AVALIAÇÃO DE DESEMPENHO DE SISTEMAS PARALELOS BASEADA EM DESCRIÇÃO SIMPLIFICADA DO PROGRAMA E DA ARQUITETURA

Thatyana de Faria Piola

Dissertação apresentada ao Instituto de Física de São Carlos, Universidade de São Paulo, para obtenção do título de Mestre em Ciências "Física Aplicada - Opção Física Computacional"

Orientador: *Prof. Dr. Gonzalo Travieso*

SÃO CARLOS

2002

Piola, Thatyana de Faria

Avaliação de Desempenho de Sistemas Paralelos Baseada em Descrição Simplificada do Programa e da Arquitetura/ Thatyana de Faria Piola

São Carlos, 2002.

102 p.

Dissertação (Mestrado) - Instituto de Física de São Carlos, 2002

Orientador: Prof. Dr. Gonzalo Travieso.

1. Paralelismo. 2. Simulação. 3.Linguagem. I.Título.

Aos meus pais Oswaldo e Ana Isabel
pelo amor, apoio e compreensão.

Agradecimentos

Ao Prof. Dr. Gonzalo, pela amizade, confiança, paciência e pela grande competência na orientação do trabalho, além do constante otimismo.

Ao Enzo, pelo amor, carinho e compreensão. Ao qual me faltam palavras que agradeçam todo apoio e companheirismo.

Aos meus irmãos Thiago e Thaísa pela compreensão, e à Luciana. Aos meus pais Oswaldo e Ana Isabel pelas oportunidades que me propiciaram

Agradeço à Patrícia e Marilde pelo incentivo, apoio e confiança.

Agradeço à Paulino e Francisco pelas discussões, companheirismo e por terem se revelado grandes amigos.

Aos amigos do grupo: Elaine Patricia, companheira e confidente mesmo longe, à Elô, Zem e André, pela amizade. Agradeço a todas as pessoas que direta ou indiretamente contribuíram para a realização do trabalho.

Agradeço à FAPESP pelo suporte financeiro.

Sumário

1 Introdução **1**

2 Desempenho de Sistemas Paralelos **4**

2.1 Processamento de Alto Desempenho 4

2.2 Modelo de Máquina Paralela 6

2.3 Desempenho 8

2.3.1 Métricas de Desempenho 9

2.4 Métodos de Previsão de Desempenho 11

2.4.1 Lei de Amdahl 12

2.4.2 Método de Foster 13

2.4.3 Método de Anglano 15

2.4.4 Método de Liang 18

2.4.5 Método de Fahringer e Zima 19

2.4.6 Método de Parashar e Hariri 21

3 Descrição do Método Proposto **24**

3.1 Ping-pong 25

3.2 Soma de Matrizes 28

4 Linguagem de Protótipo **32**

4.1 Sintaxe da Linguagem 33

4.2 Desenvolvimento do Tradutor 42

4.2.1 Analisador Sintático - BISON 43
4.2.2 Analisador Léxico - FLEX 45
4.3 Compilação 47

5 Simulador 50
5.1 Descrição do Simulador Implementado 54
5.1.1 Parâmetros 54
5.1.2 Organização 58
5.2 Funcionamento do Simulador 60

6 Resultados 66
6.1 Ping-Pong 67
6.2 Anel 68
6.3 Diferenças Finitas 73
6.4 Mandelbrot 74

7 Conclusão e Trabalhos Futuros 81

A Regras do Bison 88

B Regras do Flex 98

Lista de Figuras

2.1 Tendências de desempenho 5
2.2 Multicomputador 7
2.3 Metodologia de previsão de desempenho de Anglano . . 17

3.1 Ping-Pong 27
3.2 Programa ping-pong escrito em MPI 27
3.3 Descrição simplificada do programa Ping-pong 27
3.4 Programa Soma de matrizes escrito em MPI 29
3.5 Descrição simplificada do programa soma de matrizes . 29
3.6 Funcionamento do Tradutor 31

4.1 Programa produto escalar escrito em C e MPI 32
4.2 Descrição simplificada do programa produto escalar . . 33
4.3 Regras iniciais da linguagem 33
4.4 Expressões permitidas na Linguagem 34
4.5 Definição de uma função 35
4.6 Definição de Variação 35
4.7 Lista de comandos definidos 36
4.8 Definição do comando IF-ELSE 37
4.9 Definição do comando WHILE 37
4.10 Definição do comando FOR 37
4.11 Definição da operação Compute 38
4.12 Definição das operações de Comunicação Ponto-a-Ponto 39

4.13 Definição das operações de Comunicação Coletiva . . . 41
4.14 Diagrama de Execução 44
4.15 Regras gramaticais em formato Bison 45
4.16 Regras do analisador léxico - Flex 47
4.17 Código antes da compilação 48
4.18 Código do programa ping-pong compilado 49

5.1 Ligação entre a rede e os processadores 54
5.2 Tempo de transmissão 56
5.3 Tempo de transmissão - Mensagens menores de 1024 *bytes* 56
5.4 Tempo de transmissão - Mensagens maiores de 1024 e menores de 5000 *bytes* 57
5.5 Tempo de transmissão - Mensagens maiores de 5000 *bytes* 58
5.6 Modelo do Simulador de Rede 58
5.7 Implementação da função *send* definida na linguagem . 61
5.8 While com variação 63
5.9 Broadcast no código da figura 5.8 64
5.10 Função Broadcast 64

6.1 Compilação dos programas 66
6.2 Código do programa Ping-pong 68
6.3 Tempos de transmissão do programa ping-pong 69
6.4 Anel 69
6.5 Código do programa Anel 70
6.6 Anel com total de processos pares e ímpares 73
6.7 Tempos de transmissão do programa Anel 74
6.8 Código do programa Diferenças Finitas 75
6.9 Diferenças Finitas 76

6.10 Tempos de transmissão do programa Diferenças Finitas 76
6.11 Mandelbrot 78
6.12 Código do programa *Mandelbrot* 78
6.13 Tempos de transmissão do programa Mandelbrot 80

Lista de Tabelas

3.1 Tempos de execução do programa Ping-pong 28
3.2 Tempos de execução do programa soma de matrizes . . 30

5.1 Tempo de transmissão das mensagens 55

6.1 Comparação dos tempos de execução - Ping-pong . . . 71
6.2 Tempos de transmissão do programa Anel 72
6.3 Tempos de transmissão do programa Diferenças Finitas 77
6.4 Tempos de transmissão do programa Mandelbrot 79

Resumo

Este trabalho apresenta o desenvolvimento de uma linguagem para descrição simplificada de algoritmos paralelos, um tradutor e um simulador de rede. Com vistas à avaliação de desempenho, a linguagem permite uma prototipagem fácil e abrangente para descrever vários tipos de programas paralelos, envolvendo estruturas de controle, repetição e as partes de comunicação e computação. Para interpretar o código descrito na linguagem, foi desenvolvido um tradutor que traduz o código simplificado descrito pela linguagem desenvolvida, gerando código C++. O simulador de rede computa os tempos envolvidos nas comunicações. O simulador interage com o código gerado pelo tradutor. Para validação foram utilizados alguns programas de testes e o resultado da simulação comparado com o da execução em um *cluster* de computadores pessoais.

Abstract

This work presents the development of a language for simplified description of parallel algorithms, a language translator and a network simulator. The language aims to allow an easy parallel program prototyping for performance evaluation purposes and aims to be enough comprehensive to describe several kinds of parallel programs including execution control structures, repetition, communication and computation parts. A translator that translates the simplified code described by the language was developed, producing C++ code. A network simulator computes the communication times. The simulator interacts with the code produced by the translator. For validation some tests programs were used and the simulation results compared with the execution times in a cluster of personal computers.

Capítulo 1

Introdução

A computação paralela tem emergido como uma ferramenta indispensável para a resolução de problemas em muitas áreas científicas, sendo paralelismo em muitos casos, o único caminho para conseguir o desempenho requerido, que é mais elevado do que o possível com um único processador.

O principal objetivo de sistemas paralelos é o desempenho, uma vez que o crescimento no desempenho dos processadores ainda não é suficiente para algumas aplicações, sendo necessário a utilização de sistemas com diversos deles para obter um bom desempenho.

Devido a isto, a análise de desempenho é uma parte muito importante em sistemas paralelos, para avaliar diferentes estruturas de aplicações paralelas; prever o desempenho de uma aplicação em um dado sistema; identificar as características que têm impacto significante no tempo de execução da aplicação; avaliar o desempenho de um sistema, um componente do sistema ou mesmo de uma aplicação.

Estudos de desempenho exploram alternativas, em todos os aspectos de sistemas paralelos, visando melhorar o desempenho de sistemas disponíveis e seus componentes. Ao avaliar o desempenho

de um sistema ou de uma aplicação, buscam-se dentre outros a identificação de gargalos no sistema e a previsão de desempenho de uma determinada aplicação em um dado sistema paralelo.

A análise em um código pode ser feita através de análise preditiva, onde são desenvolvidos modelos matemáticos para o algoritmo paralelo antes de sua implementação, ou mesmo usando simulação simplificada. Um problema que surge com o desenvolvimento de modelos matemáticos é que eles raramente consideram a estrutura da rede.

Dispondo-se do programa desenvolvido, poder-se-ia utilizar simuladores detalhados para avaliar qual o efeito de uma mudança na máquina, por exemplo, avaliar se compensa para a aplicação mudar a rede de interconexão. O uso desses simuladores detalhados além de exigir muitos recursos computacionais, requer o código pronto.

Como proposta intermediária para solucionar os problemas dos modelos matemáticos e dos simuladores detalhados, este trabalho se dedica ao desenvolvimento de uma linguagem para descrição simplificada de algoritmos paralelos, um tradutor e um simulador de rede a serem usados na previsão de desempenho. A linguagem visa permitir uma prototipagem simples, buscando possibilitar o desenvolvimento de uma descrição de alto nível de um algoritmo paralelo. Um tradutor é utilizado para traduzir a descrição simplificada do código escrito utilizando a linguagem desenvolvida em um código C++. Um simulador simplificado de rede é usado para testar o funcionamento do sistema, computando os tempos envolvidos na comunicação. A interface entre o tradutor e o simulador foi feita de forma a facilitar o encaixe modular de outros simuladores.

No capítulo 2 apresentamos características de processamento

paralelo e o modelo de máquina paralela utilizado (subseção 2.2). Na subseção 2.3.1 descrevemos algumas métricas de desempenho, como eficiência, *speedup* e análise de escalabilidade. Apresentamos na seção 2.4 alguns métodos de previsão de desempenho, como por exemplo, a lei de Amdahl, o modelo de Foster, de Anglano entre outros.

Apresentamos no capítulo 3 o método desenvolvido que utiliza uma representação simplificada do programa que terá sua execução simulada para gerar dados sobre os instantes de início e término das operações de comunicação. Nesse capítulo ainda apresentamos alguns exemplos do uso do método.

No capítulo 4 descrevemos detalhadamente a linguagem de protótipos, assim como sua sintaxe. Descrevemos o desenvolvimento do tradutor utilizado para tradução da descrição simplificada do código utilizando a linguagem, e apresentamos de forma simplificada os analisadores sintático e léxico (Bison na subseção 4.2.1 e Flex na subseção 4.2.2, respectivamente). O processo de compilação do código é mostrado na seção 4.3.

Em seguida, no capítulo 5 apresentamos a descrição do simulador desenvolvido no trabalho. Na descrição do simulador, apresentamos conceitos importantes na operação do simulador, e descrevemos seu funcionamento na seção 5.2.

Os resultados do trabalho são apresentados no capítulo 6, onde são descritos os testes realizados utilizando o *cluster* descrito no capítulo 6 e no simulador desenvolvido no trabalho. Apresentamos logo em seguida, neste mesmo capítulo os tempos obtidos nos testes realizados.

No capítulo 7 apresentamos a conclusão e sugestões para trabalhos futuros.

Capítulo 2

Desempenho de Sistemas Paralelos

2.1 Processamento de Alto Desempenho

O crescimento explosivo no desempenho e capacidade de sistemas de computadores tem sido apreciável, devido ao avanço na tecnologia VLSI (*Very Large Scale Integration*) que permite redução do tempo requerido para realizar a operação mais primitiva em um processador (*período de clock*) e a integração de um número maior de componentes em um *chip*. A arquitetura de computadores transforma o potencial da tecnologia em desempenho e capacidade cada vez maiores do sistema de computadores. O paralelismo exerce um papel central nessa história.

Arquitetura de computadores, tecnologia e aplicações desenvolvem-se juntas e têm fortes interações. A arquitetura paralela não é exceção e inclui uma nova dimensão que é o número de processadores. Considerando as características de desempenho individual do processador, seu crescimento ao longo do tempo para várias classes de computadores pode ser visto na figura 2.1 de acordo com o

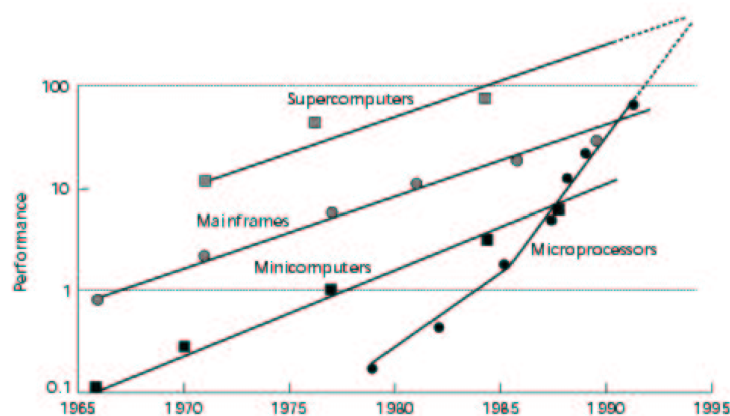

Figura 2.1: Tendências de desempenho de micros, minicomputadores, *mainframes* e supercomputadores ao longo do tempo. Fonte: Culler [1].

levantamento feito por Hennessy e Jouppi (1991) [1], onde as linhas tracejadas representam uma simples extrapolação das tendências.

Em paralelismo um grande volume de recursos significa que mais operações podem ser realizadas de uma vez em paralelo. Arquiteturas de computadores paralelos devem organizar estes recursos para que trabalhem bem juntos.

Segundo Foster em [2], computadores paralelos são um conjunto de processadores que são capazes de trabalhar cooperativamente para resolver um problema computacional. Eles são interessantes porque oferecem potencial para concentrar recursos computacionais, processadores, memória e largura de banda de entrada/saída, em grandes problemas computacionais.

O desempenho de microprocessadores tem aumentado a uma taxa de 50% ao ano. As vantagens do uso de processadores pequenos, baratos, de baixa potência e produzidos em massa como blocos de construção para sistemas de computadores com muitos processadores são intuitivamente lógicas. Uma máquina paralela com centenas deles pode satisfazer as necessidades que seriam atendidas

por um processador individual em cerca de 10 anos, se mantida a taxa de crescimento atual.

A rápida difusão de computadores em usos comerciais, científicos e educacionais deve-se à primeira padronização de um modelo de máquina simples, o *computador de von Neumann*. Este modelo simples permite aos programadores projetarem algoritmos para uma máquina abstrata de von Neumann e não para uma máquina específica. Um desenvolvimento semelhante era necessário para a difusão de arquiteturas paralelas. Examinando a evolução das principais arquiteturas paralelas, notamos uma recente convergência para máquinas escaláveis através de uma máquina paralela genérica [1].

2.2 Modelo de Máquina Paralela

Um modelo de máquina paralela genérica é o *multicomputador*, visto na figura 2.2. Esse modelo é simples por facilitar a compreensão e programação, e realista pois assegura que programas desenvolvidos para ele executem com eficiência razoável em computadores paralelos reais.

O multicomputador compreende um número de nós (ou computadores de von Neumann), ligados por uma rede de interconexão, onde cada computador executa seu próprio programa. Esse programa pode acessar a memória local ou enviar e receber mensagens através da rede. As mensagens são usadas para comunicação com outros nós, ou para ler e escrever em memórias remotas (localizadas em outros nós). Um atributo do modelo de multicomputador é que os acessos à memória local (mesmo nó) são menos caros do que os acessos à memória remota. Assim, leitura e escrita são menos

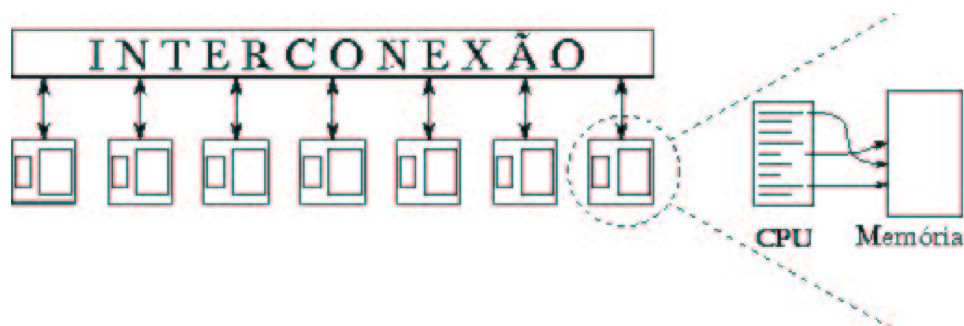

Figura 2.2: Multicomputador

custosas do que envio e recepção. Portanto, é desejável que acessos a dados locais sejam mais freqüentes do que a dados remotos.

Na arquitetura multicomputador, o custo de enviar uma mensagem entre duas tarefas localizadas em processadores diferentes pode ser representado por dois parâmetros: o tempo de *startup* da mensagem (t_s, tempo requerido para iniciar a comunicação) e o tempo de transmissão de um *byte* (t_w, que é determinado pela largura de banda do canal de comunicação ligando os processadores fonte e destino). O tempo requerido para enviar uma mensagem pode ser visto na equação (2.1), onde *d* indica o número de bytes na mensagem.

$$T_{msg} = t_s + t_w d \tag{2.1}$$

O modelo do multicomputador pode falhar se a rede de interconexão dos computadores tem propriedades diferentes do ideal, particularmente se uma aplicação gera muitas mensagens. Por exemplo, dois processadores podem necessitar enviar dados sobre o mesmo fio ao mesmo tempo. Tipicamente, somente uma mensagem poderá ser transmitida simultaneamente, assim a outra mensagem será atrasada. Portanto, basta imaginar dois processadores compartilhando a largura de banda disponível na rede. Contenção

adicional pode ocorrer se as mensagens colidem e precisam ser retransmitidas.

O impacto de competição por largura de banda é intenso em algoritmos em que todos os processadores estão enviando e recebendo mensagens ao mesmo tempo e no qual processadores não podem continuar com outra computação enquanto estão esperando.

