Eduardo Petrolini Calzeta
Thiago Alexandre Domingues de Souza

Ferramenta para predição de desempenho de programas paralelos através da simulação do grafo de execução

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

São José do Rio Preto
Novembro de 2008

Eduardo Petrolini Calzeta
Thiago Alexandre Domingues de Souza

Ferramenta para predição de desempenho de programas paralelos através da simulação do grafo de execução

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Orientador:
Prof. Dr. Aleardo Manacero Júnior

São José do Rio Preto
Novembro de 2008

Eduardo Petrolini Calzeta
Thiago Alexandre Domingues de Souza

Ferramenta para predição de desempenho de programas paralelos através da simulação do grafo de execução

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Eduardo Petrolini Calzeta

Thiago Alexandre Domingues de Souza

Prof. Dr. Aleardo Manacero Júnior

Banca examinadora:
Prof. Dr. Carlos Roberto Valêncio
Profa. Dra. Renata Spolon Lobato

Dedicatória

Aos nossos familiares.

Agradecimentos

Agradecemos em primeiro lugar a Deus pela vida que nos concedeu.

Aos nossos pais, pelo esforço, paciência e incentivo durante toda a nossa vida. Sem este apoio não estaríamos concluindo mais uma etapa de nossas vidas.

Ao professor Aleardo, por ter confiado este projeto em nossas mãos. Pelas orientações e soluções que muitas vezes não encontrávamos.

Aos verdadeiros amigos de curso, pelos grandes momentos que passamos juntos, pelas dificuldades e estudos em grupo, que esta amizade construída possa durar por toda nossa vida.

Aos demais professores, pelo ensinamento sólido que nos possibilitará vencer as dificuldades que a vida irá nos impor.

A FAPESP, pelo incentivo financeiro concedido e confiança nos resultados de nosso projeto.

Resumo

O desenvolvimento de sistemas computacionais de alto desempenho demanda um elevado esforço de otimização. Isso ocorre pelos altos custos associados ao hardware e a necessidade do aproveitamento máximo dos recursos disponíveis. Isso requer do programador uma maior atenção ao desempenho de seu programa.

Neste contexto, a análise de desempenho exerce papel fundamental para a redução de custos. Existem diversas ferramentas de análise de desempenho que auxiliam o projetista na identificação de pontos críticos do sistema, e assim propor soluções que melhorem sua execução. No entanto, essas ferramentas ou fazem aproximações do ambiente paralelo, o que diminui a precisão dos resultados obtidos, ou fazem uso do sistema real para sua medição, no entanto, essa alternativa nem sempre é viável.

Este trabalho implementa uma metodologia de medição de desempenho baseada em simulação, sem a necessidade do ambiente real de execução, nem mesmo o código fonte do programa avaliado. Nela faz-se a reescrita do código executável, compilado para a arquitetura x86, em seu grafo de execução correspondente. Esta estrutura armazena o tempo de processamento de cada instrução contida no código fornecido. Com os dados do grafo, incluindo informações sobre trocas de mensagens MPI, este programa tem sua execução simulada. Este procedimento é feito de acordo com os parâmetros fornecidos sobre o ambiente de execução, tais como número de processos, velocidade de processamento, largura de banda, etc.

Apresentaram-se resultados comparativos de medições realizadas sobre programas escritos no paradigma mestre-escravo, que indicam um bom desempenho de precisão da metodologia.

Abstract

The development of high-performance computing systems requires a large effort in their optimization. This occurs due to the high costs associated with their hardware and the need for an efficient use of the available resources. This demands, from the programmer, some careful attention to the performance of the system being developed.

In this context, the performance analysis performs a key role in the costs reduction. There are many tools available for performance evaluation helping designers to identify critical points in the system, and to propose solutions that improve its execution. However these tools either make approximations of the parallel environment, reducing the accuracy of the achieved results, or make use of the real system for its measurements, which is not always feasible.

This work implements a methodology for performance analysis based on simulation, which does not require a real execution environment, neither the source code of the evaluated program. It makes the rewriting of the executable code, compiled for x86 architecture, to its associated execution graph. This structure stores the processing time for each instruction included in the provided code.

With the data in the graph, including data about MPI exchange messages, the program has its execution simulated. This procedure is made according to the execution environment parameters, such as number of processes, clock rate, bandwidth, etc.

Comparative results from measurements performed over master-slave programs are presented. Such results indicate a good accuracy performance achieved by this methodology.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A eficiência de um sistema computacional está intrinsecamente relacionada ao seu tempo de execução, essa dependência pode ser otimizada com algoritmos mais aprimorados ou ao aumento do poder computacional do ambiente. Normalmente a segunda opção é escolhida, sem levar em consideração possíveis melhorias no sistema utilizado. Essa alternativa implica em custos associados, como aquisição de novos equipamentos de *hardware* e contratação de pessoal qualificado para sua manutenção. Além disso, como o custo é diretamente proporcional a sua utilização, a medida que o sistema não utiliza todo seu poder computacional isso implica em um investimento desnecessário.

A relevância da utilização do sistema é ainda mais importante em computação de alto desempenho, haja vista que o custo de novos equipamentos é muito elevado. Deste modo faz-se necessário o uso de ferramentas para predição de desempenho de sistemas, de maneira que seja avaliada a viabilidade de aquisição de novos equipamentos. Além disso, a ferramenta de predição deve fornecer um mecanismo de identificação dos gargalos do programa utilizado, para possíveis correções e otimizações.

1.1 Objetivos

O objetivo deste trabalho é o desenvolvimento de uma ferramenta para predição de desempenho de programas paralelos denominada *Grasptool*. Para tanto é necessário o conhecimento prévio do ambiente a simular o programa, como largura de banda da rede, velocidade média de processamento das máquinas, tamanho médio das mensagens trocadas, etc. Em posse dessas informações e do código executável do sistema, é feito o desmonte do programa em mnemônicos e códigos de instruções. A próxima etapa é a geração de um grafo de execução do programa, esta estrutura reflete fielmente todos os passos do programa. Por fim, a partir grafo de execução, o programa de entrada é simulado e são apresentados dados estatísticos em relação a sua execução.

Com esta abordagem técnicas invasivas de medições são evitadas, pois as medições são

realizadas pela simulação do grafo de execução que não oferece nenhuma interferência do simulador. Além disso, a abordagem faz com que programas complexos possam ser obtidos de forma automática sem a interferência do programador ao longo do código.

1.2 Organização da monografia

No capítulo dois é apresentada toda base teórica que fundamenta o projeto, o texto mostra diversas referências em análise de desempenho para sistemas paralelos, suas vantagens e desvantagens associadas à implementação. Além disso, fornece todo embasamento formal do modelo utilizado neste projeto.

Em seqüência, no capítulo três, são apresentados detalhes de implementação do projeto, bem como pseudocódigos e suas relações de dependência. Já no capítulo quatro são mostrados testes e os resultados que validam a eficiência do trabalho proposto. Finalmente, no capítulo cinco, são expostas as conclusões e perspectivas futuras de desenvolvimento do projeto.

Capítulo 2

Fundamentação Teórica

O desempenho de sistemas computacionais é crítico em relação ao tempo. Isso se deve à necessidade do usuário em obter resultados de forma cada vez mais rápida, sem desperdiçar recursos computacionais e a utilidade da informação após seu processamento.

Neste escopo, a análise de desempenho oferece ferramentas para avaliação de sistemas, e assim identificar se os recursos estão sendo utilizados de forma ótima ou então localizar seus gargalos. Estas ferramentas podem ser classificadas em dois grupos: por monitoração (pelo sistema operacional ou *hardware* especializado) ou pela modificação do código (fonte, objeto ou executável). O primeiro possui resultados mais precisos, no entanto isso nem sempre é viável devido aos seus altos custos associados. Já no segundo há uma perda de precisão, devido às técnicas invasivas empregadas, embora os custos sejam mais baixos.

Ao longo deste capítulo serão discutidas algumas medidas de desempenho em sistemas paralelos, métodos utilizados para predição e análise de desempenho e finalmente o método proposto por [7].

2.1 Medidas de desempenho

De maneira geral as medidas de desempenho podem ser divididas entre as geradas por modelos teóricos, usados principalmente através da modelagem analítica do código; e experimentais, obtidos através da execução do código. A vantagem da modelagem analítica se deve a sua capacidade de predizer o desempenho de um sistema sem dispor de seu ambiente de execução, no entanto, sua eficiência é dependente da sua modelagem.

Já por meio de medidas de desempenho experimentais são obtidas estimativas mais fiéis da execução do sistema, embora seja necessário dispor do ambiente de execução, fato que nem sempre é viável devido aos altos custos de *hardware* utilizados em computação de alto desempenho.

Em geral, quando o código do sistema está disponível, é possível avaliar o desempenho do sistema com as seguintes técnicas:

- **Monitoração por hardware:** usando dispositivos especializados, os quais são acoplados nos *hardwares* dos equipamentos, é possível capturar os eventos do sistema. Técnica custosa devido a necessidade de todo o ambiente de análise e, além disso, os custos associados ao equipamento de captura e de pessoal capacitado.
- **Monitoração pelo sistema operacional:** utiliza as interrupções do sistema para análise de sua execução. Técnica precisa, mas lenta devido às inúmeras interrupções lançadas.
- **Modificação do código fonte:** técnica invasiva que adiciona códigos especiais para a captura das medições desejadas. Resultados são relativamente imprecisos devido a inserção desses códigos.
- **Modificação do código objeto:** insere comandos no código objeto, possui os mesmos problemas de precisão da modificação do código fonte.
- **Modificação do código executável:** insere comandos no código executável, apresenta os mesmos problemas de precisão.

Ainda é possível classificar as técnicas de modificação de código segundo a informação resultante da medição ou pela forma como é realizada.

- ***Profiling*:** técnica mais utilizada. Baseia-se na obtenção de dados do contador de programas (*Program Counter*) do processador em intervalos fixos, assim é possível obter informações sobre quanto tempo cada trecho de código é executado. Utilizado em algumas ferramentas como gprof[14], dpat e jprof[15].
- **Contagem de eventos:** o código é modificado de forma a contar determinados eventos desejados, como chamadas de funções. Não é muito utilizado devido a sua alta taxa de processamento empregado.
- **Medição de intervalos de tempos:** é uma variante de *profiling*, em que se trocam as amostragens do primeiro por chamadas para medir o relógio do sistema inseridas no código do programa.
- **Extração de traços dos eventos:** também conhecida como *event tracing*, presente em ferramentas como IDTrace [16] e ALPES [17], fornece resultados precisos, embora exija um alto custo computacional para isso.

