada bloco do programa é executado; verificar a comunicação e sincronização realizada, além de determinar onde o sistema gasta seu tempo durante sua execução.

Profile (Perfil)

A técnica de *profile* registra a quantidade de tempo gasto em diferentes partes de um programa. Esta informação é freqüentemente de grande importância para destacar problemas de desempenho [34]. Os *profiling* podem ser obtidos automaticamente, com custo relativamente baixo, e podem fornecer uma visão geral do comportamento do programa, permitindo ao programador identificar partes problemáticas do programa sem gerar enormes quantidades de dados.

O *profiling* em sistemas seqüenciais é mais simples do que o realizado em sistemas paralelos, devido à menor quantidade de dados coletados. Uma ferramenta utilizada para encontrar gargalos em programas seqüenciais apontando para as rotinas que consomem mais tempo de CPU, é o gprof [24]. O gprof permite saber onde o programa gasta seu tempo e quais funções são chamadas durante sua execução. Esta informação pode mostrar quais pedaços do programa estão mais lentos do que o esperado, e esses por sua vez, podem ser reescritos para torná-los mais rápidos. É possível mostrar quais funções estão sendo chamadas com mais freqüência do que o esperado.

O *profiling* avalia as rotinas em todos os níveis de decomposição do programa. É facilmente usado e pode ser compilado no programa sem qualquer planejamento especial pelo programador. A execução em um sistema de tempo compartilhado usa somente os serviços fornecidos pelo sistema operacional e compiladores. A sobrecarga do uso de gprof é baixa, em termos do tempo de execução e volume de informação de *profiling* registrada [35].

Em geral, *profiles* estão disponíveis na maioria dos computadores paralelos, mas variam amplamente em suas funcionalidades e sofisticações. O mais básico faz pouco mais do que a coleta de dados do *profiling* seqüencial em cada processador; e o mais sofisticado fornece vários mecanismos para reduzir estes dados, exibí-los e relacioná-los ao código fonte [6].

Counters (Contadores)

São locais de armazenamento que podem ser incrementados cada vez que um evento específico ocorre. Podem ser usados para registrar o número de chamadas aos procedimentos, número total de mensagens, volume total das mensagens, ou o número de mensagens enviadas entre cada par de processos.

Counters complementam *profiles* fornecendo informações que não são facilmente obtidas usando técnicas de amostragem. Eles podem fornecer o número e o tamanho total de mensagens, informações que podem ser combinadas com tempo de comunicação de dados de um *profile* para determinar a eficiência das operações de comunicação. Os contadores são geralmente considerados como minimamente intrusivos e envolvem o gerenciamento de pouca quantidade

de dados [36].

Uma variação útil de um *counter* é o temporizador de intervalos que é usado para determinar o tempo gasto na execução de um determinado pedaço do código. Esta informação pode ser acumulada em um *counter* para fornecer uma determinação exata do tempo total gasto executando um componente do programa.

Traces (Rastros de Execução)

Trace é a técnica de instrumentação mais geral. É utilizada para medir tempos de comunicação, registrar eventos de emissão e recepção, exibir gargalos, e registrar onde os processos gastam o tempo executando uma aplicação. Ela pode também ser usada para obter informação global ou informação detalhada de contagem e tempo. *Tracing* pode ser realizado em vários níveis de abstração e existem várias técnicas de implementação.

Trace é a forma mais detalhada para a coleta de dados. Sistemas baseados em *tracing* geram arquivos contendo registros de eventos representando ocorrências significativas na execução do programa, tais como, chamadas a procedimentos ou envio de mensagens. Registros de *tracing* podem incluir informações como o tipo de evento e o nome do procedimento ou a tarefa destino, e podem ser gerados automaticamente ou sob controle do programador. É a técnica de coleta de dados que apresenta maior quantidade de dados em relação às técnicas de *profile* e *counter*.

Cada registro inclui o identificador do evento, quando e o local onde o evento ocorreu, por exemplo, o número da linha no código fonte, entre outras informações [37]. O volume de dados produzidos pelo *tracing* de execução pode ser enorme, principalmente em sistemas paralelos. Desse modo, quando registros de dados são produzidos por múltiplos processadores, eles podem ser extremamente difíceis de interpretar.

O padrão MPI [16, 17] define uma interface que permite a implementação de *tracing*, denominada pelo padrão de “interface de *profiling*”. A chave para isto é que qualquer implementação de MPI deve permitir que um programa chame funções MPI, através de MPI_Xxx ou PMPI_Xxx.

A interface de *trace* MPI é somente uma interface, e não existe nenhum método que apresente a forma pela qual a informação é coletada ou mesmo como essa informação coletada é salva, filtrada ou apresentada.

2.2 Transformação dos Dados

A transformação dos dados é aplicada com o objetivo de reduzir o volume total dos dados. Pode ser usada para determinar valores médios ou outras estatísticas. A forma mais simples

de análise filtra dados que não satisfazem critérios especificados pelo usuário, retendo e computando métricas apenas dos dados pertinentes (por exemplo: padrão de métricas estatísticas tais como média, mínimo, máximo, percentuais, variância). Algumas métricas de desempenho comumente usadas são:

- **Tempo de Resposta:** é definido como o intervalo entre a requisição do usuário e a resposta obtida.
- **Throughput:** é a quantidade de dados transferidos de um lugar a outro, ou a quantidade de dados processados em um determinado espaço de tempo. A taxa de transferência efetiva de um determinado sistema (uma rede de roteadores por exemplo) pode ser menor que a taxa de entrada devido às perdas e atrasos no sistema.
- **Utilização:** a utilização de um recurso é medida como a fração de tempo que o recurso está ocupado atendendo requisições. Assim, esta é a taxa de tempo ocupado e o tempo total gasto sobre um dado período. O período durante o qual um recurso não está sendo usado é chamado de tempo ocioso.
- **Confiabilidade:** a confiabilidade de um sistema é geralmente medida pela probabilidade de erros ou pelo tempo médio entre erros.
- **Disponibilidade:** a disponibilidade de um sistema é definida como a fração de tempo que o sistema está disponível para atender requisições do usuário.
- **Taxa custo/desempenho:** é geralmente usada como uma métrica para comparar dois ou mais sistemas. O custo inclui o custo de licença de *hardware/software*, instalação e manutenção sobre um número de anos. O desempenho é medido em termos de *throughput* sobre um dado tempo de resposta.

Uma forma de fazer a transformação dos dados é utilizar estratégias para a redução dos dados. Algumas estratégias usadas para a redução são [38]:

Estratégia de Agregação da Informação : esta estratégia realiza a redução de *trace* permitindo a perda de informação. Por exemplo, Nickolayev em [39], propõe um método em tempo de execução que avalia periodicamente a similaridade do comportamento dos processos para gerar *tracing* somente para alguns deles e esses serão representantes dos outros processos similares. Tavera em [40], concentra-se na frequência de eventos para

selecionar o nível de detalhes apropriado para a geração do *trace*. Para fazer isso, é necessário um valor limiar na ocorrência de eventos, dessa forma registra somente o número de eventos ocorridos no nível mais grosso.

Yan e Schmidt em [41], reduzem o número de eventos registrados substituindo eventos medidos por eventos atuais. Por exemplo, quando o mesmo comando ocorre em um *loop* gerando uma sequência de muitos eventos, seu método registra somente dois valores, isto é, o número de ocorrências e o tempo de execução médio para estes eventos. Assim, os tempos de execução não são registrados para todos os eventos.

Estratégia de Compressão Sem Perda de Dados : esta estratégia reduz o tamanho do arquivo de *trace* sem a perda de informação. Além da estratégia de agregação da informação mencionada acima, Yan e Schmidt [41] reduzem o tamanho substituindo seqüências repetitivas com fórmulas. Por exemplo, eles registram uma fórmula de 0^31^2 em vez de uma seqüência de 00011. Como esse método foca na representação de uma seqüência de dados, ele é eficaz para a parte regular do *trace*, tais como processo remetente/destinatário e o *tag* de comunicação, ambos registrados para eventos de comunicação. Entretanto, ele não é tão eficiente para a parte irregular, por exemplo, o tempo de ocorrência para eventos.

Estratégia de Instrumentação Dinâmica : esta estratégia controla dinamicamente em tempo de execução a geração de *traces* durante a execução do programa [42, 43]. Embora esta estratégia seja útil para programas de grande escala, ela não pode suportar análise baseada em caminho crítico [44], o qual requer o *trace* inteiro do início ao final da execução do programa.

Para problemas de desempenho mais complexos, análises detalhadas baseadas em eventos de *tracing* podem ser exigidas [30]. No entanto, para manter as vantagens de *trace* minimizando o volume de dados e intrusão, os analistas devem reduzir o número de métricas que são necessárias para identificar gargalos (dimensionalidade da métrica) e o número de locais onde os dados devem ser capturados (pluralidade da métrica).

Existem várias possíveis abordagens para reduzir métricas de dimensionalidade e pluralidade, que podem ser o agrupamento estatístico que reduz o número de tarefas paralelas por omitir dados de grupos de processos com um comportamento similar e a busca de projeção que reduz a dimensionalidade identificando métricas importantes [30, 45].

Agrupamento Estatístico O agrupamento estatístico é uma técnica padrão para identificar classes de dados equivalentes. A maioria das técnicas de agrupamento são hierárquicas ou por particionamento. Algoritmos de agrupamento hierárquico determinam a hierarquia de grupos dentro de um conjunto de dados. Algoritmos de agrupamento por particionamento modificam iterativamente o particionamento inicial do conjunto de dados até atingir um agrupamento ótimo [40].

O agrupamento estatístico busca identificar grupos de processos, onde as métricas para cada processo no grupo traçam trajetórias similares. Computando o número total de grupos usando métricas de desempenho de cada processo, eventos de *tracing* do sistema podem capturar e extrair *trace* somente dos membros representantes de cada grupo, reduzindo o volume total de dados [45].

Em [39] o comportamento do programa é descrito como caminhos em um espaço métrico multidimensional. No caso de programas projetados para grandes sistemas paralelos em decomposição funcional, cada função executa em um conjunto de processadores ou processos diferentes, particionando os processadores em classes equivalentes. No caso de programas SPMD, todos os processadores executam o mesmo código, diferenciado em controle de fluxo por dados locais, e processadores com dados similares também formam classes de comportamentos equivalentes.

Cada classe de grupos de processos equivalentes contém processadores cujas métricas são similares às outras dentro do grupo, e diferentes para processadores de outros grupos. Portanto, pode-se ver o comportamento de qualquer processador dentro de um grupo como a aproximação para o comportamento de todos os outros processadores no grupo. A similaridade comportamental de processadores no mesmo grupo permite reduzir o volume total de *traces* registrando somente os *traces* de um ou mais processadores representativos de cada grupo. O grau de redução é dado pela taxa do número total de processadores ao número de grupos representativos. Para execuções altamente paralelas, o número de grupos é pequeno, o que resulta em grandes reduções de dados.

Como o comportamento dos processadores e do programa muda durante a execução, o número de grupos, seu tamanho e suas formas variam. Portanto, qualquer técnica de agrupamento eficiente deve se adaptar a estas mudanças identificando os grupos e representantes dos grupos dinamicamente (isto é, quando a aplicação paralela executa). Antes que o agrupamento estatístico seja estabelecido, algumas questões como identificação de bons algoritmos de agrupamento, seleção de representantes de grupos, avaliação do custo de computação do agrupamento, e a quantificação da relação entre o volume de dados e a precisão do desempenho, devem ser resolvidas. Qualquer algoritmo de agrupamento dinâmico deve rastrear mudanças

no comportamento do programa identificando novas classes de comportamento equivalentes, selecionando representantes de cada classe, e salvando dados desses representantes.

Busca de Projeção Embora o agrupamento estatístico reduza o número de processos P (ou tarefas) dos quais os dados devem ser registrados, o agrupamento estatístico não reduz o número de métricas n e a dimensionalidade do espaço métrico. Mesmo após o agrupamento identificar um pequeno número de processos representantes, o volume total dos dados pode permanecer alto. A busca de projeção, procura identificar um subconjunto das métricas que capturam a maior parte da variação estatística.

Conceitualmente, em qualquer tempo t , muitas das n métricas estão altamente correlacionadas. A busca de projeção multidimensional seleciona no mínimo k métricas correlacionadas do espaço métrico multidimensional; onde k é a busca de projeção bi ou tridimensional. Usando busca de projeção, *tracing* em tempo real precisa registrar somente aquele subconjunto das métricas importantes para a projeção.

2.3 Visualização dos Dados

Um método utilizado para tratar grandes volumes de dados complexos é o método de visualização, que retrata graficamente os dados para uma melhor compreensão. Nesta seção serão apresentadas algumas técnicas para visualizar o comportamento de sistemas paralelos e distribuídos. O principal foco está sobre as técnicas gráficas para avaliar e melhorar o desempenho.

Visualização é um componente vital no desenvolvimento e aplicação de ferramentas de avaliação de desempenho e compreensão do comportamento de sistemas paralelos [46]. A natureza complexa da informação de desempenho e os complicados modelos operacionais do comportamento da execução paralela fornecem um rico contexto para a visualização. Ferramentas para análise de desempenho paralelo têm incorporado técnicas de visualização de várias maneiras.

Técnicas de visualização podem ser aplicadas ao projeto do programa, avaliação de desempenho e depuração. A depuração de programas paralelos permanece uma tarefa extremamente desafiante e ferramentas para ajudar nessa área são necessárias [47].

Apresentações textuais de dados que descrevem a execução de um sistema paralelo são inerentemente sequenciais e podem ser difíceis de assimilar. Visualizações gráficas estão tornando-se ferramentas úteis para o entendimento do comportamento do sistema e execução do programa, e para depuração e avaliação de desempenho.

A tarefa de visualização de desempenho é dividida em dois passos. O primeiro passo é

realizado pela instrumentação do programa (apresentada na seção 2.1) que coleta dados da execução do programa paralelo. A segunda parte está preocupada com a visualização e o uso destes dados coletados para mostrar o comportamento do programa durante a execução [48].

Exibições visuais de dados de desempenho podem ser classificadas em: orientada por processador (*hardware*) ou orientada por processos (*software*). Podem descrever eventos de baixo nível (*load/store*, *send/receive*) ou mostrar atividades de nível mais alto que sejam mais significativas para a aplicação. Podem ainda mostrar eventos individuais em processadores individuais como uma função de tempo, ou somente tempo e espaço total. Podem fornecer uma imagem estática ou animação do comportamento dinâmico de sistemas paralelos.

Exibições orientadas a memória podem ser mais apropriadas para sistemas de memória compartilhada, enquanto que exibições orientadas a comunicação são mais apropriadas para sistemas de memória distribuída. Em um multiprocessador simétrico de memória compartilhada, ou um sistema em que paralelismo é automatizado ou implícito, a identificação de processadores individuais pode ser de pouco interesse, ao passo que para programas explicitamente paralelos a identificação do processador pode ser de vital importância [3].

Abaixo são apresentadas algumas formas de exibições orientadas por processador e comunicação.

Exibição por Processador : a utilização do processador é uma métrica usada para avaliar o desempenho de programas paralelos. Informação sobre como o programa usa o processador de uma máquina paralela fornece com frequência compreensão sobre decomposição de tarefas e distribuição de dados [49].

O tipo mais básico de exibição orientada por processador mostra a porcentagem do tempo de execução gasto pelo processador durante a execução. Esse tipo de exibição fornece informações sobre a eficiência do programa e o balanceamento de carga nos processadores.

No ajuste de desempenho, procura-se minimizar o desbalanceamento de carga, aumentar a concorrência, e minimizar a sobrecarga. Exibições orientadas por processador descrevem graficamente quantidades em termos de baixo nível da informação sobre processadores individuais, como por exemplo, se um processador está ocupado ou ocioso. A sobrecarga existente em sistemas devido à comunicação interprocessador é relativamente fácil de monitorar; por outro lado, a sobrecarga dependente da aplicação, como computação redundante, não é facilmente monitorado sem ajuda do usuário. Devido a isto, a exibição orientada por processador descreve apenas a sobrecarga em sistemas e não tenta capturar possível sobrecarga em nível da aplicação inerente no algoritmo.

Um modo de exibir a utilização do processador é através do diagrama de *Kiviat*. A forma tradicional da exibição em 2D inicia com vários raios partindo de um ponto central. Um raio representa um processador em computação paralela, e a quantidade medida é a porcentagem de tempo que o processador gasta computando durante um certo intervalo de tempo. Quanto maior o raio, maior a utilização do processador e melhor o desempenho. A forma do diagrama *Kiviat* é também um bom indicador de balanceamento de carga entre os processadores [49].

Outra forma de exibir a utilização do processador é através do uso do gráfico de *Gantt* que é um gráfico de barras horizontais, onde cada barra representa o estado de cada processador em função do tempo. As barras podem simplesmente representar os estados de computação, ocioso, comunicação e/ou indicar o componente ou o procedimento do programa que está executando em cada processador.

O uso de diagrama de *Kiviat* e do gráfico *Gantt* é eficaz em fornecer informações sobre concorrência, balanceamento de carga e sobrecarga em computação paralela. Essas exibições podem alertar o usuário para problemas de desempenho [3].

Exibição por Comunicação : a comunicação interprocessador é a principal fonte de sobrecarga em sistemas paralelos, sendo importante fornecer informações sobre as atividades de comunicação no programa paralelo para o usuário. A frequência, volume e o padrão de comunicação determinam a eficiência de programas paralelos.

A comunicação interprocessador pode ser descrita logicamente (sem considerar a rede de interconexão) ou fisicamente (em termos do trajeto percorrido em uma rede específica). A exibição lógica de comunicação é independente de qualquer topologia de rede particular e portanto, pode ser portátil entre arquiteturas paralelas. Pode ser feita também a exibição do tráfego de comunicação, sendo necessário informações mais detalhadas do que apenas os processos origem e destino para cada mensagem.

2.4 Data Warehousing

O processo de *Data Warehousing* é formado por uma coleção de tecnologias que tem por objetivo dar suporte à tomada de decisão [50].

As bases de dados utilizadas para armazenar as informações do processo de *Data Warehousing* em um banco de dados, é chamado *Data Warehouse*. O *Data Warehouse* possibilita sumarizar e analisar grandes volumes de dados, sendo o núcleo dos sistemas de informações gerenciais e de apoio à decisão.

Uma característica importante é a integração dos dados que ocorre quando os dados passam do ambiente operacional para o *Data Warehouse*. O processo de introdução dos dados no *Data Warehouse* é conduzido sempre tendo em vista a preocupação de eliminar muitas inconsistências das aplicações, no que se refere a codificação, convenções de atribuição de nome, atributos físicos, unidades de medida, etc. Assim, existem dois tipos de dados em um ambiente *Data Warehouse*: os dados primitivos que são aqueles dados obtidos do ambiente operacional que não sofreram mudanças; e os dados derivados que são dados primitivos que foram transformados de alguma forma.

Por definição [22], os dados em um *Data Warehouse* não são voláteis, ou seja, eles não mudam, salvo quando é necessário fazer correções de dados previamente carregados. Os dados estão disponíveis somente para leitura e não podem ser alterados.

Um outro aspecto importante do *Data Warehouse* é a granularidade. A granularidade diz respeito ao nível de detalhe ou de resumo contido nas unidades de dados existentes no *Data Warehouse*. Quanto mais detalhes, mais baixo o nível de granularidade. Quanto menos detalhes, mais resumido e mais elevado o nível de granularidade.

O *Data Warehouse* é composto por quatro módulos: acesso, transformação, base multidimensional e distribuição.

O módulo de acesso é responsável pela migração dos dados do ambiente operacional para o ambiente do *Data Warehouse*. Nessa migração existem processos que realizam leituras periódicas no ambiente operacional.

No módulo de transformação, os dados primitivos ou operacionais previamente carregados do ambiente operacional são lidos e integrados, pois não há como garantir que todos os dados operacionais estejam gravados da mesma forma e nem como garantir que o mesmo tipo de campo não tenha nomes distintos em várias aplicações, ou até mesmo que os dados sejam concatenados em um único campo em um aplicativo e que em outro seja separado em dois ou três campos. Para que os dados sejam gravados corretamente e de maneira integrada no *Data Warehouse* é necessário que ocorra um rastreamento comparativo entre os campos existentes e os campos do *Data Warehouse*.

A base multidimensional ou cubo de dados representa os dados armazenados usando o Modelo Estrela [50]. Para cada modelo estrela existe uma tabela central, conhecida como tabela dos fatos, que contém campos com valores numéricos que possibilitam aplicar sumarizadores como média, contagem e soma. Ligada a tabela fato, existem várias outras tabelas, conhecidas como tabela dimensão. As tabelas dimensão contém campos que devem ser claros e significativos para o usuário. Um campo pode ser especializado ou generalizado em outro campo, sucessivamente formando uma hierarquia. O campo de menor hierarquia define a menor gra-

nularidade da informação e o campo de maior hierarquia define a maior granularidade. Uma ou mais bases multidimensionais constituem um *Data Mart* que é construído para uma finalidade e grupos de usuários específicos [51].

Figura 2.2: Exemplo do Modelo Estrela para Vendas

A figura 2.2, mostra o modelo estrela para vendas de supermercado. A tabela fatos (tabela central) é chamada vendas é caracterizada pelos campos valor e quantidade. Existe ligada ao fato vendas três dimensões: produto, região e tempo. A dimensão produto contém os campos da granularidade mais alta para a mais baixa: categoria, fabricante e nome do produto. A dimensão região contém os campos: país, estado e cidade. A dimensão tempo contém os campos: ano, mês, semana e dia.

O módulo de distribuição é constituído pela implementação de ferramentas que exploram e visualizam os dados de um *Data Warehouse*. Essas ferramentas são responsáveis por fazer consultas e/ou gerar relatórios, seja de forma resumida, seja de forma explicativa e com gráficos e outros resumos analíticos. A ferramenta mais popular de distribuição é o OLAP (*On-line Analytical Processing* ou Processamento Analítico em Tempo Real), que será descrita na subseção 2.4.1.