2.3 Desempenho

Desenvolvimento de aplicações eficientes para computação paralela de alto desempenho é um processo não-trivial e requer profundo entendimento não somente da aplicação mas também do ambiente computacional. Ferramentas de avaliação permitem a um desenvolvedor visualizar os efeitos de várias alternativas de projeto. Ferramentas de previsão de desempenho fornecem um meio mais prático para avaliar alternativas de projeto disponíveis e tomar decisões no projeto.

A computação paralela tem emergido como uma ferramenta indispensável para resolver muitos problemas científicos, sendo paralelismo em muitos casos, o único caminho para conseguir o desempenho requerido, que é mais elevado do que o que é possível com um único processador. Estudos de desempenho têm explorado alternativas para todos os aspectos de sistemas paralelos visando melhorar o desempenho de sistemas disponíveis e de seus componentes individuais.

O desempenho de um programa paralelo é uma questão complexa. Devemos considerar, além do tempo de execução e escalabilidade, os mecanismos pelos quais os dados são gerados, armazenados e transmitidos pela rede, movidos para e do disco, e passados entre

diferentes estágios de uma computação. As métricas pelas quais medimos o desempenho podem ser, entre outras: tempo de execução, eficiência paralela, necessidade de memória, *throughput*, latência, taxas de entrada/saída, *throughput* da rede, custo de projeto, custo de implementação, custo de verificação, reuso, necessidade de *hardware*, custo de *hardware*, custo de manutenção, portabilidade e escalabilidade.

Outras métricas são utilizadas para avaliar componentes específicos de arquiteturas paralelas, por exemplo, GFLOPS (potência computacional da CPU) e MB/s (medida de largura de banda do canal). A importância dessas diversas métricas varia de acordo com o problema. Uma especificação pode colocar exigências com relação a certas métricas, necessitar que outras sejam otimizadas, e ainda ignorar outras. A subseção 2.3.1 apresenta algumas métricas.

2.3.1 Métricas de Desempenho

Aplicações paralelas podem ser caracterizadas de diferentes maneiras dependendo do objetivo para o qual são usadas. As métricas de desempenho de software são divididas em *métricas estáticas* (são derivadas do modelo adotado para descrição da aplicação paralela; capturam as características inerentes da aplicação com respeito a paralelismo e sincronização) e *métricas dinâmicas* (descrevem o comportamento em tempo de execução de uma aplicação em termos de atividades concorrentes em vários processadores; capturam os efeitos das interações entre a aplicação e a arquitetura paralela utilizada).

Nos ocuparemos aqui apenas com métricas que podem ser derivadas do tempo de execução.

Eficiência e *Speedup*

O *speedup* é a métrica de desempenho mais usada no domínio paralelo, dá uma medida de melhora de desempenho da aplicação quando executada em um sistema paralelo. O *speedup* definido na equação (2.2), é uma métrica de alto nível visto que ele captura efeitos combinados de todos os fatores típicos de computação paralela (balanceamento de carga, sobrecarga, quantidade de comunicação). A quantidade relativa de *speedup* é o fator pelo qual o tempo de execução é reduzido em P processadores.

$$S = \frac{T_1}{T_P} \tag{2.2}$$

onde T_1 é o tempo de execução em um processador e T_P é o tempo em P processadores.

A eficiência representa a fração do tempo que os processadores gastam trabalhando, é uma métrica que pode fornecer uma medida em alguns casos mais adequada da qualidade do algoritmo paralelo. Ela caracteriza a eficácia (leva em conta o custo e o ganho de uma computação) com que o algoritmo usa os recursos computacionais de um computador paralelo de uma maneira que não depende diretamente do tamanho do problema. A eficiência pode ser definida como mostra a equação (2.3).

$$E = \frac{S}{P} \tag{2.3}$$

Análise de Escalabilidade

As métricas de escalabilidade descrevem as características da aplicação em termos de ganhos relativos ou perda de desempenho em função do número de recursos alocados. Idealmente, a eficiência

do programa deve ser mantida, mesmo aumentando o número de processadores.

Pode-se avaliar a escalabilidade do algoritmo paralelo, isto é, quão eficientemente ele pode usar um número crescente de processadores.

Escalabilidade com problema de tamanho fixo Um aspecto importante da análise de desempenho é o estudo de como o desempenho do algoritmo varia com o tamanho do problema, o número de processadores, e o custo de comunicação. Deve-se verificar a capacidade do algoritmo paralelo rodar mais rápido de acordo com a disponibilidade de processadores. Para quantificar escalabilidade pode-se determinar como o tempo de execução e a eficiência variam com o crescimento do número de processadores para um problema de tamanho fixo.

Escalabilidade com problema de tamanho variável Computadores paralelos grandes são usados não somente para resolver problemas de tamanho fixo mais rapidamente, mas também para resolver problemas maiores. As análises de algoritmos chamados *análises com tamanhos de problema variável*, consideram não como a eficiência varia com o número de processadores, mas como a quantidade de computação realizada deve aumentar com o número de processadores para manter a eficiência constante.

2.4 Métodos de Previsão de Desempenho

Metodologias de avaliação de desempenho têm sido aplicadas para analisar sistemas paralelos junto com métricas específicas de desempenho. As metodologias de avaliação de desempenho são de

grande utilidade para identificar gargalos no sistema, prever desempenho e projetar sistemas futuros.

Os objetivos da análise de desempenho são avaliar o desempenho de um sistema ou um componente de sistema ou uma aplicação; investigar a combinação entre aplicações requeridas e características da arquitetura de sistemas; identificar características que tenham impacto significante no tempo de execução da aplicação; prever o desempenho de uma aplicação particular num dado sistema paralelo; avaliar estruturas alternativas de aplicações paralelas [3].

A literatura de previsão de desempenho é bastante ampla. Abaixo discutimos alguns métodos sem buscar completude na apresentação, mas apenas tentando mostrar os tipos de soluções empregadas.

2.4.1 Lei de Amdahl

Todos os programas paralelos têm um componente seqüencial que limita o *speedup* que pode ser alcançado por um computador paralelo. A lei de Amdahl pode ser enunciada como segue: se o componente seqüencial de um algoritmo representa $1/s$ do tempo de execução de um programa, então o máximo *speedup* possível em um computador paralelo é *s*.

A lei de Amdahl não serve para avaliar o desempenho do programa, pois não leva em conta comunicação. Não fornece informação sobre o tempo de execução esperado para dado número de processadores.

Pode ser relevante quando programas seqüenciais são incrementalmente paralelizados. Para desenvolver um software paralelo, um programa seqüencial é primeiro analisado para identificar compo-

nentes que são adaptados para execução paralela. A lei de Amdahl é aplicável a este caso pois o custo computacional de componentes que não são paralelizados fornece um limite inferior no tempo de execução do programa paralelo.

2.4.2 Método de Foster

O tempo de execução de um programa paralelo é o tempo que transcorre de quando o primeiro processador inicia a execução do problema até o término da execução do último processador [2]. Durante a execução cada processador está computando, comunicando ou ocioso. T^i_{comp}, T^i_{comun} e T^i_{desoc}, são tempos gastos computando, comunicando e ocioso no processador *i*. O tempo total de execução pode ser definido de duas maneiras:

- Como a somatória dos tempos de computação, comunicação e ocioso para qualquer dos componentes, como mostra a equação (2.4).

$$T = T^j_{comp} + T^j_{comun} + T^j_{desoc} \tag{2.4}$$

- Como a somatória dos tempos em todos os processadores dividido pelo número de processadores *P*. Como mostra a equação (2.5).

$$\begin{aligned} T &= \frac{1}{P}(T_{comp} + T_{comun} + T_{desoc}) \\ &= \frac{1}{P}\left(\sum_{i=0}^{P-1} T^i_{comp} + \sum_{i=0}^{P-1} T^i_{comun} + \sum_{i=0}^{P-1} T^i_{desoc}\right) \end{aligned} \tag{2.5}$$

A última definição é freqüentemente mais utilizada, pois é mais fácil determinar o tempo de computação e comunicação realizado

por um algoritmo paralelo do que o tempo gasto computando e comunicando em processadores individuais. Os três componentes do tempo total de execução serão discutidos em seguida.

Tempo de Computação É o tempo gasto realizando computação. Normalmente depende de algumas medidas do tamanho do problema. Se o algoritmo paralelo replica computação, então o tempo de computação dependerá do número de tarefas ou processadores. Dependerá também de características de processadores e seu sistema de memória.

Tempo de Comunicação É o tempo que a tarefa gasta enviando e recebendo mensagens. Existem dois tipos de comunicação: a *comunicação interprocessador* (duas tarefas comunicando estão localizadas em diferentes processadores) e a *comunicação intraprocessador* (duas tarefas comunicando estão localizadas no mesmo processador). O custo das comunicações intraprocessador e interprocessador são em muitas máquinas paralelas comparáveis [2].

Tempo Ocioso Comunicação e computação são especificadas em um algoritmo paralelo, portanto é relativamente simples determinar sua contribuição para o tempo de execução. O tempo ocioso T_{desoc}, pode ser difícil de determinar, pois freqüentemente depende da ordem na qual as operações são realizadas.

Um processador pode estar ocioso por falta de computações a efetuar (pode ser evitado pelo uso de algoritmos de balanceamento de cargas) ou por estar esperando dados de outros processadores (aguardando comunicação — pode ser evitado pela sobreposição de comunicações com computações, ou chaveando o processador para outra tarefa).

Algumas limitações que surgem com esse modelo são: o tempo ocioso é difícil de avaliar, como descrito acima. Como o modelo só usa valores de tempo fixo, não cuida de variação dos tempos. Permite apenas o uso de modelos simples de rede, pois usa o tempo de comunicação definido pelo multicomputador (descrito na seção 2.2).

2.4.3 Método de Anglano

Anglano em [4] estuda o problema de previsão de desempenho para programas paralelos executados em *clusters* de *workstations* heterogêneas onde contenção de recursos está presente. Anglano desenvolve uma metodologia para a construção de modelos de desempenho cujas análises permitem a avaliação do tempo de execução desses programas. Utiliza-se de *Rede de Petri Temporizada* para representar o comportamento de programas paralelos e de um modelo de contenção baseado na teoria de filas [5] para quantificar os efeitos de contenção de recursos no tempo de execução da aplicação.

O ambiente computacional estudado por Anglano é um *cluster* com *m workstations* heterogêneas conectadas por uma rede local (LAN). Cada workstation têm uma CPU, escalona seus processos localmente e independentemente usando prioridade baseada em *round robin*, executa somente um processo do programa paralelo em consideração, e pode ser compartilhada por vários usuários submetendo seus próprios *jobs* independente dos outros. Anglano assume que em um dado instante somente um programa *paralelo* está em execução no *cluster*, e que a memória principal de cada workstation é grande o suficiente para alojar o conjunto de processos paralelos que ela executa.

A metodologia de previsão de desempenho é baseada na construção de um modelo de Rede de Petri Temporizada (TPN) [6] da

aplicação paralela. Com a TPN é possível especificar informação temporizada associando um atraso de disparo a cada transação. O modelo TPN de uma dada aplicação é realizado como mostrado na figura 2.3, onde os retângulos representam os passos da metodologia e as elipses representam seus resultados. Um modelo de Rede de Petri não temporizada representando o comportamento dos processos da aplicação paralela e suas interações é construído inicialmente do código fonte do programa. O modelo não temporizado é transformado em um temporizado associando a suas transições a duração das várias comunicações e computações que elas representam. Essas durações dependem das características da rede e processadores, e portanto da alocação de processos para *workstations*; a quantidade de contenção de recursos que pode estar presente é obtida como resultado de um passo da caracterização do sistema. A caracterização do sistema envolve duas atividades: determinação do desempenho dos vários recursos na ausência de contenção, que pode ser realizado por meio das várias técnicas disponíveis, e determinação da caracterização de fluxo de *jobs* e tráfego, realizado por meio de metodologias de caracterização de *workloads*. Após o modelo TPN ter sido completamente especificado, ele é analisado para avaliar o tempo de execução da aplicação para uma alocação particular escolhida. Na figura 2.3 os vários passos são mostrados usando tons de cinza para indicar que o passo da caracterização do sistema é realizado somente uma vez, enquanto a construção do modelo tem que ser realizada cada vez que um novo programa é considerado.

O modelo de Rede de Petri de programas paralelos é construído separadamente modelando primeiro cada processo, e então unindo estes submodelos em um único para representar a comunicação

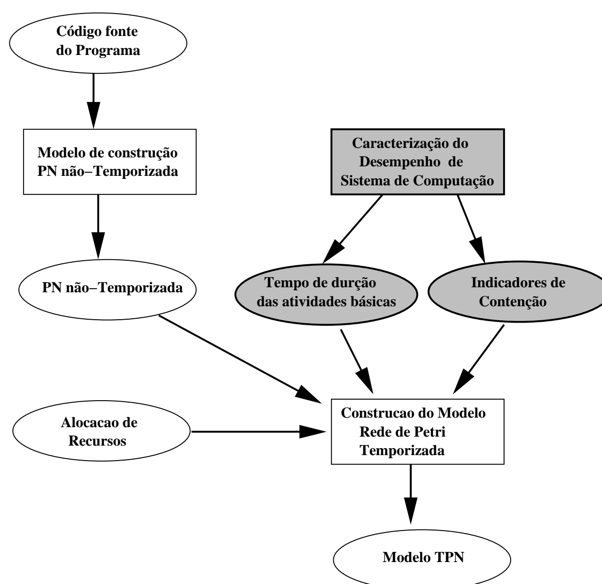

Figura 2.3: Metodologia de previsão de desempenho de Anglano

entre os processos. O modelo de um dado processo é obtido unindo os submodelos de seus segmentos, que são porções de sua computação incluída entre dois comandos consecutivos de comunicação, de acordo com a ordem em que eles ocorrem no código do processo. Para modelar o tempo de execução de vários segmentos é associado com cada transição um atraso igual à quantidade de tempo de CPU requerida. Esse atraso depende não só da quantidade de computação associada à transição e da velocidade da CPU na qual essa computação é executada, mas também da presença de outros processos competindo pelo processador. Os tempos de comunicação são representados associando atrasos apropriados com as transições correspondendo às transmissões das mensagens. O tempo de execução médio é calculado computando o tempo requerido para alcançar a marcação final do modelo TPN representado, isto é, uma marcação em que nenhuma das transações estão permitidas, o que corresponde ao término do programa.

O tempo requerido para alcançar a marcação final é computado por meio de uma simulação em que para cada transição um tempo é escalonado e adicionado em uma lista ordenada. Em um *loop*, enquanto essa lista não estiver vazia, os tempos armazenados na lista são removidos e um novo tempo é escalonado.

O método de Anglano lida com tempo de espera, com contenção, mas não lida com variações nos tempos de computação ou tamanho das mensagens.

2.4.4 Método de Liang

A aplicação paralela é modelada por Liang [7] utilizando grafos de tarefas que são uma coleção de nós e arcos. Os nós no grafo representam as tarefas enquanto os arcos representam a prioridade nos relacionamentos. Uma aplicação paralela modelada por um grafo de tarefa é chamada *job*. O ambiente de execução paralela é descrito no contexto do modelo de rede de filas [5]. As entidades básicas neste modelo são tarefas (que representam os recursos físicos) e filas (que descrevem a ordem na qual as tarefas são pegas da fila e admitidas no serviço). Os tipos básicos de tarefas que são considerados por Liang são *Seqüencial* (ou tipo S), *Parallel And* (ou tipo P_A), *Parallel Or* (ou tipo P_O) e *Probabilistic Fork* (ou tipo P_F), detalhes sobre os quatro tipos são encontrados em [7].

O objetivo de Liang é então desenvolver um algoritmo eficiente para avaliar as três principais medidas de desempenho: tempo médio de resposta de tarefas individuais, o tempo médio de resposta para *job* (ou tempo final do *job*), e o *throughput* médio de *jobs* de cada classe de *job*.

Liang em [7] apresenta um método de previsão de desempenho que é aplicado para um sistema de processamento paralelo multi-

programação e avalia o desempenho médio do sistema como o tempo de resposta e a utilização de recursos pelas tarefas. O modelo proposto é baseado na técnica MVA (Análise de Valor Médio). O método consiste em dois passos principais em cada iteração. No primeiro passo, a distribuição do tempo de resposta das tarefas é usada para avaliar o tempo de resposta do *job*. Este passo é para avaliar o atraso de sincronização entre tarefas do mesmo *job*, e para avaliar a sobreposição do tempo de resposta entre duas tarefas no sistema. No segundo passo, a sobreposição do tempo de resposta é usada para avaliar o atraso na fila e o tempo de resposta médio de cada tarefa. O algoritmo iterativo continua até o tempo de resposta da tarefa e o tempo de resposta do *job* convergir dentro de um erro aceitável.

O método de Liang não lida com comunicação, não avalia o tempo desocupado e não lida com variações.

2.4.5 Método de Fahringer e Zima

Thomas Fahringer e Hans Zima apresentam em [8] uma ferramenta de previsão de desempenho que é parte do Sistema de Compilação Fortran Vienna (VFCS). O VFCS é um sistema de paralelização automático para sistemas de memória distribuída, que traduz programas Fortran em programas de passagem de mensagem.

A ferramenta PPPT *(Parameter based Performance Prediction Tool)* é aplicada a um programa paralelo gerado pelo VFCS, que pode conter comunicação síncrona e assíncrona e recebe parâmetros do programa seqüencial computado em uma execução anterior. Computa estaticamente um conjunto de parâmetros que caracterizam o comportamento do programa paralelo. Isto inclui distribuição de trabalho, o número de dados transferidos, o tempo de transferência e

contenção de rede. A ferramenta *PPPT* foi implementada e integrada ao VFCS. A maioria dos parâmetros do programa são independentes de máquinas e portanto a ferramenta é altamente portável.

A estratégia de paralelização do VFCS é baseada em decomposição de domínio e no modelo de programação SPMD *(Single Program Multiple Data)*. A entrada para VFCS pode ser um programa Fortran 77 e uma descrição separada da distribuição de dados ou um programa escrito em Vienna Fortran que é uma linguagem independente de máquina, extensão de Fortran 77, que fornece anotações para a especificação de distribuição de dados. A saída de VFCS é um programa Fortran paralelo com passagem de mensagens explícita.

O contexto no qual a ferramenta *PPPT* é aplicada no VFCS é:

- **Primeiro passo:** o programa Fortran 77 original é compilado pelo VFCS, isto inclui análises intra e interprocedural, normalização e padronização do programa.