Um dos problemas das técnicas citadas anteriormente é a necessidade da utilização do programa na máquina real para a obtenção dos dados. Já em outras técnicas, como modelagem analítica ou simulação, são menos custosas e os resultados obtidos variam de acordo com sua modelagem. A seguir serão abordadas medidas específicas para a análise de desempenho em sistemas paralelos.

2.1.1 Medidas de desempenho em sistemas paralelos

Antes de analisar o desempenho de um programa é necessário conhecer as medidas que devem ser extraídas do sistema, tais como, tempo de execução, escalabilidade, dados sobre a rede, confiabilidade, entre outras. Já em sistemas paralelos são introduzidas novas medidas particulares ao ambiente de paralelismo. A seguir serão apresentadas as definições para esses conceitos.

- ***Speedup*:** é a medida mais importante em sistemas paralelos. Verifica o quão rápido é um programa executado em paralelo em relação a uma referência. Uma das abordagens ao *speedup* é a **Lei de Amdahl**[1] que relaciona o ganho de velocidade ao desempenho em paralelo e serial do programa, denotada pela equação 2.1:

$$speedup = \frac{(s+p)}{(s+p/N)} \tag{2.1}$$

  Onde s é a porção serial do código e p sua porção paralela, portanto $s + p = 1$ e N o número de processadores em paralelo. A equação 2.1 é válida somente quando o tamanho do problema não varia com o número de processadores. Esta é conhecida como *speedup* para tamanho fixo. Outra abordagem, apresentada por Gustafson[2], leva em consideração a manutenção fixa do tempo e não o tamanho do problema, para isso o tamanho do problema aumenta de acordo com a capacidade das máquinas. Assim, a equação abaixo relaciona o ganho de velocidade em função de sua execução seqüencial.

$$speedup\ tempo\ fixo = N + (1 - N) \cdot s \tag{2.2}$$

  É pressuposto na equação acima que com mais processadores o problema a ser resolvido também será maior.

- **Eficiência**: é uma medida que indica se o problema está aproveitando os recursos do paralelismo. Com isso é possível inferir se o problema é escalável, neste caso a eficiência tenderá a um.

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{n \cdot T(n)} \tag{2.3}$$

- **Redundância**: relação entre o paralelismo do algoritmo e o paralelismo da máquina. Quanto menor é esta relação menor é a sobrecarga de paralelismo do sistema.

$$R(n) = \frac{O(n)}{O(1)} \tag{2.4}$$

- **Utilização**: ao relacionar a equação 2.3 com 2.4 é possível medir a utilização do sistema, vista na equação 2.5. Medida relevante para se obter uma estimativa do poder computacional adequado ao sistema.

$$U(n) = R(n) \cdot E(n) = \frac{O(n)}{n \cdot T(n)} \tag{2.5}$$

- **Qualidade de paralelismo:** a qualidade do paralelismo aumenta a medida que maiores valores de eficiência e *speedup* são obtidos , e simultaneamente uma menor redundância.

$$Q(n) = \frac{S(n) \cdot E(n)}{R(n)} = \frac{T^3(1)}{n \cdot T^2(n) \cdot O(n)} \tag{2.6}$$

2.2 Métodos para predição e análise de desempenho

Com os conceitos básicos sobre medidas de desempenho já examinados, as páginas seguintes apresentam de forma sucinta os métodos para predição e análise de desempenho com suas qualidades e deficiências, especialmente sobre a metodologia adotada para obter as medidas necessárias para análise e predição.

2.2.1 Métodos analíticos

Nesta categoria enquadra-se a técnica de medição baseada em modelos analíticos, em que o uso de avaliações experimentais restringe-se a uma execução de *profiling*[1] ou muitas vezes nem ocorrem. Os métodos analíticos estão amparados essencialmente em modelos de redes de Petri estocástica generalizada (GSPN) [6] e cadeias de Markov, onde o programa é modelado como uma seqüência de estados utilizando probabilidades para decidir a mudança de um estado para outro.

Estes modelos analíticos permitem a predição de desempenho de um programa em um sistema sem a necessidade propriamente do programa ou mesmo do sistema. Em contrapartida, a modelagem analítica de programas torna-se inviável devido ao seu tamanho e complexidade, principalmente no caso de um sistema paralelo. Portanto, perde-se precisão e a consistência dos resultados de desempenho obtidos.

Uma boa característica destes modelos analíticos é a velocidade de resposta, pois estes métodos podem ser reduzidos em sistemas de equações onde vários métodos de resolução podem ser aplicados para a obtenção dos resultados, desde que o número de estados não cresça de forma exponencial, inviabilizando sua resolução.

2.2.2 Métodos baseados em *benchmarking*

Este método tem como característica sua baixa complexidade de implementação, pois suas medições são feitas diretamente no local de interesse sobre o desempenho do programa.

[1] técnica utilizada para obter o tempo que cada trecho de código gastou executando

São métodos caros, pois necessita-se da máquina real para a obtenção dos resultados. Sua estratégia é baseada na execução do programa na máquina alvo e a medição de tudo que for desejável para a análise de desempenho do sistema.

Este método é freqüentemente utilizado para avaliar o desempenho da máquina, independentemente do programa a ser executado nela. Neste caso encaixam-se as ferramentas SLALOM [5], LINPACK [9] e SPEC [10]. Porém, há o entrave de encontrar *benchmarks* (programas de tomada de medidas) que possam ser eficientes, proporcionando medidas que sejam consistentes de modo a não beneficiar uma ou outra arquitetura.

Outros problemas podem ser encontrados para a elaboração de um *benchmark*, como a caracterização da carga aplicada ao sistema computacional, como pesquisado por [3], a estruturação de procedimentos para fazer as medições [4] e também o projeto de programas que executam testes sobre a capacidade real de escalabilidade do sistema [5]. Os resultados obtidos através deste método poderiam ser dados em tempo gasto para executar a tarefa, contudo, visando à heterogeneidade das máquinas, estas medidas são convertidas em unidades relativas como, por exemplo, Mips[2] ou Flops[3].

Em um programa específico de usuário, a medida de desempenho tem que ser tomada diretamente sobre a máquina, envolvendo gastos para a compra dos equipamentos e sua manutenção. Portanto, as medidas de desempenho listadas no parágrafo anterior não precisam ser consideradas.

Apesar destes problemas, este método é bastante utilizado devido a sua relativa precisão e também pelo fato que novos programas poderão ter analisados seu desempenho neste ambiente. Outro fator positivo sobre *benchmarking* é que a maioria das técnicas faz uso de *profilers* ou traços de eventos, os quais necessitam do código binário e da máquina para serem executados. Estes fatores tornam métodos baseados em *benchmarking* atrativos para a análise de desempenho, isso faz com que grande parte das ferramentas de análise de desempenho utilize de certa forma *benchmarking* em alguma fase da análise.

2.2.3 Métodos baseados em simulação

Métodos baseados em simulação assemelham-se aos métodos analíticos. A grande diferença entre estes métodos está na forma como os modelos do sistema e seus resultados são obtidos. Nos métodos analíticos os modelos são construídos com sistemas de equações, representando o programa em análise, já em simulações têm-se regras de comportamento que ditam como os eventos ocorrem e modificam o estado do sistema.

Nos métodos analíticos a precisão é garantida pela certeza de que se o sistema de equações estiver correto, então os resultados obtidos também estarão corretos, contudo,

[2] *Millions of Instructions Per Second*
[3] *FLoating Point Operations per Second*

em simuladores isto é variável devido as condições de sua simulação e de seus parâmetros fornecidos.

Em contrapartida, o trabalho de gerar um modelo de comportamento para simulação é mais simples do que gerar um modelo de equações analíticas. Desta forma, a simulação se torna uma ferramenta com maior flexibilidade, isso fica claro quando é desejado fazer a análise de novos programas, então é possível utilizar a mesma abordagem do programa analisado anteriormente e fazer as alterações necessárias no modelo para o simulador, o que não ocorre nas equações que representam o sistema.

É importante ressaltar o uso de simuladores para análise de desempenho em conjunto com ferramentas baseadas em métodos analíticos com o objetivo de comparar as abordagens e reduzir seus estados. Finalmente, alguns simuladores de desempenho encontrados são: PDL [11], Axe [12] e PAWS [13].

2.3 Predição de desempenho através da simulação do código executável

Nesta seção será apresentado um breve detalhamento da técnica de análise de desempenho de programas paralelos por simulação proposta por [7].

2.3.1 Descrição da metodologia

O fundamento básico deste método é a "metodologia de três passos" de Herzog [8] que busca separar o modelo para o sistema em três diferentes modelos, sendo estes: detalhamento do programa que será analisado, detalhamento da máquina em que o programa será processado e por último um modelo de interação entre os dois primeiros no momento da execução.

Nesta técnica proposta por [7] o modelo para o programa é obtido através da reescrita do código executável para obtenção do grafo que represente todos os possíveis caminhos de sua execução. Em seguida, o modelo da máquina é especificado como um conjunto de parâmetros para o simulador, como velocidade dos processadores envolvidos, vazão de dados entre as unidades de processamento, tamanho da memória, etc. Por último há o modelo de interação com as taxas de acerto em memórias *cache*, carga nos processadores e sobre o suporte de comunicação.

É clara a precisão obtida com a reconstrução do código executável em um grafo de execução. Também é evidente o baixo custo da simulação, pois não há necessidade de uma máquina paralela para ter as medições. A figura 2.1 mostra os passos que o método proposto deve executar.

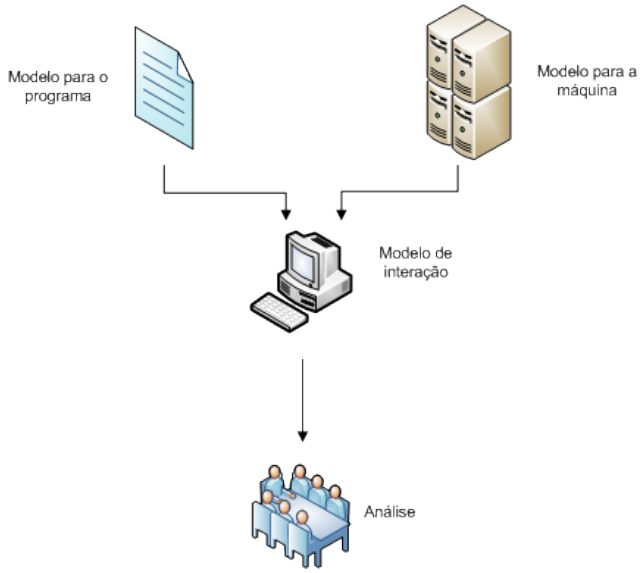


Figura 2.1: Metodologia de três passos de Herzog aplicada ao método.