2.4.1 OLAP

OLAP (*On-Line Analytical Processing*) [23, 52] é usado para resumir, consolidar e sintetizar os dados conforme múltiplas dimensões. Ele tem sido usado em aplicações, tais como, modelagem financeira, alocação de recursos, análise de variância, entre outras.

A ferramenta OLAP pode transformar dados de um *Data Warehouse* em informações estratégicas. Ela envolve desde uma básica navegação e pesquisa dos dados até os cálculos de tais dados, em uma análise de séries temporais e modelos complexos [53, 54]. Aplicações OLAP têm uma interface que privilegia o suporte a múltiplas formas de análise estatística, tanto no tipo de cálculo, quanto na facilidade de visualização de atributos e dimensões envolvidas.

A implementação do OLAP pode ser feita de quatro formas:

- **ROLAP** (*Relational Online Analytical Processing*) - é uma simulação da tecnologia OLAP, feita em banco de dados relacionais, não havendo restrições no volume de armazenamento de dados, e utilizando o modelo dimensional para gerenciar os dados. Apresenta as seguintes vantagens em relação às ferramentas MOLAP: tem tecnologia estabelecida; diversidade de plataformas, escalabilidade; permite gerenciar um banco de dados maior; tempo de resposta semelhante ao MOLAP para consultas complexas; já existem ferramentas com indexação avançada que facilitam e aumentam o desempenho; fácil de implementar para análises simples. Apresentam, no entanto, as seguintes desvantagens em relação às ferramentas MOLAP: desempenho menor; maior esforço para mudanças; maior esforço de processamento para obter as consultas multidimensionais; funcionalidade limitada para cálculos entre dimensões.
- **MOLAP** (*Multidimensional Online Analytical Processing*) - permite visões dos dados sobre várias perspectivas e representam o modelo dos negócios em três ou mais dimensões. Possibilitam uma visão multidimensional dos dados, conhecida por cubo. Possibilitam ainda a utilização por múltiplos usuários, sem apresentar queda de desempenho. As ferramentas MOLAP têm as seguintes vantagens em relação às ferramentas ROLAP: maior desempenho; maior facilidade na manutenção dos dados; menor espaço em disco e memória para armazenar os dados; facilidade na realização de análises multidimensionais complexas. Por outro lado, apresentam as seguintes desvantagens: sistemas proprietários; pouca flexibilidade; tecnologia em desenvolvimento; dificuldades com portabilidade e mudanças profundas; necessidade de manter *arrays* limitando a capacidade de memória; conseqüentemente, menor capacidade de armazenamento.
- **HOLAP** (*Hybrid Online Analytical Processing*) é um produto OLAP híbrido que pode prover análise multidimensional simultaneamente de dados armazenados em um banco de dados multidimensional e em um banco de dados relacional. As ferramentas HOLAP têm duas vertentes: as ferramentas MOLAP que estão incorporando os bancos de dados relacionais, devido à facilidade de armazenamento e maior número de interfaces com outros produtos, e as ferramentas ROLAP que estão incorporando os bancos de dados multidimensionais para oferecer aos seus clientes as vantagens das duas tecnologias.
- **WOLAP** (*Web Online Analytical Processing*) - são migrações da tecnologia OLAP para o ambiente da Internet. A interface utilizada é o *browser* (navegador), pois as análises geram documentos HTML. Ferramentas WOLAP apresentam como suas maiores vantagens: a interface é bem conhecida, conseqüentemente necessitando menor tempo de

aprendizado; a independência da plataforma; a facilidade do suporte a grupo de usuários distantes, facilitando a manutenção. Porém, apresentam algumas desvantagens: pouca confiabilidade, devido às limitações dos recursos da Internet e as interfaces e funcionalidade ainda estão em desenvolvimento e aperfeiçoamento.

Para o modelo dimensional, as análises da ferramenta OLAP são feitas sobre visões em hipercubos, permitindo assim as operações de rotacionamento e alteração da granularidade que são as operações mais importantes para as análises.

A ferramenta OLAP possibilita as seguintes operações sobre a base multidimensional [55]:

- *Roll-up* (aumento da granularidade) é a troca dos campos de uma mesma dimensão na visualização, da granularidade baixa para granularidade alta. Essa operação mostra uma visão mais genérica sobre os dados daquela dimensão. Usando o modelo multidimensional da figura 2.2 e uma visualização inicial que mostra média das valores de vendas pelas dimensões produto e região da venda, uma operação de aumento da granularidade na dimensão produto é mudar de nome do produto para categoria do produto.
- *Drill-down* (diminuição da granularidade) também troca os campos de uma mesma dimensão na visualização, no entanto, da granularidade alta para granularidade baixa. Essa operação mostra uma visão mais detalhada sobre os dados daquela dimensão. Usando o modelo multidimensional da figura 2.2 e uma visualização inicial que mostra média das valores de vendas pelas dimensões produto e região da venda, uma operação de diminuição da granularidade na dimensão região é mudar de estado para cidade.
- *Rotate* (rotacionamento) é a troca dos campos de dimensões diferentes na visualização. Essa operação mostra outras perspectivas de visualização sobre a base multidimensional. Usando o modelo multidimensional da figura 2.2 e uma visualização inicial que mostra média das valores de vendas pelas dimensões produto e região da venda, uma operação de rotacionamento é trocar a dimensão região pela dimensão tempo.

2.5 Conclusões do Capítulo

Neste capítulo foram apresentados os passos para o processo de análise de desempenho: a coleta, a transformação e a visualização dos dados. Na coleta foram apresentados os níveis mais comuns em que um sistema pode ser monitorado e as técnicas que são utilizadas na instrumentação. Na seção de transformação dos dados foram discutidas algumas métricas usadas para reduzir o volume de dados. E por fim, na seção de *Data Warehouse* e OLAP apresen-

tamos os passos a serem seguidos para a construção de um *Data Warehousing* as características da ferramenta OLAP.

Algumas ferramentas utilizadas para coletar, visualizar e analisar o desempenho em sistemas paralelos de passagem de mensagem são apresentadas no capítulo 3.

Para o desenvolvimento deste trabalho, foi utilizado o monitoramento por *software* em rotinas de bibliotecas, pela facilidade de instrumentação, por não precisar de um *hardware* dedicado, sendo portanto útil em qualquer máquina paralela, e pelo fato de que estamos interessados apenas nos dados em alto nível de abstração. A técnica de instrumentação usada foi a de *trace* por ser a mais detalhada e permitir registrar eventos de envio e recepção, registrar o tempo de execução gasto pelos processos. A seção 2.4 foi apresentanda aqui, pois para análise e visualização dos dados foi desenvolvida uma ferramenta OLAP usando uma base de dados relacional, ou seja, uma ferramenta ROLAP que é mostrada com mais detalhes no capítulo 5.

CAPÍTULO

3

Ferramentas

Neste capítulo serão apresentadas algumas ferramentas que estão disponíveis para coleta de dados, monitoramento e análise de programas paralelos [56, 57]. Com o uso das ferramentas descritas nas próximas seções, é possível, de forma mais fácil, entender o comportamento da aplicação paralela; analisar balanceamento de carga; analisar o desempenho de subrotinas ou blocos de códigos.

As ferramentas para análise de desempenho estão divididas em: *toolkits* que são apresentados na seção 3.1; a seção 3.2 apresenta as ferramentas utilizadas para fazer *profile*; a seção 3.3 apresenta as ferramentas utilizadas para fazer *trace* e na seção 3.4 são apresentadas as ferramentas utilizadas para a visualização dos dados e outros analisadores. As conclusões deste capítulo são apresentadas na seção 3.5.

3.1 Toolkits

Esta seção apresenta os *toolkits* usados para fazer *profiling* em aplicações paralelas. O *toolkit* TAU é apresentado na subseção 3.1.1. O AIMS é apresentado na subseção 3.1.2. A ferramenta HPCToolkit é apresentada na subseção 3.1.3. E a ferramenta PerfSuite é apresentada na subseção 3.1.4.

3.1.1 TAU

TAU (*Tuning and Analysis Utilities*) [58, 59, 60] é um *toolkit* de *profiling* e *tracing* portátil para análise de desempenho de programas paralelos escritos em C, C++, Fortran e Java. Todas as características da linguagem C++ são suportadas na instrumentação de TAU incluindo *templates* e *namespaces*.

Figura 3.1: Arquitetura de TAU [60]

A arquitetura de TAU pode ser vista na figura 3.1. O primeiro estágio apresentado na figura é o da instrumentação, que pode ser feita no nível do código fonte. A instrumentação pode ser feita automaticamente usando o instrumentador C++ `tau_instrumentor`, manualmente usando a API de instrumentação ou usando o `tau_run`, um instrumentador em tempo de execução.

TAU usa PDT (*Program Database Toolkit*) [61], que é uma ferramenta para análise estática e dinâmica de *software* orientado a objetos. O PDT é usado para gerar informações necessárias para instrumentar automaticamente o código fonte em C++.

O segundo estágio mostrado na figura 3.1, apresenta medidas obtidas em tempo de execução através da técnica de *profile* que mostra a distribuição do tempo de execução de cada rotina e as posições do código associadas com gargalos, além de apresentar um resumo estatístico de métricas de desempenho que caracterizam o comportamento da aplicação, e a técnica de *trace* de execução onde um programa paralelo mostra quando e onde ocorreu um evento, em termos dos processos que o executarão e a posição no código fonte.

A geração de *trace* de dados MPI pode ser feita usando a API de instrumentação TAU, inserindo chamadas para `TAU_TRACE_SENDMSG` e `TAU_TRACE_RECVMSG` no código fonte ou ligando a biblioteca MPI TAU que usa a interface de *profiling* MPI [57].

O terceiro e último estágio apresentado na figura 3.1 mostra como os dados de *trace* e *profile* gerados por TAU são visualizados. Esses dados podem ser visualizados utilizando as ferramentas de visualização de *trace* Paraver (seção 3.4.5) e Vampir (seção 3.4.7) e a ferramenta de visualização de *profiling*, paraprof [62], que fornece exibições gráficas de todos os resultados de análise de desempenho através de histogramas e exibições textuais.

Figura 3.2: Estrutura do AIMS [63]

3.1.2 AIMS

O toolkit AIMS (*Automated Instrumentation and Monitoring System*) [25, 59] fornece ferramentas de instrumentação para medidas e análise de desempenho de programas de passagem de mensagens em C e Fortran usando as bibliotecas de comunicação PVM (*Parallel Virtual Machine*) [15], MPI [16, 17], e uma variedade de mecanismos de visualização de *trace* para programas de passagem de mensagens.

A estrutura da ferramenta AIMS apresentada na figura 3.2, inclui [63, 64]:

- **Xinstrument:** É a ferramenta que adiciona rotinas de monitoramento na aplicação para coleta de dados de desempenho. O Xinstrument insere rotinas de monitoramento de desempenho na aplicação, considera o código fonte como uma coleção de construções, que são entidades lógicas de um programa, tais como condicionais, loops, subrotinas ou chamadas de comunicação. O Xinstrument permite aos usuários instrumentar apenas o código fonte de seu interesse, controlando assim a instrumentação.
- **Monitor:** fornece um conjunto de rotinas de monitoramento que mede e registra vários aspectos do desempenho do programa.
- **View Kernel (VK):** fornece animação gráfica do arquivo de *trace*. Apresenta informações tais como, subrotinas ativas, acessos a disco, mensagens enviadas entre nós e a rota das mensagens na rede.
- **Tally:** gera estatísticas de desempenho para o programa. Através dessas estatísticas é possível compreender o comportamento geral do programa. Tally gera uma lista das

Figura 3.3: Componentes do HPCToolkit [65]

estatísticas de utilização de recursos nó por nó e rotina por rotina. A rotina estatística apresenta informação fornecida por *profiles* a respeito da quantidade de tempo gasto em várias funções. Fornece fácil acesso para a porcentagem de tempo de execução gasto comunicando e o significado do tempo de comunicação em comparação com o tempo total de execução do programa.

- **Sysconfig:** revela a topologia de rede entre *workstations*.

A visualização dos *traces* resultantes pode ser feita usando AIMS View Kernel (VK), Para-Graph (seção 3.4.2) ou Pablo (seção 3.4.3).

3.1.3 HPCToolkit

HPCToolkit [59, 65] é um conjunto de ferramentas multi plataforma *open-source* para análise de desempenho baseada em *profile* de aplicações. A figura 3.3 mostra os componentes da ferramenta e seus relacionamentos.

Os componentes do *toolkit* incluem:

- **hpcrun:** uma ferramenta para coletar *profiles* por amostragem estatística de contadores de desempenho de *hardware*. Esta ferramenta usa a biblioteca PAPI [66] para acesso aos contadores de desempenho de hardware.
- **hpcprof e xprof:** ferramentas para mapear *profiles* coletados usando *hpcrun* e relatar nos *profiles* a linha do código fonte. É baseado no código do *profile* cprof/vprof (apresentado

na subseção 3.2.1).

- ***bloop***: uma ferramenta para análise binária de uma aplicação que recupera a estrutura do programa, isto é, identifica onde os *loops* são apresentados e em qual linha do código fonte.
- ***hpcview***: ferramenta usada para manipulação de dados de desempenho, métricas de computação, correlacionamento da informação da estrutura do programa, múltiplos *profiles* baseados em amostras, uma base de dados de desempenho produzida pelo código fonte do programa. A base de dados resultante é exibida por uma interface HTML.
- ***hpcviewer***: baseado em java e é usado para explorar a base de dados que consiste das informações de desempenho relacionadas com o programa fonte.

O HPCToolkit suporta as seguintes plataformas: Pentium+Linux, Opteron+Linux, Athlon+Linux, Itanium+Linux, Alpha+Tru64 and MIPS+Irix.

3.1.4 PerfSuite

PerfSuite [59, 67, 68] é uma coleção de ferramentas, utilidades, e bibliotecas para análise de desempenho de *software*, onde as principais metas são uso fácil, compreensão, interoperabilidade e simplicidade. Este *software* pode fornecer um bom ponto de entrada para análise de desempenho mais detalhada e pode apontar uma maneira para selecionar outras ferramentas e/ou técnicas usando *software* mais especializado se necessário (por exemplo, ferramentas/bibliotecas de grupos de pesquisas acadêmicos ou *software* comercial).

É composto pelos seguintes componentes:

- Linha de Comando
 - ***psinv***: fornece acesso para informações sobre as características de uma máquina (isto é, tipo e características do processador, informação de *cache*, contadores de desempenho disponível).
 - ***psprocess***: auxilia com um número de tarefas relacionadas ao pré e pós processamento de medidas de desempenho.
 - ***psrun***: uma ferramenta que realiza *counting* e *profiling* de aplicações baseadas em MPI e em *thread* para o desempenho de *hardware*. Pode também relatar informações sobre o uso do recurso da aplicação, tais como, o uso de memória, falha de página, entre outras. Não requer nenhuma mudança no código fonte ou recompilação da aplicação.

- Ferramentas Gráficas

- **OptView**: auxilia com interpretação eficaz da otimização do compilador na aplicação. A instrumentação pode ser feita utilizando essa ferramenta.
- **PerfExplore**: ferramenta para visualização dos dados do *counter* relacionados ao desempenho *hardware*.
- **ProfView**: ferramenta usada para visualização e exploração de dados de *profile* que suporta múltiplos modelos de apresentação e fontes de dados.
- **PSConfig**: uma ferramenta de fácil gerenciamento de arquivos de configuração do PerfSuite.

- Ferramentas Baseadas na Web

- **AmdahlCalc**: uma ferramenta que estima o *speedup* de programas paralelos.
- **PerfExchange**: uma infra-estrutura para coletar, manipular, armazenar e compartilhar os dados relacionados ao desempenho.

- Bibliotecas *Software/Pacotes Scripting*

- **libperfsuite**: uma biblioteca que fornece rotinas para coleta, informação sobre o uso de recursos e características da máquina, assim como diversas rotinas chamadas de C/C++ e Fortran.
- **libpshwpc**: uma biblioteca que fornece fácil acesso à informação de eventos de desempenho do *hardware* de aplicações baseadas em *threads*. Aplicações MPI utilizam a interface baseada em PMPI que requer somente uma relincagem e nenhuma mudança no código fonte.
- **libpspmi**: biblioteca de *profiling* MPI que fornece um resumo estatístico de alto nível das atividades de passagem de mensagem.
- **PSTcl**: uma interface de *scripting* baseado em Tcl para rotinas relacionadas ao PerfSuite.

3.2 Profiles

Esta seção apresenta as ferramentas usadas para fazer *profile* de programas paralelos. A ferramenta VProf é apresentada na subseção 3.2.1. A ferramenta Dynaprof é apresentada na subseção 3.2.2. A subseção 3.2.3 apresenta as ferramentas *prof* e *gprof*.

3.2.1 VProf

O VProf (*The Visual Profiler*) [69], é uma ferramenta para otimizar o desempenho de programas. Requer somente relincagem estática de uma aplicação que foi compilada com opção de otimização e a opção “-g”. A aplicação é então executada para coletar os dados de *profiling* que são escritos em arquivos *vmon.out*, um por processo. Estes arquivos são usados para gerar resumos de desempenho classificados pela linha do código fonte, pelo arquivo e pela função. Esta informação pode ser exibida de forma gráfica com o *vprof* ou pelo console com o *cprof*.

3.2.2 Dynaprof

Dynaprof [59, 70] é uma ferramenta de análise de desempenho projetada para inserir instrumentação de medidas de desempenho diretamente na aplicação em tempo de execução. Dynaprof usa DPCL ou DynInst para instrumentar na forma de instruções. Suporta as instruções implementadas em PAPI para coletar dados de contadores de *hardware* e instruções para medir o tempo gasto por processo ou por *thread*. Como a instrumentação ocorre em tempo de execução em arquivos executáveis binários, não existe a necessidade de recompilar as aplicações ou ligar com qualquer biblioteca.

Quando chamadas a subrotinas são adicionadas, o desempenho da aplicação pode mudar especialmente para regiões com computação intensiva. Um benefício adicional é a remoção das dependências na compilação do processo. O tipo e o formato da instrumentação pode ser mudado sem recompilar a aplicação.

Essa ferramenta foi projetada na Universidade do Tennessee pelo *Innovative Computing Lab*, e está disponível para AIX, IRIX, Solaris e Linux.

3.2.3 prof e gprof

prof e *gprof* [59] são duas ferramentas de *profiling* que estão disponíveis na maioria dos sistemas Unix, tais como, IBM AIX, Sun Solaris e HP/Compaq. É um simples processo que não requer modificação do código fonte. O usuário simplesmente compila o programa usando flags apropriados. Quando o programa está rodando, um processo monitor é iniciado automaticamente. *prof* e *gprof* podem ser usados com programas escritos em C/C++ e Fortran, e dependendo da implementação, estas ferramentas podem ser usadas com *thread* paralelos e programas MPI. Têm sido usados com sucesso com programas de média escala (com centenas de tarefas).

Gprof pode também ser usado com *gcov*, um analisador de código da GNU para dizer quais linhas foram executadas, quantas vezes e o por quanto tempo. Esse analisador é usado somente

com o compilador gcc.

3.3 Trace

Esta seção apresenta algumas ferramentas utilizadas para fazer *trace* em programas. A seção 3.3.1 apresenta a ferramenta MPITrace. A ferramenta SiGMA é apresentada na subseção 3.3.2.

3.3.1 Mpitrace

MpiTrace [59] é uma biblioteca de baixa sobrecarga que relata o tempo gasto dentro de cada rotina de comunicação MPI. Incorpora um método direto para converter estruturas de horas em segundos, o que é muito mais rápido do que usar outras rotinas de conversão disponível. Está disponível somente para sistemas AIX. Existe uma biblioteca mpihpm que inclui estes *trace* com os contadores *toolkit* HPM para coletar dados em arquiteturas Power-4.

3.3.2 SiGMA

SiGMA [71] é um *toolkit* para analisar gargalos e ineficiências devido à hierarquia de memória e para ajudar os programadores a entender as referências de memória em programas científicos que são causadas por má uso do subsistema de memória. Informações detalhadas, tais como, entender o comportamento de *cache* de novos algoritmos, investigar como diferentes partes de um programa competem e interagem dentro do subsistema de memória. Algumas das características chave em SiGMA são:

- Infraestrutura de análise de memória e uma família de ferramentas para entender o subsistema de memória.
- Um foco em estatísticas detalhadas para complementar os contadores de *hardware* existentes.
- Capacidade de tratar aplicações escritas em Fortran e C.
- Infra-estrutura para responder questões de estrutura de dados ou parâmetros arquiteturais que podem ser úteis para melhorar o desempenho de programas.

A abordagem tomada por SiGMA inclui os seguintes passos:

1. Instrumenta a aplicação automaticamente implementando uma ferramenta pré existente que localiza e instrumenta todas instruções que referem à posições de memória.

2. Executa a aplicação instrumentada para capturar informação sobre o uso de memória, produzindo um *trace* compacto.
3. Processa o *trace* compactado com simulação pós-execução e ferramentas de análise, fornecendo assim as informações aos programadores.

SiGMA é usado apenas em plataformas AIX e recentemente tornou-se parte do novo *toolkit* IBM HPC.

3.4 Visualizadores e Outros Analisadores

Esta seção apresenta algumas ferramentas usadas para a coleta, visualização e análise de programas paralelos. A ferramenta MPE é apresentada na subseção 3.4.1. A subseção 3.4.2 descreve a ferramenta Paragraph. A ferramenta Pablo é apresentada na subseção 3.4.3. As ferramentas Paraver e Paradyn são apresentadas nas subseções 3.4.4 e 3.4.5 respectivamente. As subseções 3.4.6 e 3.4.7 apresentam as ferramentas Upshot e Vampir.

3.4.1 MPE

A biblioteca MPE (*Multi-Processing Environment*) [72], foi desenvolvida para o uso com o MPICH mas pode ser usada com outras implementações MPI. É distribuída com a implementação MPICH e fornece um número de facilidades, incluindo depuração, *logging*, gráficos e algumas rotinas comuns. A biblioteca MPE fornece várias maneiras de gerar arquivos de *log* (*logfiles*) os quais podem ser visualizados com ferramentas gráficas. A maneira mais fácil de gerar arquivos de *log* é utilizar a interface de *profile* MPI com a biblioteca MPE.