- **Segundo Passo:** o *Weight Finder* (um *profiler* avançado e altamente otimizado para programas Fortran) é aplicado. Ele realiza uma única execução de *profiling*, uma versão instrumentada do programa, derivando valores para parâmetros desconhecidos.

- **Paralelização:** o VFCS gera um programa paralelo de passagem de mensagem otimizado.

- **Computando parâmetros do programa paralelo:** uma série de parâmetros do programa paralelo é computado estaticamente. Baseado nesses parâmetros o paralelizador é capaz de reconhecer a posição e a natureza de gargalos de desempenho.

> Isto permite a aplicação de uma seqüência de transformações e busca de espaço para partição de dados.

Os parâmetros do programa paralelo caracterizam certos aspectos de seu desempenho. Os parâmetros computados pelo PPPT incluem distribuição de trabalho (define como o trabalho total no programa paralelo é distribuído através dos processadores), quantidade de dados transferidos, número de transferência de dados, tempo de transferência, contenção de rede. Uma explicação mais detalhada de como calcular valores para os parâmetros citados acima pode ser encontrada em [8].

Através de alguns experimentos realizados em [8], acredita-se que o tempo de transferência e o número de transferências são os parâmetros mais importantes. Se o tempo de transferência e o número de transferências são similares para várias versões do programa, então seria dada prioridade maior para a distribuição de trabalho.

O método de Fahringer e Zima é altamente integrado no desenvolvimento do programa e automático (sem esforço), mas por outro lado, não gera informações ao programador que possam ajudar a melhorar o algoritmo; como o método é baseado mais em *profiling*, não fornece previsões; o programa precisa estar pronto, ou seja, precisa ser compilado pelo VFCS, e com isso o método é utilizado apenas para Vienna Fortran.

2.4.6 Método de Parashar e Hariri

Parashar e Hariri em [9] apresentam uma abordagem intepretativa para previsão de desempenho que pode ser efetivamente usada durante o desenvolvimento de *software*. A essência da abordagem é a aplicação de técnicas de interpretação para prever o desempenho

através de uma caracterização apropriada do sistema de computação de alto desempenho e da aplicação.

Segundo Parashar e Hariri em [9] a previsão de desempenho interpretativa é uma abordagem precisa de baixo custo para avaliar o tempo de execução da aplicação realizada. Uma metodologia de abstração do sistema e da aplicação é definida para abstrair hierarquicamente o sistema e a descrição da aplicação em um conjunto de parâmetros bem definidos que representam seu desempenho e comportamento. Previsão de desempenho é alcançada interpretando o custo de execução da aplicação abstraída em termos de parâmetros do sistema abstraído.

Parashar e Hariri usam a abordagem interpretativa para desenvolver um *framework* de previsão de desempenho para aplicações HPF/Fortran 90D. Fortran de alto desempenho (HPF) é baseado na linguagem Fortran 90D e tem por meta desenvolver um dialeto de Fortran que pode ser usado em diversas máquinas paralelas. A idéia de HPF (e Fortran 90D) é desenvolver um conjunto mínimo de extensões para Fortran 90 suportar o modelo de programação paralela de dados.

Foi criado um *framework* de previsão de desempenho HPF/ Fortran 90D, o ESP [9]. ESP é baseado na tecnologia de compilador HPF *source-to-source* que traduz HPF em códigos SPMD Fortran 77 e passagem de mensagens. Previsão de desempenho HPF/Fortran 90D é realizada em duas fases. Na primeira fase usa-se tecnologia de compilação HPF para produzir um programa SPMD consistindo de Fortran 77 mais chamadas para rotinas de tempo de execução. Na segunda fase, usa-se então a abordagem de interpretação para abstrair e interpretar o desempenho da aplicação.

A fase de interpretação é implementada como uma seqüência de

fases que são:

1. **Fase de Abstração:** a abstração da aplicação é realizada caracterizando a aplicação em unidades de abstração da aplicação. Cada unidade representa um padrão de construção da aplicação parametrizando seu comportamento. As unidade são combinadas para abstrair a estrutura de controle da aplicação formando o grafo de abstração da aplicação (são formados por comandos seqüenciais e comunicações, sendo que as comunicações não têm ligação entre si) e grafos de abstração da aplicação sincronizada (são formados por comandos seqüenciais e comunicações, sendo que as comunicações possuem ligação entre si)

2. **Fase de Interpretação:** para cada unidade de abstração da aplicação em grafos de abstração da aplicação sincronizada, a função de interpretação é usada para gerar métricas de desempenho associadas às unidades. Métricas mantidas em cada unidade são computações, comunicações e tempo de sobrecarga.

3. **Fase de Saída:** a fase final (de saída) avalia métricas de desempenho para o usuário.

O método de Parashar e Hariri permite a geração automática do modelo, fornecendo informações de desempenho. É aplicado apenas para HPF.

Capítulo 3

Descrição do Método Proposto

Com o desenvolvimento de modelos matemáticos é possível detectar e resolver problemas de desempenho antes do desenvolvimento do código e todos os seus detalhes. Quando os modelos incluem dados sobre os custos de comunicação, eles raramente consideram a estrutura da rede, e se baseiam simplesmente em parâmetros como latência e largura de banda. Também fatores de interação de tempos entre os processos são desprezados.

Com o uso de simuladores detalhados é possível avaliar qual o efeito de uma mudança de máquina, por exemplo, avaliar se compensa para uma dada aplicação mudar de rede de interconexão. A desvantagem em utilizar simuladores detalhados é que eles simulam instrução a instrução a execução do programa, exigindo muitos recursos e um código pronto.

Para preencher a lacuna existente entre os modelos matemáticos e os simuladores detalhados, utilizamos uma representação simplificada do programa que terá sua execução simulada para gerar dados sobre instantes de início e término das operações que poderão ser usados para calcular tempos de computação e desocupado. A verificação dos tempos fornecerá informações importantes

para análise de desempenho e para depuração de problemas de desempenho.

É necessário desenvolver uma descrição de alto nível do programa paralelo indicando as partes de computação e comunicação envolvidas; avaliar os tempos de execução de diversas partes de computação (valores médios e desvios-padrão no caso em que as operações tenham tempos variáveis) e o tamanho das mensagens enviadas (valores médios e desvios-padrão, caso haja variações). A partir dessa descrição será gerado um código que será executado por um simulador que inclui a simulação da rede escolhida.

A linguagem desenvolvida no trabalho é utilizada para a descrição simplificada de um algoritmo paralelo, envolvendo estruturas de seleção, iteração, comunicação ponto-a-ponto e comunicação coletiva como descrito no capítulo 4. Com a construção de um esqueleto — ou protótipo — do algoritmo paralelo é possível indicar as comunicações, computações e quantidade de dados envolvidos. Para calcular a quantidade de dados são utilizadas variações estatísticas como distribuição gaussiana. O uso de um simulador permite verificar as características de execução do algoritmo, como tempo gasto durante a comunicação ou mesmo computação (mais detalhes sobre o simulador são descritos no capítulo 5).

Exemplos de uso do método e de códigos de programas escritos utilizando a biblioteca MPI e C++ são discutidos abaixo.

3.1 Ping-pong

O código do programa ping-pong mostra a troca de mensagens realizada entre dois processos como visto na figura 3.1. A figura 3.2 mostra o código descrito em MPI e a figura 3.3 apresenta o código

utilizando a descrição simplificada obtida na linguagem desenvolvida no trabalho.

No código mostrado na figura 3.2 foram utilizadas as funções da biblioteca MPI, MPI_Send e MPI_Recv, para a realização da troca de mensagem entre os dois processos. O mesmo código foi escrito de uma maneira simplificada como pode ser visto na figura 3.3, onde as funções *send* e *receive* desenvolvidas na linguagem, têm a mesma função que as operações em MPI. A função *send* mostrada na figura 3.3 possui apenas dois parâmetros que indicam o processo destino e a quantidade de dados a serem transmitidos. Para o cálculo dos dados foram utilizados dois valores que indicam a média e o desvio-padrão (variações estatísticas). Como a quantidade de dados enviada no programa é fixa, o desvio-padrão foi especificado com 0. A função *receive* indica apenas o número do processo que enviou os dados.

Os valores apresentados nas funções da figura 3.3 foram obtidos de acordo com os valores das funções da figura 3.2. Na operação *send* da figura 3.3 o 1 indica o número do processo a quem será enviada a mensagem e na figura 3.2, na função *MPI_Send* o número do processo é indicado por 1. A quantidade de dados enviada aos processos é indicada por 8192, tanto na figura 3.2 quanto na figura 3.3.

A tabela 3.1 apresenta os tempos obtidos, na comunicação de cada processo. A quantidade de dados transmitidos é de 8192 *bytes*. São também apresentados os tempos máximos de execução. Os tempos foram obtidos a partir da execução do programa ping-pong no *cluster* (descrito no capítulo 6) e no simulador (descrito no capítulo 5). A diferença nos valores de tempo será comentada no capítulo 5.

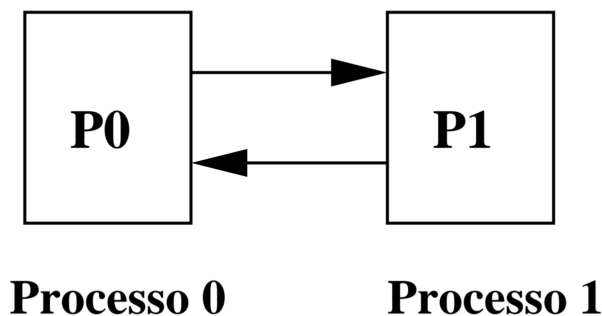

Figura 3.1: Ping-Pong

```
loop = 100;
if (rank == 0) {
  for (i = 0; i < loop; i++) {
     MPI_Send(tam,8192,MPI_CHAR,1,TAG,
              MPI_COMM_WORLD);
     MPI_Recv(tam,8192,MPI_CHAR,1,TAG,
              MPI_COMM_WORLD, &status);
  }
}
else {
  for (i = 0; i < loop; i++) {
     MPI_Recv(tam,8192,MPI_CHAR,0,TAG,
              MPI_COMM_WORLD, &status);
     MPI_Send(tam,8192,MPI_CHAR,0,TAG,
              MPI_COMM_WORLD);
  }
}
```

Figura 3.2: Programa ping-pong escrito em MPI

```
loop = 100;
if (rank==0) {
  for (i,loop){
    send(1,(8192,0));
    receive(1);
  };
}
else {
  for (i,loop){
    receive(0);
    send(0,(8192,0));
  };
};
```

Figura 3.3: Descrição simplificada do programa Ping-pong

Tabela 3.1: Tempos obtidos com a execução do programa ping-pong no *cluster* e no simulador

Processos	*TempoCluster*(seg)	*TempoSimulador*(seg)
0	0.206023	0.205818
1	0.206014	0.205818
Máximo	0.206023	0.205818

3.2 Soma de Matrizes

É realizada a soma de duas matrizes N x N. As matrizes estão inicialmente no processo 0, onde deverá também ser colocado o resultado. A figura 3.4 apresenta o trecho de um programa de soma de matrizes feito em MPI e a figura 3.5 mostra o trecho do mesmo programa feito utilizando a descrição simplificada obtida na linguagem.

A cada processador são atribuídos N/P linhas consecutivas para realização da soma (supõe-se N divisível por P). No trecho do programa da figura 3.4 o comando MPI_Scatter realiza a distribuição dos elementos das linhas das matrizes originais entre os processadores. Após essa distribuição ser feita é realizada a soma entre os elementos e depois o comando MPI_Gather realiza a coleta do resultado obtido da soma de todos os processadores e o armazena na variável *resultado*.

Na descrição simplificada mostrada na figura 3.5 a função *scatter* é usada duas vezes, isso porque é necessário a distribuição dos dados das duas matrizes, em que são distribuídos N*N/P elementos para cada um dos processos e essa quantidade é ainda multiplicada por 4, pois está sendo transmitido um inteiro (4 *bytes*) como mostrado na figura 3.4. A função *compute* indica a computação re-

```
/* Distribuição, loop e coleta dos elementos para soma
   dos elementos de uma matriz
*/
   if (rank == 0){
     MPI_Scatter(&a[0][0], N*N/P, MPI_INT, &d[0][0], N*N/P,
                  MPI_INT, 0, MPI_COMM_WORLD);
     MPI_Scatter(&b[0][0], N*N/P, MPI_INT, &e[0][0], N*N/P,
                  MPI_INT, 0, MPI_COMM_WORLD);

     for(i = 0; i < (N/P); i++)
       for(j = 0; j < N; j++)
        soma[i][j] = d[i][j] + e[i][j];

     MPI_Gather(&soma[0][0], N*N/P, MPI_INT, &resultado[0][0],
                 N*N/P, MPI_INT, 0, MPI_COMM_WORLD);
    }
```

Figura 3.4: Trecho do programa de soma de matrizes feito usando MPI

```
/* Cálculo para soma de matrizes */
   N = 720;
   scatter(0,((N*N/P)*4,0));
   scatter(0,((N*N/P)*4,0));
   compute(1.7e-7*N*N/P,0);
   gather (((N*N/P)*4,0),0);
```

Figura 3.5: Trecho do programa de soma de matrizes utilizando a descrição simplificada

alizada pelos processos, essa computação é mostrada na figura 3.4 pelo *loop*. E por fim a função *gather* coleta N*N/P dados de cada processo. O valor $1.7 \cdot 10^{-7}$ apresentado na figura 3.5 foi obtido através do tempo medido na execução da operação de soma no *loop* da figura 3.4.

Tabela 3.2: Tempos em segundos obtidos com a execução do programa soma de matrizes no *cluster* e no *simulador*

Processos	*TempoCluster*(seg)	*TempoSimulador*(seg)
02	0.396882	0.321790
03	0.459568	0.400277
04	0.481668	0.439970
05	0.500551	0.464147
06	0.511739	0.480564
08	0.528667	0.501761
09	0.533350	0.509126
10	0.539058	0.515199
12	0.546451	0.524758
15	0.557570	0.535216
16	0.558417	0.538056

A tabela 3.2 apresenta os tempos obtidos na execução do programa de soma de matrizes (figura 3.4 e figura 3.5) realizado no *cluster* e no simulador. O programa foi executado variando o número de processos. Os testes não foram executados com 7, 11, 13 e 14 processos, pois esses não são divisores de N.

Os comandos utilizados na descrição simplificada dos exemplos apresentados são descritos no capítulo 4.

Para transformar a descrição simplificada das figuras 3.3 e 3.5

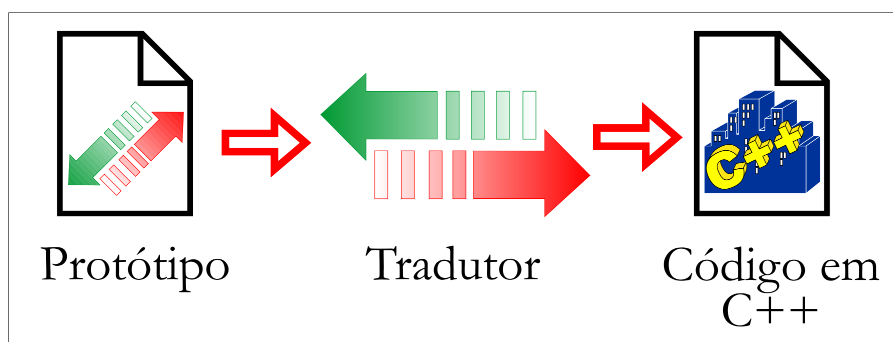

Figura 3.6: Funcionamento do Tradutor

em um código que será executado pelo simulador, foi necessário o desenvolvimento de um tradutor que é utilizado para traduzir a descrição simplificada (esqueleto) em um código C++, como mostra a figura 3.6. Mais detalhes sobre o funcionamento do tradutor podem ser visto na seção 4.2.

O simulador proposto é baseado no modelo de máquina paralela, o multicomputador descrito na seção 2.2. Mais detalhes sobre o funcionamento do simulador de rede podem ser vistos no capítulo 5.

Resultados mais detalhados são apresentados no capítulo 6.

Capítulo 4

Linguagem de Protótipo

A definição da linguagem visa permitir uma prototipagem simples para possibilitar a descrição de diversos tipos de algoritmos paralelos. A linguagem busca possibilitar o desenvolvimento de uma descrição de alto nível de um algoritmo paralelo, indicando as partes de computação e as comunicações envolvidas.

Para o desenvolvimento de um protótipo do algoritmo utilizando a linguagem proposta é necessário definir as comunicações e as computações envolvidas, avaliar os tempos de execução das partes de computação e o tamanho das mensagens a serem enviadas. Um exemplo disso pode ser visto no trecho do programa que calcula o produto escalar mostrado na figura 4.1 construído utilizando C e

```
/* Loop para cálculo do produto escalar
   N = tamanho dos arrays
   P = número de processadores
*/
   produto = 0;
   for (i=0;i<(N/P);i++)
       produto = produto + c[i]*d[i];
   MPI_Reduce(&produto,&resultado,1,MPI_INT,MPI_SUM,
              0,MPI_COMM_WORLD);
```

Figura 4.1: Trecho do programa de produto escalar utilizando C e MPI

```
/* Cálculo do produto escalar */
   compute(0.0000125*N/P,0);
   reduce(0,(4,0));
```

Figura 4.2: Trecho do programa de produto escalar utilizando a definição da linguagem

```
   programa: lista_comando
   lista_comando: comando ;
                | comando ; lista_comando
   bloco: { lista_comando };
```

Figura 4.3: Regras iniciais da linguagem

MPI. A figura 4.2 mostra o protótipo do mesmo trecho de programa construído utilizando as regras da linguagem. O valor na função *compute* é o tempo de execução de uma iteração no laço da figura 4.1, N é o tamanho do vetor (a quantidade de dados) e P é o número de processos. Na operação *reduce* o valor 4 indica a quantidade de dados a ser enviada (um inteiro).

4.1 Sintaxe da Linguagem

Algumas regras da gramática da linguagem são apresentadas na figura 4.3. Um programa é formado por uma lista de comandos, e uma lista de comandos é formada por um comando seguido de ponto-e-vírgula, ou ainda esse mesmo pode ser seguido de uma lista de comandos. Como em C, para delimitar os blocos de comandos, a linguagem utiliza as chaves { e }. Um bloco de comandos é formado por uma lista de comandos sempre entre chaves.