2.3.2 Módulos funcionais do método

O modelo proposto por [7] é constituído basicamente por três módulos, o primeiro módulo é responsável pela construção do grafo de execução a partir do código binário, em seguida o grafo é otimizado com o objetivo de reduzir o tempo de simulação e finalmente, com as informações do ambiente de simulação o grafo é simulado. Esses módulos serão alvo de maiores detalhes a seguir.

Geração do grafo de execução

Um programa pode ser representado por um grafo dirigido, onde os vértices são pontos de execução e suas arestas são os caminhos possíveis, como pode ser visto na figura 2.2.

Antes de serem exploradas as estratégias deste modelo para a geração do grafo de execução, serão definidas cinco categorias básicas de vértices, que são elucidadas em seqüência:

- **Inicial:** representa o início do programa, portanto, o grau de incidência do vértice é nulo.
- **Passagem:** possuem uma aresta de entrada e outra de saída, aplicadas em vértices sem desvios condicionais.
- **Decisão:** há um vértice de entrada e dois ou mais vértices de decisão. Podem ser utilizados em testes e laços.
- **Agrupamento:** vértices com grau de incidência maior que um, como, por exemplo, ao término de um laço. Representam um agrupamento de vértices.

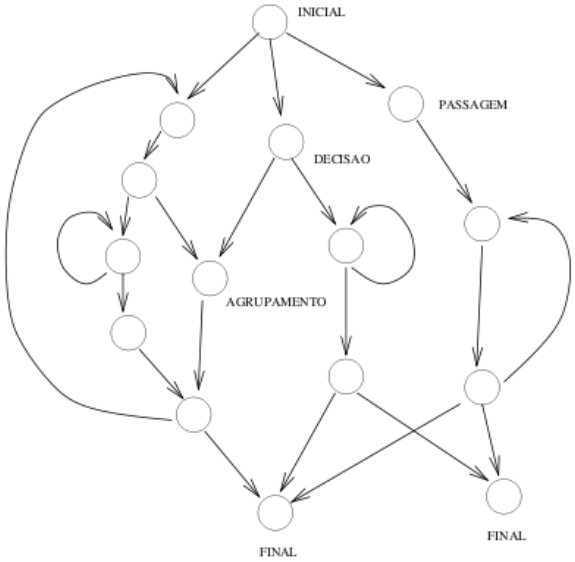


Figura 2.2: Grafo de execução de um programa.

· **Final:** vértices sem arestas de saída. Representam o término do programa.

No entanto, uma visão estática sobre os vértices não expressa a sua verdadeira função em tempo de execução, para isso as seguintes características podem ser atribuídas aos vértices:

· **Vértice de Execução:** representam computações locais, processamento matemático, lógico, testes, etc.

· **Vértice de Sincronismo:** indica que a passagem ao próximo vértice depende de outros fatores, como por exemplo, receber uma mensagem de outro processo.

· **Vértice de Comunicação:** depende do canal de comunicação para prosseguir ao próximo vértice.

Uma vez definidos os tipos de vértices e suas categorias dinâmicas, serão abordadas as estratégias para geração do grafo de execução, estas serão separadas em três etapas básicas: leitura do código executável, interpretação das instruções de máquina e agrupamento das instruções.

1. **Leitura do código executável:** nesta etapa o código executável é lido a partir de seu desmonte (*disassembly*), ou seja, são obtidas instruções de máquina particulares do processador utilizado na criação do código binário com o mapeamento de endereços lógico de cada sub-rotina.

2. **Interpretação das instruções de máquina:** a interpretação é feita a partir do mapeamento de cada instrução para seu significado semântico, as três categorias básicas são: saltos incondicionais (*jump*), saltos condicionais (*branch*) e instruções computacionais, que incluem as instruções que não são de salto. Assim um salto condicional gera um vértice de decisão, já um salto incondicional cria um vértice de passagem, final e de agrupamento.

3. **Agrupamento das instruções:** devem ser agrupadas seqüências de instruções as quais não ocorrem saltos, assim é reduzido o total de vértices para a simulação.

Otimização do grafo de execução

Não é permitido reduzir o grafo de execução sem alguns critérios que preservem a integridade do programa modelado, o tempo gasto pelo vértice para fazer a simulação e a precisão pretendida com a simulação. Nestes termos há um conflito de interesses, pois menor quantidade vértices significa resultados mais rápidos, porém, menos precisos.

O que se busca então é uma relação entre tempo de simulação e precisão dos resultados. Algumas abordagens baseiam-se em técnicas de otimização em compiladores e outras técnicas surgiram das peculiaridades do modelo de vértices usados no grafo de execução. Todavia, estas técnicas só podem ser aplicadas em um determinado conjunto de vértices. A seguir apresentam-se as três principais técnicas de otimização que podem ser habilitadas.

1. **Aglutinação de vértices de passagem.**

   É possível incorporar um vértice de passagem ao vértice destino de sua aresta de saída, contudo esta operação não pode ser realizada se o vértice em questão representa um ponto de sincronismo no programa ou então se o vértice de destino for do tipo agrupamento.

   Esta operação faz com que o tempo do vértice a ser aglutinado seja acrescentado ao vértice de destino. A aresta incidente do vértice de passagem passa a incidir no vértice de destino deste, possibilitando assim a remoção de um ponto de controle. A figura 2.3 ilustra o procedimento mencionado acima.

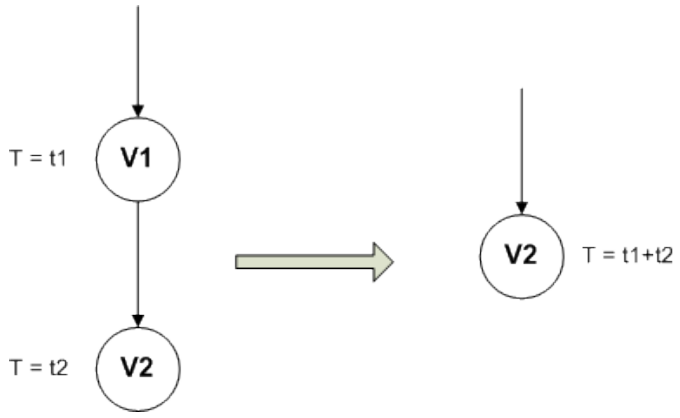
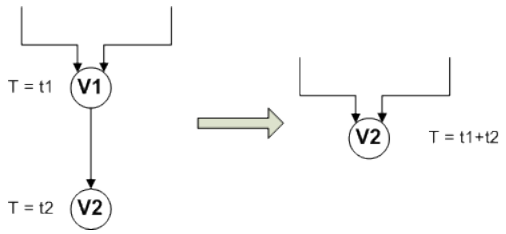


Figura 2.3: Aglutinação de vértices passagem.

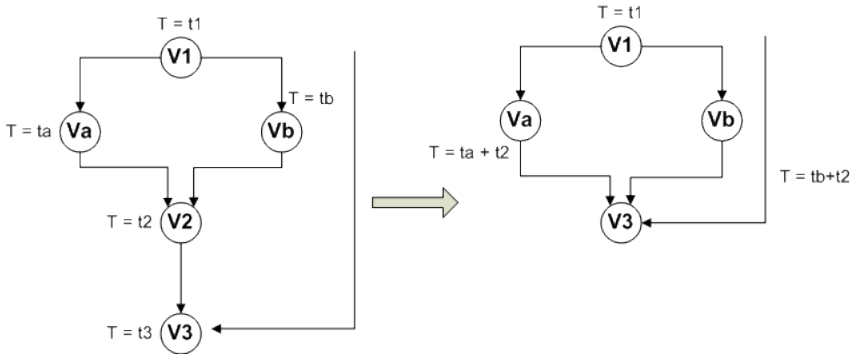
2. **Aglutinação de vértices de agrupamento.**

   De forma semelhante ao caso anterior, é possível incorporar vértices de aglutinação ao vértice destino de sua aresta emergente, mas isto só ocorre se o destino não possuir outras arestas incidentes, como mostra a figura 2.4(a).

   Também é possível reduzir um grafo mesmo quando mais de uma aresta incide sobre o nó de origem, como ilustrado na figura 2.4(b). Esta situação exige um tratamento especial, pois a remoção do vértice *V2* é feita passando suas informações para seus antecessores e esses passam a referenciar o sucessor do vértice *V2*. Esta aglutinação é possível porque as demais arestas que incidem em *V3* não pertencem aos ramos entre *V1* e *V2*.



(a) Incorporação para frente.



(b) Incorporação para trás.

Figura 2.4: Aglutinação de vértices de agrupamento.

3. **Redução de vértices comuns em ramos distintos**

   Esta técnica é semelhante à eliminação de código comum aos ramos de um teste de decisão. Aqui diminui-se o número de vértices dentro de vários ramos que partem de um vértice de decisão. Isso só é possível se os vértices não são de sincronismo ou comunicação.

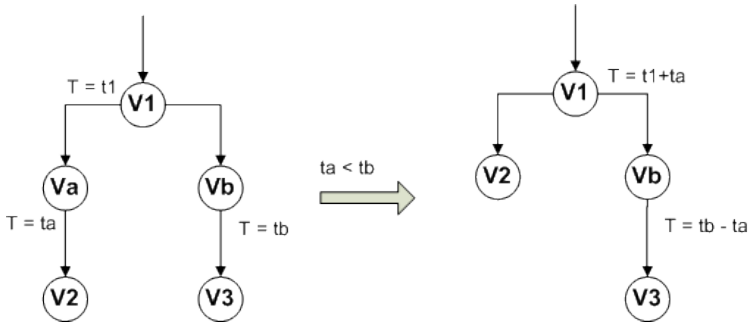


Figura 2.5: Redução de vértices comuns.

   Como é mostrado na figura 2.5, consegue-se a eliminação de pelo menos um dos vértices iniciais. Contudo, é possível obter uma redução maior quando os tempos consumidos por vários sucessores do vértice de decisão forem iguais ao de menor tempo. Portanto, aqueles vértices que atendem aos requisitos serão eliminados, mantendo aqueles cujos tempos são maiores que esse tempo mínimo.