O MPE tenta fornecer aos programadores um conjunto de ferramentas para a análise de desempenho. Essas ferramentas incluem um conjunto de bibliotecas de *profile* que fornecem uma coleção de rotinas para a criação de arquivos de *log*, sendo que esses arquivos podem ser criados manualmente inserindo chamadas MPE no programa MPI, automaticamente utilizando bibliotecas MPE apropriadas ou então, combinando os dois métodos; inclui também um conjunto de programas para conversão do formato e impressão do arquivo de *log*. A visualização dos arquivos de *log* pode ser feita através das ferramentas gráficas *Upshot* e *Jumpshot*, apresentadas na subseção 3.4.6.

O MPE fornece três formatos diferentes de arquivos de *log*, ALOG, CLOG e SLOG. O formato ALOG é visualizado pela ferramenta *Upshot*. Arquivos CLOG consistem de uma simples lista de eventos marcados por tempo que podem ser visualizados com a ferramenta *Jumpshot-2*.

O padrão SLOG armazena dados como estados, ou seja, eventos com uma certa duração. Esses arquivos podem ser visualizados com a ferramenta *Jumpshot-3*. Os estados em um arquivo SLOG são particionados em *frames* de dados, cada um sendo pequeno suficiente para ser processado eficientemente pela ferramenta de visualização. SLOG e *Jumpshot-3* são projetados para serem capazes de tratar arquivos de *log* contendo *gigabytes* de dados.

3.4.2 ParaGraph

ParaGraph [73], é uma ferramenta de exibição gráfica para visualizar o comportamento e desempenho de programas paralelos que usam MPI [16, 17]. As principais metas, segundo Heath [73], em projetar ParaGraph foram: facilidade de entendimento, facilidade de uso e portabilidade. Essa ferramenta foi originalmente desenvolvida para analisar *traces* gerados por uma biblioteca de passagem de mensagens chamada PICL (*Portable Instrumented Communication Library*) [74], mas pode em princípio ser usada para examinar qualquer *trace* compatível com o formato gerado pela biblioteca PICL. Como muitos sistemas de passagem de mensagens, PICL pode ser instrumentada para gerar *trace* de execução, sem intervenção do programador [6].

Uma vez que o arquivo de *trace* for especificado, a informação visual do programa paralelo pode ser feita através de exibições da utilização do processador e das comunicações envolvidas.

A exibição da utilização do processador permite ao usuário distinguir o tempo gasto em computação, comunicação e ocioso. Essas exibições podem ser usadas para identificar desbalanceamento de carga, comunicação excessiva em componentes do código e custos de tempo ocioso. Exibições de comunicação podem ser usadas para obter informações detalhadas de volume e comunicação, podendo estudar o relacionamento por exemplo, entre tempo ocioso e as comunicações realizadas.

Como ParaGraph opera em *traces* gerados automaticamente, ele pode ser usado com pequena intervenção do programador. Algumas exibições fornecidas por ParaGraph são [73]:

- Exibição por Utilização: a exibição por utilização está preocupada principalmente com a utilização do processador. Essa exibição é útil para determinar a eficácia com que os processadores são usados e o quão uniformemente as cargas de trabalho são distribuídas entre os processadores.
- Exibição por Comunicação: as exibições por comunicação estão preocupadas principalmente com a comunicação interprocessador. São úteis em determinar a frequência e volume da comunicação. O tráfego de comunicação mostra a comunicação total no sistema em função do tempo.

- Outras Exibições: a ferramenta Paragraph fornece ainda outras exibições como por exemplo, exibição por tarefa, por *trace*, resumo estatístico, estado do processador, caminho crítico, entre outras.

As exibições feitas por ParaGraph são particularmente intuitivas, embora a falta de *scroll* dentro das janelas de exibições possa ser frustrante [6].

3.4.3 Pablo

Pablo [6] foi projetado para organizar e visualizar informações coletadas de programas executados em processadores paralelos [27]. Fornece uma variedade de mecanismos para coletar, transformar e visualizar dados e o programador pode incorporar ainda novos formatos de dados, mecanismos de coleta de dados, módulos de redução de dados e amostragem. Pablo é um ambiente para análise de desempenho projetado para fornecer captura de dados, análise e apresentação. A figura 3.4 mostra um exemplo da interface da ferramenta Pablo.

Figura 3.4: Interface Pablo [27]

A instrumentação realizada em Pablo pode ser feita utilizando *traces*, *counters* e intervalos de tempo [40]. A biblioteca de registro Pablo (*Trace Library*) inclui uma biblioteca base

para gravar registros de eventos e extensões para registrar informação de desempenho sobre chamadas MPI, MPI I/O, e pedidos I/O. Todos os registros gerados pela biblioteca de registro *Trace Library* estão no formato SDDF (*Self Defining Data Format*). Os registros são escritos com informações de tempo e parâmetro das chamadas MPI no formato SDDF. Um arquivo separado SDDF é criado para cada processo MPI.

Pablo fornece uma variedade de módulos de amostragem e redução de dados que podem ser inseridos juntos para formar uma rede de visualização e análise de dados. Por exemplo, a maioria das exibições fornecidas por ParaGraph (descrito na subseção 3.4.2) pode ser construída usando módulos Pablo.

Um outro módulo do ambiente Pablo é o SvPablo, que é uma interface gráfica para instrumentar código fonte e explorar dados de desempenho em tempo de execução. Aplicações podem ser instrumentadas interativamente ou automaticamente. O SvPablo pode ser usado para instrumentar todas subrotinas e *loops* externos, incluindo chamadas MPI. Após a aplicação instrumentada ter sido executada, SvPablo relaciona os dados de desempenho em tempo de execução com o código fonte da aplicação e realiza análises estatísticas. SvPablo mostra uma representação gráfica que liga visualmente a informação de desempenho ao código fonte original [57].

3.4.4 Paradyn

O propósito do projeto Paradyn [42, 57] é fornecer uma ferramenta para medida de desempenho que escala para programas executando em grandes sistemas (milhares de nós) paralelos e distribuídos e que automatiza a busca por gargalos no desempenho. O projeto de Paradyn tenta evitar a sobrecarga de tempo e espaço tipicamente associada com ferramentas baseadas em *trace*. Sua abordagem é instrumentar dinamicamente a aplicação e controlar automaticamente a instrumentação na busca de problemas de desempenho [28].

Algumas características de Paradyn incluem: o usuário não tem que modificar o código ou usar um compilador especial; fornece visualizações através de histogramas de tempo, gráfico em barras e tabelas; suporte multi-plataforma (Solaris, Linux, Windows 2000 e XP, e AIX); suporte para passagem de mensagens MPI e PVM [75]; a capacidade para monitorar e exibir os dados de desempenho, e isolar problemas de desempenho para intervalos da execução do programa.

A instrumentação feita por Paradyn é inserida na execução da aplicação e a análise e a exibição do desempenho é feita em tempo real. Paradyn decide qual dado de desempenho coletar enquanto o programa está sendo executado. O usuário não tem que modificar o código

fonte da aplicação ou usar um compilador especial pois Paradyn instrumenta a imagem binária do programa usando a biblioteca de instrumentação dinâmica DyninstAPI [76].

A interface de visualização de Paradyn [77] fornece um mecanismo para adicionar facilmente novas visualizações ao sistema. Essa interface é flexível e tem sido usada para outras finalidades além de visualização, tais como prever desempenho para direcionar a aplicação ou registrar dados de desempenho. Um exemplo de visualização fornecida pela ferramenta Paradyn pode ser visto na figura 3.5 que apresenta um módulo de visualização que recebe seus dados através da interface *VisiLib* [78]. A tabela de visualização possui na primeira coluna o nome dos programas utilizados e nas demais colunas as métricas utilizadas. Os campos *active_processes*, *cpu*, *exec_time*, *procedure_calls* indicam as métricas, e os campos *operations*, *CPUs*, *CPUs*, *operation/sec* indicam as unidades das métricas definidas na primeira linha.

Table Visualization				
File	Actions	View	Phase: Global	
	active_processes	cpu	exec_time	procedure_calls
	operations	CPUs	CPUs	operations/sec
anneal.c	1	0	0.0034	58
bubba.c	1	0	0	0
channel.c	1	0	0	0
graph.c	1	0.15	0.11	6.9e+02
outchan.c	1	0	0	0
partition.c	1	0.19	0.2	2.5e+02
p_makeMG	1	0.19	0.19	20
Whole Program	1	0.39	1	1.2e+03

Figura 3.5: Tabela de Visualização Paradyn [28]

3.4.5 Paraver

Paraver [79, 80] é uma ferramenta de análise e visualização que pode ser usada para analisar programas MPI, OpenMP, Java, entre outros. Paraver fornece uma grande quantidade de informações sobre o comportamento da aplicação. Uma das principais características de Paraver é a flexibilidade para representar *traces* vindos de diferentes ambientes.

A instrumentação do código paralelo que usa o modelo de programação de passagem de mensagens MPI é feita utilizando a ferramenta MPITrace. A ferramenta OMPTrace é utilizada para instrumentar códigos paralelos que usam o modelo de programação OpenMP. Os códigos paralelos baseados no modelo de programação OpenMP e MPI são instrumentados utilizando a ferramenta OMPITrace.

A estrutura do Paraver pode ser vista na figura 3.6 e consiste de três níveis de módulos que são:

Módulo de Filtro é o primeiro módulo que trabalhará no arquivo de *trace*. O módulo de

Figura 3.6: Estrutura da ferramenta Paraver [79]

filtro passa ao próximo nível uma visão parcial do arquivo de *trace*. Este módulo filtra algumas comunicações e/ou alguns eventos que não passarão para os próximos módulos. O usuário pode especificar quais eventos serão passados ao módulo semântico, como o tipo de evento, *tag* das mensagens ou processo destino.

Módulo Semântico computa os valores que serão transferidos para o módulo de representação.

A computação dos valores é baseada em um ou vários registros retornados pelo módulo de filtro. O módulo semântico é o nível mais importante porque ele extrai e dá sentido aos valores no arquivo de *trace*. Os arquivos de *trace* têm muita informação que pode ser extraída e este módulo seleciona o que será extraído; esta informação é chamada de “semântica” do arquivo de *trace*.

Módulo de Representação recebe os valores computados pelo módulo semântico e os exibe de diferentes maneiras. Assim, o arquivo de *trace* de cada processo pode ser exibido de forma gráfica.

O módulo de representação é composto de três módulos:

Módulo de Visualização permite ao usuário mostrar os valores computados pelo módulo semântico na janela de exibição. Apresenta também a visualização gráfica do arquivo de *trace*. É responsável por gerenciar as janelas de exibição.

Módulo Textual dá uma representação textual do *trace* de registro em volta de pontos específicos.

Módulo Analisador analisa um subconjunto ou o arquivo *trace* completo. O módulo analisador realiza vários tipos de análise, como por exemplo, número de *bytes* en-

viados, número de operações *send*, número de operações *receive*, tamanho médio das mensagens, número de eventos, entre outras. O módulo analisador pode mostrar quatro análises das citadas anteriormente ao mesmo tempo, mas pode computar também os resultados de todas as outras funções. Pode ser computado o valor de média, variância, tamanho médio das mensagens, entre outras.

3.4.6 Upshot

Upshot [6, 81], é um pacote de visualização de *trace* para programas de passagem de mensagens. Pode ser usado para visualizar *traces* de uma variedade de sistemas de passagem de mensagens: em particular, eventos de *trace* que podem ser gerados automaticamente usando uma versão instrumentada de MPI ou MPE apresentado na subseção 3.4.1. A ferramenta *Upshot* pode visualizar os arquivos gerados pela biblioteca MPE no formato ALOG. Um exemplo de visualização utilizando a ferramenta *Upshot* é mostrado na figura 3.7.

Figura 3.7: Visualização em *Upshot* [6]

Uma versão mais nova da ferramenta *Upshot* é a ferramenta *Nupshot* [82] que é também uma ferramenta de visualização de desempenho que exibe o desempenho de arquivos de *trace* no formato *Clog* ou no formato da biblioteca PICL. Essa ferramenta possui dois tipos de exibição, uma que mostra os estados dos processos e passagens de mensagens, denominada *Timeline*, e uma segunda exibição que mostra histogramas de estatísticas para várias chamadas MPI, denominada de *Mountain Ranges*. A ferramenta *Nupshot* pode visualizar os arquivos com extensão

ALOG e CLOG. A figura 3.8 mostra a exibição da ferramenta *Nupshot*.

Figura 3.8: Exibição da ferramenta *Nupshot* [83]

A ferramenta *Jumpshot* [84] é uma versão da ferramenta *Upshot* em Java. *Jumpshot* é parte da implementação MPI MPICH, mas ela pode ser usada também com o MPI LAM. É uma ferramenta para exibição de arquivos de *log* que são gerados pela biblioteca MPE no formato SLOG e pode também visualizar os outros arquivos com as mesmas extensões que são utilizados pelas ferramentas *Upshot* e *Nupshot* discutidas acima. A figura 3.9 mostra a exibição de um arquivo de *log* (CLOG) visualizado pela ferramenta *Jumpshot*.

Figura 3.9: Exibição *Jumpshot* [72]

3.4.7 Vampir

A ferramenta Vampir (*Visualization and Analysis of MPI Programs*) [56, 57] é usada para análise e visualização de *trace* paralelo MPI disponível comercialmente. A interface da ferramenta Vampir pode ser vista na figura 3.10.

Figura 3.10: Interface da ferramenta Vampir [85]

A instrumentação é feita ligando a aplicação com a biblioteca VAMPIRtrace após adicionar chamadas VAMPIRtrace no código fonte. VAMPIRtrace registra todas as chamadas MPI, incluindo chamadas MPI I/O e rotinas de comunicação ponto-a-ponto e coletiva. A coleta de dados de *trace* pode ser feita dinamicamente em tempo de execução. A execução do código instrumentado produz um arquivo de *trace* que pode ser visualizado usando Vampir. Os arquivos de *trace* gerados pelos processos MPI quando o programa está executando são automaticamente coletados e incorporados em um único arquivo de *trace*.

Vampir fornece várias exibições para analisar graficamente características de desempenho e comportamento de uma aplicação paralela em tempo de execução. As várias exibições gráficas apresentam importantes aspectos do comportamento em tempo de execução da aplicação; visão

cronológica detalhada de eventos e comunicação; análise estatística da execução do programa e das operações de comunicação que podem ser calculadas para certos intervalos de tempo em processos específicos.

3.5 Conclusões do Capítulo

Neste capítulo foram apresentadas várias ferramentas que são utilizadas na coleta, transformação e visualização dos dados. A seção 3.1 apresenta *toolkits* como TAU, AIMS, HPCTollkit e Perf-Suite usados para *profiling* e *tracing* para análise de desempenho de programas paralelos. Na seção 3.2 são apresentadas as ferramentas VProf, Dynaprof, prof e gprof utilizadas para fazer apenas *profiling* de programas paralelos. Ferramentas usadas para fazer *tracing* de programas como Mpitrace, SiGMA são apresentadas na seção 3.3. E na seção 3.4 são apresentadas as ferramentas usadas para a coleta, visualização e análise de programas paralelos, como por exemplo, ParaGraph, Pablo, Paradyn, Paraver, Vampir, entre outras.

A ferramenta desenvolvida aqui neste trabalho difere das demais ferramentas apresentadas, pelo fato de que ela permite o levantamento de dados importante para análise de fatores de desempenho de programas paralelos de forma hierárquica, ou seja, permite coletar as informações acompanhando o nível de abstração usado pelo usuário em seu programa, como pode ser visto no capítulo 4. Com relação às ferramentas de visualização apresentadas aqui, foi desenvolvida uma ferramenta visual para auxiliar o usuário na análise de desempenho de programas paralelos que utiliza o processo de *Data Warehousing*, e mais detalhes dessa ferramenta são apresentados no capítulo 5.

CAPÍTULO

4

Ferramenta de Análise Hierárquica

*E*ste capítulo apresenta os objetivos e as características do trabalho proposto, além de mostrar detalhes de seu desenvolvimento. Este trabalho define uma ferramenta que permite o levantamento de dados importantes para análise de fatores que limitam o desempenho de programas paralelos. A ferramenta acompanha os vários níveis de abstrações, que podem ser representados por várias bibliotecas que foram utilizadas pelo usuário durante o desenvolvimento de seu programa paralelo. Dessa forma, a ferramenta permite que o usuário possa identificar gargalos ou fontes de desempenho ruim em sua aplicação em todos os níveis de bibliotecas utilizadas, ou seja, acompanhando a hierarquia das bibliotecas.

A ferramenta desenvolvida aqui é chamada de HieraAnalyses e sua arquitetura pode ser vista na figura 4.1. Para o desenvolvimento da ferramenta foi utilizado instrumentação por *software* em rotinas de biblioteca de forma estática, pela facilidade de instrumentação e por não precisar de um *hardware* dedicado, sendo útil em qualquer máquina paralela. A técnica escolhida para a coleta dos dados foi a de *trace*, pois dentre as outras técnicas apresentadas no capítulo 2, é a mais detalhada. A instrumentação foi feita nas rotinas da biblioteca MPI e do *framework* OOPS. Caso o usuário utilize outras bibliotecas em sua aplicação, as mesmas terão que ter seu código instrumentado.

A ferramenta HieraAnalyses é composta dos módulos de coleta (*HieraCollector*, descrito na seção 4.1) e do módulo de transformação dos dados (*HieraTransform*, descrito na seção 4.2). Para auxiliar na análise dos programas foi desenvolvida uma ferramenta visual para análise dos dados obtidos pelo módulo de coleta que é descrita no capítulo 5.

O módulo HieraCollector é responsável por coletar e armazenar os dados no nível de abstração utilizado pelo usuário, esses dados são armazenados em arquivos que são estruturados no formato XML (*Extensible Markup Language*). O módulo HieraTransform define formas de representação dos dados coletados e formas de se realizar medidas. A representação define

Figura 4.1: Arquitetura da biblioteca HieraAnalyses

classes que manipulam um grafo dos dados coletados durante a execução do programa, dos diversos níveis de abstração. Para a análise dos dados são definidas outras classes que realizam medidas sobre esses dados.

A seção 4.1 apresenta o módulo de coleta dos dados. Esta seção mostra como é feita a coleta usando o MPI (*Message Passing Interface*) [16, 17] e o *framework* OOPS (*Object Oriented Parallel System*) [21]. Os dados gerados pela coleta são usados pelo módulo de transformação que está descrito na seção 4.2.

4.1 Módulo de Coleta dos Dados

Como foi visto no capítulo 2 (na seção 2.1), a coleta de dados é a primeira etapa dentro do processo de análise de desempenho.

A coleta de dados em sistemas paralelos é mais complexa do que a de sistemas sequenciais, já que manipula uma quantidade maior de dados. Essa quantidade maior de dados é proveniente de cada processo em execução (equivalentes aos dados gerados em um programa sequencial) e à existência de uma nova dimensão a ser tratada: a comunicação entre os processos e suas sincronizações relativas.

Neste trabalho o módulo responsável pela coleta de dados é chamado de HieraCollector (*Hierarchical Collector*). Sua implementação foi feita utilizando a linguagem de programação C [86]. A técnica escolhida para a coleta dos dados foi a de *trace*, pois dentre as outras técnicas

apresentadas no capítulo 2, é a mais detalhada. Os dados a serem coletados são referentes as operações do MPI e aos métodos do OOPS, uma vez que apenas as rotinas das duas bibliotecas foram instrumentadas.

O usuário que deseja coletar os dados de um nível de abstração em seu sistema paralelo usando o módulo HieraCollector, não terá que alterar o código fonte de sua aplicação, desde que o desenvolvedor das bibliotecas utilizadas tenha preparado seu código adequadamente.

Os dados coletados de todos os níveis de abstração são armazenados em arquivos estruturados no formato XML (*Extensible Markup Language*) [87]. XML é uma linguagem de marcação onde os *tags* são utilizados para estruturar a informação independente do aplicativo utilizado. Essa estrutura reflete uma organização lógica hierárquica que é representada através de uma estrutura de dados de uma árvore. Dessa forma, a organização hierárquica possibilita que a análise das informações coletadas no módulo HieraCollector possa ser feita em diversos níveis.

Para a manipulação de arquivos XML existem duas APIs (*Application Programming Interface*): SAX (*Simple API for XML*) e DOM (*Document Object Model*). O módulo de coleta HieraCollector usa DOM que é API padrão definida pelo W3C [87]. O DOM define uma forma de acesso e manipulação à árvore de objetos correspondente ao documento XML. Suporta manipulação da estrutura e conteúdo do documento, permitindo a criação de um novo documento XML, a navegação na árvore XML, modificar, remover ou adicionar nós (elementos, atributos, textos, comentários).

Existem várias implementações da API DOM para diversas linguagens e plataformas, uma delas é a biblioteca LibXML [88]. Essa biblioteca será usada por este módulo. A biblioteca LibXML é um *parser* para a linguagem XML implementado na linguagem C pela GNOME Foundation. É uma biblioteca de código livre e portátil para um grande número de plataformas.

O documento XML pode estar associado a uma gramática que rege sua estrutura e composição. Existem várias formas para validar a gramática de um documento XML. O módulo de coleta HieraCollector usa o DTD (*Document Type Definition*) [89].

O DTD corresponde às marcações declarativas que especificam a seqüência e aninhamento de elementos; valores e tipos de atributos exigidos ou permitidos; nomes de entidades internas e externas necessárias e das anotações utilizadas. O DTD pode ser interno (quando é definido no próprio documento XML) ou externo (quando é definido em arquivo separado do documento XML).