O *token ANYSOURCE* mostrado na figura 4.4, pode ser usado em uma operação *receive* para indicar que um processo pode receber mensagens de qualquer processo. Um comando de atribuição é simplesmente um identificador seguido por um sinal de igual “=”, e

```
ident: IDENT
source: ANYSOURCE
atribuicao: ident = expressao
expressao:  ident
          | funcao
          | NUM
          | expressao + expressao
          | expressao - expressao
          | expressao * expressao
          | expressao / expressao
          | expressao % expressao
          | ( expressao )
comparacao: expressao == expressao
          | expressao != expressao
          | expressao >  expressao
          | expressao >= expressao
          | expressao <  expressao
          | expressao <= expressao
```

Figura 4.4: Expressões permitidas na Linguagem

uma expressão como mostrado na figura 4.4.

As expressões são formadas por elementos básicos como dados e operadores. Os dados podem ser representados por identificadores, números positivos ou negativos. Os operadores são definidos em duas classes: os operadores aritméticos e os operadores relacionais. As expressões aritméticas podem ser formadas de expressões seguidas do operador aritmético e outra expressão, como mostrado na regra da gramática da figura 4.4. Como pode ser visto, uma expressão pode ou não estar entre parêntesis ( ). As expressões relacionais (comparação) referem-se às relações que os valores (expressões) podem ter uns com os outros. O uso destas expressões relacionais é semelhante aos correspondentes operadores de C e C++. A regra para esses operadores pode ser vista na figura 4.4. Todos os números e expressões são considerados valores de ponto-flutuante.

A linguagem permite a definição de uma lista de parâmetros que

```
lista_param: expressao
           | expressao , lista_param
funcao: ident ( lista_param )
```

Figura 4.5: Definição de uma função

```
variacao: expressao , expressao
```

Figura 4.6: Definição de Variação

pode ser uma única expressão, ou uma expressão seguida de uma lista de parâmetros (figura 4.5); além de permitir também a definição de função que é formada por um identificador seguido de lista de parâmetros entre parêntesis como mostra a figura 4.5. A linguagem não permite a definição de funções, mas possibilita o uso de funções da biblioteca C através dessa sintaxe: a chamada da função é passada identicamente para o código gerado.

Durante a definição e desenvolvimento da linguagem houve a necessidade da definição de uma construção sintática para descrever valores estatiscamente flutuantes, construção essa que chamamos de “**variação**”. A variação é composta de uma expressão seguida de vírgula mais uma expressão, como na figura 4.6. Geralmente usamos a variação com dois valores numéricos, que indicam a média e o desvio-padrão de uma distribuição gaussiana. Durante a simulação utilizamos esses dois valores numéricos para sortear números de acordo com essa distribuição. Variações são utilizadas por exemplo para determinar a quantidade de dados a serem transmitidos.

Além disso, foram também definidos diversos comandos. Os comandos definidos envolvem comandos de seleção, de iteração, de comunicação ponto-a-ponto e comandos de comunicação coletiva. A figura 4.7 mostra a lista dos comandos definidos na linguagem. Sempre no final de cada comando, existe um ponto-e-vírgula. Por

```
comando: atribuicao
       | bloco
       | if_comando
       | while_comando
       | for_comando
       | send_comando
       | receive_comando
       | compute_comando
       | broadcast_comando
       | gather_comando
       | all_gather_comando
       | scatter_comando
       | all_to_all_comando
       | reduce_comando
       | all_reduce_comando
```

Figura 4.7: Lista de comandos definidos

exemplo, sempre que utilizamos o comando *if* definido na figura 4.8, o ponto-e-vírgula aparece no final da } referente ao bloco de comandos do *else* se esse existir, caso contrário no final da } correspondente ao término do bloco de comandos do *if*.

Comando de Seleção : Existem dois comandos condicionais, ambos representados usando if-else (figura 4.8).

- **IF-ELSE:** A forma do comando *if* é mostrada na figura 4.8. A cláusula *else* é opcional. Se a condição da comparação for verdadeira o bloco que segue o *if* será executado; caso contrário, se existir, o bloco do *else* será executado. Somente será executado o código associado ao *if* ou ao *else*, nunca os dois. A comparação citada na figura 4.8 corresponde a uma das expressões sob o item correspondente na figura 4.4.

  Outro comando *if* é o que usa uma expressão. A expressão no comando *if* indica a porcentagem de vezes que será executado o bloco do *if* ao invés do *else*.

```
if ( comparacao ) bloco
if ( comparacao ) bloco else bloco
if ( expressao  ) bloco
if ( expressao  ) bloco else bloco
```

Figura 4.8: Definição do comando IF-ELSE

```
while ( comparacao ) bloco
while ( variacao ) bloco
```

Figura 4.9: Definição do comando WHILE

Iteração : Os comandos de iteração que foram definidos na linguagem são *while* e o *for*.

- **WHILE:** A sintaxe do comando *while* pode ser vista na figura 4.9, sendo que o laço é repetido enquanto a comparação for verdadeira. Quando usamos uma variação, o laço é executado o número de vezes indicado por sorteio de acordo com a variação.
- **FOR:** A figura 4.10 mostra a forma do comando *for*, onde temos um *ident* (identificador) que controla o laço; o identificador é seguido de vírgula e uma expressão. O laço é executado o número de vezes indicado pela expressão, com o identificador variando de 0 ao valor de expressão menos um.

Computação : Durante a realização da computação não existe nenhum tipo de comunicação.

- **COMPUTE:** A função da operação *compute* é simular um intervalo de computação. Sua sintaxe é mostrada na fi-

```
for ( ident, expressao )
```

Figura 4.10: Definição do comando FOR

```
compute ( variacao );
```

Figura 4.11: Definição da operação Compute

gura 4.11. A variação indica a quantidade de tempo (em segundos) tomada pela computação.

Tanto os comandos de comunicação ponto-a-ponto quanto os comandos de comunicação coletiva foram desenvolvidos baseados nos comandos existentes em MPI (*Message Passing Interface*) [10].

MPI é um padrão de biblioteca de passagem de mensagens para sistemas paralelos. Suas rotinas são construídas em torno dos conceitos (que também foram utilizados no desenvolvimento do trabalho) de processos e mensagens.

Segundo Travieso em [11], o elemento básico de computação em MPI é chamado *processo*. Um processo é um entidade de execução seqüencial, que pode ser comparada com um programa seqüencial em execução. A diferença consiste em que um processo MPI tem a possibilidade de cooperar com outros processos MPI para a execução de uma tarefa. Um programa consiste então na especificação do código a ser executado por um conjunto de processos que cooperam na solução de um problema.

As funções de comunicação ponto-a-ponto e coletiva da biblioteca MPI foram escolhidas como modelo pois esse é o padrão de passagem de mensagem utilizado nos trabalhos de programação paralela em nosso laboratório.

Comunicação Ponto-a-Ponto: O termo comunicação *ponto-a-ponto* é utilizado para indicar os tipos de comunicação que envolvem um par de processos. Essas comunicações são efetuadas por primitivas *send* (no remetente) e *receive* (no destinatário). As

```
send    ( expressao , ( variacao ) );
send    ( expressao , ( variacao ) , expressao );
receive ( expressao );
receive ( expressao, ident );
receive ( source );
receive ( source , ident , ident );
```

Figura 4.12: Definição das operações de Comunicação Ponto-a-Ponto

operações *send* e *receive* utilizadas aqui, são do tipo *bloqueante*, onde um processo só pode continuar o processamento quando o *send* ou *receive* correspondente já tiver sido postado, caso contrário ele fica esperando.

- **SEND:** O *send* simula o envio de dados para outros processos. Pode ser usado de duas formas. A sintaxe é mostrada na figura 4.12. A primeira forma da operação *send* é composta por uma expressão que indica o processo destino da mensagem, e a variação que indica a quantidade de dados que será enviada para o processo destino. A segunda forma da operação *send* tem os dois primeiros campos com a mesma função da primeira forma, a única diferença é que essa possui um terceiro campo composto de uma expressão que indica um *tag* do processo usado para distinguir diferentes tipos de mensagens (como em MPI).
- **RECEIVE:** A função do *receive* é simular a recepção dos dados enviados pelo *send*. As formas da operação *receive* são mostradas na figura 4.12. A primeira forma composta por uma expressão indica o processo origem do qual se deseja receber a mensagem. A segunda forma é composta por uma expressão (mesma função da primeira forma) e o *ident* que é o identificador que receberá o *tag* da men-

sagem. Na terceira forma a operação *receive* é composta por um *source* (a sintaxe do *source* é apresentada na figura 4.4) que indica que a mensagem recebida pode haver sido enviada por qualquer um dos processos. O *source* na última forma da operação *receive* tem a mesma função da terceira, o primeiro *ident* indica de qual processo a mensagem foi recebida e o último *ident* indica o *tag* da mensagem.

Um exemplo utilizando as comunicações ponto-a-ponto pode ser visto na figura 3.1 (ping-pong), onde um processo envia uma mensagem a outro processo que a devolve ao processo de origem.

Comunicação Coletiva: Comunicações coletivas são aquelas que envolvem um grupo de processos, ao invés de apenas dois processos. As operações coletivas que foram definidas na linguagem são: *broadcast, gather, all_gather, scatter, all_to_all, reduce, all_reduce*. As regras para gramática dessas operações podem ser vistas na figura 4.13.

Em todas as operações coletivas apresentadas na figura 4.13, todos os processos devem realizar uma chamada para a rotina da operação. Quando essas operações envolvem uma *raiz*, o valor da expressão que calcula a raiz deve ser igual em todos os processos.

- **BROADCAST:** É uma operação em que um dado presente em um dos processos é transmitido para todos os outros processos. O processo que irá transmitir os dados pode ser chamado de processo *raiz*. A figura 4.13 mostra como é a sintaxe da função *broadcast*. A expressão indicada na

```
broadcast  (  expressao ,( variacao ));
gather     (( variacao ), expressao );
all_gather ( variacao  );
scatter    ( expressao , ( variacao ));
all_to_all ( variacao );
reduce     ( expressao , ( variacao ));
all_reduce ( variacao );
```

Figura 4.13: Definição das operações de Comunicação Coletiva

operação de *broadcast* é o processo (raiz) que irá transmitir seus dados, e a variação indica a quantidade de dados transmitidos para os outros processos.

- **GATHER:** Nesta operação, o processo *raiz* coleta dados dos outros processos e os armazena. Neste caso, a variação (figura 4.13) indica a quantidade de dados a ser enviada ao processo *raiz* por cada processo e a expressão é o número do processo raiz.

- **ALL_GATHER:** A operação *all_gather* é semelhante à operação *gather*, mas em vez de apenas o processo raiz colher os dados, todos os outros processos também os recebem. A regra da gramática desse comando pode ser vista na figura 4.13.

- **SCATTER:** A operação *scatter* é uma operação inversa à operação *gather*. Na operação de distribuição *scatter*, o processo *raiz* distribui seus dados entre os outros processos. Os parâmetros utilizados na figura 4.13 têm a mesma finalidade que os parâmetros da função *gather*: a variação indica a quantidade de dados que será distribuída para cada processo e a expressão indica o processo que irá realizar a distribuição.

- **ALL_TO_ALL:** Esta é uma operação que combina a dis-

tribuição (*scatter*) e a coleta (*gather*). Todos os processos enviam quantidades iguais de seus dados para todos os outros processos que os coletam. A regra para a construção dessa função pode ser vista na figura 4.13, sendo que a variação indica a quantidade de dados enviada para cada processo.

- **REDUCE:** A operação de redução é aquela em que cada um dos processos fornece um valor para o cálculo de algum resultado global, como por exemplo uma somatória, ou um produtório. A regra da função *reduce* é mostrada na figura 4.13, onde a expressão indica o processo raiz que receberá o resultado final da operação global e a variação indica a quantidade de dados fornecida por cada processo.
- **ALL_REDUCE:** A operação *all_reduce* é parecida com a operação *reduce*, com a diferença que não só o processo raiz recebe o resultado, mas todos os outros processos também o recebem. A sintaxe dessa operação pode ser vista na figura 4.13.

Maiores informações sobre as operações coletivas podem ser encontradas em [10].

4.2 Desenvolvimento do Tradutor

Um tradutor é utilizado para traduzir a descrição simplificada do código escrito utilizando a linguagem de protótipo em um código C++. Foi implementado para avaliação das regras de semântica em uma árvore gramatical utilizando o caminhamento em árvore binária *em-ordem* [12]. Nesse caminhamento os nós são visitados apenas uma

vez, na seguinte ordem: primeiro visita o filho da esquerda, visita o nó raiz, e visita o filho da direita (EPD). Para seu desenvolvimento foram utilizadas ferramentas de construção de compiladores como Bison [13] (gerador de *parser*) e Flex [14] (analisador léxico), GCC e G++ (compiladores para C e C++ respectivamente).

Conforme mostra a figura 4.14, após o protótipo ser escrito utilizando a linguagem ele é então traduzido, ou seja, todas as regras de semânticas e de sintaxe descritas no código são avaliadas. O tradutor gera uma saída para cada sentença que tenha sido gerada pela gramática através da execução na ordem do caminhamento da árvore citada acima. As saídas do tradutor são escritas em um arquivo. A saída gerada pelo tradutor é um código C++ que será utilizado posteriormente na simulação do sistema.

4.2.1 Analisador Sintático - BISON

O Bison [13] é um gerador de analisador sintático para uma linguagem com gramática livre de contexto. A forma mais comum de apresentar as regras é através da *Backus-Naur Form* ou notação BNF. A notação BNF é uma metalinguagem para linguagens de programação. Uma gramática BNF é composta por um conjunto finito de regras que definem uma linguagem de programação.

Nas regras gramaticais para a linguagem, cada tipo de unidade sintática ou grupo é chamado *símbolo*. Aqueles que são construídos por grupos menores de acordo com as regras gramaticais são denominados *símbolos não-terminais*; aqueles que não podem ser subdivididos são chamados *símbolos terminais*. Um trecho de entrada correspondente a um único símbolo terminal é chamado *token* e um trecho correspondente a um único símbolo não-terminal é denominado *grupo*.

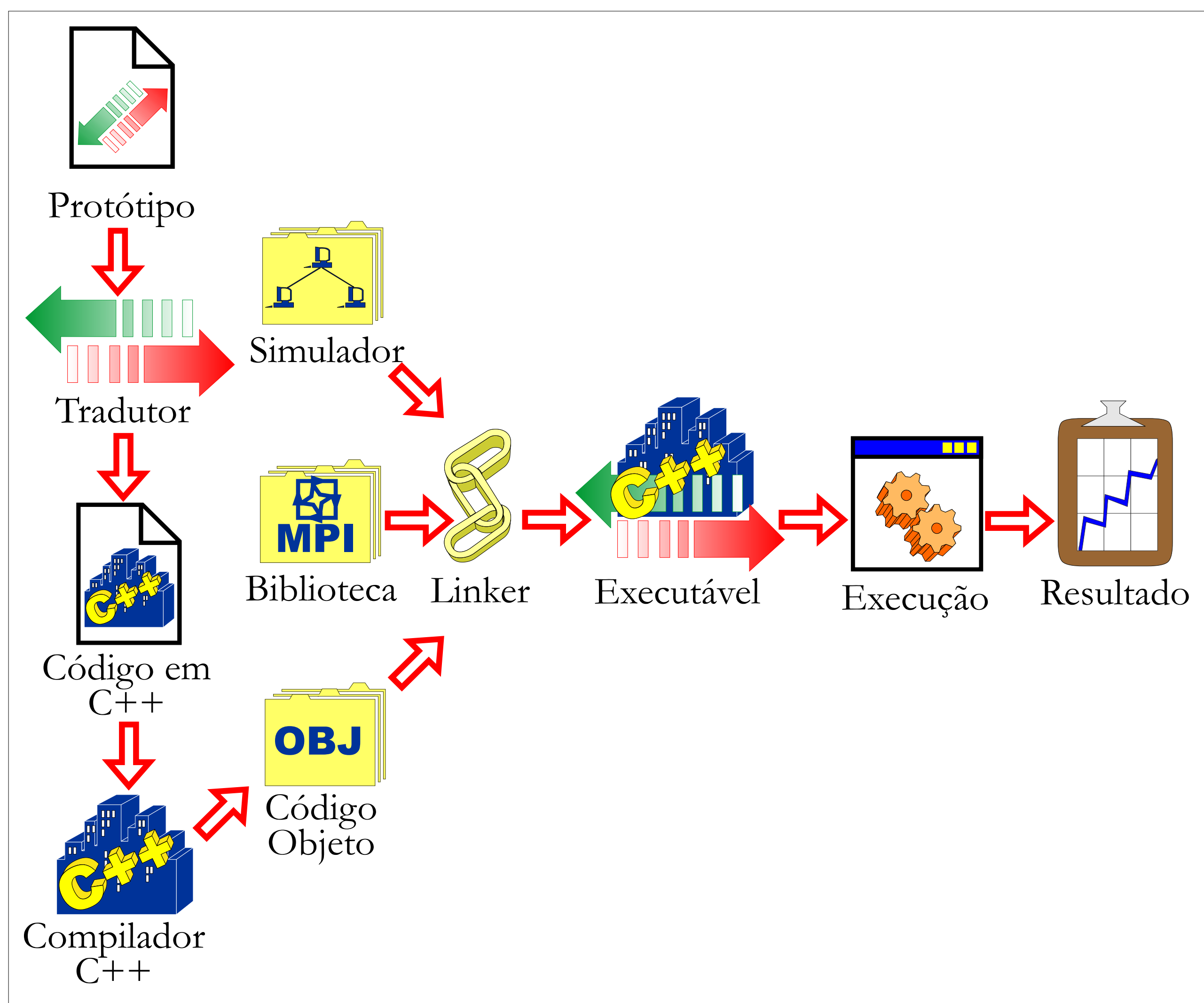

Figura 4.14: Diagrama de Execução

```
ident: IDENT
source: ANYSOURCE
atribuicao: ident '=' exp
exp:
     ident
   | NUM
   | exp '+' exp
   | exp '-' exp
   | exp '*' exp
   | exp '/' exp
   | exp '%' exp
   | OPEN_BRACE exp CLOSE_BRACE
if_comando:
    IF OPEN_BRACE comparação CLOSE_BRACE bloco
  | IF OPEN_BRACE comparação CLOSE_BRACE bloco ELSE bloco
  | IF OPEN_BRACE expressão  CLOSE_BRACE bloco ELSE bloco
while_comando:
    while OPEN_BRACE comparacao CLOSE_BRACE bloco
  | while OPEN_BRACE variacao CLOSE_BRACE bloco
compute_comando:
    compute OPEN_BRACE variacao CLOSE_BRACE
```

Figura 4.15: Trecho das regras gramaticais da linguagem de protótipo em formato Bison

Uma regra é composta pelo seu lado esquerdo e direito, separado por dois pontos (:). O lado esquerdo contém um símbolo não-terminal. O lado direito contém símbolos terminais ou não-terminais, podendo ter cada símbolo não-terminal várias regras alternadas, juntadas por uma barra | que é lida como "ou".