Simulação do grafo de execução

Este último módulo do método proposto está baseado em uma estrutura centralizada de controle sobre a ocorrência de eventos, que são passagens de um vértice para o outro dentro do grafo de execução. A seguir uma síntese sobre o funcionamento do simulador, das técnicas estatísticas utilizadas e da coleta e tratamento dos resultados das simulações.

1. **Estratégia de operação**

   A figura 2.6 mostra o algoritmo utilizado pelo simulador. Neste algoritmo não serão examinados a fundo os passos 1 e 7 pois esses triviais. O primeiro passo trata da leitura do grafo de execução, o qual pode ser de um único programa do tipo SPMD[4] ou múltiplos programas do tipo mestre-escravo. O sétimo passo é uma interface entre o simulador e o usuário onde se apresentará as medidas coletadas pelo simulador. Em função disto, passa-se à descrição dos demais passos.

[4] *Simple Program Multiple Data*

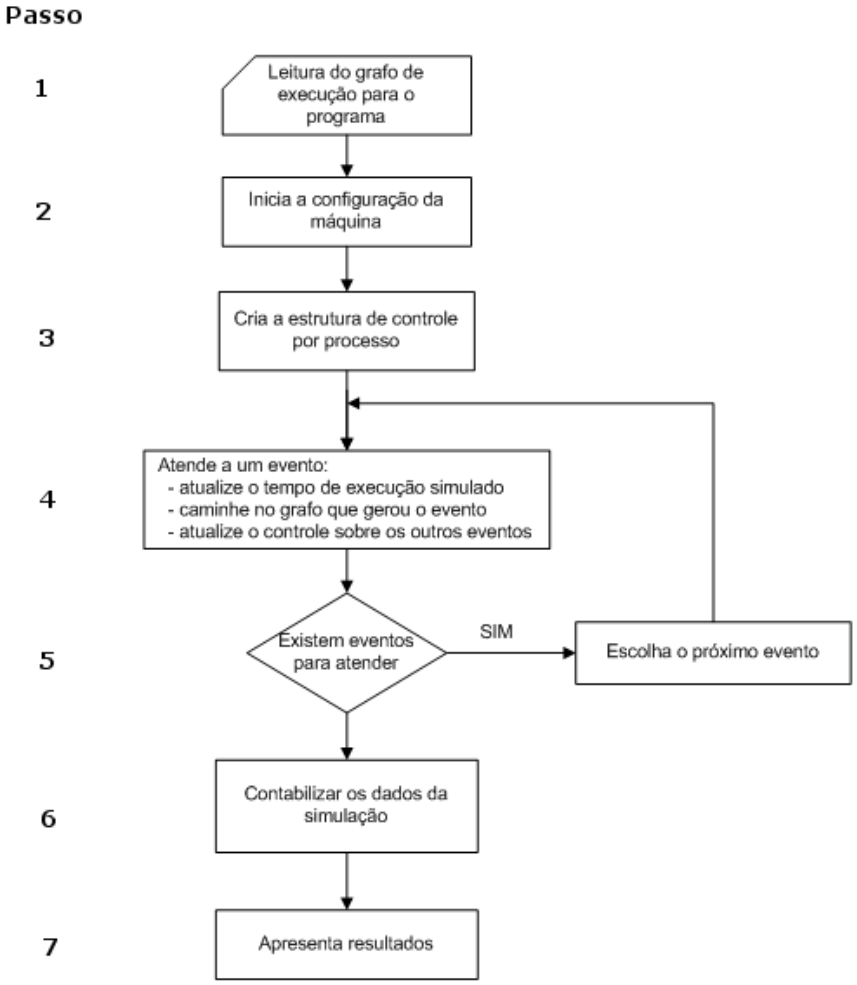


Figura 2.6: Algoritmo de simulação.

(a) **Passo 2**

Neste passo obtém-se a configuração da máquina e do ambiente de execução como descrito na subseção "Descrição da metodologia". Estas informações têm grande utilidade para a composição dos modelos de máquina e interação programa-máquina dentro da metodologia de três passos de Herzog.

(b) **Passo 3**

Após os passos 1 e 2 serem realizados, implementa-se uma estrutura de controle para cada um dos processos que serão executados. Estrutura essa que conterá informações sobre o estado do processo, qual a próxima aresta a se seguir e o momento de ocorrência do próximo evento. Outras duas estruturas são criadas, a primeira emula um processo para o meio de comunicação disponível, ou seja, um processo equivalente aos demais processos para cada ligação entre processadores. A segunda estrutura é do tipo "blackboard"contendo informações correntes sobre os processos, o que significa a existência de apontadores para

as estruturas individuais, listas de processos bloqueados e um apontador para o próximo evento a ocorrer.

(c) **Passo 4**

Cada evento atendido pelo simulador reflete em ações realizadas por parte dele. Estas ações são: a atualização do tempo de execução simulado, percurso pelo grafo em direção ao próximo vértice e, finalmente, os demais processos têm suas estruturas atualizadas.

(d) **Passo 5**

Aqui é verificado o critério de parada da simulação, que se finaliza somente quando todos os processos estejam no modo "encerrado". São excluídos os processos relativos ao meio de comunicação que não atingem este estado. Caso existam outros processos sendo executados é dada continuidade a simulação e é escolhido o próximo evento a ocorrer. A decisão é feita selecionando o evento com menor instante de ocorrência entre todos em execução. São deixados de fora aqueles que estão bloqueados por sincronismo ou comunicação.

(e) **Passo 6**

Concluída a simulação é necessário coletar e quantificar as informações geradas pelo simulador, sendo elas: tempo total de execução, "*speedup*" relativo, tempos gastos com processamento e comunicação, taxas de ocupação dos processadores dos suportes de comunicação, pontos ótimos de operação para o sistema e escalabilidade do programa.

2. **Estratégia de decisão pelo caminho a seguir**

O modelo proposto não estipula um comportamento de decisão sobre um desvio nem a quantidade de vezes que um laço será executado. No protótipo descrito foram usadas as funções de distribuição de probabilidade uniforme, normal e exponencial para simular esta tomada de decisão. A tabela 2.1 mostra uma rápida descrição dessas funções e seu respectivo uso.

A precisão do simulador está relacionada basicamente com a adaptação coerente entre os geradores de números aleatórios e os pontos em que eles são aplicados.

2.4 Considerações finais

Neste capítulo foram abordadas diversas medidas de desempenho pertinentes em sistemas paralelos, como o *speedup* e sua abordagem estática e dinâmica. Em seguida foram apresentados os métodos para análise e predição de desempenho, bem como suas vantagens

Tabela 2.1: Funções de distribuição de probabilidade e suas aplicações.

fdp	**aplicação**
uniforme	usada em testes de desvio condicional quando só caminhos são equiprováveis.
normal	usada em testes de desvio condicional com caminhos não-equiprováveis.
exponencial	usada em testes de continuidade de ciclos, arranjando-se que um número gerado indique diretamente o número de repetições. Também é usado para gerar os atrasos sofridos no acesso ao canal de comunicação.

e desvantagens. Finalmente, foi exposto o método proposto por [7] através da predição de desempenho por meio da simulação do grafo de execução.

Atualmente, existe um protótipo desta ferramenta implementado em C por [7], no entanto, esta implementação é dirigida à arquitetura de processadores MIPS e voltada para a simulação de programas paralelos em PVM[5]. Além disso, existe a necessidade de uma interface gráfica amigável ao usuário. No próximo capítulo serão abordados os detalhes da implementação em Java deste modelo, agora utilizando a plataforma x86[6] e voltado para predição de desempenho em um ambiente MPI[7].

[5]Parallel Virtual Machine
[6]Série de processadores baseados na arquitetura Intel©
[7]Message Passing Interface

Capítulo 3

Detalhamento e desenvolvimento do projeto

3.1 Conceitos básicos

A seguir serão apresentados alguns conceitos sobre as tecnologias utilizadas ao longo deste projeto, bem como a opção de uma tecnologia em detrimento de outras. Essas considerações são relevantes para entender o caminho trilhado no desenvolvimento do projeto em relação a suas implementações.

3.1.1 x86

Arquitetura de processadores desenvolvida pela Intel© no fim dos anos 70. Tornou-se popular em computadores pessoais ao longo dos anos, seus primeiros processadores foram nomeados com a terminação 86 (8086, 80186, 80286, 80386), daí surge o termo x86.

Esta arquitetura define um conjunto de instruções básicas para a família de processadores Intel©, apesar da adição de novas instruções específicas para tarefas de diversas áreas, as instruções primitivas permaneceram inalteradas, isso permitiu a geração de código binário compatível entre as diferentes versões de processadores.

A arquitetura foi escolhida devido a sua grande disseminação, em computação pessoal, corporativa e principalmente de alto desempenho. Além disso, a compatibilidade de códigos binários entre diferentes modelos de processadores torna mais simples a identificação das instruções geradas.

3.1.2 MPI

Message Passing Interface é uma especificação utilizada para o desenvolvimento de programas paralelos em computação de alto desempenho. Este padrão foi implementado

através de bibliotecas de funções em diversas linguagens de programação, como C/C++ e Fortran. Devido ao desenvolvimento dessa API[1] embutido em linguagens largamente utilizadas não foi necessário desenvolver uma nova linguagem, e conseqüentemente um novo compilador, para aplicações paralelas.

O sucesso deste padrão se deve a sua simplicidade de programação na troca de mensagens entre processos. Além disso, a possibilidade de reaproveitamento de códigos legados dessas linguagens, escritos, por exemplo, utilizando *sockets* como comunicação entre os nós de computação, evita-se esforço de porte desses programas a uma nova linguagem, basta adaptar os trechos de comunicação dos códigos legados a biblioteca de MPI.

Existem diversas implementações do padrão MPI, entre as mais comuns estão MPICH[19] e OpenMPI[20], além de diversos esforços de desenvolvimento comerciais como IntelMPI[21] e HP-MPI[22]. Neste trabalho foi utilizado principalmente o padrão OpenMPI, mas foram obtidos resultados semelhantes com o MPICH.

3.1.3 Java

A linguagem de programação Java foi escolhida devido a sua independência de plataforma em ambientes heterogêneos de *hardware* e sistemas operacionais. Graças a tecnologia desenvolvida pela *Sun Microsystems*© é possível compilar um programa em um sistema operacional *Unix-like* e executá-lo em um ambiente *Microsoft Windows*©. Isso ocorre devido a geração de *byte-codes* e não de códigos binários nativos do processador - como acontece em outras linguagens, esses *byte-codes* são interpretados por uma *Java Virtual Machine* (JVM) que o traduz para a arquitetura onde o programa será executado.