O módulo HieraCollector é implementado para cada nível de abstração, seja para o nível do MPI ou o nível do OOPS. A subseção 4.1.1 descreve como o HieraCollector realiza a coleta de dados no nível do MPI e para exemplificar o uso do módulo de coleta, é apresentado como exemplo o programa do passeio do cavalo. A subseção 4.1.2 descreve como o

HieraCollector realiza a coleta no nível de orientação a objetos mostrando a hierarquia existente nos dados coletados e para exemplificar seu uso, foi utilizado um programa que resolve o problema de Poisson [90] usando o método Jacobi.

4.1.1 HieraCollector para MPI

O módulo de coleta HieraCollector para o MPI tem por objetivo armazenar os dados coletados das operações realizadas no MPI.

A escolha pelo uso da biblioteca MPI (*Message Passing Interface*) é pelo fato de que o MPI é amplamente usado em certas classes de máquinas paralelas, especialmente aquelas com memória distribuída. As vantagens do uso de MPI são [16]: portabilidade; permite comunicação eficiente; permite que implementações possam ser usadas em um ambiente heterogêneo; permite o uso das linguagens C, C++ e Fortran; define uma interface que não é tão diferente de PVM [15]; fornece extensões que permitem maior flexibilidade.

Para a geração dos arquivos de coleta de dados no nível do MPI foi necessária a alteração no arquivo *header* da biblioteca MPI (*mpi.h*). As alterações foram feitas com a utilização de diretivas de pré-processamento [91] inseridas no arquivo *mpi.h* permitindo controlar o que será apresentado ao compilador.

Uma vez que o usuário deseje coletar os dados de seu programa no nível do MPI, basta adicionar a diretiva de pré-processamento `-D _HIERATRACEMPI` em seu *makefile*, ou utilizar um *script* que foi desenvolvido junto com a biblioteca *HieraAnalyses*. A execução é realizada como a de um programa MPI normal e essas modificações garantem que o usuário não precise fazer nenhuma alteração no seu código fonte.

Pelas alterações feitas é possível coletar o nome do arquivo que executou a operação e a linha que essa operação aparece no código fonte da aplicação durante a compilação do código. Uma parte das alterações ocorridas no arquivo *mpi.h* pode ser vista no quadro abaixo.

```

1 #ifndef _HIERATRACEMPI
2 extern int MPI_Barrier __ARGS((MPI_Comm));
3 extern int MPI_Bcast __ARGS((void *, int, MPI_Datatype, int, MPI_Comm));
4 extern int MPI_Recv __ARGS((void *, int, MPI_Datatype, int, int, MPI_Comm,
5     MPI_Status *));
6 extern int MPI_Send __ARGS((void *, int, MPI_Datatype, int, int, MPI_Comm));
7 ...
8 #else
9 #include <trace.h>
10 #define MPI_Barrier(com) HC_Barrier(__FILE__, __LINE__, com)
11 #define MPI_Bcast(buffer, cont, type, root, com) HC_Bcast(__FILE__, __LINE__,
12     buffer, cont, type, root, com)
13 #define MPI_Recv(buffer, cont, type, rem, tag, com, status) HC_Recv(__FILE__,

```

```

14  __LINE__,buffer,cont,type,rem,tag,com,status)
15  #define MPI_Send(buffer,cont,type,dest,tag,com) HC_Send(__FILE__,__LINE__,
16  buffer,cont,type,dest,tag,com)

```

Uma vez que o usuário tenha adicionado a diretiva de processamento `HIERATRACEMPI` na compilação de seu programa, quando o compilador encontrar uma chamada `MPI_Send` no programa, automaticamente é chamado a operação `HC_Send` definida no arquivo *header trace.h* como mostrado na linha 9 do quadro acima. As operações definidas no arquivo *trace.h* são as mesmas operações definidas na biblioteca do MPI, com a diferença que todas permitem a realização da coleta dos dados através do uso de funções da biblioteca LibXML.

Durante a execução do programa são gerados: um arquivo de configuração e vários arquivos de coleta de dados. O arquivo de configuração contém informações de configuração sobre os arquivos de coleta. Cada arquivo de coleta contém informações sobre um processador. Todos os arquivos são gerados no formato XML e esses serão usados pelo módulo *HieraTransform*, sendo a gramática desses arquivos validada por DTD.

O DTD para o arquivo de configuração pode ser visto no quadro abaixo. Um documento XML válido para o DTD do quadro abaixo deve ter um elemento raiz chamado *hieraCollector* com informações sobre o nome da aplicação que foi executada e o número de processos do programa MPI. O elemento raiz pode ter *n* subelementos *collect_file* (*n* é igual ao número de processos) que descrevem o nome do arquivo de coleta de dados e o identificador do processador.

```

1  <!ELEMENT hieraCollector (collect_file)*>
2  <!ATTLIST hieraCollector
3    application  CDATA #REQUIRED
4    count_proc   CDATA "0">
5  <!ELEMENT collect_file EMPTY>
6  <!ATTLIST collect_file
7    rank ID #REQUIRED>

```

Uma parte do DTD para os arquivos de coleta, tanto para o MPI quanto para o OOPS, é mostrada no quadro abaixo. O elemento raiz *processor* desse DTD possui informações sobre o identificador, o tempo inicial e final de execução do processador. O elemento raiz pode ter zero ou mais elementos do tipo *hieraMPI* ou *hieraOOPS*. Esses elementos podem ser operações ponto-a-ponto, coletivas e grupos para o *hieraMPI* sendo que as informações de cada elemento dependem da operação realizada, ou podem ser classes e métodos para o elemento *hieraOOPS*. O elemento *hieraMPI* é uma lista (como mostra o quadro abaixo nas linhas 7 a 27) contendo informações como o nome da operação, nome do arquivo, a linha onde a operação foi

executada, os tempos inicial e final e vários outros campos dependendo da operação realizada. O restante das linhas mostra a parte do DTD usada quando a coleta é realizada com o OOPS e será explicada na subseção 4.1.2.

```

1 <!ELEMENT processor(hieraMPI | hieraOOPS)*>
2 <!ATTLIST processor
3   rank      ID      #REQUIRED
4   init      CDATA #REQUIRED
5   finalize  CDATA #REQUIRED >
6 <!ELEMENT hieraMPI EMPTY>
7 <!ATTLIST hieraMPI
8   operation (address|allgather|allgatherv|allreduce|alltoall|alltoallv|barrier|
9     broadcast|bsend|bsend_init|buffer_attach|buffer_detach|cancel|comm_create|
10    comm_dup|comm_split|gather|gatherv|get_count|get_elements|ibsend|
11    intercomm_create|intercomm_merge|iprobe|irecv|irsend|isend|issend|pack|
12    pack_size|probe|receive|recv_init|reduce|reduce_scatter|request_free|rsend|
13    rsend_init|scan|scatter|scatterv|send|send_init|sendrecv|sendrecv_replace|
14    ssend|ssend_init|start|startall|test|testall|testany|test_cancelled|
15    testsome|type_commit|type_contiguous|type_extent|type_free|type_hindexed|
16    type_hvector|type_indexed|type_lb|type_size|type_struct|type_ub|
17    type_vector|unpack|wait|waitall|waitany|waitsome) #REQUIRED
18   file      CDATA #REQUIRED
19   line      CDATA #REQUIRED
20   start_time CDATA #REQUIRED
21   finish_time CDATA #REQUIRED
22   count     CDATA #IMPLIED
23   type      CDATA #IMPLIED
24   dest      CDATA #IMPLIED
25   tag       CDATA #IMPLIED
26   com       CDATA #IMPLIED
27   ...
28 >
29 <!ELEMENT hieraOOPS (hieraOOPS | hieraMPI | EMPTY)*>
30 <!ATTLIST hieraOOPS
31   class (unknown|distributionBlocked|distributionCyclic|distributionNone|matrix|
32     vector|vectorRepl|vectorSequ|group|partner|topology|topologyGrid|
33     topologyLinear|topologyPlain|topologyPipe|topologyTorus|
34     workgroup) #REQUIRED
35   method (allreduce|barrier|bcast|col|distributionBlocked|distributionCyclic|
36     distributionNone|fromEast|fromNext|fromNorth|fromSouth|fromWest|fromNE|
37     fromNW|fromSE|fromSW|gather|gatherv|globalToLocal|isInGroup|localSize|
38     localToGlobal|matrix|max|min|partner|prod|recv|reduce|row|scatter|scatterv|
39     send|split|store|subGroup|sum|syncGhostsCart|topologyGrid|topologyPipe|
40     toEast|toNorth|toPrevious|toSouth|toWest|toNE|toNW|toSE|toSW|vector|
41     vectorRepl|vectorSequ) #REQUIRED
42   file      CDATA #REQUIRED
43   line      CDATA #REQUIRED
44   start_time CDATA #REQUIRED
45   finish_time CDATA #REQUIRED
46   ...

```

47 >

As operações ponto-a-ponto e coletivas definidas para a coleta de dados foram: `address`, `allgather`, `allgatherv`, `allreduce`, `alltoall`, `alltoallv`, `barrier`, `bcast`, `bsend`, `bsend_init`, `buffer_attach`, `buffer_detach`, `cancel`, `comm_create`, `comm_dup`, `comm_split`, `gather`, `gatherv`, `get_count`, `getElements`, `intercomm_create`, `intercomm_merge`, `iprobe`, `ibsend`, `irecv`, `irsend`, `isend`, `issend`, `pack`, `pack_size`, `probe`, `recv`, `recv_init`, `reduce`, `reduce_scatter`, `request_free`, `rsend`, `rsend_init`, `scan`, `scatter`, `scatterv`, `send`, `send_init`, `sendrecv`, `sendrecv_replace`, `ssend`, `ssend_init`, `start`, `startall`, `test`, `test_cancelled`, `testall`, `testany`, `testsome`, `type_commit`, `type_extent`, `type_free`, `type_hindexed`, `type_hvector`, `type_indexed`, `type_lb`, `type_size`, `type_struct`, `type_ub`, `type_vector`, `unpack`, `wait`, `waitall`, `waitany`, `waitsome`.

No caso do MPI é gerado apenas um nível na hierarquia, ou seja, não existe nenhuma dependência entre os níveis. Para aplicações que possuem mais de um nível na hierarquia, é preciso definir um novo DTD que corresponda às regras utilizadas por esses outros níveis, e novos arquivos XML serão gerados contendo informações referentes de todo o nível hierárquico.

Exemplo usando HieraCollector para MPI

Para exemplificar o uso do módulo HieraCollector para o MPI, utilizaremos um problema clássico do xadrez: encontrar o número de soluções do problema do passeio do cavalo em um tabuleiro de tamanho arbitrário. O problema do passeio do cavalo consiste em encontrar todas as soluções de o cavalo percorrer todo o tabuleiro de xadrez e visitar cada casa apenas uma vez, partindo de cada casa do tabuleiro. Os movimentos possíveis do cavalo são os permitidos pelas regras usuais do xadrez: na forma de “L”, duas casas em uma direção e uma na direção perpendicular as das outras casas.

A aplicação do passeio do cavalo utiliza o método mestre/escravo para sua implementação. Nesse problema há um processo mestre que distribui as casas iniciais entre os diversos processos escravos e fica verificando qual processo escravo terminou e envia mais casas até que todos tenham terminado. Os processos escravos são responsáveis por realizar a procura de soluções, dadas as casas iniciais recebidas pelo processo mestre e por retornar a quantidade de soluções encontradas. A aplicação termina quando o último resultado for recebido pelo processo mestre.

O programa foi executado utilizando 4 processos e um tabuleiro de tamanho 5×5 . O arquivo de configuração XML gerado pode ser visto no quadro abaixo que tem `hieraCollector` como elemento raiz, o nome da aplicação indicado por `application` e o número de processos `count_proc`. O elemento `collect_file` possui os dados que são referentes a cada processo em execução. Por exemplo, a linha 4 mostra que o processo 0 (`rank=P0`) irá armazenar os seus

dados coletados no arquivo chamado `traceMPI0.xml`. Os demais elementos `collect_file` são referentes aos demais processos em execução.

```

1 <?xml version="1.0"?>
2 <!DOCTYPE hieraCollector SYSTEM "hieraCollector.dtd">
3 <hieraCollector application="cavalo" count_proc="4">
4   <collect_file rank="P0">traceMPI0.xml</collect_file>
5   <collect_file rank="P1">traceMPI1.xml</collect_file>
6   <collect_file rank="P2">traceMPI2.xml</collect_file>
7   <collect_file rank="P3">traceMPI3.xml</collect_file>
8 </hieraCollector>

```

Um exemplo do arquivo contendo alguns dos dados coletados referentes às operações realizadas pelo processo 0 durante a execução do programa do passeio do cavalo, pode ser visto no quadro abaixo. O arquivo mostra o elemento raiz `processor` com o `rank=P0`, o tempo inicial e o tempo final da execução do processo 0. O elemento raiz `processor` possui ainda uma lista de elementos `hieraMPI` que são as operações que foram realizadas pelo processo 0, tais como `send` e `receive` (linha 5 a 21), e os respectivos dados coletados. Os vários subelementos `hieraMPI` mostrados no quadro são escritos nos arquivos de coleta com o uso das funções `xmlNewTextChild` que cria um novo elemento ou subelemento (`hieraMPI`) e `xmlNewProp` que cria uma propriedade de um nó (`operation`, `file`, `line`, entre outros).

```

1 <?xml version="1.0"?>
2 <!DOCTYPE processor SYSTEM "trace.dtd">
3 <?xml-stylesheet type="text/xsl" href="visual.xsl"?>
4 <processor rank="P0" init="0.05057" finalize="19.60675">
5   <hieraMPI operation="send" file="cavalo.cc" line="253" start_time="0.05130" finish_time="0.05131"
6     count="1" type="MPI_INT" dest="1" tag="2" comm="MPI_COMM_WORLD"/>
7   <hieraMPI operation="send" file="cavalo.cc" line="253" start_time="0.05135" finish_time="0.05136"
8     count="1" type="MPI_INT" dest="2" tag="2" comm="MPI_COMM_WORLD"/>
9   <hieraMPI operation="send" file="cavalo.cc" line="253" start_time="0.05139" finish_time="0.05139"
10    count="1" type="MPI_INT" dest="3" tag="2" comm="MPI_COMM_WORLD"/>
11  <hieraMPI operation="send" file="cavalo.cc" line="253" start_time="0.05143" finish_time="0.05143"
12    count="1" type="MPI_INT" dest="4" tag="2" comm="MPI_COMM_WORLD"/>
13  <hieraMPI operation="receive" file="cavalo.cc" line="278" start_time="0.05173" finish_time="0.05175"
14    count="1" type="MPI_INT" rem="2" tag="0" comm="MPI_COMM_WORLD"/>
15  <hieraMPI operation="send" file="cavalo.cc" line="279" start_time="0.05179" finish_time="0.05179"
16    count="2" type="MPI_INT" dest="2" tag="1" comm="MPI_COMM_WORLD"/>
17  <hieraMPI operation="receive" file="cavalo.cc" line="278" start_time="0.05182" finish_time="0.05183"
18    count="1" type="MPI_INT" rem="3" tag="0" comm="MPI_COMM_WORLD"/>
19  <hieraMPI operation="send" file="cavalo.cc" line="279" start_time="0.05186" finish_time="0.05186"
20    count="2" type="MPI_INT" dest="3" tag="1" comm="MPI_COMM_WORLD"/>
21  <hieraMPI operation="receive" file="cavalo.cc" line="278" start_time="0.05189" finish_time="0.05340"
22    count="1" type="MPI_INT" rem="1" tag="0" comm="MPI_COMM_WORLD"/>
23  ...
24 </processor>

```

Outros programas MPI foram usados para testar o módulo de coleta e seus dados serão usados pelo módulo de transformação para sua análise.

4.1.2 HieraCollector para OOPS

O módulo de coleta HieraCollector para o OOPS (*Object Oriented Parallel System*) [21, 92] tem por objetivo armazenar os dados coletados referentes às classes e métodos executados pela aplicação OOPS de forma hierárquica. O *framework* foi usado aqui neste trabalho por fornecer abstrações de alto nível, o que permitirá coletar e analisar os dados no nível de orientação a objetos mostrando a hierarquia entre os dados coletados.

O *framework* OOPS é uma ferramenta que usa orientação a objetos para suportar a programação de aplicações científicas para execução paralela. Fornece abstrações de alto nível para evitar que o programador se envolva com detalhes de implementação. Para considerações de desempenho, alguns aspectos paralelos, tais como, decomposição e distribuição dos dados não são completamente ocultados do programador da aplicação. Para isso, o OOPS encapsula algumas técnicas de programação freqüentemente usadas para sistemas paralelos.

Os elementos da execução paralela no OOPS são chamados de processadores virtuais e podem ser associados com processos MPI executando em uma máquina paralela. Processadores virtuais são organizados em grupos, e um grupo de processadores pode ser usado para construir uma topologia. A topologia fornece a comunicação (que pode ser ponto-a-ponto ou coletiva) através de passagem de mensagem entre os processadores virtuais de um grupo, definindo assim, a relação de vizinhos entre eles.

Figura 4.2: Exemplos de Topologias com seus vizinhos e a estrutura usada para a comunicação ponto-a-ponto [92]

O ponto de entrada de um programa OOPS é a classe Main que inicializa e finaliza procedimentos do sistema paralelo. As comunicações das topologias OOPS podem enviar elementos de qualquer tipo básico. Topologias para estruturas de comunicação uni e bi-dimensional (mostradas na figura 4.2) são: TopologyLinear onde um *array* linear de processadores virtuais

comunica cada um com o anterior e o próximo; *TopologyRing* é similar a um anel, mas com conexões entre o primeiro e o último processador; e *TopologyGrid* onde um *array* bi-dimensional com conexões entre vizinhos nas direções diagonal e cartesiana. Os elementos de um *container* são distribuídos conforme a distribuição especificada durante a sua criação por classes derivadas de *Distribution*. As distribuições fornecidas são: *block*, *cyclic* e *block-cyclic*. A figura 4.3 mostra as diferentes distribuições aplicadas a matrizes, cuja cor dos elementos são as mesmas dos processadores virtuais da topologia. A especificação de uma boa topologia e distribuição é fundamental para alcançar localidade e balanceamentos de carga nas aplicações.

Figura 4.3: Diferentes distribuições em quatro processadores virtuais [92]

A diferença do módulo de coleta *HieraCollector* do MPI para o módulo de coleta do OOPS é que aqui os dados são armazenados de forma hierárquica, pois os métodos do OOPS possuem dependência entre si. Não é preciso nenhuma alteração no código fonte do usuário para coletar os dados, basta compilar o código com a opção `-D _HIERATRACEOOPS`. A execução é feita como a de um programa MPI normal.

A parte do DTD para os arquivos de coleta do OOPS é mostrada no quadro abaixo. Como mostrado no quadro na subseção 4.1.1, o elemento raiz *processor* possui informações sobre o identificador, o tempo inicial e final de execução do processador. O elemento raiz pode ter zero ou mais elementos que podem ser do tipo *hieraMPI* e *hieraOOPS*. O elemento *hieraOOPS* pode conter subelementos do tipo *hieraOOPS*, *hieraMPI* ou pode estar vazio, ou seja, não conter nenhum subelemento. Os subelementos do tipo *hieraOOPS*, mostrados no quadro abaixo, podem conter informações sobre o nome da classe, o nome do método de uma determinada classe, o nome do arquivo, o número da linha onde o método foi executado, os tempos inicial e final, e várias outras informações que dependem do tipo de método que foi executado, como mostrado nas linhas 11 a 24. As informações contidas nos subelementos *hieraMPI* são as mesmas citadas na subseção 4.1.1.

1 <!ELEMENT processor(hieraMPI | hieraOOPS)*>
 2 <!ATTLIST processor

```

3   rank      ID      #REQUIRED
4   init      CDATA #REQUIRED
5   finalize  CDATA #REQUIRED >
6 <!ELEMENT hieraMPI EMPTY>
7 <!ATTLIST hieraMPI
8   ...
9 >
10 <!ELEMENT hieraOOPS (hieraOOPS | hieraMPI | EMPTY)*>
11 <!ATTLIST hieraOOPS
12   class (unknown|distributionBlocked|distributionCyclic|distributionNone|
13     matrix|vector|vectorRepl|vectorSequ|group|partner|topology|topologyGrid|
14     topologyLinear|topologyPlain|topologyPipe|topologyTorus|
15     workgroup) #REQUIRED
16   method (allreduce|barrier|bcast|col|distributionBlocked|distributionCyclic|
17     distributionNone|fromEast|fromNext|fromNorth|fromSouth|fromWest|fromNE|
18     fromNW|fromSE|fromSW|gather|gatherv|globalToLocal|isInGroup|localSize|
19     localToGlobal|matrix|max|min|partner|prod|recv|reduce|row|scatter|scatterv|
20     send|split|store|subGroup|sum|syncGhostsCart|topologyGrid|topologyPipe|
21     toEast|toNorth|toPrevious|toSouth|toWest|toNE|toNW|toSE|toSW|vector|
22     vectorRepl|vectorSequ) #REQUIRED
23   file      CDATA #REQUIRED
24   line      CDATA #REQUIRED
25   start_time CDATA #REQUIRED
26   finish_time CDATA #REQUIRED
27   ...
28 >

```

Durante a execução do programa são gerados: um arquivo de configuração e vários arquivos de coleta de dados, do mesmo modo como é feito no MPI. O arquivo de configuração contém as informações sobre os arquivos de coleta. Cada arquivo de coleta contém as informações sobre um processador. Todos os arquivos serão usados pelo módulo HieraTransform e estão no formato XML. O arquivo de configuração e os arquivos de coleta podem ser vistos no exemplo mostrado abaixo, com a diferença apenas no nome dos arquivos referentes a cada processo em execução.