Algumas das regras utilizadas, como por exemplo a regra das expressões e do comando *if*, utilizando a notação BNF usada pelo Bison podem ser vistas na figura 4.15. O arquivo completo com as regras Bison para a linguagem se encontra no apêndice A.

4.2.2 Analisador Léxico - FLEX

O Flex [14] é uma ferramenta para gerar programas que reconhecem *strings* como símbolos válidos na linguagem. O Flex lê a descrição de um *"scanner"* de um arquivo de entrada, ou da entrada padrão

se nenhum nome de arquivo é dado. A descrição é na forma de pares de expressões regulares e código C. Um arquivo C é gerado como saída.

Quando um “scanner” gerado é executado, ele analisa suas entradas procurando *strings* que combinem com seus moldes (as expressões regulares). Se ele encontra mais do que uma, pega a que combine com a maioria do texto. Se ele encontra duas ou mais combinações de mesmo comprimento, a regra listada primeiro no arquivo de entrada *Flex* é escolhida. Quando nenhuma combinação é encontrada, então a regra *default* é executada: o próximo caracter na entrada é copiado para a saída.

Cada molde em uma regra tem uma ação correspondente, que pode ser qualquer comando C. Se uma ação é vazia, então quando o molde combina o *token* de entrada é simplesmente descartado.

A variável *yylval* utilizada na declaração das regras mostradas na figura 4.16 é utilizada para retornar os valores associados com os tokens. No Bison por exemplo, um valor inteiro é armazenado em *yylval* e o token retornado pelo Flex é NUM.

Algumas regras definidas na linguagem utilizando *Flex* podem ser vistas na figura 4.16. A primeira regra apresentada na figura, a regra dos números, retorna números inteiros (positivos ou negativos) ou um número *double* (um ou mais dígitos seguidos por um “.”, que pode ser seguido de zero ou mais dígitos).

A regra dos identificadores combina com uma letra (minúscula ou maiúscula) seguida por zero ou mais letras e dígitos. As palavras entre "" indicadas nos comandos são aquelas que o analisador léxico reconhece no arquivo de entrada como palavras-chave. O mesmo acontece com os operadores relacionais, o que está entre "" são os símbolos que retornarão os tokens definidos na parte sintática da

```
/* Números */
 [-+]?[0-9]+("."[0-9]*)?([eE]"-"?[0-9]*)?
        { yylval.pont->val = atof(yytext);
          return NUM;
        }
/* Identificadores */
 [a-zA-Z][a-zA-z0-9]* { strncpy(yylval.pont->nome,
                                yytext, 256);
                        return IDENT;
                      }
/* Comandos */
 "if"        { return IF; }
 "else"      { return ELSE; }
 "send"      { return SEND; }
 "receive"   { return RECEIVE; }

/* Operadores Relacionais */
 "=="   { return EQ; }
 "!="   { return NE; }
 ">"    { return GQ; }
 "<="   { return LE; }
```

Figura 4.16: Trecho das regras Flex do analisador léxico para a linguagem de protótipo

linguagem.

Mais detalhes sobre o analisador léxico *Flex* podem ser encontrados em [14]. O arquivo com todas as regras Flex da linguagem é apresentado no apêndice B.

4.3 Compilação

Um protótipo utilizando a linguagem proposta pode ser gerado a partir de qualquer editor de texto. O arquivo não precisa ser salvo com nenhuma extensão específica. A figura 4.17 mostra o código escrito utilizando a linguagem antes de ser compilado. O código mostrado é uma operação ping-pong em que se tem dois processos, um que envia mensagem para o outro processo que a devolve.

Para compilarmos o código acima precisamos apenas digitar *prot*

```
loop = 100;
if (rank == 0){
  for (i,loop){
    send(1,(8192,0));
    receive(1);
  };
}
else {
  for (i,loop){
    receive(0);
    send(0,(8192,0));
  };
};
```

Figura 4.17: Código antes da compilação

(nome do tradutor) e o nome do arquivo a ser compilado. No caso do exemplo do ping-pong mostrado na figura 4.17 sua compilação ficaria (supondo que o código esteja num arquivo de nome ping-pong):

```
prot ping-pong
```

O tradutor gera então o código em C++ que pode ser visto na figura 4.18. Esse código gerado será executado pelo simulador de rede descrito no capítulo 5.

```
#include<iostream.h>
#include<stdio.h>
#include<math.h>

#include "biblioteca.h"

void processador(int rank){
  double loop=100;

  if (rank==0) {

    for(int i=0; i<(loop); i++) {
      double tmp00000=Var(8192,0);
      send(1,tmp00000);
      receive(1);
    }
  }
  else {
    for(int i=0; i<(loop); i++) {
      receive(0);
      double tmp00001=Var(8192,0);
      send(0,tmp00001);
    }
  }
}
```

Figura 4.18: Código gerado pela compilação do programa Ping-pong

Capítulo 5

Simulador

Simuladores têm se tornado uma ferramenta indispensável em diversas áreas científicas e tecnológicas. Diversas razões contribuem para sua popularidade, entre elas: a possibilidade de estudar as características de um sistema antes que ele seja construído; a possibilidade de estudar características do sistema que seriam difíceis, impossíveis ou muito custosas de serem estudadas em laboratório; a capacidade de verificar modelos teóricos, pela comparação de simulação baseada nesses modelos com resultados experimentais. Quatro fases distintas são identificadas [15]. São elas:

- **Identificação do Sistema:** um sistema é definido como uma coleção de objetos, seus relacionamentos e comportamentos, relevantes para um conjunto de propósitos. Um modelo sempre depende do objetivo e contexto da aplicação.

- **Representação do Sistema:** onde imagens simbólicas de objetos, relacionamentos e comportamentos são estruturadas como parte de algum *framework* maior de opiniões, suposições e teorias da resolução do problema. Teorias podem ser descritivas, usadas para prever e explicar, ou podem ser normativas,

usadas para prescrever o que deveria ser feito. Diferentes tipos de teorias produzem diferentes tipos de modelos.

- **Projeto do Modelo:** um modelo é uma representação das estruturas e processos. Modelos podem crescer rapidamente para formar estruturas muito complicadas. Abstração e agregação são técnicas importantes e podem manter modelos de fenômenos complexos gerenciáveis. Deve ser tomado grande cuidado em preservar as caracteristicas essenciais ao sistema.

- **Codificação do Modelo:** o código do modelo procura uma representação formal das estruturas dos símbolos e suas transformações dentro das estruturas de dados e procedimentos computacionais em alguma linguagem de programação.

A modelagem segundo Jensen e Kreutzer apresentada em [16, 15] pode ser realizada de várias formas, como por exemplo:

Métodos de Monte Carlo: são usados para estimar probabilidades de estados de modelos através de experimentação dirigida por amostragem. Somente correlações entre valores de entrada e de saída são explorados. Os métodos de Monte Carlo podem ser empregados para estudar fenômenos complexos que não são bem entendidos para análises mais detalhadas de causa e efeito, ou quando soluções para modelos não probabilísticos são complicadas ou computacionalmente caras. O primeiro tipo de modelo freqüentemente surge no contexto de estudos sociais e fenômenos econômicos (como previsão econômica), ao passo que o segundo tipo é bem exemplificado por métodos numéricos para resolução de integrais múltiplas.

Simulação Contínua: a técnica de modelagem contínua originou-se

da tradição de construção de dispositivos análogos para modelagem e simulação de sistemas. Modelos contínuos são empregados se o comportamento do sistema depende mais do fluxo agregado de processos do que de estados individuais e suas modificações. As atividades predominantes do modelo causam mudanças nas suas variáveis. A precisão é dependente da escolha do incremento de tempo.

Simulação de Eventos Discretos: modela sistemas como coleções estruturadas de objetos e um conjunto de relações e transformações. Enquanto simulação contínua abstrai descontinuidade, modelos dirigidos a eventos assumem que nada de relevante acontece entre sucessivas transições de estado. Embora uma grande variedade de sistemas possam ser estudados usando técnicas de simulação discreta, o paradigma de rede de filas é o mais popular. O objetivo é explorar os efeitos de limitações de capacidade e estratégias de roteamento no fluxo de algumas classes de transações. Os objetos de interesse são tipos diferentes de recursos, filas e itens de carga de trabalho. Teorias de fila podem ser usadas para analisar tais modelos, mas somente se eles têm uma estrutura relativamente simples.

Cinco componentes de qualquer simulação de eventos discretos são:

1. Elementos para modelar estrutura e execução. Uma coleção de objetos interagindo através de seqüências de ações.
2. Um relógio para o gerenciamento de tempo. É tipicamente assíncrono, salta o valor do relógio para o tempo do próximo evento, uma vez que se estabeleceu que nenhuma ação adicional pode ser executada até esse instante.

3. Processos aleatórios para representar aspectos menos essenciais.
4. Elementos para experimentação estatística e relatório, para compreender e analisar dados.
5. Uma agenda para escalonar e guardar sinais de eventos pendentes. *Arrays* podem ser usados, mas listas ligadas são mais flexíveis.

Simulação Combinada Contínua/Discreta: tenta integrar eventos discretos e a abordagem de mudança contínua para o estudo do comportamento de um sistema. Um modelo de simulação combinada pode refletir um processo que progride de uma maneira essencialmente contínua, com eventos discretos modelando descontinuidade ocasional, ou pode ser dirigido por um escalonador baseado em eventos, com alguns subprocessos regulares representados continuamente.

O simulador desenvolvido neste trabalho é baseado em simulação orientada a eventos — dado o estado inicial do sistema computam-se os tempos de respostas do sistema a esse estado, e as mudanças ocorridas são registradas como eventos. A cada evento é associado o instante em que ele ocorrerá. Assim, subsistemas mais lentos geram eventos que ocorrerão em instantes posteriores, enquanto subsistemas mais rápidos geram eventos que ocorrerão brevemente. Desta forma, o andamento do tempo de simulação se adapta dinamicamente às velocidades dos subsistemas, evitando realização de cálculos desnecessários.

A simulação dirigida a eventos é adequada no nosso caso, pois estamos preocupados em analisar apenas o que acontece no ínicio e no final de cada evento de comunicação, e não ficar a todo instante

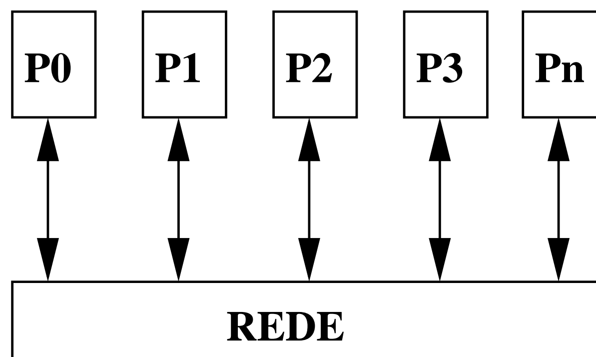

Figura 5.1: Ligação entre a rede e os processadores

verificando o estado do sistema como acontece com a simulação contínua e nem fazer apenas análises estatísticas dos valores de entrada e saída como no método de Monte Carlo.

5.1 Descrição do Simulador Implementado

O simulador baseia-se no modelo da máquina paralela, o multicomputador, descrito na subseção 2.2. A figura 5.1 mostra que os vários processadores comunicam-se entre si através da rede.

5.1.1 Parâmetros

O custo de envio de uma mensagem de um processador para outro é calculado utilizando a equação (2.1). Os valores de t_s e t_w foram calculados utilizando os dados e os tempos obtidos na execução de um programa ping-pong no *cluster* descrito no capítulo 6. A partir dos dados, mostrados na tabela (5.1), utilizamos o método dos mínimos quadrados para ajustar os valores dos parâmetros t_s e t_w.

O gráfico mostrado na figura 5.2 apresenta o tempo de transmissão e o tamanho das mensagens transmitidas de acordo com os

Tabela 5.1: Tempo de transmissão em função do tamanho da mensagem

Tamanhos(bytes)	*Tempos*(seg)	*Tamanhos*(bytes)	*Tempos*(seg)
8	0.000056	8192	0.001005
16	0.000058	16384	0.001743
32	0.000062	32768	0.003189
64	0.000069	65536	0.006090
128	0.000083	131072	0.011967
256	0.000112	262144	0.023646
512	0.000168	524288	0.046897
1024	0.000276	1048576	0.093495
2048	0.000356	2097152	0.186626
4096	0.000529		

dados mostrados na tabela 5.1. Como podemos observar o gráfico apresenta uma curva com três comportamentos diferentes. Os gráficos 5.3, 5.4 e 5.5 apresentam separadamente os três comportamentos da curva.

São apresentados três gráficos com os dados contidos na tabela 5.1. O gráfico mostrado na figura 5.3 apresenta o tempo de transmissão e as mensagens menores ou iguais à 1024 *bytes*, onde a transmissão é feita em apenas um pacote. Quando as mensagens são menores do que 5000 *bytes* a comunicação feita pelo MVIA [17] é do tipo não bloqueante. Os valores dos parâmetros t_s e t_w encontrados nesse caso são $55\mu s$ e $0.22\mu s$ respectivamente.

Para mensagens maiores do que 1024 *bytes* e menores do que 5000 *bytes*, a comunicação ainda é do tipo não bloqueante e os valores dos parâmetros t_s e t_w são $190\mu s$ e $0.083\mu s$ respectivamente.

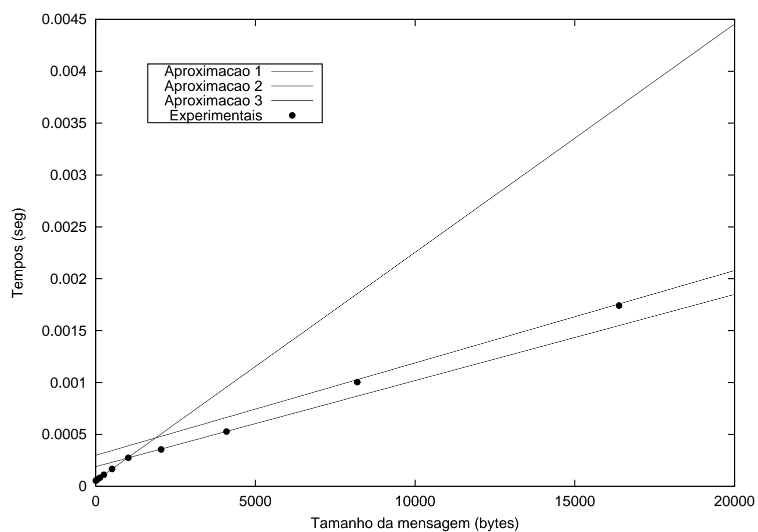

Figura 5.2: Tempo de transmissão

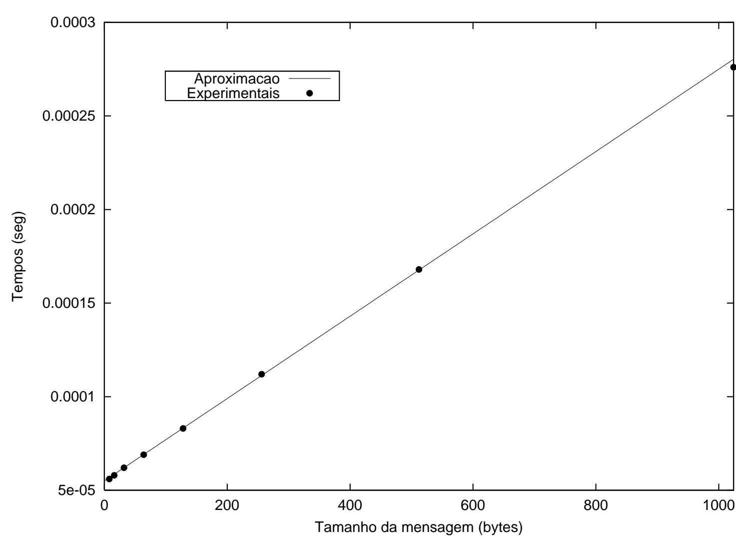

Figura 5.3: Tempo de transmissão - Mensagens menores de 1024 *bytes*

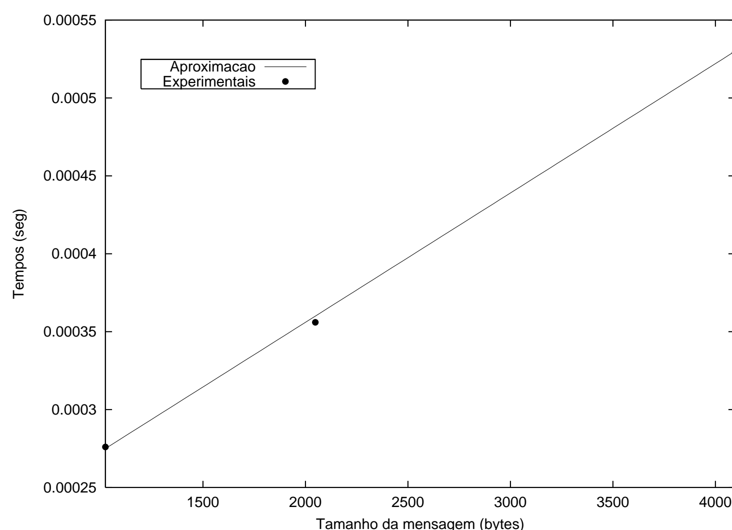

Figura 5.4: Tempo de transmissão - Mensagens maiores de 1024 e menores de 5000 *bytes*

A figura 5.4 apresenta o gráfico com mensagens maiores de 1024 e menores de 5000 *bytes*.

Os valores dos parâmetros t_s e t_w para mensagens acima de 5000 *bytes* são de $300\mu s$ e $0.089\mu s$ respectivamente, e a comunicação passa a ser bloqueante. A figura 5.5 apresenta o gráfico mostrando o tempo de transmissão e o tamanho das mensagens de 5000 até 2MB.

Os dados mostrados na tabela 5.1 foram obtidos a partir da execução no *cluster* do programa ping-pong, sendo que os tempos relatados foram os tempos mínimos obtidos em sua execução.

O simulador verifica então, qual o tamanho das mensagens e calcula o tempo da transmissão de acordo com os valores de t_s e t_w discutidos nesta seção.