Além das características que uma linguagem orientada a objetos proporciona como herança, encapsulamento e polimorfismo, Java fornece a coleta de lixo (*garbage collector*) automática, com isso o programador não precisa se preocupar com a liberação de memória manual ao não utilizar um objeto, isso é feito automaticamente pela JVM quando um objeto não é referenciado. Devido a essas e outras características o desenvolvimento de projetos mais robustos e confiáveis tornam-se tarefas mais simples ao programador.

Outra vantagem da linguagem Java é sua fácil associação entre diversos ambientes de desenvolvimento, como *web*, celulares e mais recentemente a TV digital. Com isso é possível utilizar as classes de um projeto desenvolvido para *desktop* em um ambiente *web* por exemplo.

3.1.4 *Disassembler*

Um desmontador ou *disassembler* é uma ferramenta capaz de converter a linguagem de máquina gerada para uma arquitetura alvo em seus correspondentes símbolos de cada

[1] *Application Programming Interface*

instrução. Estes símbolos, também conhecidos como mnemônicos, auxiliam o programador de *assembly* no desenvolvimento de programas em baixo nível, já que não é necessário memorizar os códigos (*opcodes*) das instruções.

Em posse dessas informações é possível reconstruir o código executável em instruções conhecidas e assim obter um grafo de execução do programa, que é um dos objetivos deste trabalho. A ferramenta utilizada para desmonte é o *objdump* [18] e pode ser facilmente encontrada em um ambiente *Linux*. Com esse programa é possível obter além dos mnemônicos, os *opcodes* das instruções com seus endereços lógicos e, além disso, o nome dado às funções pelo programador.

3.2 Geração do grafo de execução

A geração do grafo de execução a partir de um código executável é feita, como mencionada no capítulo anterior, através de três etapas básicas: leitura do código executável, interpretação das instruções de máquina e agrupamento de instruções. A seguir serão apresentados os detalhes da implementação de cada etapa.

3.2.1 Leitura do código executável

Esta etapa é mais simples em relação às outras, o código executável é passado como parâmetro para a execução do *disassembler*, ao término de sua execução é gerado um único arquivo com o endereço lógico das instruções, seus *opcodes* e os mnemônicos, assim como o nome das funções contidas no código binário. Na figura 3.1 é apresentado um trecho do arquivo gerado pelo *objdump*.

```
080487e4 <main>:
 80487e4:       8d 4c 24 04             lea    0x4(%esp),%ecx
 80487e8:       83 e4 f0                and    $0xfffffff0,%esp
 80487eb:       ff 71 fc                pushl  -0x4(%ecx)
 80487ee:       55                      push   %ebp
 80487ef:       89 e5                   mov    %esp,%ebp
 80487f1:       51                      push   %ecx
 80487f2:       81 ec 14 01 00 00       sub    $0x114,%esp
 80487f8:       8b 41 04                mov    0x4(%ecx),%eax
 80487fb:       89 85 08 ff ff ff       mov    %eax,-0xf8(%ebp)
 8048801:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 8048807:       89 45 f8                mov    %eax,-0x8(%ebp)
 804880a:       31 c0                   xor    %eax,%eax
 804880c:       8d 85 08 ff ff ff       lea    -0xf8(%ebp),%eax
 8048812:       89 44 24 04             mov    %eax,0x4(%esp)
 8048816:       89 0c 24                mov    %ecx,(%esp)
```

Figura 3.1: Trecho do arquivo gerado pelo objdump.

Em seguida as funções geradas são separadas em arquivos que contém apenas uma

função. Cada arquivo é nomeado com a data atual (número de milissegundos a partir de 1970, devido a convenção de datas em Java), seguido do endereço lógico da função e uma extensão ".gttxt"(GraspTool TeXT). A adição da data impede a sobrescrição de arquivos, caso sejam executadas duas instâncias do gerador simultaneamente, já a extensão facilita a identificação do tipo de arquivo e sua remoção posterior. Dados como nome do arquivo criado, endereço lógico inicial e nome da função são armazenados na estrutura mostrada na figura 3.2 para utilização em fases posteriores.

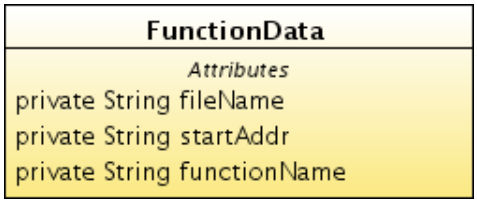


Figura 3.2: Estrutura de dados com informação do arquivo criado.

3.2.2 Interpretação das instruções de máquina

Em posse dos arquivos separados por funções é possível analisar cada função do programa isoladamente, isso facilita a reutilização de trechos analisados anteriormente. Cada arquivo é lido e em cada linha são extraídos os *opcodes* da instrução e estes são passados para uma classe especializada na identificação da instrução.

A partir da identificação da instrução a classe armazena o número de ciclos computacionais gastos para a execução da instrução e a classifica em um dos tipos básicos, que são explicados a seguir.

- **Instrução de execução:** a maior parte das instruções se enquadra neste tipo, são instruções de movimentação entre registradores, operações de lógica/aritmética, etc. Estas são de fato as operações que representam a execução do programa.
- **Chamada de função:** representa a chamada para uma função específica do programa, a função atual é colocada em uma pilha de execução e voltará a executar ao fim da função chamada. Operação pode ser denotada pelo mnemônico "CALL".
- **Salto incondicional:** é definido como um salto (*JUMP*) para um endereço lógico do código, essa categoria ainda é divida em duas subcategorias, que identificam se o salto é para um endereço posterior ou anterior ao endereço de execução atual.
- **Salto condicional:** são operações utilizadas em testes de decisão e laços de iteração, diversas instruções se encaixam nesta categoria (JZ, JNZ, JE, JNL, JA, etc.). Assim como na categoria de salto incondicional, esta também é divida em duas subcategorias que identificam se o salto é posterior ou anterior ao endereço lógico atual.

- **Retorno de função:** representa o fim da execução de uma função, essa identificação é fundamental, pois indica que a execução do programa deve seguir para a função chamadora. Esta operação é geralmente identificada pelo mnemônico RET (retorno).

- **Instrução de comunicação:** as primitivas de comunicação em MPI, como *Send*, *Recv* e *Sendrecv*, são categorizadas pois dependem de fatores externos para sua execução, como capacidade e disponibilidade do meio de comunicação, sincronismo entre as trocas de mensagens e tempo de processamento nas máquinas remotas.

Ao identificar certas instruções é necessário obter dados relevantes sobre sua operação. No caso de instruções de salto é preciso calcular seu endereço de salto, que está em seu código de instrução e assim identificar se o salto é para frente ou atrás do endereço lógico atual.

Para obter o deslocamento do salto é preciso conhecer quantos bits representam a instrução para assim manipular o deslocamento. Para calcular o endereço de salto é necessário inverter os códigos que representam o endereço e avaliar se o primeiro número é maior ou igual a 128, ou seja, o primeiro bit está ativado, em caso afirmativo o salto é para trás, neste caso é necessário fazer uma negação desse endereço para obtenção de seu deslocamento. Caso contrário, ou seja, o salto é para frente, basta adicionar o deslocamento. Em ambos os casos a soma ou subtração para obter o endereço de salto é feita em relação ao endereço da próxima instrução.

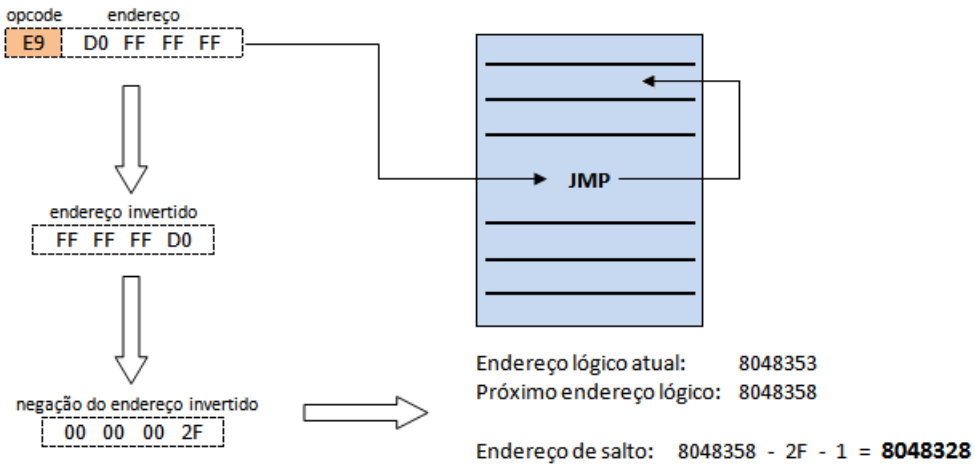


Figura 3.3: Obtenção do deslocamento para trás.

As funções de comunicação também precisam de tratamento especial. A identificação dessas operações é feita a partir da chamada da função MPI, normalmente os identificadores dessas funções seguem o formato MPI_Send, MPI_Recv e MPI_Sendrecv.

3.2.3 Agrupamento de instruções

O objetivo desta fase é agrupar as instruções de execução que estejam em seqüência no programa e gerar apenas um vértice para este bloco de instruções, contendo o somatório dos ciclos de cpu gastos por cada instrução. Essa medida reduz o número de vértices do grafo, conseqüentemente, o consumo de memória, além de acelerar o processo de simulação, devido a diminuição de vértices a serem visitados. Este agrupamento é válido, haja vista que as instruções de execução em seqüência formam um bloco de código que sempre será executado.

A estratégia adotada para o agrupamento é a interpretação das instruções e acúmulo de seus ciclos de cpu, isso é feito até que uma instrução diferente de execução seja encontrada, quando isso ocorre são gerados dois vértices: o primeiro para a instrução de execução e o segundo para a instrução diferente encontrada, esta pode ser um salto, uma chamada/retorno de função ou de comunicação.

Alguns aspectos devem ser levados em consideração na criação de um vértice, no caso de saltos para um endereço posterior ao atual, um novo vértice de destino é criado e seu endereço é adicionado a um *Set*[2], pois caso o desvio seja para o meio de um vértice de execução este terá de ser dividido em dois vértices.