As classes e seus respectivos métodos definidos para a coleta de dados foram: DistributionBlocked com os métodos distributionBlocked, globalToLocal, localSize, localToGlobal; DistributionCyclic com os métodos distributionCyclic, globalToLocal, localSize, localToGlobal; DistributionNone com os métodos distributionNone, globalToLocal, localSize, localToGlobal; Matrix com os métodos matrix, localToGlobal, syncGhosts, syncGhostsCart, load, store; Vector com os métodos vector, localSize, localToGlobal, sum, syncGhosts, product_elementToElement, product_scalar, multiply, load, store; VectorRepl com os métodos sum, load; VectorSequ com os métodos vectorSequ, realloc; Group com os métodos barrier, bcast, gather, isInGroup, max, min, prod, recv, scatter, send, split, sum; Partner com os métodos partner, send, recv; Topology

com os métodos `send`, `recv`, `split`, `barrier`, `bcast`, `max`, `min`, `sum`, `prod`, `gather`, `scan`, `scatter`; `TopologyGrid` com os métodos `topologyGrid`, `toNorth`, `fromNorth`, `toSouth`, `fromSouth`, `toEast`, `fromEast`, `toWest`, `fromWest`, `toNW`, `fromNW`, `toNe`, `fromNE`, `toSW`, `fromSW`, `toSE`, `fromSE`; `TopologyLinear` com os métodos `toPrevious`, `fromNext`; `TopologyPlain` com os métodos `send`, `recv`; `TopologyPipeRing` com o método `topologyPipeRing`; `TopologyPipe` com os métodos `topologyPipe`, `toNext`, `fromPrevious`; `TopologyTorus` com o método `topologyTorus`; `Workgroup` com o método `bcast`.

Exemplo usando o HieraCollector para OOPS

Várias soluções científicas apresentam soluções tipo estêncil, por exemplo as que tratam de equações diferenciais parciais. Para exemplificar o uso do módulo de coleta com o OOPS, foi usada uma solução para resolver o problema de Poisson [90]. No caso bidimensional, têm-se as seguintes equações para o problema de Poisson:

$$\begin{aligned} \frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} &= f(x, y) && \text{no interior} \\ u(x, y) &= g(x, y) && \text{no contorno} \end{aligned} \quad (4.1)$$

Para a solução numérica, é usada uma matriz para representar $u(x, y)$ como uma malha discretizada representada por $u_{i,j}$ e o método Jacobi que é um algoritmo iterativo simples para resolver este problema é usado. Seguindo as aproximações de [90] para a equação 4.1, tem-se a seguinte equação para o Jacobi:

$$u_{i,j}^{t+1} = \frac{1}{4} \left(u_{i-1,j}^t + u_{i,j-1}^t + u_{i+1,j}^t + u_{i,j+1}^t \right) \quad (4.2)$$

onde t é o número da iteração, e i e j indicam a posição do elemento na matriz.

Com a equação 4.2 o novo valor para um elemento é calculado em função do valor de seus elementos vizinhos no passo anterior e o cálculo é repetido iterativamente até uma condição de término ser satisfeita.

Um esquema da equação 4.2 pode ser visto na Figura 4.1.2(a) que mostra um diagrama de entêncil que utiliza os valores dos seus vizinhos para calcular seu novo valor. Como o conjunto de vizinhos requisitados é fixo e com acesso regular, a Figura 4.1.2(b) mostra um esquema da dependência dos dados para os cálculos em um processador. A Figura 4.1.2(c) mostra os elementos de um processador adicionado de uma borda para armazenamento dos dados de seus vizinhos, e essa borda é denominada de sombra ou fantasma.

Figura 4.4: (a) Diagrama de um estêncil. (b) Esquema da dependência dos dados. (c) Elementos de um processador [92]

O programa utilizando o método Jacobi foi executado com 4 processos e o arquivo de configuração XML gerado é mostrado no quadro abaixo que tem `hieraCollector` como elemento raiz, o nome da aplicação indicado por `application` e o número de processos `count_proc`. O elemento `collect_file` possui os dados que são referentes a cada processo em execução. Por exemplo, a linha 3 mostra que o processo 0 (`rank=P0`) irá armazenar os seus dados coletados no arquivo chamado `trace00PS0.xml`. Os demais elementos `collect_file` são referentes aos demais processos em execução.

```

1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <hieraCollector application="jacobi" count_proc="5">
3   <collect_file rank="P0">trace00PS0.xml</collect_file>
4   <collect_file rank="P1">trace00PS1.xml</collect_file>
5   <collect_file rank="P2">trace00PS2.xml</collect_file>
6   <collect_file rank="P3">trace00PS3.xml</collect_file>
7 </hieraCollector>

```

Um exemplo do arquivo `trace00PS0` do processo 0 contendo dados coletados referentes as operações realizadas pelo processo 0 pode ser visto no quadro abaixo. O arquivo mostra na linha 4 o elemento raiz `processor` com `rank='P0'`, o tempo inicial e final (`init` e `finalize`) da execução do processo 0. O elemento raiz `processor` possui uma lista de elementos `hiera00PS` referente às classe e aos métodos que foram executados pelo processo 0. Por exemplo, a linha 6 mostra a execução do método `TopologyGrid` construtor da classe `TopologyGrid` chamado no arquivo fonte `jacobi.cc` na linha 17 com seus respectivos dados, `gsize` que indica o tamanho da grade, `cols` que indica o número de colunas, o tempo inicial e o tempo final.

A hierarquia pode ser observada da linha 18 até a linha 30, onde o elemento raiz nesse caso é a chamada do método `syncGhostsCart` da classe `Matrix` na linha 18. Pode-se observar que esse elemento “raiz” (no nível mais alto da hierarquia), possui dois filhos que são os métodos `toNorth` e `toEast`, nas linhas 20 e 22 respectivamente. Seguindo na hierarquia dos métodos, o método `toEast` (linha 22) tem um filho que é o método `send` da classe `TopologyGrid` na linha

24, e esse por sua vez, também tem um filho que é o método send da classe Group na linha 27 que tem como filho no nível mais baixo da hierarquia o elemento hieraMPI que é a execução da operação send na linha 30.

Com a apresentação desse exemplo verificamos como a coleta é feita de forma hierárquica pelo módulo HieraCollector da biblioteca HieraAnalyses utilizando o OOPS.

```

1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE processor SYSTEM "trace.dtd">
3 <?xml-stylesheet type="text/xsl" href="visual.xsl"?>
4 <processor rank="P0" init="0.00478" finalize="2.89697">
5   ...
6   <hieraOOPS file="jacobi.cc" line="17" class="OOPS::TopologyGrid"
7     method="TopologyGrid" gsize="4" cols="2" start_time="0.02940"
8     finish_time="0.05502"/>
9   <hieraOOPS file="jacobi.cc" line="31" class="OOPS::Matrix<double>"
10    method="localToGlobal" i="-1" j="-1" start_time="0.05565"
11    finish_time="0.05620"/>
12   <hieraOOPS file="Matrix.h" line="869" class="OOPS::DistributionBlocked"
13    method="localToGlobal" i="-1" N="10" P="2" ID="0" localglobal="-1"
14    start_time="0.05587" finish_time="0.05601"/>
15   <hieraOOPS file="Matrix.h" line="870" class="OOPS::DistributionBlocked"
16    method="localToGlobal" i="-1" N="10" P="2" ID="0" localglobal="-1"
17    start_time="0.05603" finish_time="0.05619"/>
18   <hieraOOPS file="jacobi.cc" line="44" class="OOPS::Matrix<double>"
19    method="syncGhostsCart" start_time="0.06439" finish_time="0.06553">
20     <hieraOOPS file="Matrix.h" line="1456" class="OOPS::TopologyGrid"
21      method="toNorth" count="5" start_time="0.06444" finish_time="0.06448"/>
22     <hieraOOPS file="Matrix.h" line="1461" class="OOPS::TopologyGrid"
23      method="toEast" count="5" start_time="0.06448" finish_time="0.06472">
24      <hieraOOPS file="TopologyGrid.cc" line="7319" class="OOPS::TopologyGrid"
25       method="send" count="5" partner="1" tag="10" start_time="0.06454"
26       finish_time="0.06472">
27        <hieraOOPS file="Topology.cc" line="1921" class="OOPS::Group"
28         method="send" count="5" partner="1" tag="10" start_time="0.06458"
29         finish_time="0.06471">
30         <hieraMPI operation="send" file="Topology.cc" line="1921" count="5"
31          type="MPI_DOUBLE" dest="1" tag="10" comm="OOPS_Comm1"
32          start_time="0.06458" finish_time="0.06470"/>
33        </hieraOOPS>
34      </hieraOOPS>
35    </hieraOOPS>
36   ...
37 </hieraOOPS>
38 </processor>

```

Após os dados serem coletados de forma hierárquica eles serão transformados pelo módulo HieraTransform.

4.2 Módulo de Transformação dos Dados

O módulo de transformação dos dados HieraTransform, tem por objetivo transformar os dados coletados dos vários níveis hierárquicos extraindo medidas que permitam a realização da análise do programa paralelo. Este módulo usa o paradigma de orientação a objetos e é implementado na linguagem C++ [91, 93].

As informações coletadas no programa pelo módulo HieraCollector e armazenadas em arquivos XML são inseridas em uma estrutura de grafo. O grafo gerado possui vários tipos de nós, como os nós de processador, nós de operação ponto-a-ponto (*send*, *receive*), nós de operações coletivas (*broadcast*, *gather*, *scatter*) e nós de métodos conforme pode ser visto na figura 4.5.

Figura 4.5: Ligações do Grafo

As ligações entre os nós do grafo possibilitam dois tipos de percursos:

- **Percurso por Operações:** onde todas operações de cada processador são visitadas apenas uma vez. Nesse percurso pode-se obter informações detalhadas sobre o processador e os tipos de operações. Um percurso completo pode ser visto na figura 4.6, onde a ordem de visita é numerada da seguinte forma: (1)P0, (2)send, (3)bcast, (4)send, (5)P1, (6)recv, (7)bcast, (8)sendrecv, (9)P2, (10)send, (11)bcast, (12)recv, (13)P3, (14)recv, (15)bcast.
- **Percurso entre Operações Parceiras:** são visitadas todas operações e suas parceiras de cada processador, sendo que as operações podem ser visitadas mais de uma vez. Nesse percurso pode-se obter informações detalhadas sobre o processador, a operação e suas parceiras. A visita entre uma operação e sua parceira depende do tipo de comunicação. As operações ponto-a-ponto só se relacionam com outra operação ponto-a-ponto, por

Figura 4.6: Percurso por Operações

exemplo, Send-Receive, Send-SendrecvReceive, Receive-Send. As operações coletivas também só se relacionam entre si, por exemplo, uma operação Scatter com a operação Scatter correspondente. O percurso entre as operações e suas parceiras pode ser visto na figura 4.7, onde a ordem de visita é numerada da seguinte forma: (1) P0, (2) send, (3) rcv, (4) bcast, (5) bcast, (6) bcast, (7) bcast, (8) send, (9) sendrecv, (10) P1, (11) rcv, (12) send, (13) bcast, (14) sendrecv, (15) rcv, (16) send, (17) P2, (18) send, (19) rcv, (20) bcast, (21) rcv, (22) sendrecv, (23) P3, (24) rcv, (25) send, (26) bcast.

Figura 4.7: Percurso entre Operações Parceiras

Para a implementação do grafo, foi utilizado um vetor para representar as ligações entre os processadores, uma lista encadeada que representa a ligação entre as operações, uma lista duplamente encadeada para representar as operações ponto-a-ponto, e listas especializadas para representar cada tipo de operação coletiva.

O diagrama de classes em UML (*Unified Modeling Language*) [94], mostrado na figura 4.8, retrata as construções de herança, associação semântica, composição e agregação do módulo de transformação. As classes que compõem o diagrama da figura 4.8 são:

Figura 4.8: Diagrama de classes UML HieraTransform

htProcessors: define os processadores envolvidos na execução da aplicação paralela. Essa classe define métodos que inserem os dados de um arquivo XML gerados pela coleta em uma lista (classe htHieraList).

htHieraList: define uma lista cronológica para cada processador, onde cada nó representa os dados computados de uma operação.

htFactoryHieraNode: esta é uma classe abstrata que aloca os nós da lista na memória a partir dos elementos (que podem ser hieraMPI e hieraOOPS) lidos do arquivo XML.

htHieraNode: é uma classe abstrata que representa um nó da lista de operações de comunicação. Essa classe armazena características como o tipo de coleta que foi feita, se é MPI ou OOPS; nome do arquivo; a linha onde a operação apareceu no arquivo fonte, o tempo inicial e final da execução da operação.

Figura 4.9: Nó criado na classe htHieraNode

O nó criado na classe htHieraNode contém os campos mostrados na figura 4.9: *trace* que indica se o *trace* é MPI ou OOPS; *file* o nome do arquivo do código fonte; *line* o número da linha no código fonte referente a operação realizada; *start_time* o tempo inicial; *finish_time* o tempo final da operação; *next* o ponteiro para a próxima operação na lista e *nextChild* o ponteiro para o nó filho seja MPI ou OOPS.

htMPINode: é uma classe abstrata especializada da classe htHieraNode. Essa classe armazena características das operações MPI, como por exemplo, o tipo de comunicação, se é coletiva, ponto-a-ponto ou grupos de comunicador; o nome da operação mpi, e os parceiros de entrada e de saída. O diagrama UML dessa classe e suas classes filhas é mostrado na figura 4.10.

O nó criado na classe htMPINode contém os campos mostrados na figura 4.11: *mpi* que indica que o *trace* é MPI; *communication* indica o tipo de comunicação existente, se é ponto-a-ponto, coletiva ou grupo; *operation* indica qual é a operação realizada; *file* o nome do arquivo do código fonte; *line* o número da linha no código fonte referente a operação realizada; *start_time* o tempo inicial; *finish_time* o tempo final da operação; *partnerIn* é o número de parceiros de entrada e *partnerOut* é o número de parceiros de saída.

Figura 4.10: Diagrama UML da classe htMPINode

Figura 4.11: Nó criado na classe htMPINode

htPointToPoint: é uma classe abstrata especializada da classe htMPINode e armazena características como tamanho das mensagens, o tipo de dados da mensagens e o tag da mensagem. Algumas especializações desta classe são: htSend, htSend_init, htRecv, htSendrecv, htBsend, htRsend, htSsend, htSendrecv_replace, entre outras. Algumas destas especializações podem ser vistas na figura 4.12.

htCollective: é uma classe abstrata especializada da classe htMPINode que armazena características como o tamanho da mensagem e o tipo de dado. Algumas especializações dessa classe são: htBcast, htAllgather, htGather, htAllgatherv, htReduce, htAllReduce, htAlltoall, htScatter, htAlltoallv, htGatherv e htScatterv, como pode

Figura 4.12: Diagrama UML da classe htPointToPoint

Figura 4.13: Diagrama UML da classe htCollective

Figura 4.14: Diagrama UML da classe htComm

ser visto na figura 4.13.

htComm: é uma classe abstrata especializada da classe **htMPINode** e armazena características referentes às operações que envolvem comunicadores. As especializações desta classe podem ser vistas na figura 4.14.

htOOPSNode: como na classe **htMPINode**, essa também é uma classe abstrata especializada da classe **htHieraNode**. Essa classe armazena características das classes e métodos que compõem o OOPS, como por exemplo, o nome da classe e do método. As especializações dessa classe são: **htFromNorth**, **htFromSouth**, **htMatrix**, **htVector**, **htToNE**, **htToNW**, **htSum**, **htProd**, entre outras, como pode ser visto na figura 4.15

O nó criado na classe **htOOPSNode** contém os campos mostrados na figura 4.16: **OOPS** que indica que o *trace* é OOPS; **class** indica o nome da classe usada; **method** indica qual é o

Figura 4.15: Diagrama de Classes do htOOPSNode

OOPS	class	method	file	line	start_time	finish_time
------	-------	--------	------	------	------------	-------------

Figura 4.16: Nó criado na classe htOOPSNode

método executado; file o nome do arquivo do código fonte; line o número da linha no código fonte referente ao método realizado; start_time o tempo inicial; finish_time o tempo final.

Pode-se aplicar medidas, como contagem, média, desvio padrão ou variância, sobre um agrupamento de informações que são obtidas através de um percurso possibilitando ao usuário realizar a análise de sua aplicação. Portanto, a análise depende de um grafo sobre o qual podem ser aplicadas uma ou mais medidas.

Em uma mesma aplicação pode-se analisar, por exemplo: o tipo de coleta realizada; a contagem dos tipos de operações; a contagem de tipos de operação de comunicação executadas por cada processador em uma determinada linha do código fonte usando percurso por operação.

O agrupamento define quais informações compõem as medidas. Para definir um agrupamento deve-se observar quais itens deverão constituir o agrupamento, como por exemplo, tipo de operação e processador.

Para realizar o agrupamento dos dados coletados foi utilizada a implementação da *Red-Black Tree* [95] implementada pela STL (*Standard Template Library*) [96].

A STL fornece um conjunto de classes, tais como *containers* e *arrays* associativos, que podem ser usados com qualquer tipo de dado já definido e com tipos de dados definidos pelo usuário que suporta algumas operações elementares (tais como cópia e atribuição). Os algorit-

mos STL são independentes de *containers*, o que reduz significativamente a complexidade da biblioteca. Os resultados da STL são obtidos através do uso de *templates*. A STL foi criada como a primeira biblioteca de algoritmos e estruturas de dados genérica, com quatro idéias em mente: programação genérica, abstração sem perda de eficiência, o modelo computacional de *Von Neumann* e valores semânticos.

A árvore *Red-Black* usada para a implementação, é um tipo de árvore de busca binária balanceada, usada para implementar *arrays* associativos. Essa estrutura foi criada em 1972 por Rudolf Bayer e é também chamada de *Symmetric Binary B-Trees* [95]. Uma árvore *Red-Black* é uma árvore de busca binária onde cada nó tem um atributo de cor, cujo valor é preto ou vermelho. Requerimentos adicionais de qualquer árvore *Red-Black* são: um nó é vermelho ou preto; o nó raiz é preto; todos os nós folhas são pretos; os nós filhos de todo nó vermelho são pretos, assim todo nó vermelho deve ter um pai preto; todo caminho de um nó para um nó folha descendente contém o mesmo número de nós pretos, sem contar o nó folha.

As operações somente de leitura não requerem nenhuma modificação daquelas usadas para árvores de busca binária, porque toda árvore *Red-Black* é uma especialização de uma simples árvore de busca binária. Entretanto, o resultado imediato de uma inserção ou remoção pode violar as propriedades de uma árvore *Red-Black*. A reconstrução das propriedades da *Red-Black* requer um pequeno número de mudanças nas cores e não mais do que três rotações (duas delas para a inserção). Embora as operações de inserção e deleção sejam complicadas, o tempo permanece $O(\log n)$.

A utilização da árvore *Red-Black* fez-se necessária devido a:

- a quantidade variável de itens à qual uma medida será aplicada. Não se sabe previamente a quantidade de itens que serão contabilizados, inviabilizando assim a utilização de uma estrutura estática, como por exemplo, um vetor;
- o alto custo em localizar um item que será contabilizado. Para contabilizar um item, esse deve ser localizado na estrutura que pode ter uma grande quantidade de itens, inviabilizando assim uma busca seqüencial, como por exemplo, busca em uma lista encadeada.

Os nós inseridos na árvore podem ser do tipo *hieramPI* ou *hieraOOPS*. Os nós *hieramPI* são nós de operações ponto-a-ponto, coletivas ou de grupo do MPI. Os nós *hieraOOPS* são nós de classes e métodos que foram executados pelo OOPS. Uma vez que esses nós contendo os dados de MPI ou OOPS são inseridos na árvore, é possível aplicar medidas para que se possa realizar a análise da aplicação.

4.2.1 Experimentos do HieraTransform para MPI

Para que seja possível fazer a análise dos dados coletados de forma mais fácil, foram desenvolvidas quatro aplicações que demonstram as várias possibilidades de análise dos dados. A primeira aplicação realiza uma contagem agrupada por operações em cada nível da hierarquia. A segunda aplicação realiza a contagem do número de vezes que uma operação ou um método foi realizado por qual processador e se a coleta era para MPI ou OOPS. A terceira aplicação mostra o número de vezes que uma operação ou método foi executado. A quarta aplicação mostra o tempo total e o tempo de comunicação de cada processador.

As aplicações foram executadas em quatro nós de um *cluster* composto por oito nós. Cada nó possui processador Pentium 4 Hyper Thread de 3.0 Ghz, 2GB de memória RAM e HD de 160Gb. O sistema operacional usado é o Gentoo Linux [97].

Programa do Passeio de Cavalo

Para exemplificar o uso do módulo de transformação dos dados, HieraTransform, foi utilizado o mesmo exemplo do módulo HieraCollector apresentado na subseção 4.1.1, o problema do passeio do cavalo executado com 4 processos. A tabela 4.1 mostra a contagem do número de operações realizadas que estão no mesmo nível da hierarquia, no caso do MPI, existe apenas um nível na hierarquia. A coluna Processos indica os processos envolvidos na execução da aplicação do passeio do cavalo; a coluna Nível mostra o nível da hierarquia; a coluna Contador mostra a quantidade de operações que foram inseridas no nível 0 da hierarquia.

Tabela 4.1: Contagem do número de operações por nível da hierarquia

Processos	Nível	Contador
0	0	59
1	0	19
2	0	21
3	0	19

A tabela 4.2 mostra os dados da contagem do número de operações realizadas, a linha em que as operações foram executadas e o tipo de coleta feita. A coluna Processos indica o número dos processos; a coluna Operação indica quais foram as operações realizadas por um determinado processo; a coluna Coleta indica o tipo de coleta que foi realizada, nesse caso foi mpi; a coluna Arquivo mostra o nome do arquivo fonte em que a operação foi realizada; a coluna Linha indica a linha no código fonte onde a operação foi executada e a coluna Contador

indica o número de vezes que a operação foi realizada. Por exemplo, o processo 2 realizou 9 vezes as operações *receive* da linha 319 e *send* da linha 318.