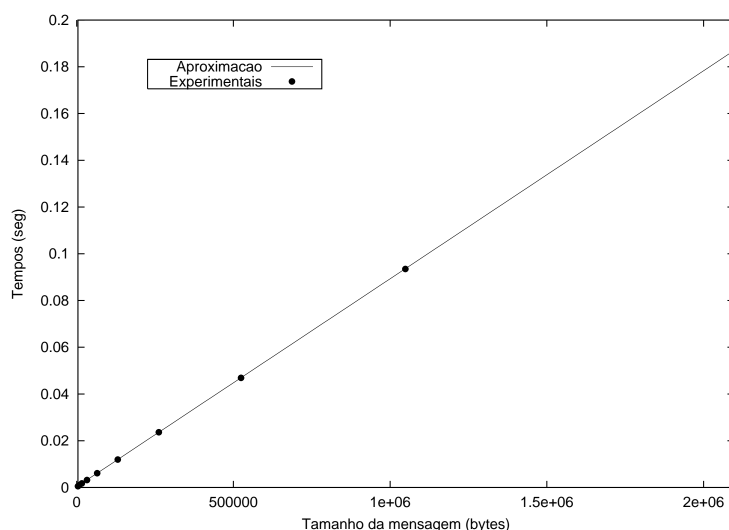

Figura 5.5: Tempo de transmissão - Mensagens maiores de 5000 *bytes*

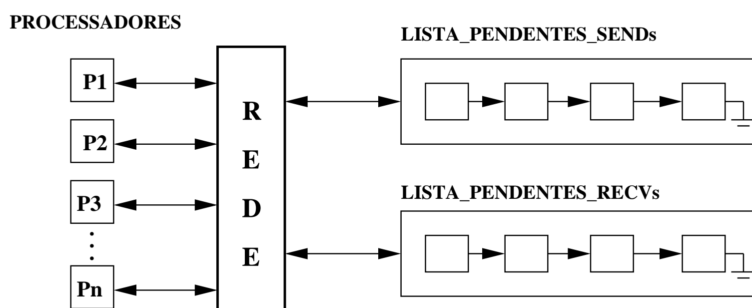

Figura 5.6: Modelo do Simulador de Rede

5.1.2 Organização

A figura 5.6 mostra a arquitetura do modelo do simulador desenvolvido para uso neste trabalho, com elementos para simular vários processadores, e uma rede de interconexão que usa listas de eventos pendentes para envio e recepção.

Conceitos importantes na operação do simulador (figura 5.6) são:

- **Mensagens:** os processos comunicam-se com a rede através

de mensagens. Uma mensagem contém dados a serem transmitidos, processo remetente, processo destinatário, além de informações de identificação. As mensagens são representadas por estruturas com informações de identificação que podem ser do tipo SND, RECV e FIM.

O tipo *SND* indica que a mensagem é do tipo *send*, ou seja, algum processo está enviando a mensagem para seu processo parceiro (o destinatário). Quando o identificador da mensagem é *RECV* indica que o processo destino está pronto para a recepção. Caso o identificador da mensagem seja do tipo *FIM* indica que o processo terminou todas as suas comunicações.

- **Eventos:** os eventos são cargas de comunicação geradas pela simulação dos processadores. Cada evento (carga) gerado possui: um identificador que indica se o evento gerado é do tipo *send* (SND), ou do tipo *receive* (RECV); possui um campo que indica o tamanho da mensagem (o número de bytes) a ser transferida; cada evento tem associado um tempo, que é o valor do tempo de ocorrência. Os eventos são gerados a partir das mensagens enviadas ao simulador de rede e são armazenados nas listas de eventos pendentes.

- **Lista de Eventos Pendentes:** o simulador possui duas listas de eventos pendentes. Uma lista para eventos gerados a partir de um *send* e outra lista para eventos gerados a partir de um *receive*. Essas listas são utilizadas sempre que um novo evento (carga) for gerado durante a simulação.

- **Processadores:** simulam os processadores gerando mensagens. As mensagens são geradas a partir de códigos criados utilizando a linguagem apresentada no capítulo 4 que são exe-

cutados no simulador.

5.2 Funcionamento do Simulador

A figura 4.14 apresenta o diagrama de execução da linguagem e do simulador para obtenção dos resultados que são mostrados no capítulo 6. Seguindo a figura temos que primeiramente escrever o código utilizando a linguagem definida no capítulo 4, em seguida o código será traduzido utilizando o tradutor descrito no capítulo 4 na seção 4.2 que gera um código C++. O código C++ gerado será compilado gerando um código objeto. Esse código objeto será ligado com a biblioteca e o simulador gerando assim o código executável. Após a execução podemos obter os resultados que são mostrados no capítulo 6.

Como descrito anteriormente o simulador opera com mensagens, listas de eventos pendentes, processadores e rede que quando começa a receber mensagens com eventos do tipo *SND* ou *RECV*, inicia a simulação da transmissão. A rede se comunica com os processos origem e destino para enviar o valor do tempo atualizado. O valor do tempo é calculado utilizando a equação (2.1).

Para simular a execução paralela dos processadores e cuidar da sincronização do acesso aos recursos da rede foi utilizado MPI [10] como sistema de comunicação entre processos.

O simulador de rede e processadores são executados como processos distintos usando funções MPI para a comunicação. Os processadores executam os códigos gerados pela linguagem (capítulo 4) e esses códigos fazem chamadas para funções que foram definidas e que se comunicam com o simulador de rede. A figura 5.7 apresenta como exemplo a implementação da função *send*.

```
#include<iostream.h>
#include<stdio.h>
#include<math.h>
#include<mpi.h>

#include "biblioteca.h"
#include "rede.h"

double send(double p, double d, int tag){
  int pi = int(p);
  int rank;

  MPI_Status status;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  fprintf(arqTempo, " Tempo inicial na funcao
                      Send = %f\n", T/1000000.0);

  MPI_Send(&T, 1, MPI_DOUBLE, 0, SND, MPI_COMM_WORLD);
  MPI_Send(&pi, 1, MPI_INT, 0, SND, MPI_COMM_WORLD);
  MPI_Send(&d, 1, MPI_DOUBLE, 0, SND, MPI_COMM_WORLD);
  MPI_Send(&tag, 1, MPI_INT, 0, SND, MPI_COMM_WORLD);
  MPI_Recv(&T, 1, MPI_DOUBLE, 0, SND, MPI_COMM_WORLD,&status);

  fprintf(arqTempo, " Tempo final na funcao
                      Send = %f\n", T/1000000.0);
}
```

Figura 5.7: Implementação da função *send* definida na linguagem

O simulador de rede possui um vetor contendo informação de estado sobre todos os processos. Os valores atribuídos aos processos contidos no vetor podem ser de três tipos: COMP (quando o processo está realizando alguma computação), ESP (quando o processo está esperando alguma mensagem) e TERM (quando o processo terminou todas as suas comunicações). Inicialmente todos os processos contidos no vetor estão no estado COMP.

As variáveis *p*, *d* e *tag* na figura 5.7, indicam o número do processo parceiro, a quantidade de dados a serem transmitidos e o *tag* da mensagem respectivamente. Cada processo de um processador mantém um relógio que será atualizado de acordo com as informações do simulador. Os processadores enviam então dados sobre a comunicação (o tempo, o parceiro, a quantidade de dados e o *tag*; como mostram as três operações de envio na figura 5.7), para o simulador de rede. O simulador recebe esse dados e os processa. Através dos dados (mensagens) recebidos, o simulador de rede verifica qual é o seu tipo. Existem no simulador dois tipos de listas de eventos pendentes, a de *send* e a de *receive*.

Inicialmente quando a rede recebe uma mensagem, seja ela do tipo *SND* ou *RECV*, a rede gera um evento que é inserido, dependendo do tipo da mensagem, na lista de eventos pendentes de *send* ou *receive*, e o processo que enviou a mensagem, é então marcado no vetor como ESP. Após o evento ter sido inserido em uma das duas listas, o vetor com os processos é percorrido e se algum dos processos estiverem ainda marcados como COMP, a rede continua recebendo as mensagens dos processos. Quando todos os processos no vetor estiverem marcados como ESP, o simulador de rede percorre então as duas listas comparando os eventos que foram inseridos, quando um evento de uma das listas combina com o outro,

```
while (7,1){
  bcast(i,(7,0));
};
```

Figura 5.8: While com variação

os dois são então retirados das listas.

O simulador de rede calcula e devolve o tempo de conclusão das transmissões para os processadores, que os utilizam para atualizar seus relógios. A atualização dos relógios dos processos é feita através da função MPI_Recv no término da função mostrada na figura 5.7.

O tempo de resposta que é enviado aos processos é calculado como mostrado na equação (2.1) com t_s e t_w escolhidos de acordo com o tamanho das mensagens, como discutido na seção 5.1.1. O processo que realiza um *send* ou um *receive* só terá seu tempo atualizado quando a mensagem correspondente já tiver sido postada, pois as operações são do tipo bloqueante. Depois de devolver o tempo atualizado para os processos remetente e destinatário, esses são então marcados no vetor como COMP.

A função *compute* mostrada na figura 4.11 apenas incrementa o relógio após simular um intervalo de computação, não tendo assim nenhuma comunicação com o simulador de rede.

Um tratamento especial é necessário para os comandos *while* e *if* com variação. O código mostrado na figura 5.8 apresenta o comando *while* utilizando variação, seguido de um *broadcast*. Supondo que temos dois processos (P0 e P1), e os dois começam a executar o código da figura 5.8; o número sorteado na variação para os dois processos pode ser diferente, um menor que o outro; assim um dos processos terminaria primeiro sua execução, fazendo com que o processo com o número maior sorteado ficasse esperando no pró-

```
if (rank == 0){
 tmp00001 = Var(7,1);
};
MPI_Bcast(&tmp00001,1,MPI_DOUBLE,0,simCom);
while(tmp00001 > 0){
  bcast(i,(7,0));
  tmp00001--;
};
```

Figura 5.9: Broadcast no código da figura 5.8

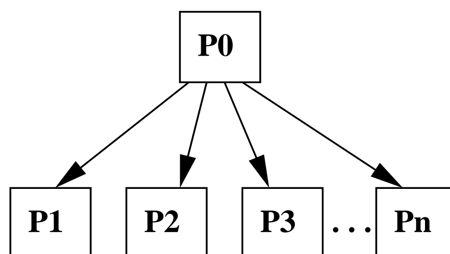

Figura 5.10: Função Broadcast

ximo *broadcast*. Para evitar que isso aconteça, primeiro sorteamos um número de acordo com a variação no processo 0, e esse é então passado a todos os outros processadores, como pode ser visto na figura 5.9. Na figura 5.9 é mostrado o comunicador *simCom* que é utilizado por todos os processos, menos pelo simulador de rede. Algo semelhante é feito com o comando *if* quando ele utiliza uma expressão (figura 4.8).

As operações coletivas apresentadas no capítulo 4 são implementadas usando as funções *send* e *receive* definidas na linguagem. Por exemplo, na função *broadcast* o processo raiz faz um *send* enviando a quantidade de dados para todos os outros processos que os recebem, como mostra a figura 5.10.

Quando todos os processadores terminam as suas operações eles enviam uma mensagem com o identificador do tipo FIM para o simulador de rede, que quando recebe essa mensagem marca no

vetor o processador que a enviou como TERM, e quando recebe de todos os processadores termina a simulação. As informações sobre os tempos na chamada e conclusão de todas as operações de comunicação são registradas em arquivos (um para cada processador), conforme mostra a figura 5.7.

O modelo de simulação utilizado sem uma agenda centralizada só é válido por causa do modelo simples de rede simulado, onde as diferentes mensagens não têm dependência entre si.

Capítulo 6

Resultados

Para validar o simulador foram realizados testes utilizando o *cluster* que foi adquirido pelo nosso grupo de pesquisa (processo FAPESP 1998/14681-8), e testes utilizando a linguagem (capítulo 4) e o simulador (capítulo 5) desenvolvidos no trabalho.

Para os testes foi utilizado um *cluster* de 16 nós (CPU) interligados por duas redes (*10Mb/s* e *100Mb/s*). Cada nó possui um processador AMD K6-III e 256Mb de memória RAM local. O sistema operacional que está sendo utilizado é *Linux*, distribuição *SLACKWARE 8.0* [18] e o sistema de passagem de mensagem é o MPI (Message Passing Interface), implementação MPICH [19] sobre M-VIA (MVICH).

Para realização dos testes foram utilizados programas construídos utilizando C, C++ e MPI. A compilação dos programas foi feita como mostra a figura 6.1.

O comando *mpicc* (mostrado na figura 6.1) é utilizado quando os

```
mpicc nome-do-arquivo -o nome-do-arquivo-executável
mpiCC nome-do-arquivo -o nome-do-arquivo-executável
```

Figura 6.1: Compilação dos programas

programas são criados em C, e quando são criados utilizando C++ o comando utilizado na compilação é o *mpiCC*. Após compilado o programa, para executá-lo basta utilizar a linha de comando abaixo.

```
mpirun -np número-de-processos nome-do-arquivo-executável
```

6.1 Ping-Pong

Os testes com o programa ping-pong foram realizados utilizando dois processos enviando e recebendo mensagens. Os testes foram realizados com diferentes quantidades de dados, como pode ser visto na tabela 6.1.

O código do programa ping-pong utilizado para os testes, pode ser visto na figura 6.2. Como podemos observar no código, os processos 0 e 1 trocam mensagens entre si, ou seja, o processo 0 envia mensagem ao processo 1 que a recebe e esse então, envia uma mensagem ao processo 0. Essa troca é simulada 100 vezes.

O programa ping-pong pode ser utilizado para verificação do funcionamento do simulador de rede e verificação dos tempos de comunicação calculados. Como pode ser visto na figura 4.17 a comunicação é feita apenas entre dois processos que enviam e recebem mensagens alternadamente. Neste teste haverá apenas um par de eventos pendente por vez.

A tabela 6.1 mostra a comparação feita com os tempos obtidos na execução do programa ping-pong com mensagens de diversos tamanhos no *Cluster* e no Simulador. A diferença percentual apresentada na tabela 6.1 foi calculada a partir da equação (6.1).

$$P = \frac{\text{TempoSimulador} - \text{TempoCluster}}{\text{TempoCluster}} \cdot 100 \tag{6.1}$$

```
loop = 100;
if (rank==0) {
  for (i,loop){
    send(1,(8192,0));
    receive(1);
  };
}
else {
  for (i,loop){
    receive(0);
    send(0,(8192,0));
  };
};
```

Figura 6.2: Código do programa Ping-pong

O gráfico apresentado na figura 6.3 mostra os tempos de transmissão obtidos através da execução do programa ping-pong (mostrado na figura 6.2) no *cluster* e no simulador desenvolvido no trabalho (capítulo 5). Os dados mostrados no gráfico são os apresentados na tabela 6.1.

6.2 Anel

O exemplo do anel é mostrado na figura 6.4, onde se tem um número n de processos. Todos os processos enviam seus dados para seu processo sucessor, e recebem dados de seu processo antecessor. O código do programa anel pode ser visto na figura 6.5, onde o processo 0 (primeiro processo) sempre recebe dados do último processo. E o último processo sempre envia seus dados para o processo 0. Esse programa foi utilizado para teste, pois através dele é possível verificar o funcionamento do simulador com diversas comunicações pendentes simultaneamente.

Para os testes realizados no programa do anel foram utilizados de 2 a 16 processos, e a quantidade de dados a ser transmitida

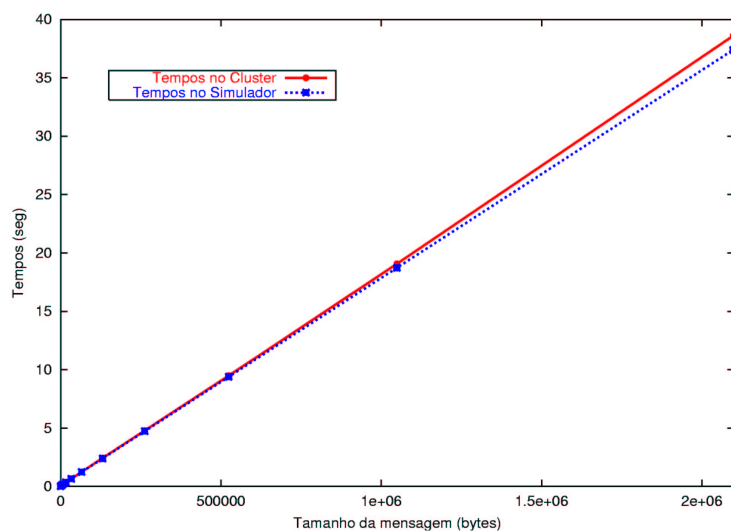

Figura 6.3: Tempos de transmissão obtidos no programa ping-pong executado no *Cluster* e no Simulador

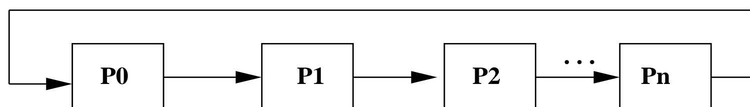

Figura 6.4: Anel

```
if (rank % 2 == 0){
 if (rank < P-1) {
   send(rank+1, (10000,0));
 }
 else {
   send(0,(10000,0));
 };
 if (rank > 0) {
   receive(rank-1);
 }
 else {
   receive(P-1);
 };
}
else{
  if (rank > 0) {
    receive(rank-1);
  }
  else {
    receive(P-1);
  };
  if (rank < P-1) {
    send(rank+1, (10000,0));
  }
  else {
    send(0,(10000,0));
  };
};
```

Figura 6.5: Código do programa Anel

Tabela 6.1: Comparação dos tempos de execução do programa ping-pong no *cluster* e no simulador

Tamanho(bytes)	*TempoCluster*(seg)	*TempoSimulador*(seg)	*Diferença*
8	0.011954	0.011352	−5.04%
16	0.012258	0.011704	−4.52%
32	0.013108	0.012408	−5.34%
64	0.014573	0.013816	−5.19%
128	0.017482	0.016632	−4.86%
256	0.023330	0.022264	−4.57%
512	0.035203	0.033528	−4.76%
1024	0.059027	0.056056	−5.03%
2048	0.076439	0.071997	−5.81%
4096	0.111411	0.105994	−4.86%
8192	0.206023	0.205818	−0.099%
16384	0.351350	0.351635	0.08%
32768	0.645088	0.643270	1.27%
65536	1.226511	1.226541	0.002%
131072	2.428278	2.393082	−1.45%
262144	4.776908	4.726163	−1.06%
524288	9.491226	9.392326	−1.04%
1048576	19.068304	18.724724	−1.80%
2097152	38.591679	37.389270	−3.12%

de 10000 *bytes*. A tabela 6.2 mostra os tempos obtidos usando 16 processos e a diferença percentual entre os tempos utilizando a equação (6.1).