Já no caso de salto com endereço anterior ao atual, é verificada na tabela de vértices (uma tabela *Hash*[3] que mapeia um endereço para um vértice) se o vértice v_1 existe, em caso afirmativo basta v_2 referenciar v_1, caso contrário este endereço está no meio de v_1 , pois os demais tipos de vértices anteriores já foram criados, portanto será necessário dividir v_1 em dois, $v_{1,1}$ fará parte da primeira metade e estará ligado a segunda metade $v_{1,2}$ e o vértice de desvio terá uma referência a $v_{1,2}$, no cálculo de ciclos de $v_{1,1}$ e $v_{1,2}$ é feita uma proporção de acordo com a quantidade de endereços de $v_{1,1}$ e $v_{1,2}$, de forma a distribuir os ciclos de v_1. Esse processo pode ser visto na figura 3.4.

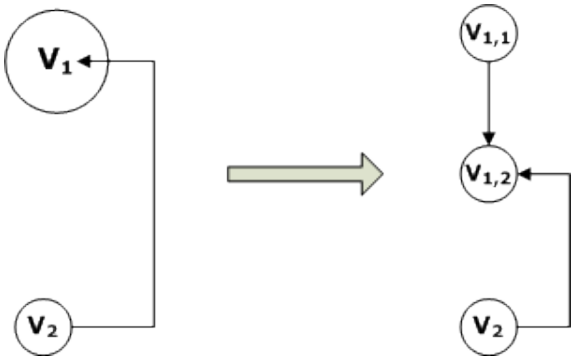


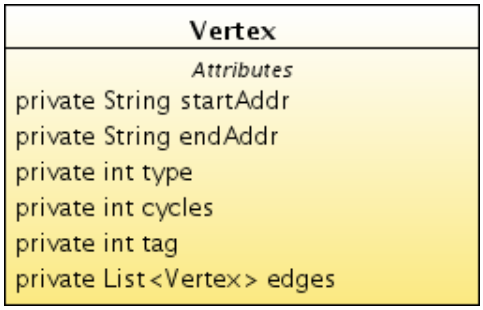
Figura 3.4: Divisão de um vértice de execução.

[2]Estrutura de dados de acesso rápido que não permite elementos duplicados
[3]Estrutura mapeia uma chave em um valor

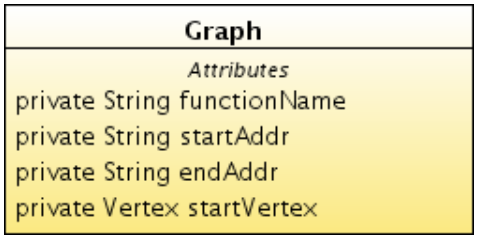
A adição de vértices de outras categorias é trivial, a tabela de vértices é consultada e caso o vértice exista este é referenciado pelo novo vértice, caso contrário um novo vértice é criado e, conseqüentemente, referenciado. A estrutura de dados utilizada para o armazenamento de um vértice é mostrada na figura 3.5(a), nela são armazenados dados como endereço inicial e final do vértice, tipo de instrução, número de ciclos necessários para execução, uma *tag* que identifica a comunicação e uma lista de vértices adjacentes.

É importante ressaltar que a adição de um novo vértice deve manter a conexidade do grafo, neste caso a primeira aresta de um vértice representa o próximo endereço lógico, isso ocorre até mesmo para instruções de salto incondicionais. As demais arestas podem representar o endereço para uma chamada de função, o endereço de salto ou então o endereço para um vértice de comunicação.

Cada função, que representa um grafo conexo de execução, é armazenada em uma lista da estrutura mostrada na figura 3.5(b). Dados como nome da função, seus endereços inicial/final da função e uma referência para o primeiro vértice da função são armazenados. Esta lista de grafos será a estrutura mais utilizada durante a simulação do grafo de execução.



(a) Estrutura dos vértices do grafo.



(b) Estrutura dos grafos do programa.

Figura 3.5: Estruturas de dados utilizadas para geração do grafo de execução

3.2.4 Interface gráfica do Gerador do grafo de execução

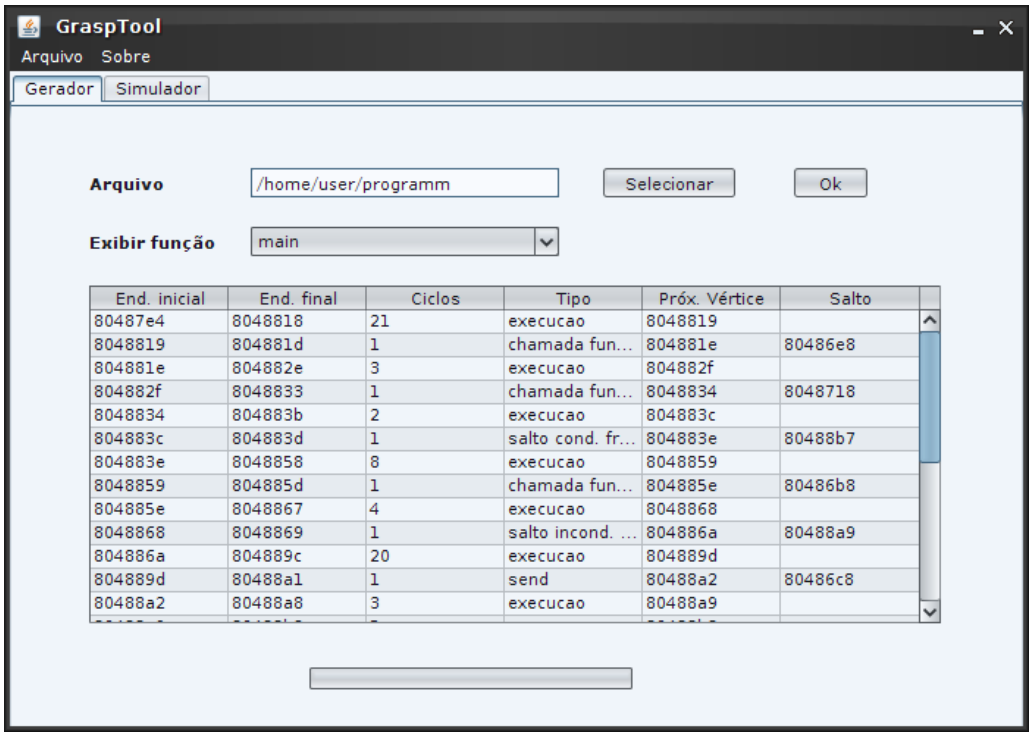
A interface gráfica do Gerador do grafo de execução é bastante simples e intuitiva, como mostrado na figura 3.6(a). Para sua execução o usuário deve fornecer o caminho para um programa executável ao clicar no botão "Selecionar"ou então através do menu "Arquivo/Abrir", a seguir o usuário deve confirmar sua execução. Durante a geração do

grafo, o usuário pode navegar nos menus do programa e inclusive interrompê-lo, isso ocorre graças a sua execução em uma *thread*[4] separada da interface gráfica.

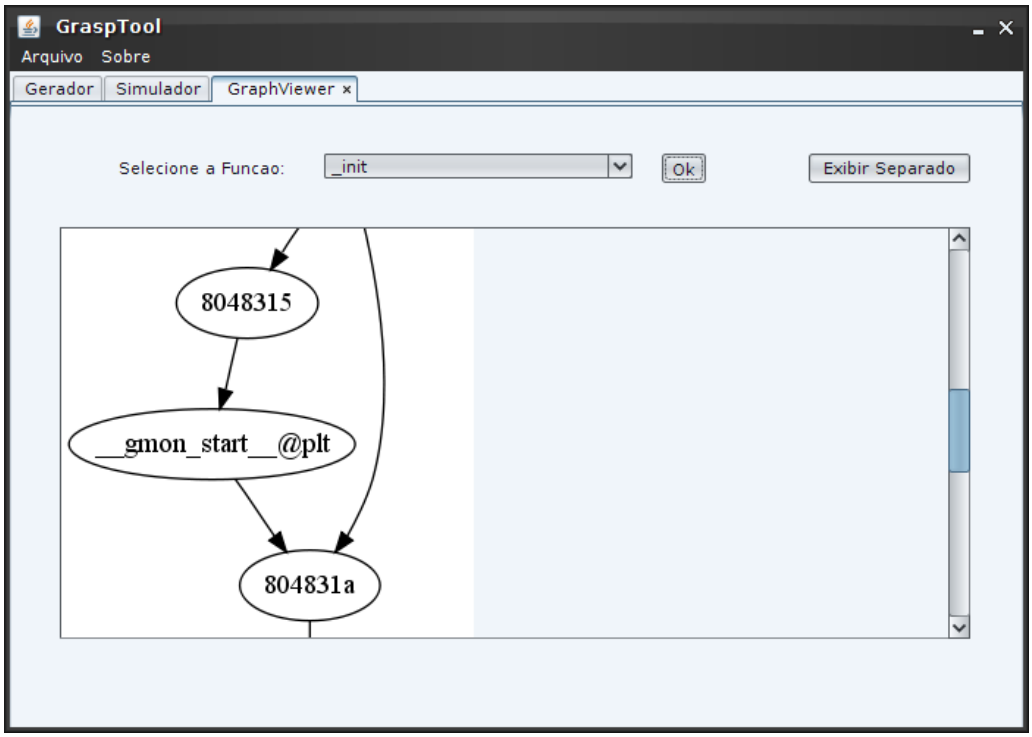
Após a obtenção do grafo de execução é possível verificar detalhes dos vértices adicionados em um grafo, como por exemplo, seus endereços inicial/final, número de ciclos gastos no vértice, tipo de instrução e o próximo endereço. Isso também pode ser visto na figura 3.6(a).

Outra funcionalidade adotada é a exibição dos grafos em forma de fluxogramas, mostrada na figura 3.6(b). Para isso foi utilizado o programa gratuito Graphviz [23] na plataforma Linux para a geração das imagens dos grafos. Ao selecionar no menu "Arquivo/Visualizar grafo", uma nova aba é criada para a visualização dos grafos. Quando o usuário seleciona uma função para visualização, o seu grafo é percorrido e seus vértices são adicionados na formatação utilizada pelo *software*. Por fim, este arquivo é passado como parâmetro para o Graphviz.

[4] *Thread* é um processo leve, possui sua própria pilha de execução, contador de programa, etc.



(a) Tela principal do gerador.



(b) Exibição de um grafo de uma função.

Figura 3.6: Interface do gerador do grafo de execução.