Tabela 4.2: Contagem do número de operações e tipo de coleta

Processos	Operação	Coleta	Arquivo	Linha	Contador
0	receive	mpi	cavalo.cc	278	25
0	receive	mpi	cavalo.cc	285	03
0	send	mpi	cavalo.cc	253	03
0	send	mpi	cavalo.cc	279	25
0	send	mpi	cavalo.cc	287	03
1	receive	mpi	cavalo.cc	256	01
1	receive	mpi	cavalo.cc	315	01
1	receive	mpi	cavalo.cc	319	08
1	send	mpi	cavalo.cc	314	01
1	send	mpi	cavalo.cc	318	08
2	receive	mpi	cavalo.cc	256	01
2	receive	mpi	cavalo.cc	315	01
2	receive	mpi	cavalo.cc	319	09
2	send	mpi	cavalo.cc	314	01
2	send	mpi	cavalo.cc	318	09
3	receive	mpi	cavalo.cc	256	01
3	receive	mpi	cavalo.cc	315	01
3	receive	mpi	cavalo.cc	319	08
3	send	mpi	cavalo.cc	314	01
3	send	mpi	cavalo.cc	318	08

A tabela 4.3 mostra a quantidade de operações que foram realizadas pela aplicação. A coluna *Operação* indica as operações que foram executadas por todos os processos. A coluna *Contador* indica o número de vezes que a operação foi executada. Por exemplo, nesta aplicação, tanto a operação *send* quanto o *receive* foram executados 59 vezes.

Tabela 4.3: Contagem do número de operações de comunicação realizadas

Operação	Contador
receive	59
send	59

A tabela 4.4 apresenta o tempo total de execução e o tempo de comunicação gasto por cada processo. A coluna *Processos* mostra os processos envolvidos na execução do programa passeio do cavalo; a coluna *Tempo de Comunicação* apresenta os tempos que cada processo gastou com comunicação e a coluna *Tempo Total* mostra o tempo total gasto por cada processo na execução do programa.

Tabela 4.4: Tempo total de execução e comunicação

Processos	Tempo de Comunicação (seg)	Tempo Total (seg)
0	12.808120	12.810770
1	0.001550	12.810890
2	0.001800	11.915490
3	0.001370	11.413400

A tabela 4.5 apresenta o tempo total de execução da aplicação do passeio do cavalo com e sem *trace*. Pode-se observar que a diferença entre os tempos obtidos foi entre 2.05% a 3.51%.

Tabela 4.5: Comparação do tempo total de execução com e sem *trace*

Processos	Com Trace (seg)	Sem Trace (seg)
0	12.810770	12.376298
1	12.810890	12.376237
2	11.915490	11.538283
3	11.413400	11.183921

Programa de Dinâmica Molecular usando MPI

Este código implementa o algoritmo de decomposição de forças descrito em [98] para dinâmica molecular de partículas *Lennard-Jones*. Um dado número de partículas é distribuído em uma caixa tri-dimensional (com condições periódicas de contorno) de acordo com algumas condições iniciais para posição e velocidade. A evolução da posição e da velocidade das partículas é avaliada iterativamente levando em consideração as interações entre as partículas. As interações são computadas usando o potencial *Lennard-Jones*; este é um potencial de curto alcance permitindo algumas otimizações que evitam a computação de interações de pares de partículas distantes.

O algoritmo de decomposição de forças usa uma organização especial para distribuir a computação da interação entre os pares de partículas entre os processos. As partículas são distribuídas em blocos entre os processos. Depois os processos são organizados em uma grade bidimensional. Quando as interações precisam ser computadas, cada processo coleta a informação sobre todas as partículas em processos da mesma linha em uma lista e todas as partículas nos processos da mesma coluna em outra; as interações entre as partículas na linha com as partículas na coluna são avaliadas. Veja [98] para maiores explicações e outros detalhes do algoritmo.

Essa aplicação foi executada com 4 processos. A tabela 4.6 mostra a contagem do número de operações que estão contidas no nível 0 da hierarquia. A coluna Processos mostra os

processos envolvidos na execução da aplicação. A coluna *Nível* mostra o nível na hierarquia, neste caso existe apenas o nível 0. A coluna *Contador* mostra o número de operações contidas no nível 0. Por exemplo, o processo 0 teve 2396 operações no nível 0 da hierarquia, executando 10% mais em relação aos demais processos.

Tabela 4.6: Contagem do número de operações em cada nível da hierarquia

Processos	Nível	Contador
0	0	2396
1	0	2144
2	0	2144
3	0	2144

A tabela 4.7 mostra a contagem do número de operações realizadas, o tipo de coleta feita, a linha onde a operação foi executada no código fonte e o número de vezes que foi executada. A coluna *Processos* mostra todos os processos envolvidos na execução. A coluna *Operação* mostra quais foram as operações realizadas por cada processo; a coluna *Arquivo* indica o nome do arquivo onde a operação foi executada; a coluna *Coleta* mostra em qual nível foi realizada a coleta, nesse caso, o MPI; a coluna *Linha* indica a linha no código fonte onde a operação foi realizada e a última coluna *Contador* mostra a contagem do número de vezes que cada operação foi executada. Como o tamanho do arquivo de coleta foi muito grande, são mostrados apenas os dados referentes ao processos 0. A tabela mostra que a operação *reduce* nas linhas 171 e 175 são as operações mais executadas.

A tabela 4.8 mostra na coluna *Operação* a lista com as operações que foram executadas na aplicação e a coluna *Contador* mostra o número de vezes que cada operação foi realizada. Nesta tabela pode ser visto que a operação de redução é a operação mais executada em relação as demais operações, nesse caso a operação *reduce* foi executada 6400 vezes.

A tabela 4.9 apresenta os tempos de comunicação e o tempo total gasto pelos processos durante a execução da aplicação de dinâmica molecular. A coluna *Processos* mostra os processos envolvidos na execução da aplicação. A coluna *Tempo de Comunicação* mostra o tempo de comunicação gasto por cada processo durante a execução e a coluna *Tempo Total* apresenta o tempo total gasto pelos processos na execução da aplicação. Com os tempos mostrados na tabela, pôde-se observar que o tempo de comunicação é uma importante fração no tempo total de execução.

A tabela 4.10 apresenta os tempos obtidos com a execução da aplicação de dinâmica molecular usando a versão com e sem *trace*. Com os dados apresentados nesta tabela, observa-se a influência da instrumentação no tempo de execução. A diferença entre os tempos obtidos

Tabela 4.7: Contagem do número de operações realizadas por linha e o tipo de coleta

Processos	Coleta	Operação	Arquivo	Linha	Contador
0	mpi	allgather	md.c	107	1
0	mpi	allgather	md.c	108	1
0	mpi	allgatherv	md.c	137	1
0	mpi	allgatherv	md.c	139	1
0	mpi	allgatherv	md.c	150	200
0	mpi	allgatherv	md.c	156	200
0	mpi	broadcast	md.c	37	1
0	mpi	broadcast	md.c	38	1
0	mpi	broadcast	md.c	39	1
0	mpi	broadcast	md.c	40	1
0	mpi	broadcast	md.c	41	1
0	mpi	broadcast	md.c	42	1
0	mpi	broadcast	md.c	43	1
0	mpi	broadcast	md.c	44	1
0	mpi	broadcast	md.c	45	1
0	mpi	comm_split	md.c	101	1
0	mpi	comm_split	md.c	102	1
0	mpi	receive	auxfmd.c	51	9
0	mpi	receive	auxfmd.c	90	180
0	mpi	receive	auxfmd.c	92	180
0	mpi	reduce	md.c	171	400
0	mpi	reduce	md.c	175	400
0	mpi	reduce	md.c	183	200
0	mpi	reduce	md.c	184	200
0	mpi	reduce	md.c	185	200
0	mpi	reduce	md.c	186	200
0	mpi	scan	md.c	95	1
0	mpi	send	auxfmd.c	58	9
0	mpi	type_commit	md.c	93	1
0	mpi	type_contiguous	md.c	92	1

Tabela 4.8: Contagem usando percurso por operações agrupando somente as operações de comunicação

Operação	Contador
allgather	8
allgatherv	1608
broadcast	36
comm_split	8
receive	378
reduce	6400
scan	4
send	378
type_commit	4
type_contiguous	4

usando as duas versões foi entre 5.91% e 6.40%.

Tabela 4.9: Tempo total de execução e comunicação

Processos	Comunicação (seg)	Tempo Total (seg)
0	0.495700	1.784300
1	0.915030	1.753440
2	0.991050	1.753510
3	0.972300	1.753490

Tabela 4.10: Comparação do tempo total de execução com e sem *trace*

Processos	Com Trace (seg)	Sem Trace (seg)
0	1.784300	1.676900
1	1.753440	1.655580
2	1.753510	1.655580
3	1.753490	1.655630

4.2.2 Experimentos do HieraTransform para OOPS

Para que seja possível realizar a análise dos dados coletados de forma mais fácil, foram desenvolvidas quatro aplicações. As aplicações usadas aqui, foram as mesmas aplicações usadas no módulo de transformação HieraTransform para o MPI na subseção 4.2.1.

Programa Jacobi

Para exemplificar o uso do módulo de transformação dos dados, HieraTransform, foi utilizado o mesmo exemplo do módulo HieraCollector, o problema de Poisson usando o método Jacobi descrito na subseção 4.1.2.

Essa aplicação foi executada utilizando 4 processos. A tabela 4.11 mostra a contagem do número de operações em cada nível da hierarquia. A coluna *Nível* mostra os vários níveis existentes nessa aplicação. As demais colunas mostram os processadores e o número de operações em cada nível da hierarquia. Por exemplo, o processador 2 tem 1000 operações no nível 3.

Tabela 4.11: Contagem do número de operações em cada nível da hierarquia

Nível	Proc0	Proc1	Proc2	Proc3
0	530	530	530	530
1	2131	2131	2131	2131
2	1125	1125	1125	1125
3	1000	1000	1000	1000

Tabela 4.12: Contagem do número de operações realizadas por linha, arquivo e o tipo de coleta

Processos	Coleta	Operação	Arquivo	Linha	Contador
1	mpi	allreduce	Topology.cc	3230	125
1	mpi	receive	Topology.cc	2175	500
1	mpi	send	Topology.cc	1921	500
1	oops	distributionBlocked	Application	0	1
1	oops	fromEast	Matrix.h	1491	250
1	oops	fromNext	Matrix.h	1486	250
1	oops	fromSouth	Matrix.h	1476	250
1	oops	fromWest	Matrix.h	1481	250
1	oops	localSize	Matrix.h	1264	2
1	oops	localSize	Matrix.h	1265	2
1	oops	localToGlobal	jacobi.cc	31	49
1	oops	localToGlobal	Matrix.h	869	49
1	oops	localToGlobal	Matrix.h	870	49
1	oops	matrix	Application	0	2
1	oops	max	jacobi.cc	58	125
1	oops	max	Topology.cc	3230	125
1	oops	receive	Topology.cc	2175	500
1	oops	receive	TopologyGrid.cc	6061	250
1	oops	receive	TopologyGrid.cc	6899	250
1	oops	send	Topology.cc	1921	500
1	oops	send	TopologyGrid.cc	5642	250
1	oops	send	TopologyGrid.cc	6480	250
1	oops	split	Topology.cc	2221	2
1	oops	split	TopologyGrid.cc	4273	1
1	oops	split	TopologyGrid.cc	4274	1
1	oops	syncGhostsCart	jacobi.cc	44	125
1	oops	syncGhostsCart	jacobi.cc	49	125
1	oops	topologyGrid	Application	0	1
1	oops	topologyPipe	Constructor	0	2
1	oops	toEast	Matrix.h	1461	250
1	oops	toNorth	Matrix.h	1456	250
1	oops	toSouth	Matrix.h	1466	250
1	oops	toWest	Matrix.h	1471	250

A tabela 4.12 mostra a contagem do número de operações realizadas, o tipo de coleta feita, a linha onde a operação foi executada no código fonte e número de vezes que foi executada. Nesta tabela foram apresentados apenas os dados referentes ao processo 1. A coluna Processos mostra os processos envolvidos na execução. A coluna Coleta mostra em qual nível foi realizada a coleta, nesse caso tem operações do MPI e métodos do OOPS; a coluna Operação mostra quais foram as operações realizadas por cada processo; a coluna Linha indica a linha no código fonte onde a operação foi realizada e a última coluna mostra a contagem do número de vezes que cada operação foi executada.

A tabela 4.13 mostra na coluna Operação/Método a lista com as operações e métodos que foram executados na aplicação e a coluna Contador mostra o número de vezes que cada operação foi realizada. Por exemplo, a operação send foi executada 2000 vezes e o método fromNext foi executado 1000 vezes.

A tabela 4.14 apresenta o tempo total obtido da execução da aplicação jacobi com e sem trace. Por exemplo, o processo 3 na versão do programa jacobi que usa trace gastou 2.425846 segundos e o processo 3 da versão que não usa trace gastou 2.314920 segundos. A diferença de

Tabela 4.13: Contagem usando percurso por operações agrupando somente as operações de comunicação

Operação/Método	Contador	Operação/Método	Contador
allreduce	500	max	1000
receive	2000	receive	4000
send	2000	send	4000
distributionBlocked	4	split	16
fromEast	1000	syncGhostsCart	1000
fromNext	1000	topologyGrid	12
fromSouth	1000	toEast	1000
fromWest	1000	toNorth	1000
localSize	16	toSouth	1000
localToGlobal	588	toWest	1000
matrix	8		

tempo entre as duas versões ficou entre 4.79% e 11.15%.

Tabela 4.14: Comparação do tempo total de execução com e sem *trace*

Processos	Com Trace (seg)	Sem Trace (seg)
0	2.572783	2.314560
1	2.573641	2.316270
2	2.428085	2.315120
3	2.425846	2.314920

Programa de Dinâmica Molecular usando OOPS

O programa de dinâmica molecular usado aqui é o mesmo descrito na subseção 4.2.1, só que usando OOPS.

Essa aplicação foi executada com 4 processos. A tabela 4.15 mostra a contagem do número de operações que estão contidas em todos os níveis da hierarquia. A coluna Processos mostra o número de processos envolvidos na execução da aplicação. As demais colunas mostram a quantidade de operações e métodos em cada nível da hierarquia. Por exemplo, o processo 0 tem 165.476 operações no nível 1. Esta tabela mostra o número de operações executadas em cada nível de abstração 0, 1 e 2. Observou-se que um número muito maior no número de operações em relação ao código dessa aplicação na versão do MPI. Isto se deve ao fato de que a versão atual do OOPS não tem operações de redução em seções de *arrays*, como apresentado no MPI, e portanto as reduções devem ser executadas em um *loop* para cada partícula.

A tabela 4.16 mostra a contagem do número de operações realizadas, o tipo de coleta feita,

Tabela 4.15: Contagem do número de operações em cada nível da hierarquia

Processos	Nível 0	Nível 1	Nível 2
0	101756	165476	100811
1	101756	100948	100811
2	101756	100948	100811
3	101756	100948	100811

o nome do arquivo fonte, a linha onde a operação foi executada no código fonte e o número de vezes que a operação ou o métodos foram executados. A coluna Processos mostra os processos envolvidos na execução da aplicação, e devido a grande quantidade de dados foi mostrado apenas os dados referentes ao processo 2. A coluna Coleta mostra em qual nível foi realizada a coleta, nesse caso, o MPI e OOPS. A coluna Operação mostra as operações realizadas por cada processo; a coluna Arquivo indica o nome do arquivo fonte onde a operação foi executada. A coluna Linha indica a linha no código fonte onde a operação foi realizada e a última coluna Contador mostra a contagem do número de vezes que cada operação foi executada. Nesta tabela pôde-se notar como a maioria das operações de redução do MPI (um total de 100000) são executadas em requisição das operações de sum nas linhas 181 e 188 do arquivo fonte `molecularDynamicsOOPS.cc`.

A tabela 4.17 mostra na coluna Operação/Método a lista com as operações do MPI e o métodos do OOPS que foram executados no código fonte da aplicação. A coluna Contador mostra o número de vezes que cada operação foi realizada. Por exemplo, a operação `reduce` do MPI foi realizada 400000 vezes e o método `sum` foi realizado 806400 vezes durante a execução de todos os processos.

A tabela 4.18 apresenta a comparação do tempo total de execução de cada processo do programa de dinâmica com e sem *trace*. Os tempos apresentados pela versão do programa que usa *trace* foi de 239.407456 segundos para o processo 0 e o tempo do processo 0 para a versão que não usa *trace* foi de 237.457 segundos. A diferença de tempo entre eles foi entre 0.77% a 0.82%, o que é um valor pequeno em relação ao alto número de operações registradas.

4.3 Conclusões do Capítulo

Este capítulo apresentou a biblioteca de Análise Hierárquica *Hierarchical Analyses*. O módulo de coleta `HieraCollector` usou a técnica de *trace* por permitir a coleta de forma mais detalhada, o armazenamento dos dados coletados é feito em arquivos estruturados no formato XML seguindo uma organização lógica e hierárquica possibilitando que a análise fosse feita em vários

Tabela 4.16: Contagem do número de operações realizadas por linha e o tipo de coleta

Processos	Coleta	Operação	Arquivo	Linha	Contador
2	mpi	allgather	Topology.cc	2655	2
2	mpi	allgatherv	Topology.cc	2622	2
2	mpi	allreduce	Topology.cc	3672	200
2	mpi	allreduce	Topology.cc	3837	600
2	mpi	broadcast	Topology.cc	2335	5
2	mpi	broadcast	Topology.cc	2423	3
2	mpi	broadcast	Topology.cc	2434	1
2	mpi	reduce	Topology.cc	3892	100000
2	mpi	scan	Topology.cc	4202	1
2	oops	broadcast	molecularDynamicsOOPS.cc	45	1
2	oops	broadcast	molecularDynamicsOOPS.cc	46	1
2	oops	broadcast	Topology.cc	2423	3
2	oops	broadcast	Topology.cc	2434	1
2	oops	distributionBlocked	Application	0	1
2	oops	gather	Topology.cc	2622	2
2	oops	gather	Topology.cc	2655	2
2	oops	gather	Topology.h	3019	60
2	oops	load	Vector.h	1606	4
2	oops	localSize	Vector.h	1285	4
2	oops	localSize	Vector.h	1531	4
2	oops	scan	molecularDynamicsOOPS.cc	105	1
2	oops	scan	Topology.cc	4202	1
2	oops	scatter	Vector.h	1531	4
2	oops	scatter	Topology.h	3045	4
2	oops	split	Topology.cc	2221	2
2	oops	split	TopologyGrid.cc	4273	1
2	oops	store	Vector.h	1613	60
2	oops	sum	molecularDynamicsOOPS.cc	181	50000
2	oops	sum	molecularDynamicsOOPS.cc	188	50000

níveis. Esse módulo foi implementado para o MPI para o OOPS. A implementação do módulo HieraCollector usando as rotinas da biblioteca do MPI permitiu mostrar a comunicação e a sincronização entre os vários processos, apresentando também os dados referentes a cada operação realizada, o que pôde ser visto nos exemplos apresentados na subseção 4.1.1. Com a implementação do módulo HieraCollector para as classes e os métodos do *framework* OOPS foi possível mostrar de forma mais clara a coleta no nível de orientação a objetos exibindo a hierarquia entre os dados coletados, como pôde ser visto na subseção 4.1.2.

O módulo de transformação HieraTransform utiliza os arquivos XML gerados na coleta para aplicar medidas para a análise de desempenho. Os dados do arquivo XML são inseridos em uma estrutura de grafo que tem nós de operações e métodos. Com o uso desse grafo foi possível implementar dois tipos de percursos entre seus nós permitindo agrupar os dados para que se possa aplicar medidas para realizar a análise. Os nós lidos no grafo são inseridos em uma árvore para que se possa realizar o agrupamento e aplicar uma medida para a análise dos dados.

Tabela 4.17: Contagem usando percurso por operações agrupando somente as operações de comunicação

Operação/Método	Contador
allgather	8
allgatherv	8
allreduce	3200
broadcast	36
reduce	400000
scan	4
broadcast	72
distributionBlocked	4
gather	3712
load	16
localSize	32800
localToGlobal	32016
scan	8
scatter	32
split	16
store	240
sum	806400
topologyGrid	4
topologyPipe	12

Tabela 4.18: Comparação do tempo total de execução com e sem *trace*

Processos	Com Trace (seg)	Sem Trace (seg)
0	239.407456	237.457
1	239.288433	237.458
2	239.288600	237.459
3	239.288144	237.471

Com o uso das aplicações desenvolvidas foi possível realizar a análise dos dados de forma mais fácil, como mostrado na subseção 4.2.1 e na subseção 4.2.2.

No capítulo 5 será apresentada a ferramenta HieraOLAP que pode ser usada para visualização e análise dos dados.

CAPÍTULO

5

Ferramenta de Análise HieraOLAP

Este capítulo apresenta uma outra contribuição inovadora deste trabalho, que é auxiliar a tomada de decisão na análise de desempenho de aplicações paralelas usando o processo de *Data Warehousing*. Para isso, foi desenvolvida uma ferramenta chamada HieraOLAP que é composta dos quatro módulos do processo de *Data Warehousing*: acesso, transformação, base multidimensional e distribuição.

O módulo de acesso, que será descrito na seção 5.1, é responsável pela extração dos dados coletados nas aplicações paralelas. O módulo de transformação, que será descrito na seção 5.2, integra os dados primitivos (dados coletados) provenientes do módulo de acesso em dados derivados para o modelo multidimensional.

O módulo base multidimensional, que será detalhado na seção 5.3, utiliza dois modelos estrela que compõem um *Data Mart* para o propósito de análise de desempenho dos desenvolvedores de aplicações paralelas. Um modelo estrela representa os dados das operações do MPI e o outro representa as classes e métodos do *framework* OOPS.

No módulo distribuição, que será apresentado na seção 5.4, implementa uma ferramenta OLAP que utiliza os dados coletados pelo HieraCollector e que serão recuperados pelo módulo de acesso, padronizados pelo módulo de transformação e armazenados no módulo base multidimensional.

Essa ferramenta foi desenvolvida em C++ usando o compilador *Borland C++ Builder* e as bibliotecas: VCL (*Visual Component Library*) [99] da Borland para os componentes gráficos de janela, LibXML2 [87] da Gnome para manipulação de arquivos XML e a *Hierarchical Analyses* descrita no capítulo 4. O armazenamento dos dados na ferramenta utiliza o banco de dados relacional chamado Paradox [100].