Como pode ser visto nos tempos resultantes, os tempos com to-

Tabela 6.2: Tempos de transmissão (em segundos) de 10000 bytes obtidos na execução do programa anel no Cluster e no Simulador

Processos	*TempoCluster*(seg)	*TempoSimulador*(seg)	*Diferença*
2	0.002569	0.002380	−7.35%
3	0.003786	0.003570	−5.70%
4	0.002607	0.002380	−8.70%
5	0.003781	0.003570	−5.58%
6	0.002699	0.002380	−11.82%
7	0.003676	0.003570	−2.88%
8	0.002621	0.002380	−9.19%
9	0.003694	0.003570	−3.36%
10	0.002661	0.002380	−10.56%
11	0.003769	0.003570	−5.28%
12	0.002772	0.002380	−14.14%
13	0.003694	0.003570	−3.36%
14	0.002814	0.002380	−15.42%
15	0.003723	0.003570	−4.11%
16	0.002791	0.002380	−14.73%

tal de processos ímpares é maior que com o total de processos pares, isto acontece porque um número par de processos precisa de apenas dois ciclos para terminarem a comunicação, como pode ser visto na figura 6.6, enquanto que os processos ímpares precisam de três ciclos de comunicação (um ciclo a mais). O simulador reproduz exatamente esse comportamento.

O gráfico apresentado na figura 6.7 mostra os tempos de transmissão obtidos na execução do programa anel (figura 6.5) utilizando o *cluster* e o simulador. Os dados utilizados são os apresentados na

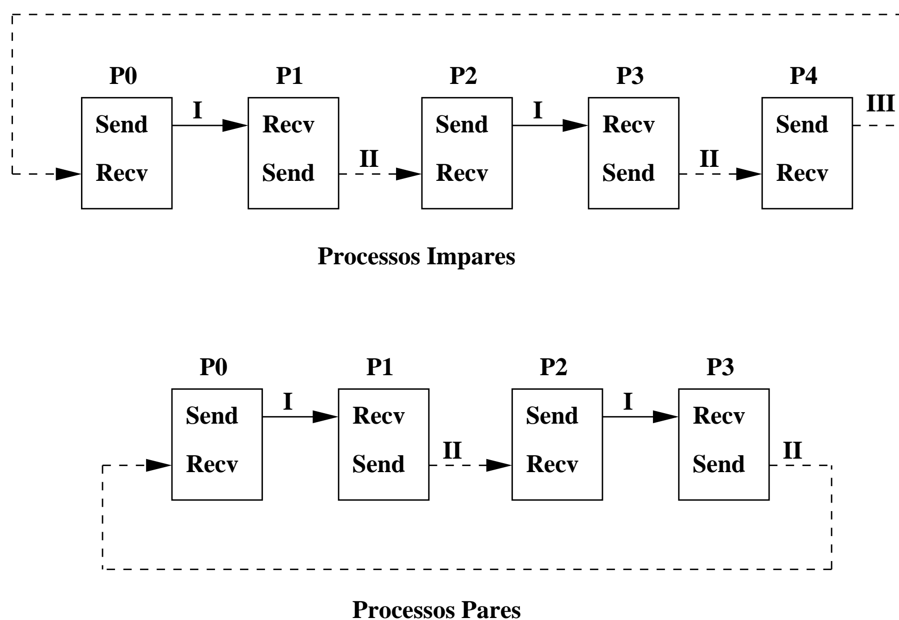

Figura 6.6: Anel com total de processos pares e ímpares

tabela 6.2.

Esses resultados demonstram como o simulador é capaz de capturar o efeito de desbalanceamento de cargas (neste caso, desbalanceamento na distribuição das comunicações).

6.3 Diferenças Finitas

O programa de diferenças finitas foi escolhido pois é amplamente utilizado na resolução numérica de equações diferenciais parciais e simulações de fenômenos físicos, como previsão do tempo. O código do programa de diferenças finitas é mostrado na figura 6.8. É definida uma matriz de tamanho N x M, onde N e M são iguais a 720. Os processos comunicam-se com seus vizinhos superiores, inferiores, da esquerda e da direita quando precisam de dados para realizar alguma computação, como mostra a figura 6.9. O programa mostra as diversas comunicações existentes entre os processos,

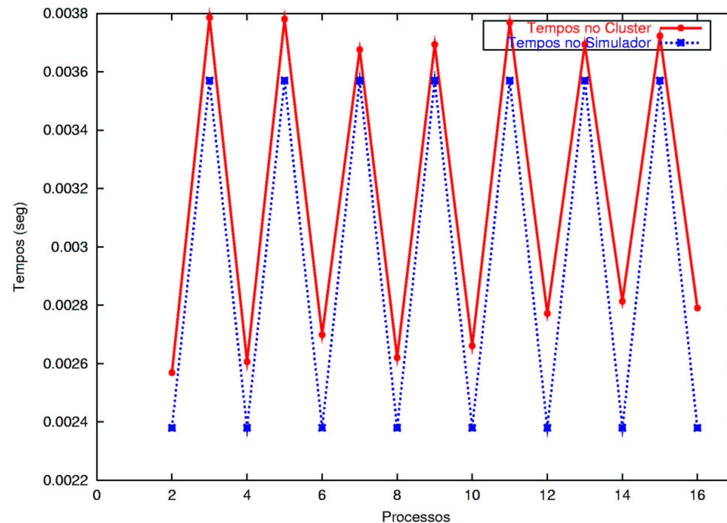

Figura 6.7: Tempos de transmissão obtidos na execução do programa anel no *Cluster* e no Simulador

além de indicar a computação que cada processo realiza.

A tabela 6.3 apresenta os tempos obtidos na execução do programa diferenças finitas utilizando o *cluster* e o simulador. Foram utilizados de 2 a 16 processos, e a quantidade de dados foi *N=720*. Como podemos ver na tabela 6.3, não constam os números de processos 7, 11, 13, e 14, isso porque o valor N (720) não é divisível por eles. A tabela também apresenta a diferença percentual entre os tempos obtidos utilizando a equação (6.1).

A figura 6.10 apresenta o gráfico contendo os tempos de transmissão obtidos na execução do programa de diferenças finitas (mostrado na figura 6.8) no *cluster* e no simulador. Os dados mostrados no gráfico 6.10 são os apresentados na tabela 6.3.

6.4 Mandelbrot

O conjunto *Mandelbrot* nomeado por Benoit Mandelbrot em [20] é um fractal, com uma estrutura detalhada que é realçada pelas cores

```
N = 720; M = 720;  /* tamanho da matriz */
Q = 4; /* número de linhas  */
R = 4; /* número de colunas */
auxLinha = rank/Q;
auxColuna = rank % Q;

for (i, 100) {
  if ((auxColuna % 2) == 0){
    if (auxColuna != 0){
      send(rank-1,(8*M/Q,0));
      receive(rank-1);
     };
    if ((rank+1)% Q != 0){
      send(rank+1,(8*M/Q,0));
      receive(rank+1);
    };
  }
  else {
    if (auxColuna != 0){
      receive(rank-1);
      send(rank-1,(8*M/Q,0));
    };
    if ((rank+1)% Q != 0){
      receive(rank+1);
      send(rank+1,(8*M/Q,0));
    };
  };
  if ((auxLinha % 2) == 0){
    if (rank-Q >= 0){
      send(rank-Q,(8*N/R,0));
      receive(rank-Q);
    };
    if (rank+Q <= P-1){
      send(rank+Q,(8*N/R,0));
      receive(rank+Q);
    };
  }
  else {
    if (rank-Q >= 0){
      receive(rank-Q);
      send(rank-Q,(8*N/R,0));
    };
    if (rank+Q <= P-1){
      receive(rank+Q);
      send(rank+Q,(8*N/R,0));
    };
  };
  compute(1.8e-7*(M/Q)*(N/R),0);
};
```

Figura 6.8: Código do programa Diferenças Finitas

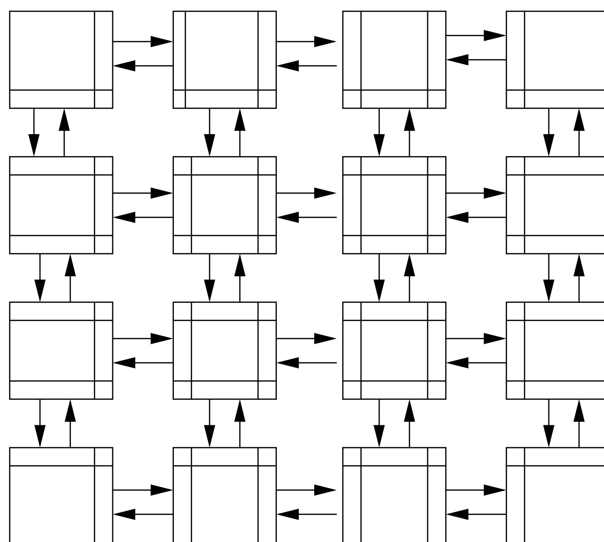

Figura 6.9: Diferenças Finitas

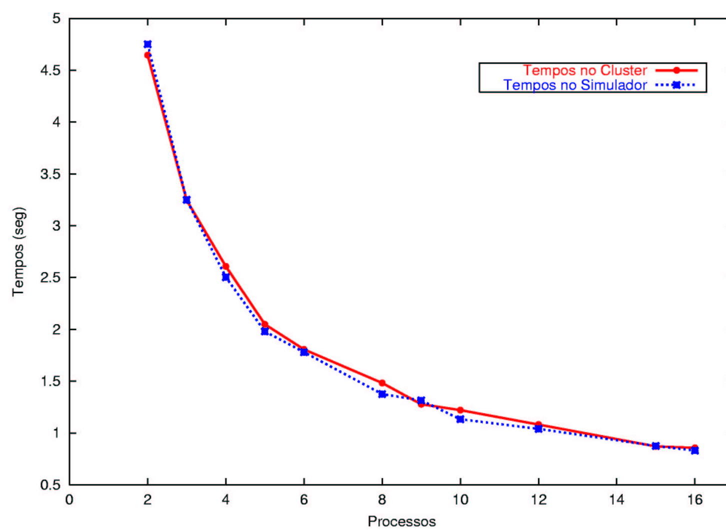

Figura 6.10: Tempos de transmissão obtidos na execução do programa de diferenças finitas no *Cluster* e no Simulador

Tabela 6.3: Tempos de transmissão (em segundos) obtido na execução do programa diferenças finitas no Cluster e no Simulador

Processos	*TempoCluster*(seg)	*TempoSimulador*(seg)	*Diferença*
2	4.646377	4.751410	2.26%
3	3.245838	3.249910	0.13%
4	2.608685	2.504420	−3.40%
5	2.049221	1.980946	−3.33%
6	1.806902	1.780400	−1.47%
8	1.483335	1.376175	−7.22%
9	1.278673	1.315823	2.91%
10	1.221409	1.133632	−7.19%
12	1.081989	1.041072	−3.78%
15	0.872335	0.876294	0.45%
16	0.858162	0.831126	−3.15%

geradas pelo computador [21]. Esse programa foi escolhido para podermos testar o comportamento do simulador em problemas com variação nos tempos de computação.

Para implementar o *mandelbrot* foi utilizado o método gerente/trabalhador. O gerente distribui blocos de dados (cada parte quadriculada da figura 6.11) entre os diversos trabalhadores, e fica verificando qual processo terminou e envia mais dados até que todos tenham terminado. Os processos trabalhadores recebem os dados, calculam e enviam os resultados para o processo gerente.

A figura 6.12 apresenta o código do programa *mandelbrot*.

A tabela 6.4 apresenta os tempos obtidos na execução do programa *mandelbrot* no simulador e no *cluster*. Os testes foram realizados utilizando 2 a 16 processadores.

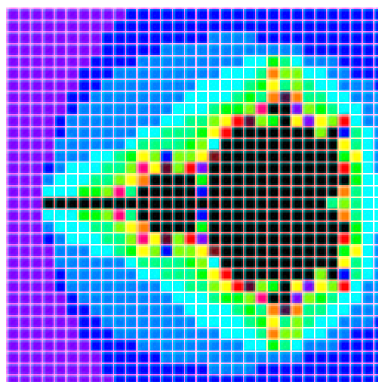

Figura 6.11: Mandelbrot

```
N = 32*32;
t=0;
s=0;
if (rank == 0){
  for(i,P-1){
    send(i+1,(4*4,0),0);
  };
  j = N-P+1;
  while(j > 0){
    receive(any_source,s,t);
    j = j - 1 ;
    send(s,(4*4,0),0);
  };
  for(i, P-1){
    receive(any_source,s,t);
    send(s,(0,0),1);
  };
}
else {
  while(t != 1){
    if (rank != 0){
      receive(0,t);
      if (t == 0) {
        compute(0.015856,0.017587);
        send(0,(4*4,0),0);
      };
    };
  };
};
```

Figura 6.12: Código do programa *Mandelbrot*

Tabela 6.4: Tempos de transmissão (em segundos) obtidos na execução do programa *mandelbrot* no Cluster e no Simulador

Processos	*TempoCluster*(seg)	*TempoSimulador*(seg)	*Diferenças*
2	16.752398	16.490400	−1.56%
3	8.381043	8.245860	−1.61%
4	5.589511	5.455047	−2.41%
5	4.195682	4.150972	−1.07%
6	3.361424	3.409770	1.44%
7	2.810386	2.871185	2.16%
8	2.413412	2.360752	−2.18%
9	2.118351	2.074348	−2.08%
10	1.887796	1.856642	−1.65%
11	1.701871	1.675324	−1.56%
12	1.549665	1.522568	−1.75%
13	1.425213	1.380239	−3.16%
14	1.323915	1.296611	−2.06%
15	1.232846	1.243576	0.87%
16	1.155935	1.164245	0.72%

O gráfico mostrado na figura 6.13 apresenta os tempos de transmissão da execução do programa *mandelbrot* (mostrado na figura 6.12) no *cluster* e no simulador.

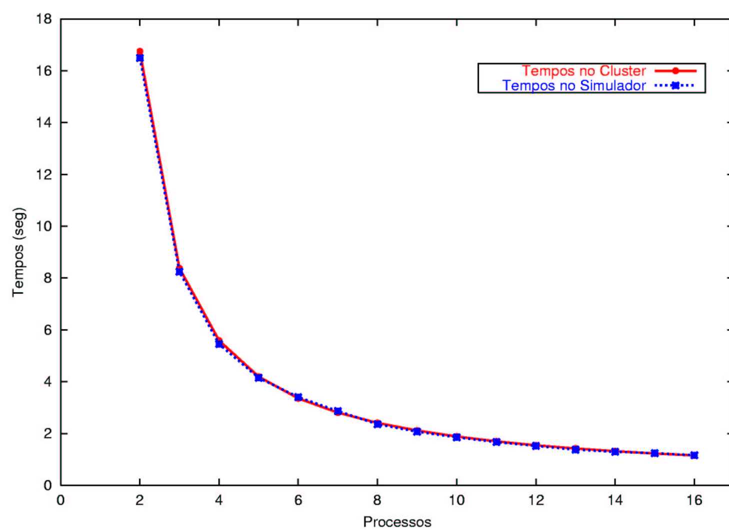

Figura 6.13: Tempos de transmissão obtidos na execução no *Cluster* e no Simulador do programa *Mandelbrot*

Capítulo 7

Conclusão e Trabalhos Futuros

Neste trabalho foi desenvolvido uma linguagem para a construção de protótipos de algoritmos paralelos, que permite uma descrição de alto nível do algoritmo especificando as comuicações, computações e a quantidade de dados envolvida.

Para a avaliação de um algoritmo paralelo, deve-se implementá-lo utilizando a linguagem. Com a implementação pronta, esse é então traduzido pelo tradutor que gera um código em C++. Para gerar o executável, o código em C++ é compilado e ligado com a biblioteca MPI e o simulador de rede. O simulador de rede simula o ambiente configurável com diversos processos e a troca de mensagens ocorrida entre eles através da rede.

Os tempos de comunicação entre os processos são calculados para avaliação de desempenho durante a simulação. Com base nos resultados obtidos com a simulação pode-se avaliar se vale a pena ou não a implementação do algoritmo.

Para validação do simulador foram implementados quatro algoritmos para análise em relação a um *cluster* de computadores pessoais. O primeiro teste realizado foi o do algoritmo ping-pong que visava avaliar os tempos de comunicação entre dois processos, va-

riando o tamanho das mensagens (8 *bytes* a 2 Mb). O teste realizado com esse algoritmo permitiu a verificação do funcionamento do simulador de rede e a verificação dos tempos de comunicação calculados.

O segundo teste feito utilizou o algoritmo do anel transmitindo mensagens de tamanho fixo de 10000 *bytes*, variando apenas o número de processos envolvidos (2 a 16 processos). A utilização do teste com este programa permitiu verificar o funcionamento do simulador com diversas comunicações pendentes simultaneamente.

O terceiro algoritmo testado foi o de diferenças finitas, que utiliza uma matriz N x N, onde N era igual a 720, variando o número de processos (2 a 16 processos). Com a execução do programa de diferenças finitas foi possível ver as diversas comunicações existentes entre os processos, além da computação realizada em cada processo.

O quarto e último teste realizado foi com o algoritmo do *mandelbrot* gerente/trabalhador. Nesse algoritmo variamos apenas o número de processos (2 a 16 processos). Esse teste nos permitiu verificar qual seria o comportamento do simulador usando programas com variação nos tempos de computação.

Através dos testes realizados, constatamos que o simulador é capaz de reproduzir o comportamento da rede e a troca de mensagens realizada entre os processos. Desta forma é possível prever o desempenho de programas antes de sua implementação

Tendo em vista os tempos obtidos na execução desses algoritmos, pode-se concluir que os algoritmos implementados na linguagem têm um erro médio de aproximadamente 3.6% mais rápido em relação ao algoritmo implementado utilizando a biblioteca MPI no *cluster* de teste. Esse erro se deve principalmente ao ajustamento

dos parâmetros t_s e t_w discutido no capítulo 5.

A maior contribuição desse trabalho é que para avaliar o quanto eficiente é o algoritmo paralelo, o programador não precisa construir novos modelos matemáticos e nem fazer uma simulação detalhada de sua aplicação. Basta apenas construir o algoritmo de sua aplicação utilizando a linguagem proposta e verificar se os resultados obtidos na execução de sua aplicação utilizando o simulador serão viáveis ou não para a aplicação em questão.

Algumas limitações que surgem com a utilização da linguagem e do simulador e que podem ser usadas como ponto de partida para trabalhos futuros são listadas abaixo:

- É permitido apenas o uso de um único programa com vários dados (SPMD). Um modo de resolver essa limitação, seria utilizar vários comandos *if* testando o *rank* dos processos, assim cada processo executaria um programa diferente, como feito no exemplo do *mandelbrot* (capítulo 6).