3.3 Implementação do Simulador

Nesta seção será discutida a implementação do simulador responsável pela coleta, interpretação e simulação das medidas de desempenho provenientes do grafo gerado como mostrado anteriormente. O simulador está presente no terceiro passo da metodologia de Herzog [8]. A seguir, segue a descrição da implementação do simulador e suas funcionalidades.

3.3.1 Inicialização do Simulador

O simulador deve receber duas entradas de dados para realizar suas tarefas. A primeira entrada de dados é o grafo de execução criado pelo Gerador e a segunda entrada é o ambiente onde o sistema será simulado; este por sua vez é fornecido pelo usuário. Desta forma, o simulador poderá criar o modelo de interação do sistema como descrito em [8]. A seguir serão abordadas as entradas de dados.

Grafo de Execução

Entrada de dados proveniente do Gerador e contém o grafo de execução de um programa executável a ser simulado. É a principal estrutura de dados do simulador e serve de base para a criação de outras estruturas utilizadas ao longo da simulação. Ela é constituída por uma floresta de grafos, onde cada grafo representa uma função do programa executável. Algumas funções são herdadas da própria API da linguagem, como a preparação para execução do programa, a entrada/saída de dados, a comunicação entre as máquinas e as demais são desenvolvidas pelo próprio programador.

Dados do Usuário

Esta última entrada está relacionada a máquina alvo, ou seja, o usuário irá fornecer detalhes do ambiente em que o programa a ser simulado é executado. A figura 3.7 mostra a tela de entrada de dados do usuário.

A primeira informação que o usuário deve fornecer ao simulador está relacionada a largura de banda. Esta informação é relevante para a estimativa do atraso durante as trocas de mensagens MPI, as entradas fornecidas contêm velocidades padrão em adaptadores de rede. Sua unidade é representada em Mbps[5].

A seguir, temos a informação do *clock* das máquinas ou o *clock* médio das máquinas caso o programa seja executado em um ambiente heterogêneo. Com esta informação e o número de ciclos gastos pelo programa é possível fazer uma análise do tempo gasto para executar o programa, mostrada na equação 3.1.

[5] *Megabits per second*

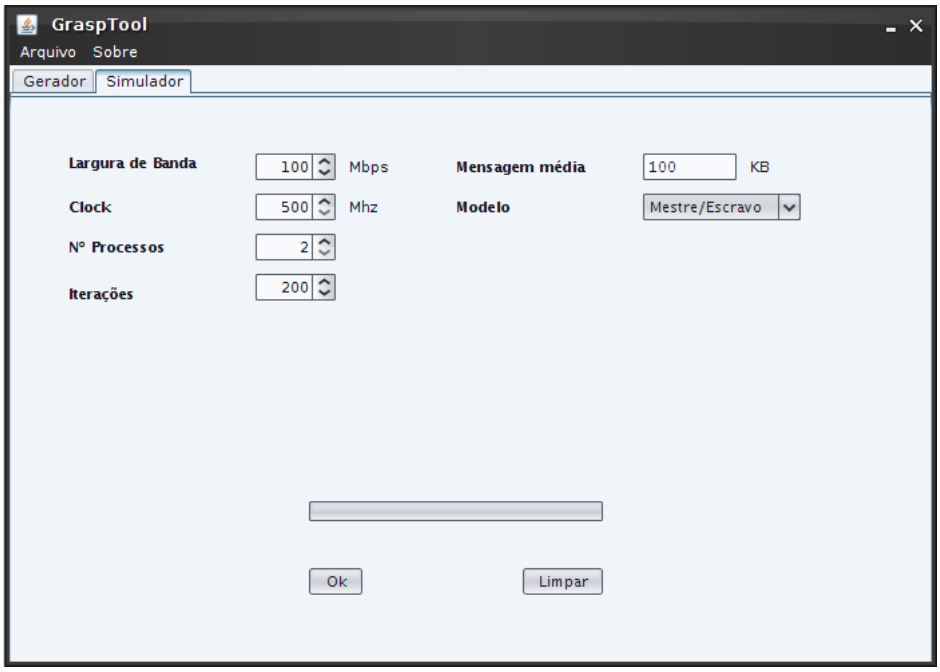


Figura 3.7: Tela principal do simulador.

$$Tempo = \frac{ciclos}{clock} \tag{3.1}$$

Também deve ser informado ao simulador o número de processos a serem simulados. Os valores fornecidos estão em potência de dois e representam a quantidade de máquinas que executariam o programa simulado em um ambiente real.

Em seguida informa-se o tamanho médio da mensagem, que será expresso em Kb[6], para o cálculo do atraso que a rede oferece ao ambiente de execução quando ocorre uma comunicação entre processos.

O usuário também deverá informar ao simulador a quantidade máxima que um laço deve ser executado. Essa informação é relevante a medida que nenhum laço irá executar eternamente, portanto, este valor atua como um limitante superior ao número de iterações.

Por fim o usuário deverá escolher o modelo de programa concorrente que será simulado. Por enquanto o único modelo disponível é o mestre/escravo. No entanto, foram feitos estudos iniciais sob o paradigma SPMD, cujas abordagens podem ser vistas nas páginas a seguir. No modelo mestre/escravo, um processo é considerado mestre e é responsável pelo gerenciamento dos dados, envio do trabalho a ser realizado pelos demais processos e recebimento dos resultados. Os demais processos são considerados escravos, cuja função é receber uma porção de dados do processo mestre, realizar as computações necessárias e enviar os resultados ao processo mestre. Esse processo pode ser visto na figura 3.8. Já sob o paradigma SPMD, cada processo se comunica com seu vizinho e também pode passar

[6]Kilobytes

mensagens a uma máquina principal.

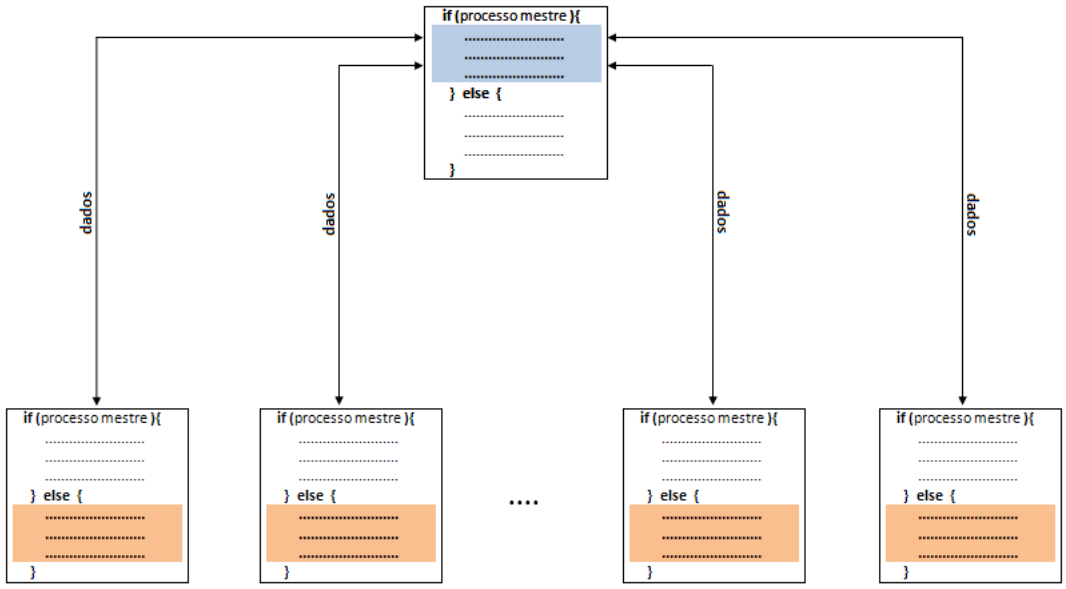


Figura 3.8: Modelo de programa sob a arquitetura mestre/escravo.

Fornecidas as entradas, o usuário pode pressionar o botão "Ok"e o simulador busca por vértices de comunicação na floresta de grafos. Isso é feito pois não é possível identificar em tempo de compilação se a troca de mensagens acontece no sentido mestre/escravo ou na via oposta por exemplo. Esta informação pode ser obtida apenas em tempo de execução, portanto, o usuário deve conhecer a *priori* se o vértice de comunicação é do mestre ou do escravo. Finalmente ocorre a simulação do código executável fornecido. A figura 3.9 mostra como a solicitação do tipo de vértice de comunicação ocorre.

3.3.2 Motor de Simulação

Nesta seção será discutido como o simulador interpretará as informações inseridas pelo usuário e pelo gerador. Serão abordadas as estratégias para simulação de programas, o modelo de execução para programas com o paradigma mestre/escravo e as funções probabilísticas para decisão em testes condicionais e laços de repetição.

Ao término dos passos descritos na subseção anterior, o simulador configura suas estruturas de dados para ajustar-se ao problema submetido. A classe de simulação recebe como parâmetro uma lista de grafos e o ambiente de execução fornecido pelo usuário. Com estas informações, seu construtor inicia os vetores utilizados com o número de processos especificados, desta forma cada processo tem isolada sua estrutura de gerenciamento da simulação.

A seguir as funções de probabilidade são carregadas. Este projeto utiliza as funções de distribuição de probabilidade normal e exponencial. Assim como mencionado no capítulo anterior, a função de distribuição de probabilidade normal é encarregada de tratar testes de

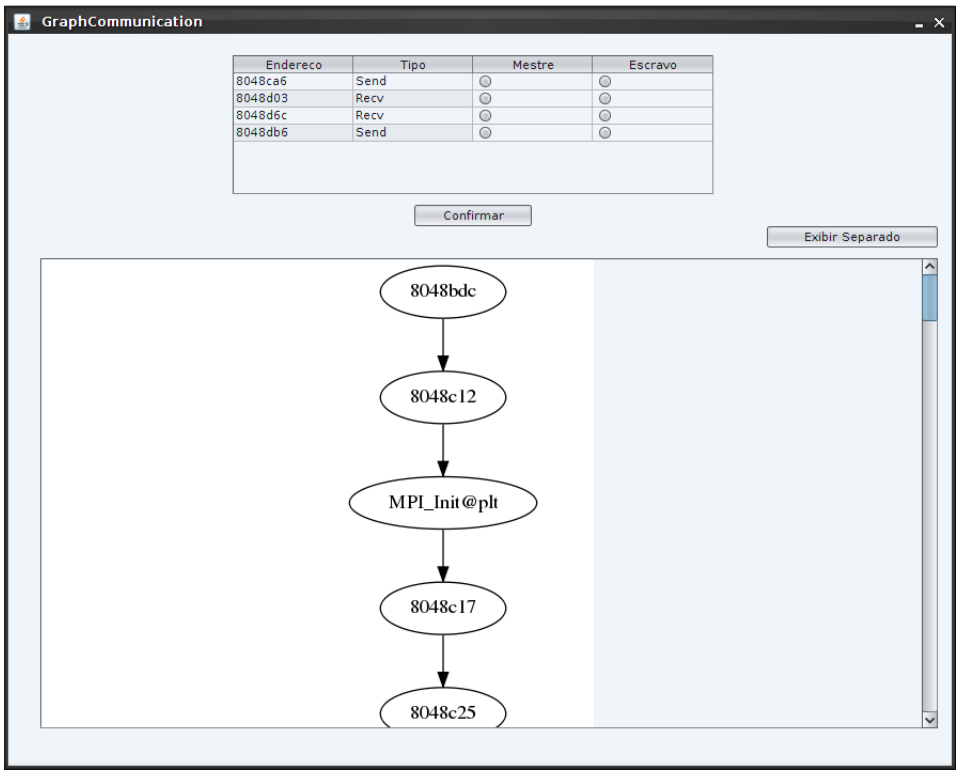


Figura 3.9: Solicitação de informação sobre o tipo de vértice.

decisão. O principal motivo de sua adoção é devido ao seu comportamento, que mais se aproxima das tomadas de decisão em testes.