A figura 5.1 apresenta a tela principal da ferramenta HieraOLAP. A figura na parte em destaque (1) mostra as opções de menu e os botões de atalho: o primeiro botão cria uma visualização

de uma face do modelo multidimensional que envolve os módulos base multidimensional e distribuição; o segundo botão importa os dados XML que envolve os módulos de acesso e transformação; os três últimos botões são usados para organizar as janelas em cascata, lado a lado horizontalmente e verticalmente. A figura na parte em destaque (2) é usada para apresentar as várias visualizações de faces do modelo multidimensional selecionadas pelo usuário.

Figura 5.1: Tela principal da Ferramenta HieraOLAP

5.1 Módulo de Acesso

O módulo de acesso é responsável pela migração dos dados da base operacional para a base multidimensional da ferramenta HieraOLAP.

Os dados operacionais são obtidos pelo `hieraCollector` com a inclusão de uma diretiva de pré-processamento chamada `_HIERATRACEMPI` ou `_HIERATRACE00PS` no processo de compilação da aplicação. Os dados do *trace* são descarregados em arquivos no formato XML, durante a execução da aplicação paralela. Mais detalhes podem ser vistos na seção 4.1 do capítulo 4.

A ferramenta HieraOLAP utiliza o carregamento dos dados diretamente do ambiente operacional. Essa forma de carregamento permite ao usuário selecionar quais execuções de uma mesma aplicação paralela serão armazenadas na base multidimensional da ferramenta. Assim,

o usuário pode gerar uma base histórica das execuções demonstrando a evolução no código fonte.

Os módulos de acesso e de transformação na ferramenta HieraOLAP fazem parte da opção Importar XML no menu Arquivo, onde o usuário deve selecionar o arquivo `hieraCollector.xml` que foi gerado durante a execução da aplicação paralela, como mostra a figura 5.2. Para que o usuário tenha uma base histórica das execuções de uma aplicação, deve selecionar o arquivo `hieraCollector.xml` de cada execução, podendo até excluí-los após a importação.

Figura 5.2: Importação do Arquivo XML

5.2 Módulo de Transformação

O módulo de transformação utiliza dados primitivos coletados pelo *trace* e dados derivados. Foram identificados treze campos primitivos importantes para a análise de desempenho:

- Aplicação: nome do arquivo executável. Esse campo é usado para análise de desempenho de aplicações que usam MPI ou OOPS.

- **Biblioteca:** nome da biblioteca, sendo que uma aplicação pode usar apenas o MPI ou OOPS mais MPI.
- **Processador:** número do processador onde foi realizado o processamento. Esse campo é usado para MPI ou OOPS.
- **Arquivo:** nome do arquivo do código fonte, sendo que uma aplicação pode ser composta por vários. Esse campo é usado para análise em MPI ou OOPS.
- **Linha:** número da linha no arquivo fonte. Campo usado para MPI ou OOPS.
- **Tempo Inicial:** tempo em milisegundos que foi iniciado o processamento. Campo usado para MPI ou OOPS.
- **Tempo Final:** tempo em milisegundos do término do processamento. Campo usado para MPI ou OOPS.
- **Operação:** nome da operação do MPI, por exemplo, `send`, `receive`, `gather`, `broadcast`, entre outras.
- **Comunicador:** tipo de comunicador usado pelas operações do MPI.
- **Tipo de Dado:** tipo de dado enviado ao processador pela operação do MPI.
- **Tamanho do Dado:** tamanho do dado enviado ao processador pela operação do MPI.
- **Classe:** nome da classe do OOPS.
- **Método:** nome do método do OOPS.

A partir dos campos primitivos, foram criados treze campos derivados que são descritos abaixo:

- **Agrupamento por Processador:** o campo primitivo número do processador onde foi realizado o processamento foi transformado em múltiplos de 2, múltiplos de 5, além de ser classificado em par ou ímpar. Por exemplo, o processador 7 corresponde a: processador 8 em múltiplos de 2, processador 10 em múltiplos de 5 e classificado como processador ímpar.
- **Agrupamento por Linha:** o campo primitivo número da linha no código fonte foi transformado em múltiplos de 5 linhas, 15 linhas e 30 linhas. Por exemplo, a linha 92 corresponde a: linha 95 no múltiplo de 5, linha 105 no múltiplo de 15, e linha 120 no múltiplo de 30.

- **Agrupamento por Tempo Inicial:** o campo primitivo tempo inicial que está na unidade de milésimos de segundos foi transformado em múltiplos de centésimos de segundos, de segundo, de 5 segundos, de 15 segundos e de 30 segundos. Por exemplo, o tempo inicial 65'432 corresponde a: 655 centésimos de segundo, 66 segundos, 70 segundos no múltiplo de 5 segundos, 75 segundos no múltiplo de 15 segundos e 90 segundos no múltiplo de 30 segundos.
- **Agrupamento por Operação:** o campo primitivo nome da operação MPI foi transformado em operações ponto-a-ponto, coletivas e grupos. Por exemplo, a operação send corresponde a uma operação ponto-a-ponto e a operação gather corresponde a uma operação coletiva.
- **Agrupamento por Destino do Dado:** o campo primitivo nome da operação MPI possibilita identificar se é dado de entrada (recebido), saída (enviado) ou de entrada/saída (recebido/enviado). Por exemplo, os dados enviados em uma operação broadcast são de saída, e os dados de uma operação sendrecv são dados de entrada e saída.
- **Agrupamento por Nível:** o nível é determinado pela quantidade de elementos antecessores até atingir a raiz do documento XML. Esse dado foi transformado em múltiplos de 2, múltiplos de 5, além de ser classificado em par ou ímpar. Por exemplo, o nível 3 corresponde a: nível 4 em múltiplos de 2, nível 5 em múltiplos de 5 e classificado como nível ímpar.
- **Tempo Gasto:** é o resultado do cálculo de subtração entre os campos primitivos tempo final e tempo inicial.
- **Quantidade de Parceiros Entrada ou Saída:** os campos primitivos operação do MPI possibilita identificar a quantidade de processadores parceiros de entrada ou parceiros de saída foi utilizada.
- **Tamanho dos Dados Entrada ou Saída:** os campos primitivos tamanho dos dados e operação do MPI possibilita identificar se o dado é de entrada ou de saída.

5.3 Módulo Base Multidimensional

O carregamento dos dados provenientes do módulo de acesso, integrados pelo módulo de transformação são inseridos em duas bases multidimensionais de um *data mart*, uma para análise de desempenho de aplicações que usam o MPI e outra para aquelas que usam o OOPS.

A figura 5.3 mostra uma representação multidimensional através do modelo estrela para a análise de desempenho do MPI. Essa figura apresenta um retângulo central que representa o fato a ser analisado que liga com vários outros retângulos chamados dimensões. O fato nessa figura é chamado de Análise de Desempenho MPI e as dimensões são tempo, dados, arquivo, operação, comunicador e processador.

Figura 5.3: Modelo Estrela para MPI

Os campos do fato que podem ser usados na análise pelo modelo estrela do MPI são: tempo gasto, quantidade de operações, quantidade de processadores, parceiros de entrada e de saída, e tamanho de dados de entrada e de saída. Sobre esses campos podem ser aplicados sumários como contagem, média e soma.

Os campos nas dimensões determinam a granularidade da informação a ser analisada pois, quanto maior a granularidade, menor é a quantidade de grupos no resultado da análise. E quanto menor a granularidade, maior é a quantidade de grupos no resultado da análise. A menor granularidade indica a informação indivisível.

A seguir é descrito o agrupamento dos campos de cada dimensão do MPI:

- **Processador:** essa dimensão indica o processador em que foi realizada a operação do MPI. O número do processador foi agrupado em: classificação par/ímpar, múltiplos de 5 processadores e múltiplos de 2 processadores.
- **Arquivo:** essa dimensão indica a linha no arquivo fonte em que foi coletada a operação do MPI. A linha em que foi coletada a operação foi agrupada em: múltiplos de 30 linhas, múltiplos de 15 linhas e múltiplos de 5 linhas.

- **Tempo:** essa dimensão indica o tempo em que foi realizada cada operação do MPI. O tempo em milisegundos da realização da operação foi agrupado em: centésimos de segundos, segundos, 5 segundos, 15 segundos e 30 segundos.
- **Operação:** essa dimensão indica a operação do MPI. O nome da operação foi agrupado por classificação de operação (coletiva, ponto-a-ponto e grupo).
- **Dados:** essa dimensão indica características dos dados envolvidos nas operações do MPI. O tipo de dado foi agrupado em classificação de entrada ou saída.
- **Comunicador:** essa dimensão indica o comunicador envolvido nas operações do MPI.

O dado indivisível apresentado nesse modelo é: o número do processador, a linha no arquivo fonte, o tempo em milisegundos, a operação realizada, o tipo do dado usado e o comunicador.

A figura 5.4 mostra uma representação multidimensional através do modelo estrela para a análise de desempenho do OOPS. O fato nessa figura é chamado de Análise de Desempenho OOPS e as dimensões são tempo, arquivo, processador, métodos e classe.

Figura 5.4: Modelo Estrela para OOPS

Os campos do fato que podem ser usados na análise pelo modelo estrela do OOPS são: tempo gasto, a quantidade de métodos, quantidade de classes, e quantidade de processadores.

A seguir é descrito o agrupamento dos campos de cada dimensão do OOPS:

- **Processador:** essa dimensão indica o processador em que foi realizada a operação do MPI. O número do processador foi agrupado em: classificação par/ímpar, múltiplos de 5 processadores e múltiplos de 2 processadores.

- **Arquivo:** essa dimensão indica a linha no arquivo fonte em que foi coletada a operação do MPI ou os métodos do OOPS. A linha em que foi coletada a operação foi agrupada em: múltiplos de 30 linhas, múltiplos de 15 linhas e múltiplos de 5 linhas.
- **Tempo:** essa dimensão indica o tempo em que foi realizada cada operação do MPI. O tempo em milissegundos da realização da operação foi agrupado em: centésimos de segundos, segundos, 5 segundos, 15 segundos e 30 segundos.
- **Nível:** essa dimensão indica a quantidade de chamadas internas até atingir o código fonte. As chamadas são realizadas pelos métodos públicos das classes do OOPS. O incremento do nível é feito a cada chamada interna pelo método. O nível foi agrupado em: classificação par/ímpar, múltiplos de 5 níveis e múltiplos de 2 níveis.
- **Classe:** essa dimensão indica o nome da classe utilizada no OOPS.
- **Método:** essa dimensão indica o nome do método utilizado no OOPS.

Os dados apresentados nesse modelo são: o número do processador, a linha no arquivo fonte, o tempo em milissegundos, quantidade de níveis, o nome da classe e o nome do método.

Para a implementação da base multidimensional foi utilizado o modelo relacional, caracterizando assim um ROLAP (descrito na subseção 2.4.1 do capítulo 2). Existem duas maneiras de realizar o mapeamento do modelo multidimensional para o modelo relacional: criar uma tabela para cada dimensão e para cada fato ou criar uma única tabela unindo todos os campos de dimensão e de fatos.

A primeira forma de mapeamento minimiza a quantidade de dados armazenados pois, não existe duplicação de informação nas tabelas dimensão. Isto acontece porque um registro na tabela dimensão pode relacionar com vários registros da tabela de fatos. No entanto, nessa forma de mapeamento exige que para cada consulta sejam realizadas operações de junção para unir dados presentes na tabela dimensão e na tabela fatos. Operações de junção com grandes quantidades de dados possuem alto custo computacional.

A segunda forma de mapeamento utiliza uma única tabela, o que exige a duplicação da informação. Isso acontece porque um mesmo valor de registro na tabela dimensão vai repetir várias vezes nessa única tabela. No entanto, essa forma de mapeamento não exige operações de junção, uma vez que os dados estão em uma única tabela. Em decorrência da grande quantidade de dados gerados pela coleta, optou-se por essa forma de mapeamento.

5.4 Distribuição usando OLAP

O módulo de distribuição usa uma visualização OLAP para o modelo multidimensional do MPI e do OOPS. Consiste na seleção das dimensões e sumário em fatos para posterior investigação da visualização através do gráfico gerado.

Os módulos base multidimensional e distribuição são visíveis na ferramenta através da opção criar face do menu Arquivo que mostra uma janela para projetar a face de visualização do modelo multidimensional da figura 5.5.

Figura 5.5: Projeto de Visualização de uma face do Modelo Multidimensional

Para melhor entendimento a figura 5.5 foi dividida em quatro regiões: (1) região de filtro, (2) região de sumarização, (3) região que define a dimensão em linha e (4) região que define a dimensão em coluna.

A região (1) define os filtros que podem ser aplicados na base multidimensional que são: por tipo de *trace* onde o usuário escolhe entre filtrar pelas operações do MPI, ou pelas classes e métodos do OOPS, ou ambos MPI e OOPS; por aplicação onde o usuário seleciona a execução de interesse; e por arquivo fonte onde o usuário seleciona o nome do arquivo fonte é de seu

interesse.

Na região (2) o usuário seleciona entre os sumarizadores de contagem, média e soma nos campos da tabela fatos do MPI e do OOPS (explicados na seção 5.3): tempo gasto, quantidade de operações, quantidade de parceiros de entrada e saída, número de processadores, tamanho dos dados de entrada e saída, quantidade de métodos e quantidade de classes.

As regiões (3) e (4) definem os campos das dimensões que serão agrupados em linha e coluna respectivamente. As dimensões para o MPI e para o OOPS são: tempo, dados, arquivo, processador, comunicador, operação, classe e método. A hierarquia entre os campos de cada dimensão podem ser vistos nas seções 5.2 e 5.3.

5.4.1 Experimento para Visualização

Para gerar a visualização de uma face do modelo multidimensional deve-se selecionar as opções de filtro, um sumarizador e duas dimensões.

Neste experimento foi utilizado os dados obtidos com a execução do programa do cavalo apresentado no capítulo 4. Esses dados foram importados previamente usando a opção Importar XML para a ferramenta HieraOLAP.

Figura 5.6: Dados da Aplicação do cavalo

A figura 5.6 mostra na região (1) cinco botões. O primeiro botão é usado para exibir ou não os dados em forma de gráfico; o segundo botão é usado para exibir ou não os dados na forma de

tabela; o terceiro botão indica o sumariizador escolhido; o quarto botão indica a dimensão em linha e o quinto botão indica a dimensão em coluna. A região (2) apresenta os dados na forma de tabela. E a região (3) apresenta os dados na forma de gráfico.

As opções selecionadas para essa visualização foram: o tipo de coleta MPI, a aplicação cavalo e o arquivo escolhido foi cavalo.cc. A sumarização escolhida foi a soma do tempo gasto; a dimensão selecionada na coluna foi Processador com a granularidade por processador e a dimensão selecionada na linha foi Operação com a granularidade por operação.

5.4.2 Experimentos para Visualização das Operações *Drill Down* e *Roll Up*

A operação de *Drill Down* permite a diminuição da granularidade nas dimensões. Para mostrar essa operação, foram utilizados os dados obtidos com a execução do programa de dinâmica molecular para o MPI apresentado no capítulo 4. Os dados foram importados previamente usando a opção Importar XML para a ferramenta HieraOLAP.

A figura 5.7 mostra duas operações de *Drill Down*, uma entre as regiões (1) e (2) e outra entre as regiões (2) e (3). A região (1) apresenta um gráfico inicial entre a dimensão Operação em linha com a granularidade Classificação da Operação e a dimensão Processador em coluna com a granularidade Agrupa Par/Ímpar.

A região (2) mostra um gráfico com a diminuição da granularidade Agrupa Par/Ímpar da dimensão Processador em coluna para a granularidade Por Processador. A região (3) apresenta um gráfico com a diminuição da granularidade Classificação da Operação da dimensão Operação em linha para a granularidade Por operação.

As opções selecionadas para a visualização da região (1) foram: o tipo de coleta MPI, a aplicação md e os arquivos escolhidos foram auxfmd.c e md.c. A sumarização escolhida foi a média do tempo gasto; a dimensão selecionada na coluna foi Processador com a granularidade Agrupa Par/Ímpar e a dimensão selecionada na linha foi Operação com a granularidade Classificação da Operação.

Para a região (2), as opções de visualização selecionadas em filtro e sumariizador são as mesmas usadas usadas na região (1). No entanto, a dimensão selecionada na linha foi Operação com a granularidade Classificação da Operação e a dimensão selecionada na coluna foi Processador com a granularidade Por Processador.

Na região (3) as alterações das opções de visualização foram apenas nas dimensões. Foi selecionada na dimensão Operação em linha com a granularidade Por Operação e a dimensão Processador em coluna com a granularidade Por Processador.

Figura 5.7: Operação *Drill Down*

A outra operação usada é a *Roll Up* que permite o aumento da granularidade nas dimensões. Para exemplificar o uso dessa operação foi utilizado o mesmo programa da operação *Drill Down*.

Duas operações de *Roll Up* são exibidas na figura 5.8, umas entre as regiões (1) e (2) e outra entre as regiões (2) e (3). A região (1) apresenta no gráfico inicial a dimensão Operação em linha com a granularidade Por Operação e a dimensão Tempo em coluna com a granularidade

Figura 5.8: Operação *Roll Up*

Agrupar 1 segundo. A região (2) mostra um gráfico com a dimensão Operação em linha com a granularidade Por Operação e a dimensão Tempo em coluna com a granularidade Agrupa 5 segundo.

O gráfico apresentado na região (3) tem a dimensão Operação em linha com a granularidade Classificação da Operação e a dimensão Tempo em coluna com a granularidade Agrupa 5 segundos.

Para as regiões (1), (2) e (3), as opções de filtro e sumarização foram as mesmas para as três regiões: tipo de coleta MPI, a aplicação md, os arquivos escolhidos foram auxfmd.c e md.c e a sumarização contagem do tempo gasto. Na região (1) a dimensão Operação em linha com a granularidade Por Operação e a dimensão Tempo com a granularidade Agrupa 1 segundo. Aplicando a operação *Roll Up* a região (2) tem a dimensão Operação em linha com a granularidade Por Operação e a dimensão Tempo com a granularidade Agrupa 5 segundos. E quando aplicada a operação novamente, a região (3) tem apenas a dimensão Operação em linha com a granularidade Classificação por Operação modificada.

5.4.3 Experimentos para visualização da operação *Rotate*

A operação *Rotate* é usada para alterar as duas dimensões (linha e coluna) escolhidas para a visualização dos dados. Para mostrar o uso da operação de rotacionamento, foram utilizados os dados obtidos com a execução do programa Jacobi com a coleta hierárquica, apresentado no capítulo 4. Os dados também foram importados previamente usando a opção Importar XML para a ferramenta HieraOLAP.

A figura 5.9 mostra duas operações *Rotate*, uma nas regiões (1) e (2) e outra na região (2) e (3). A região (1) apresenta no gráfico a dimensão Classe em linha com a granularidade Por Classe e a dimensão Processador em coluna com a granularidade Por Processador. Aplicado a operação *Rotate*, a região (2) apresenta a dimensão Classe em linha com a granularidade Por Classe e a dimensão Nível em coluna com a granularidade Agrupa 1 Nível, ou seja, houve uma mudança na dimensão coluna da região (1) para a região (2).

Aplicado a segunda operação *Rotate*, a região (3) mostra a dimensão Método em linha com a granularidade Por Método e a dimensão Classe em coluna com a granularidade Por Classe, apresentando mudança na dimensão linha e coluna.

Para as três regiões o filtro e a sumarização selecionados foram: tipo de coleta foi OOPS, a aplicação foi jacobi, o arquivo selecionado foi jacobi.cc e a sumarização foi média do tempo gasto.

5.5 Conclusões do Capítulo

Neste capítulo foi apresentada a ferramenta HieraOLAP que auxilia o usuário na análise de desempenho de programas paralelos usando o processo de *Data Warehousing*. A ferramenta HieraOLAP é composta dos quatro módulos: acesso, transformação, base multidimensional e distribuição.

Figura 5.9: Operação *Rotate*

Com a ferramenta é possível navegar nas dimensões em diferentes níveis de granularidade através das operações de *Drill Down*, *Roll Up* e *Rotate*.

Os experimentos apresentados demonstraram as funcionalidades da ferramenta manipulando o total de 8852 de registros na base multidimensional.

CAPÍTULO

6

Conclusões e Trabalhos Futuros

A computação paralela se tornou uma ferramenta indispensável em muitas áreas onde o desempenho obtido com apenas um processador não é desejado. Dessa forma, a computação paralela tenta suprir a necessidade que a computação sequencial não consegue, como por exemplo, meteorologistas querendo prever o tempo com bastante precisão; farmacêuticos querendo desenvolver remédios usando computadores sem a necessidade de testá-los em cobaias; engenheiros querendo usar computadores durante todas as fase de desenvolvimento de aviões.

Para que seja possível obter o desempenho melhor nas várias áreas, é necessário a análise de desempenho em todo o estágio de ciclo de vida de um sistema. Uma forma de analisar o desempenho desses sistemas é seguir três passos do processo de análise que são: a coleta, a transformação e a visualização dos dados.

Diversas abordagens de ferramentas foram propostas para auxiliar no processo de análise de desempenho. Com o estudo das ferramentas apresentadas, este trabalho de doutorado veio propor a implementação de uma ferramenta para análise de desempenho de forma hierárquica, ou seja, a ferramenta permitiu acompanhar os níveis de abstração utilizado pelo usuário. As principais contribuições são descritas com mais detalhes na seção 6.1.

6.1 Principais Contribuições

As principais contribuições deste trabalho foram: definição e implementação de uma ferramenta de análise hierárquica. Essa ferramenta utilizou monitoramento por *software* nas rotinas da biblioteca MPI e nas classes e métodos definidas no *framework* OOPS e a técnica de instrumentação utilizada nas rotinas foi *trace* por permitir registrar eventos de recepção e envio, registrar tempos de execução gasto pelos processos. Como apresentado no capítulo 4, a ferra-

menta HieraAnalyses é composta dos módulos de: coleta dos dados, transformação e análise de desempenho dos dados que foram coletados.