- Não se permite a implementação de algoritmos com geração dinâmica de tarefas. Isso acontece pois em MPI os processos são criados estaticamente na hora da execução, desse modo não poderíamos durante a execução criar um novo processo. Para a criação dinâmica dos processos seria necessário que fosse desenvolvido na linguagem uma função para a criação de processos e o esquema de controle da simulação usando MPI precisaria ser revisto, pois MPI também não permite a geração dinâmica de processos.

- Cada processador executa apenas uma tarefa. Em certos programas é útil alocar mais de uma tarefa por processador. No simulador implementado não é fácil introduzir essa versatili-

dade. Cada processo MPI executa uma cópia do programa, que simultaneamente toma conta do relógio de um processador. Se usarmos mais do que um processo MPI por processador, precisaríamos introduzir algum esquema complexo de sincronização dos relógios entre esses processos; para colocarmos mais do que um processo simulado no mesmo processo MPI seriam necessárias alterações profundas no código gerado.

- Não é possível criar subgrupos para as operações coletivas como em MPI. Uma forma de contornar essa limitação, seria criar na linguagem uma nova primitiva que permita a criação de novos grupos e alterar a implementação das funções coletivas para considerar esses grupos.
- Não existem cálculos, e portanto não podem ser simuladas computações ou comunicações dependentes de cálculos. Nesse caso, seria necessário o desenvolvimento de um simulador completo e não simplificado, visto que com o simulador desenvolvido as comunicações e computações são obtidas estatisticamente.
- Não existem comunicações assíncronas (por exemplo, para simular sobreposição de comunicação e computação). Comunicações assíncronas poderiam ser introduzidas através de novas primitivas na linguagem e de novas regras do simulador de rede para reação a essas primitivas.
- Utiliza apenas distribuição gaussiana nas variações. Uma forma de contornar isso seria o programador ter uma opção para escolher qual o tipo de distribuição que ele deseja usar, através do fornecimento de uma função definida pelo usuário a ser utilizada ao longo das variações.

Referências Bibliográficas

[1] D. E. Culler, J. P. Singh, and A. Gupta. *Paraller Computer Architecture: A Hardware/Software Approach*. Morgan Koufmann, 1999.

[2] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[3] Paolo Cremonesi, Emilia Rosti, Giuseppe Serazzi, and Evgenia Smirni. Performance evaluation of parallel systems. *Parallel Computing 25*, pages 1677–1698, January 1999.

[4] Cosimo Anglano. Predicting parallel applications performance on non-dedicated cluster platforms. In *International Conference on Supercomputing*, pages 172–179, 1998.

[5] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Anlysis using Queueing Network Models*. Prentice-Hall, 1984.

[6] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77*, pages 541–580, April 1989.

[7] De-Ron Liang and Satish K. Tripathi. On performance prediction of parallel computations with precedent constraints. *IEEE*

Transactions on Parallel and Distributed Systems, 11(5):491–508, May 2000.

[8] Thomas Fahringer and Hans P. Zima. A static parameter based perfomance prediction tool for parallel programs. *ACM*, pages 207–219, 1993.

[9] Manish Parashar and Salim Hariri. Interpretive performance prediction for parallel application development. *Journal of Parallel and Distributed Computing*, pages 14–47, 2000.

[10] Message Passing Interface Forum. MPI: A message-passing interface standard, `http://www-unix.mcs.anl.gov/mpi/`, 1995. (visitado em 15 de junho de 2002).

[11] Gonzalo Travieso. Message passing interface: Notas de aula. Outubro 2000.

[12] Luis Gustavo Nonato. Tipos e estruturas de dados: Notas de aula. Instituto de Ciências Matemáticas e de Computação, USP São Carlos, Março 2000.

[13] Charles Donnelly and Richerd Stallman. *BISON: The YACC-compatible Parser Generator*, November 1995. Versão 1.25.

[14] Vern Paxson. *FLEX: A fast scanner generator*, March 1995. Versão 2.5.

[15] Wolfgang Kreutzer. *System Simulation-Programming Styles and Languages*. Addison-Wesley, 1986.

[16] Lin Jensen. *Simulation Programming Notes*. Bishop's University Computer Science, 1996.

[17] MVICH: MPI for virtual interface architecture, `http://www.nersc.gov/research/ftg/mvich/`. (visitado em 20 de junho de 2002).

[18] The slackware linux project, `http://www.slackware.com`. (visitado em 01/2002).

[19] MPICH - a portable implementation of MPI, `http://www-unix.mcs.anl.gov/mpi/mpich/`. (visitado em 15 de junho de 2002).

[20] David Dewey. Introduction to mandelbrot set, `http://www.olympus.net/personal/dewey/mandelbrot.html`. (visitado em 30 de janeiro de 2002).

[21] Maya Storch and Ferdinand Keller. The mandelbrot set, `http://www.unizh.ch/nchaos/mand/index.html`. (visitado em 15 de maio de 2002).

Apêndice A

Regras do Bison

Neste apêndice são apresentadas o conjunto de regras da gramática definidas utilizando a ferramenta Bison descrita na seção 4.2.1.

```
%token <pont> FUNC
%token <pont> IF
%token <pont> PORC
%token <pont> ELSE
%token <pont> NUM

%token <pont> WHILE
%token <pont> DO
%token <pont> FOR
%token <pont> IDENT

%token <pont> SEND
%token <pont> ENVIO

%token <pont> RECEIVE
%token <pont> RECEBE
%token <pont> OBTER

%token <pont> COMPUTE

%token <pont> ANYSOURCE

%token <pont> BCAST
%token <pont> GATHER
%token <pont> ALL_GATHER
%token <pont> SCATTER
%token <pont> ALL_TO_ALL
```

```
%token <pont> REDUCE
%token <pont> ALL_REDUCE

%token <pont> OPEN_BRACE   /* ( */
%token <pont> CLOSE_BRACE  /* ) */
%token <pont> OPEN_BLOCK   /* { */
%token <pont> CLOSE_BLOCK  /* } */
%token <pont> COMMA        /* , */
%token <pont> SEMICOLON    /* ; */

%token <pont> EQ /* == */
%token <pont> GQ /* > */
%token <pont> GE /* >=*/
%token <pont> LQ /* < */
%token <pont> LE /* <= */
%token <pont> NE /* != */

%type <pont> programa
%type <pont> lista_comando
%type <pont> bloco
%type <pont> exp
%type <pont> ident
%type <pont> source
%type <pont> funcao
%type <pont> lista_param
%type <pont> atribuicao
%type <pont> comando
%type <pont> variacao
%type <pont> comparacao
%type <pont> igualdade
%type <pont> diferenca
%type <pont> maior
%type <pont> menor
%type <pont> maior_igual
%type <pont> menor_igual
%type <pont> if_comando
%type <pont> while_comando
%type <pont> for_comando
%type <pont> send_comando
%type <pont> receive_comando
%type <pont> compute_comando
%type <pont> bcast_comando
%type <pont> gather_comando
%type <pont> all_gather_comando
%type <pont> scatter_comando
%type <pont> all_to_all_comando
%type <pont> reduce_comando
%type <pont> all_reduce_comando
```

```
%right '='
%left  '-' '+'
%left  '*' '/'
%left  '%'

/* Declaracao BISON - regras de gramatica */
%%

programa: lista_comando { root = $1; }

lista_comando:
  comando SEMICOLON
    { $1->prox = 0;
      $$ = $1;
    }
  | comando SEMICOLON lista_comando
     { $1->prox = $3;
       $$ = $1;
     }

bloco:
  OPEN_BLOCK lista_comando CLOSE_BLOCK { $$ = $2; }

ident:
  IDENT  { $$ = (No*)malloc(sizeof(No));
           $$->token = IDENT;
           strcpy($$->nome, yylval.pont->nome);
           $$->esq = NULL;
           $$->dir = NULL;
         }

source:
  ANYSOURCE {  $$ = (No*)malloc(sizeof(No));
               $$->token = ANYSOURCE;
               $$->esq = NULL;
               $$->dir = NULL;
            }

atribuicao:
  ident '=' exp { $$ = (No*)malloc(sizeof(No));
                  $$->token = '=';
                  $$->esq = $1;
                  $$->dir = $3;
                }

exp:
  ident
  | funcao
  | NUM          { $$ = (No*)malloc(sizeof(No));
```

```
                  $$->token = NUM;
                  $$->val = yylval.pont->val;
                  $$->esq = NULL;
                  $$->esq = NULL;
                }
  | exp '+' exp  { $$ = (No*)malloc(sizeof(No));
                  $$->token = '+';
                  $$->esq = $1;
                  $$->dir = $3;
                }
  | exp '-' exp  { $$ = (No*)malloc(sizeof(No));
                  $$->token = '-';
                  $$->esq = $1;
                  $$->dir = $3;
                }
  | exp '*' exp  { $$ = (No*)malloc(sizeof(No));
                  $$->token = '*';
                  $$->esq = $1;
                  $$->dir = $3;
                }
  | exp '/' exp  { $$ = (No*)malloc(sizeof(No));
                  $$->token = '/';
                  $$->esq = $1;
                  $$->dir = $3;
                }
  | exp '%' exp  { $$ = (No*)malloc(sizeof(No));
                  $$->token = '%';
                  $$->esq = $1;
                  $$->dir = $3;
                }
  | OPEN_BRACE exp CLOSE_BRACE
     { $$ = (No*)malloc(sizeof(No));
       $$ = $2;
     }

lista_param:
  exp { $$ = $1; }
  |  exp COMMA lista_param
       { $$ = (No*)malloc(sizeof(No));
         $$->token = COMMA;
         $$->esq = $1;
         $$->dir = $3;
       }

funcao:
  ident OPEN_BRACE lista_param CLOSE_BRACE
     { $$ = (No*)malloc(sizeof(No));
       $$->token = FUNC;
       $$->lookahead = $1;
```

```
      $$->esq = $3;
      $$->dir = NULL;
    }

comando:
  atribuicao
  | bloco
  | if_comando
  | while_comando
  | for_comando
  | send_comando
  | receive_comando
  | compute_comando
  | bcast_comando
  | gather_comando
  | all_gather_comando
  | scatter_comando
  | all_to_all_comando
  | reduce_comando
  | all_reduce_comando
  ;

variacao:  exp COMMA exp { $$ = (No*)malloc(sizeof(No));
                           $$->token = COMMA;
                           $$->esq = $1;
                           $$->dir = $3;
                         }

comparacao: igualdade
          | diferenca
          | maior
          | maior_igual
          | menor
          | menor_igual
          ;

igualdade: exp EQ exp  { $$ = (No*)malloc(sizeof(No));
                         $$->token = EQ;
                         $$->esq = $1;
                         $$->dir = $3;
                       }

diferenca: exp NE exp  { $$ = (No*)malloc(sizeof(No));
                         $$->token = NE;
                         $$->esq = $1;
                         $$->dir = $3;
                       }

maior: exp GQ exp      { $$ = (No*)malloc(sizeof(No));
```

```
                          $$->token = GQ;
                          $$->esq = $1;
                          $$->dir = $3;
                        }

maior_igual: exp GE exp  { $$ = (No*)malloc(sizeof(No));
                           $$->token = GE;
                           $$->esq = $1;
                           $$->dir = $3;
                         }

menor: exp LQ exp        { $$ = (No*)malloc(sizeof(No));
                           $$->token = LQ;
                           $$->esq = $1;
                           $$->dir = $3;
                         }

menor_igual: exp LE exp   { $$ = (No*)malloc(sizeof(No));
                            $$->token = LE;
                            $$->esq = $1;
                            $$->dir = $3;
                          }

if_comando:
  IF OPEN_BRACE comparacao CLOSE_BRACE bloco
     { $$ = (No*)malloc(sizeof(No));
       $$->token = IF;
       $$->lookahead = $3;
       $$->esq = $5;
       $$->dir = NULL;
     }
  | IF OPEN_BRACE comparacao CLOSE_BRACE bloco ELSE bloco
      { $$ = (No*)malloc(sizeof(No));
        $$->token = IF;
        $$->lookahead = $3;
        $$->esq = $5;
        $$->dir = $7;
      }
  | IF OPEN_BRACE exp CLOSE_BRACE bloco
      { $$ = (No*)malloc(sizeof(No));
        $$->token = PORC;
        $$->lookahead = $3;
        $$->esq = $5;
        $$->dir = NULL:
      }
  | IF OPEN_BRACE exp CLOSE_BRACE bloco ELSE bloco
      { $$ = (No*)malloc(sizeof(No));
        $$->token = PORC;
        $$->lookahead = $3;
```

```
        $$->esq = $5;
        $$->dir = $7;
      }

while_comando:
  WHILE OPEN_BRACE variacao CLOSE_BRACE bloco
     { $$ = (No*)malloc(sizeof(No));
       $$->token = WHILE;
       $$->lookahead = $3;
       $$->esq = $5;
       $$->dir = NULL;
     }
  | WHILE OPEN_BRACE comparacao CLOSE_BRACE bloco
       { $$ = (No*)malloc(sizeof(No));
         $$->token = DO;
         $$->lookahead = $3;
         $$->esq = $5;
         $$->dir = NULL;
       }

 for_comando:
   FOR OPEN_BRACE ident COMMA exp CLOSE_BRACE bloco
     { $$ = (No*)malloc(sizeof(No));
       $$->token = FOR;
       $$->lookahead = $3;
       $$->lookahead1 = $5;
       $$->esq = $7;
       $$->dir = NULL;
     }

send_comando:
  SEND OPEN_BRACE exp COMMA OPEN_BRACE variacao
       CLOSE_BRACE CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = SEND;
      $$->lookahead = $3;
      $$->lookahead1 = $6;
      $$->esq = NULL;
      $$->dir = NULL;
    }
| SEND OPEN_BRACE exp COMMA OPEN_BRACE variacao CLOSE_BRACE
         COMMA exp CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = SEND;
      $$->lookahead = $3;
      $$->lookahead1 = $6;
      $$->lookahead2 = $9;
      $$->esq = NULL;
      $$->dir = NULL;
```

```
    }

receive_comando:
  RECEIVE OPEN_BRACE exp CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = RECEIVE;
      $$->lookahead = $3;
      $$->esq = NULL;
      $$->dir = NULL;
    }
| RECEIVE OPEN_BRACE exp COMMA ident CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = RECEIVE;
      $$->lookahead = $3;
      $$->lookahead1 = $5;
      $$->esq = NULL;
      $$->dir = NULL;
    }
| RECEIVE OPEN_BRACE source CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = OBTER;
      $$->lookahead = $3;
      $$->esq = NULL;
      $$->dir = NULL;
    }
| RECEIVE OPEN_BRACE source COMMA ident CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = ADMITIR;
      $$->lookahead = $3;
      $$->lookahead1 = $5;
      $$->esq = NULL;
      $$->dir = NULL;
    }

compute_comando:
  COMPUTE OPEN_BRACE variacao CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = COMPUTE;
      $$->lookahead = $3;
      $$->esq = NULL;
      $$->dir = NULL;
    }

bcast_comando:
  BCAST OPEN_BRACE exp COMMA OPEN_BRACE variacao
        CLOSE_BRACE CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = BCAST;
      $$->lookahead = $3;
```

```
      $$->lookahead1 = $6;
      $$->esq = NULL;
      $$->dir = NULL;
    }

gather_comando:
  GATHER OPEN_BRACE OPEN_BRACE variacao CLOSE_BRACE
         COMMA exp CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = GATHER;
      $$->lookahead = $4;
      $$->lookahead1 = $7;
      $$->esq = NULL;
      $$->dir = NULL;
    }

all_gather_comando:
  ALL_GATHER OPEN_BRACE variacao CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = ALL_GATHER ;
      $$->lookahead = $3;
      $$->esq = NULL;
      $$->dir = NULL;
    }

scatter_comando:
  SCATTER OPEN_BRACE exp COMMA OPEN_BRACE variacao
          CLOSE_BRACE CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = SCATTER;
      $$->lookahead = $3;
      $$->lookahead1 = $6;
      $$->esq = NULL;
      $$->dir = NULL;
    }

all_to_all_comando:
  ALL_TO_ALL OPEN_BRACE variacao CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = ALL_TO_ALL;
      $$->lookahead = $3;
      $$->esq = NULL;
      $$->dir = NULL;
    }

reduce_comando:
  REDUCE OPEN_BRACE exp COMMA OPEN_BRACE variacao
         CLOSE_BRACE CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
```

```
      $$->token = REDUCE;
      $$->lookahead = $3;
      $$->lookahead1 = $6;
      $$->esq = NULL;
      $$->dir = NULL;
    }

all_reduce_comando:
  ALL_REDUCE OPEN_BRACE variacao CLOSE_BRACE
    { $$ = (No*)malloc(sizeof(No));
      $$->token = ALL_REDUCE;
      $$->lookahead = $3;
      $$->esq = NULL;
      $$->dir = NULL;
    }
```

Apêndice B

Regras do Flex

Neste apêndice são apresentadas as regras definidas na linguagem utilizando a ferramenta Flex descrita na seção 4.2.2.

```
"if"         { return IF; }
"else"       { return ELSE; }
"while"      { return WHILE; }
"for"        { return FOR; }
"send"       { return SEND; }
"receive"    { return RECEIVE; }
"compute"    { return COMPUTE; }
"bcast"      { return BCAST; }
"gather"     { return GATHER; }
"all_gather" { return ALL_GATHER; }
"scatter"    { return SCATTER; }
"all_to_all" { return ALL_TO_ALL; }
"reduce"     { return REDUCE; }
"all_reduce" { return ALL_REDUCE; }
"funcao"     { return FUNC; }
"any_source" { return ANYSOURCE; }

[-+]?[0-9]+("."[0-9]*)?([eE]"-"?[0-9]*)?
          { yylval.pont->val = atof(yytext);
            return NUM;
          }
[a-zA-Z][a-zA-z0-9]* { strncpy(yylval.pont->nome,yytext, 256);
                         return IDENT;
                       }

"=="   { return EQ; }
"!="   { return NE; }
```

```
">"    { return GQ; }
">="   { return GE; }
"<"    { return LQ; }
"<="   { return LE; }
"("    { return OPEN_BRACE; }
")"    { return CLOSE_BRACE; }
"{"    { return OPEN_BLOCK; }
"}"    { return CLOSE_BLOCK; }
","    { return COMMA; }
";"    { return SEMICOLON; }

"="|"+"|"-"|"*"|"/"|"%" { return *yytext; }

[\t\n]  /* Ignora */
.      { return *yytext; }
```