A principal contribuição deste trabalho e que difere das demais ferramentas existentes, foi o desenvolvimento e a implementação da coleta de forma hierárquica. Como a técnica de *trace* foi a escolhida para o desenvolvimento deste trabalho, foi possível levar em consideração o nível de abstração utilizado pelo usuário, permitindo dessa forma determinar o nível hierárquico das execuções.

Para exemplificar o uso da coleta hierárquica, foi implementado a coleta nas operações da biblioteca do MPI e nas classes e métodos do *framework* OOPS por fornecer abstrações de alto nível. A implementação feita na coleta hierárquica não é invasiva no código do usuário, pois não exige nenhuma modificação no código fonte da aplicação. É necessário apenas que sejam adicionadas diretivas na compilação da aplicação quando se deseja coletar dados do MPI (`_HIERATRACEMPI`) ou do OOPS (`_HIERATRACEOOPS`).

A estruturação e a padronização dos dados coletados foi feita através da definição de um DTD, o que possibilita a reutilização dos dados da coleta hierárquica em outras aplicações. Atualmente esse DTD define a coleta hierárquica para as bibliotecas MPI e OOPS, podendo ser estendido para outras bibliotecas, o que demonstra a preocupação de extensibilidade deste projeto.

Os dados coletados são armazenados em arquivos no formato XML o que permitiu a estruturação de forma hierárquica dos dados nos arquivos. O nível hierárquico é definido em tempo de execução. No arquivo XML essa hierarquia é definida pelos elementos `hieraMPI` quando são coletados atributos das operações do MPI e `hieraOOPS` quando são coletados atributos de classes e métodos do OOPS.

Para o módulo de transformação dos dados, foi definido um grafo que permite ligações entre operações de um mesmo processador e entre operações que estão em diferentes processadores, o que possibilita a realização de análise dos dados armazenados. Dois exemplos de como realizar a análise dos dados no grafo foi mostrado através da navegação entre os nós do grafo: uma entre as operações de um mesmo processador e outra entre operações “parceiras” que podem estar em qualquer outro processador. Outras formas de percurso para a realização da análise dos dados podem ser implementadas.

Para mostrar o uso do grafo, foi implementada uma biblioteca orientada a objetos que transforma os dados coletados das operações MPI e classes e métodos OOPS em nós que compõem o grafo. Os nós no grafo foram criados para permitir aplicar algum tipo de medida para realizar a análise dos dados inseridos no grafo. Essa biblioteca possui 71 tipos diferentes de nós das operações do MPI e 54 tipos de nós para os métodos do OOPS. Novas bibliotecas po-

dem ser adicionadas através do uso de conceitos de orientação a objetos incorporadas nessa implementação.

Outra contribuição deste trabalho foi a aplicação do processo de *Data Warehousing* para auxiliar a análise de desempenho em sistemas paralelos através da definição dos módulos de acesso, transformação, base multidimensional e distribuição. Nesse processo os dados hierárquicos são transformados em dados multidimensionais.

Foi criado um *Data Mart* que tem por objetivo realizar a análise de desempenho de sistemas paralelos para usuários e desenvolvedores desses sistemas. Esse *Data Mart* é composto de dois modelos multidimensionais um para o MPI e outro para o OOPS. O modelo multidimensional do MPI totaliza 6 dimensões, 19 atributos de granularidade e 7 sumarizadores. O modelo multidimensional do OOPS totaliza 6 dimensões, 20 atributos de granularidade e 4 sumarizadores.

O uso do processo de *Data Warehousing* para análise de desempenho de sistemas paralelos foi exemplificado através da ferramenta HieraOLAP que implementa os módulos de acesso, transformação, base multidimensional e distribuição. O módulo de acesso possibilita gerar uma base histórica das execuções das aplicações. No módulo de transformação novos dados derivados foram incorporados na base multidimensional. O módulo de distribuição possibilita que análises sejam feitas através do uso de operações de consulta OLAP. Essas operações são diminuição de granularidade, aumento da granularidade e rotação de dimensões.

Com as contribuições apresentadas, foi possível observar que o trabalho apresentado segue os passos do processo de análise de desempenho de sistemas paralelos. Para a coleta dos dados que é o primeiro passo no processo, foi definido e implementado o módulo HieraCollector que utilizou a técnica de *trace* em rotinas de biblioteca; o módulo de transformação HieraTransform permitiu o uso de medidas, como contagem, nos dados coletados para a realização da análise de desempenho. A ferramenta HieraOLAP foi desenvolvida para auxiliar o usuário na análise de desempenho dos dados utilizando o processo de *Data Warehousing*.

6.2 Trabalhos Futuros

Nesta seção são apresentadas algumas sugestões para trabalhos futuros, que são:

- Realizar a coleta de forma seletiva, de modo que o usuário possa selecionar partes do programa para coletar dados que sejam mais relevantes.
- Selecionar pelo programador os níveis de abstração que terão suas execuções monitoradas através de chamadas de rotinas ou inserção de diretivas no código.

- Integração da ferramenta de coleta e análise hierárquica em um único ambiente de desenvolvimento de aplicações paralelas. A contribuição deste trabalho consiste no auxílio do processo de depuração do código fonte paralelo para localização de bugs ou identificação de gargalos no sistema.
- Aplicação de técnica para análise de desempenho que não seja dependente da experiência do usuário como é o caso do processo de *Data Warehousing*, por exemplo, *Data Mining* ou visualização de dados. O desafio deste trabalho consiste em investigar quais entre as diversas técnicas de *Data Mining* ou visual, podem ser usadas no processo de análise de programas paralelos para que o desenvolvedor desses sistemas não tenha que ser um especialista. Dessa forma, a análise pode ser feita de maneira automática sem a intervenção do usuário inexperiente.
- Estender a coleta hierárquica para outras bibliotecas orientadas a objeto, como o MPI++ ou para outras bibliotecas não orientada a objetos como o PVM.

Referências Bibliográficas

- [1] Thomas L. Sterling, Daniel Savarese, Donald J. Becker, John E. Dorband, Udaya A. Ranawake, and Charles V. PackerT. BEOWULF: A parallel Workstation for Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14, Oconomowoc, WI, 1995.
- [2] D. B. Skillicorn. Motivating Computational Grids. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 401, Washington, DC, USA, 2002. IEEE Computer Society.
- [3] Michael T. Heath. *Visualization of Parallel and Distributed Systems*, chapter 31, pages 897–916. McGraw-Hill, 1996.
- [4] Raj Jain. *The Art of Computers Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. John Wiley & Sons, Inc., first edition, 1991.
- [5] Lei Hu and Ian Gorton. Performance Evaluation for Parallel Systems: A Survey. Technical report, Department of Computer Systems, University of NSW, Sydney, Australia, October 1997.
- [6] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. 381 pag.
- [7] Francisco Aparecido Rodrigues and Gonzalo Travieso. Técnicas de Orientação ao Objeto para Computação Científica Paralela. Master's thesis, Instituto de Física de São Carlos - Universidade de São Paulo, April 2004.
- [8] L. S. Blackford, J. J. Dongarra, and R. C. Whaley. *ScaLAPACK Users' Guide*. Siam—Society for Industrial and Applied Mathematics, May 1997. ISBN 0-89871-397-8.
- [9] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed

- Memory Computers - Design Issues and Performance. Technical report, University of Tennessee, Knoxville, TN 37996, USA, 1995.
- [10] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers – Design Issues and Performance. *Computer Physics Communication – High-Performance Computing in Science*, 97(1–2):1–15, August 2 1996.
- [11] R. C Whaley. Basic Linear Algebra Communication Subprograms: Analysis and Implementation Across Multiple Parallel Architectures. LAPACK Working Note 73, Inst-UT-CS, Knoxville, TN, USA, May 1994.
- [12] L. Dongarra and R. C. Walker. User’s Guide to the BLACS v1.1. In *Computer Science Dept. Technical Report*, pages 95–281, University of Tennessee, 1995.
- [13] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David W. Walker, and R. Clinton Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In *PARA '95: Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science*, volume 1041, pages 107–114, London, UK, 1995. Springer-Verlag.
- [14] Jaeyoung Choi, Jack Dongarra, and David W. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subprograms. *Concurrency - Practice and Experience*, 8(7):517–535, September 1996.
- [15] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine. A Users’ Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.
- [16] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, Fall-Winter 1994.
- [17] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 12 1995.
- [18] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

- [19] William D. Gropp and Ewing Lusk. *User's Guide for MPICH, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996.
- [20] The LAM/MPI Team. *LAM/MPI User's Guide*, September 26 2004.
- [21] Eloiza Sonoda and Gonzalo Travieso. The OOPS Framework: High Level Abstractions for the Development of Parallel Scientific Applications. In *OOPSLA'06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 659–660, New York, NY, USA, 2006. ACM Press.
- [22] W. H. Inmon. *Como Construir o Data Warehouse*. Editora Campus, 1997.
- [23] Sanjay Goil and Alok Choudhary. A Parallel Scalable Infrastructure for OLAP and Data Mining. In *Proceedings of the 1999 International Symposium on Database Engineering and Applications*, pages 178–186, Montreal, Canada, August 1999. IEEE Computer Society.
- [24] Jay Fenlason and Richard Stallman. *GNU gprof: The GNU Profiler*, September 1997.
- [25] J. C. Yan. Performance Tuning with AIMS — An Automated Instrumentation and Monitoring System for Multicomputers. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, volume II, pages 625–633, January 1994.
- [26] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, New York, NY, USA, 1995. ACM Press.
- [27] Ruth A. Aydt. *An Informal Guide to Using Pablo*. Department of Computer Science - University of Illinois, May 12 1992.
- [28] Barton P. Miller. Paradyn Parallel Performance Tools. <http://www.paradyn.org>.
- [29] Kei-Chun Li and Kang Zhang. Supporting Scalable Performance Monitoring and Analysis of Parallel Programs. *Journal of Supercomputing*, 13(1):5–31, Jan-Feb 1999.
- [30] Luiz A. DeRose, Mario Pantano, Daniel A. Reed, and Jeffrey S. Vetter. Performance Issues in Parallel Processing Systems. In *Performance Evaluation: Origins and Direction*, pages 141–159, London, UK, 2000. Springer-Verlag.

- [31] Masoud Mansouri-Samani and Morris Sloman. Monitoring Distributed Systems - A Survey. Technical Report DOC92/93, Imperial College, Longon, UK, September 1992.
- [32] Daniel A. Reed. Experimental Analysis of Parallel Systems: Techniques and Open Problems. In *Proceedings of the 7th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, pages 25–51, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.
- [33] Edu Metz and Raimondas Lencevicius. Performance Data Collection: Hybrid Approach. In *Proceedings of the Second International Workshop on Dynamic Analysis*, pages 48–51, Edinburg, Scotland, May 2004. <http://www.cs.virginia.edu/woda2004/> (visitada Jan/2005).
- [34] Michael T. Heath, Allen D. Malony, and Diane T. Rover. The Visual Display of Parallel Performance Data. *IEEE Computer*, 28(11):21–28, November 1995.
- [35] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, June 1982.
- [36] J. Chassin de Kergommeaux, É. Maillet, and J. M. Vincent. *Monitoring Parallel Programs for Performance Tuning in Cluster Environments*, volume 5, chapter 6, pages 137–157. Nova Science Publishers, Incorporated, January 2001.
- [37] Daniel A. Reed. Performance Instrumentation Techniques for Parallel Systems. In *Performance/SIGMETRICS Tutorials*, pages 463–490, 1993.
- [38] Fumihiko Ino, Yuki Kanbe, Masao Okita, and Kenichi Hagihara. Trace Reduction for Performance Improvement Assessment of Message Passing Parallel Programs. *Systems and Computers in Japan*, 37(5):1–13, May 2006.
- [39] Oleg Y. Nickolayev, Philip C. Roth, and Daniel A. Reed. Real-Time Statistical Clustering for Event Trace Reduction. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):144–159, Summer 1997.
- [40] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

- [41] J. C. Yan and M. A. Schmidt. Constructing Space-Time Views from Fixed Size Trace Files – Getting the Best of Both Worlds. In E. H. D’Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, editors, *Proceedings of the Conference Parallel Computing: Fundamentals, Applications and New Directions*, volume 12, pages 633–640, Amsterdam, September 1998. Elsevier, North-Holland.
- [42] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.
- [43] Jerry Yan, Sekhar R. Sarukkai, and Pankaj Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software - Practice and Experience*, 25(4):429–461, April 1995.
- [44] Jeffrey K. Hollingsworth. Critical Path Profiling of Message Passing and Shared-Memory Programs. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):1029–1040, October 1998.
- [45] Daniel A. Reed, Ruth A. Aydt, Luiz DeRose, Celso L. Mendes, Randy L. Ribler, Eric Shaffer, Huseyin Simitci, Jeffrey S. Vetter, Daniel R. Wells, Shannon Whitmore, and Ying Zhang. Performance Analysis of Parallel Systems: Approaches and Open Problems. In *Proceedings of the Joint Symposium on Parallel Processing (JSPP)*, pages 239–256, Jun 1998.
- [46] Michael T. Heath, Allen D. Malony, and Diane T. Rover. Parallel Performance Visualization: from Practice to Theory. *IEEE Parallel and Distributed Technology*, 3(4):44–60, 1995.
- [47] John T. Stasko and Eileen Kraemer. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18(1):258–264, 1993.
- [48] Abdul Waheed and Diane T. Rover. Performance Visualization of Parallel Programs. In *Proceedings of the 4th Conference on Visualization*, pages 174–181, October 1993.
- [49] Steven T. Hackstadt and Allen D. Malony. Visualizing Parallel Programs and Performance. *IEEE Computer Graphics and Applications*, 15(4):12–14, July 1995.
- [50] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

- [51] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. Wiley, 2nd Edition edition, April 2002.
- [52] Sanjay Goil and Alok N. Choudhary. High Performance OLAP and Data Mining on Parallel Computers. *Data Mining and Knowledge Discovery*, 1(4):391–417, December 1997.
- [53] Sanjay Goil and Alok N. Choudhary. High Performance Data Mining using Data Cubes on Parallel Computers. In *IPPS: 12th International Parallel Processing Symposium*, pages 548–555. IEEE Computer Society Press, March 1998.
- [54] Anindya Datta, Debra VanderMeer, Krithi Ramamritham, and Bongki Moon. Applying Parallel Processing Techniques in Data Warehousing and OLAP. Technical Report Tr2001-15, Chutney Technologies, 2001.
- [55] Olivier Teste. Towards Conceptual Multidimensional Design in Decision Support Systems. In *Proceedings of the 5th East-European Conference on Advances in Databases and Information Systems*, pages 77–88, September 2001.
- [56] Shirley Browne, Jack Dongarra, and Kevin London. Review of Performance Analysis Tools for MPI Parallel Programs. *NHSE Review*, 3(1), 1998.
- [57] Shirley Moore, David Cronk, Kevin London, and Jack Dongarra. Review of Performance Analysis Tools fo MPI Parallel Programs. In *8th European PVM/MPI Users' Group Meeting*, pages 241–248, 2001.
- [58] Sameer Shende, Allen Malony, Janice Cuny, Kathie Lindlan, Peter Beckman, and Steve Karmesin. *TAU Portable Profiling and Tracing Toolkit User's Guide*, 2004. <http://www.cs.uoregon.edu/research/paracomp/tau>.
- [59] Michael Collete, Bob Corey, and John Johnson. High Performance Tools & Technologies. UCLR-TR-209289, Lawrence Livermore National Laboratory, US. Dep. of Energy, December 2004.
- [60] Allen D. Malony and Janice E. Cuny. Program Analysis Toolkit. <http://www.cs.uoregon.edu/research/paracomp/proj/>.
- [61] Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende, Bernd Mohr, Reid Rivenburgh, and Craig Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In *Proceedings of the 2000 ACM/IEEE*

- Conference on Supercomputing*, number 49. IEEE Computer Society, 2000. <http://citeseer.ist.psu.edu/lindlan00tool.html>.
- [62] Robert Bell, Allend D. Malony, and Sameer Shende. ParaProf: A Portable, Extensible and Scalable Tool for Parallel Performance Profile Analysis. In *Proc. Europar 2003 Conference*, pages 17–26, Springer 2003.
- [63] AIMS - Automated Instrumentation and Monitoring System. NASA Ames Research Center. <http://www.nas.nasa.gov/Groups/Tools/Projects/AIMS/>.
- [64] J. C. Yan, M. Schmidt, and C. Schulbach. The Automated Instrumentation and Monitoring System (AIMS). NAS Technical Report, August 1999. Version 3.7 User's Guide.
- [65] J. Mellor-Crummey, R. Fowler, and G. Marin. HPCVIEW: A Tool for Top-down Analysis of Node Performance. *The Journal of Supercomputing*, 23(1):81–104, 2002.
- [66] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.
- [67] Rick Kufrin. PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux. In *The 6th International Conference on Linux Clusters: The HPC Revolution*, Chapel Hill, NC, April 2005.
- [68] Dong Ahn, Rick Kufrin, Anand Raghuraman, and Jun-Hyuk Seo. Perfsuite. <http://perfsuite.ncsa.uiuc.edu/> (visitada Fev/2007).
- [69] Sandia National Laboratories. The Visual Profiler, 2002. <http://aros.ca.sandia.gov/~cljanss/perf/vprof/index.html> (visitada Fev/2007).
- [70] Philip J. Mucci. *Dynaprof Users Guide*, release 0.8 edition, November 2002. http://www.cs.utk.edu/~mucci/dynaprof/dynaprof.html#_Toc15965915 (visitada Fev/2007).
- [71] Simone Sbaraglia. Simulation Guided Memory Analyzer - SiGMA. Technical report, Advanced Computing Technology Center - IBM Research, July 2004.
- [72] Anthony Chan, William Gropp, and Ewing Lusk. User's Guide for MPE Extensions for MPI Programs. Technical report, Argonne National Laboratory - University of Chicago, 1998.

- [73] Michael T. Heath and Jennifer E. Finger. *ParaGraph: A Performance Visualization Tool for MPI*, August 31 2003.
- [74] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A User's Guide to PICL: A Portable Instrumented Communication Library. Technical Report, Oak Ridge National Laboratory, June 1991.
- [75] Barton P. Miller, Jeffrey K. Hollingsworth, and Mark D. Callaghan. The Paradyn Parallel Performance Tools and PVM. In J.J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*. SIAM Press, 1994.
- [76] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
- [77] Karen L. Karavanic, Jussi Myllymaki, Miron Livny, and Barton P. Miller. Integrated Visualization of Parallel Program Performance Data. *Parallel Computing*, 23(1–2):181–198, April 1997.
- [78] Barton P. Miller. *Paradyn Parallel Performance Tools - VisiLib Programmer's Guide*, release 4.2 edition, March 2005.
- [79] Jesús Labarta, Judit Gimenez, Jordi Caubet, and Francesc Escale. *PARAVER: Parallel Program Visualization and Analysis Tool*. European Center for Parallelism of Barcelona, October 2001. Version 3.1.
- [80] Paraver: The Flexible Analysis Tool. European Center for Parallelism of Barcelona - Technical University of Catalonia. <http://www.cepba.upc.es/paraver>.
- [81] Virginia Herrarte and Ewing Lusk. Studying Parallel Program Behavior with Upshot. Technical Report ANL–91-15, Argonne National Laboratory, 1991.
- [82] Edward Karrels and Ewing Lusk. Performance Analysis of MPI Programs. In Jack Dongarra and Bernard Tourancheau, editors, *Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing*, pages 195–200. SIAM Publications, 1994.
- [83] Nupshot for Trace Visualization, May 1997. <http://web.utk.edu/course/image/nupshot>.
- [84] Omer Zaki, William Gropp, Ewing Lusk, and Deborah Swider. Toward Scalable Performance Visualization with Jumpshot. *International Journal of Supercomputer Applications and High Performance Computing*, 13(2):277–288, Fall 1999.

- [85] Tunig Perfomance Analysis - Vampir/Vampirtrace. <http://www.pallas.com/e/products/vampir/>.
- [86] Herbert Schildt. *C - The Complete Reference*. McGraw-Hill, Fourth edition, 2000. 889 pag.
- [87] W3C: World Wide Web Consortium. <http://www.w3c.org>.
- [88] Libxml: The XML C parser and toolkit of Gnome. <http://xmlsoft.org>.
- [89] Harvey M. Deitel, Paul J. Deitel, T. R. Nieto, Ted Lin, and P. Sadhu. *XML How to Program*. Prentice Hall, first edition, 2001.
- [90] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
- [91] G. Travieso. Introdução à Orientação a Objetos em C++: Notas de aula. Instituto de Física, USP São Carlos, 2004.
- [92] Eloiza Helena Sonoda. *OOPS - Object-Oriented Parallel System: Um Framework de Classes para Programação Científica Paralela*. PhD thesis, Universidade de São Paulo - IFSC, Março 2006.
- [93] Bjarne Stroustrup. *C++: The Programming Language*. Addison Wesley, Third edition, 2000. 928 pág.
- [94] Meilir Page-Jones. *Fundamentos do Desenho Orientado a Objeto com UML*. Makron Books, Primeira edition, 2001. 470 pág.
- [95] Rudolf Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Inf.*, 1(4):290–306, December 1972.
- [96] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison Wesley Longman, Inc., 1999.
- [97] Gentoo linux. <http://www.gentoo.org> (visitada em 04/2007).
- [98] Steve Plimpton. Fast Parallel Algorithms for Short-range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, March 1995.

- [99] Jarrod Hollingworth, Dan Butterfield, Bob Swart, and Jamie Allsop. *C++ Builder 5: Developer's Guide*. Sams; Pap/Cdr edition, December 2000.
- [100] Kevin Mitchell. Paradox 4.x File Formats - Revision 1. <http://mediasrv.ns.ac.yu/extra/fileformat/database/db/paradox4.txt>, May 11 1996.