Já a função distribuição exponencial está relacionada a laços de repetição. Essa função foi escolhida pela sua característica aproximação do eixo das abscissas, desta forma, quanto mais um determinado teste de um laço for verdadeiro, maior será a probabilidade que o próximo teste seja falso, portanto, segue um padrão de laços. Essas funções foram inicialmente implementadas na linguagem C por [24], para este projeto foi feito o seu porte para Java.

Terminada esta primeira etapa, a simulação propriamente dita é iniciada. O pseudo-código mostrado a seguir representa a simulação. A simulação pode ser dividida em quatro etapas relacionadas ao critério de parada da simulação, ao processo que irá executar a próxima instrução, a execução da instrução e por fim, terminada a simulação, exibição dos dados coletados pelo simulador.

```
1  enquanto for verdadeiro {
2      para todos os processos ativos:
3           verifique se a pilha de funções chamadas está vazia;
4      se todas as pilhas estão vazias{
5          saia do enquanto;
6      }
7      para todos os processos ativos:
8           selecione o processo i com menor número de ciclos;
9      executar o vértice do processo i;
10 }
11 mostrar dados coletados ao usuário;
```

Critério de Parada

Como podemos verificar no algoritmo acima, a primeira parte, que compreende as linhas 2-6, verifica para todos os processos ativos se suas pilhas estão vazias, caso as pilhas de todos os processos estejam vazias os processos terminaram sua execução, portanto, a simulação é finalizada. Como pode ser notado a simulação termina de fato somente quando todos os processos estiverem com sua pilha de funções vazia.

Instrução de máquina a ser executada

Esta etapa está compreendida entre as linhas 7-8 e busca pelo processo com vértice de menor número de ciclos. Em posse deste vértice, o processo ao qual ele pertence simulará sua execução. Esta estratégia está fortemente ligada ao acumulador de ciclos dos processos que será explicado a seguir.

Execução da instrução de máquina

Esta etapa recebe como argumento o índice do processo que contém o vértice de menor ciclo para executar. O vértice é passado para seu tipo correspondente para simulação. Os vértices estão classificados, conforme mencionado na subseção 3.2.2, com os tipos: execução, chamada e retorno de função, saltos incondicional/condicional e comunicação. A seguir serão descritas suas operações durante a simulação.

1. **Vértice de execução**
   Sua tarefa é adicionar os ciclos gastos no vértice atual ao total de ciclos acumulados em cada processo. Isso ocorre devido a execução paralela dos programas, portanto, os demais processos executaram no mínimo os ciclos deste vértice. Sua última função é decrementar de cada processo o número de ciclos executados pelo vértice atual,

haja vista que estes já foram contabilizados ao total de ciclos executados. Este procedimento é mostrado na figura 3.10. Ao atualizar esses dados tem-se o cuidado de não modificar os ciclos de processos bloqueados, já que estes estão aguardando um evento.

Após atualizar os dados de ciclos, o vértice em execução avança para o próximo nó e a simulação prossegue.

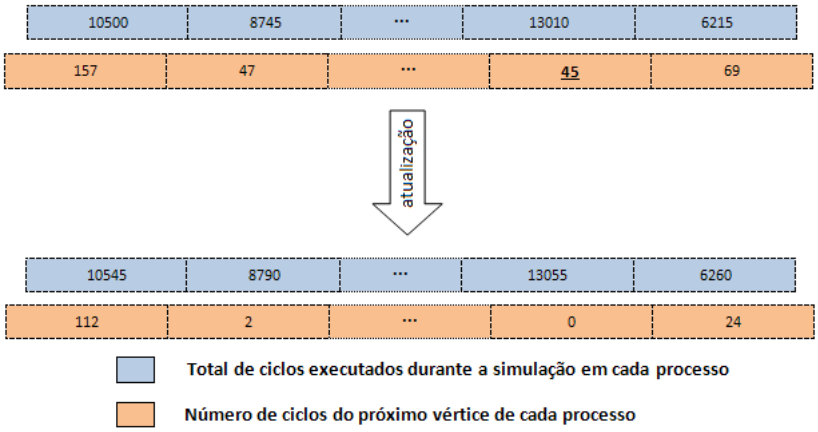


Figura 3.10: Atualização de dados de um vértice de execução.

2. **Chamada de função**

   A princípio os ciclos gastos para essa instrução são contabilizados, ou seja, a função de vértice de execução é chamada e atualiza os ciclos gastos para simular esta instrução. A seguir, uma referência para o próximo vértice é colocada em uma pilha. Assim, é possível restaurar a execução do programa ao término do procedimento chamado. O uso de pilhas se faz necessário devido a possibilidade de chamadas de função dentro de outras funções.

   Finalmente a referência para a próxima instrução é alterada para a função chamada. Assim a próxima execução deste processo será em uma nova função.

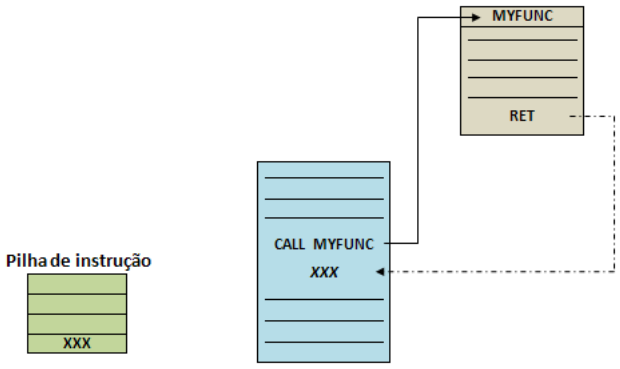


Figura 3.11: Simulação de uma chamada de função.

3. **Retorno de função**
   A simulação deste vértice verifica o estado da pilha de chamadas de função. Caso haja apenas uma chamada de função na pilha a simulação termina, devido ao fato que este é o retorno da principal função do programa, ou seja, a função chamadora dos demais procedimentos. Em caso contrário, a referência do próximo vértice é ajustada para o vértice extraído da pilha de instruções, mencionada no item anterior.

4. **Salto incondicional**
   A simulação deste vértice é simples, assim como ocorre na execução de uma instrução de salto incondicional, o salto irá ocorrer para um trecho do código independentemente de alguma condição. Portanto, basta ajustar a referência da próxima instrução do processo para o vértice de salto.

5. **Salto condicional**
   Este vértice é crucial para a predição de execução fiel do programa, já que este vértice identifica testes condicionais e laços de repetição, e em função do seu tipo é feita a escolha da melhor função de distribuição de probabilidade.

   Uma característica de vértices de decisão é que seu salto é sempre para um endereço lógico posterior ao atual, portanto, nesta situação a execução do bloco *if* ocorre em função da distribuição de probabilidade normal. Caso o bloco *if* não execute, o corpo *else* será executado ou então o trecho de código posterior ao bloco *if*, caso não exista *else*. A figura a seguir mostra um teste de decisão após seu desmonte, note que o endereço de salto na instrução de decisão (JLE) é para um endereço posterior ao atual.

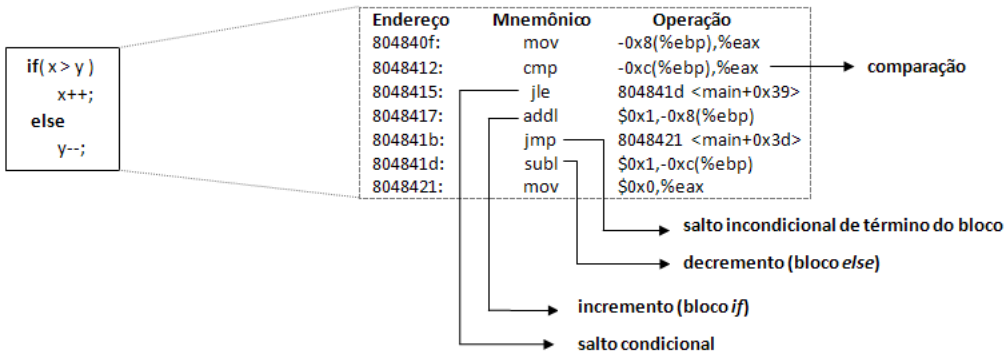


Figura 3.12: Desmonte de um teste de decisão.

Já em laços o seu salto é para um endereço lógico anterior ao atual. Neste caso aplica-se a função de distribuição de probabilidade exponencial na tomada de decisão. Outro fator que deve ser levado em consideração na tomada de decisão é se o número de iterações do laço irá ultrapassar o limite estabelecido pelo usuário, haja vista que a simulação não executará infinitamente. Para o controle das iterações é utilizado

uma tabela *hash* que mapeia o endereço do laço para o número de iterações em cada processo simulado. Além disso, o controle de iterações é utilizado na geração dos resultados, essa é uma boa estratégia para identificar os gargalos de execução.

Após a tomada de decisão, a referência para o próximo vértice a executar é ajustada para mais uma iteração do laço ou para a próxima instrução de execução.

A estrutura de laços segue o formato mostrado na figura 3.13. No início há um salto incondicional para a instrução de comparação, esta operação afetará ou não as *flags* de referência para o salto, em seguida o salto condicional é efetuado ou não, de acordo com as *flags* ativadas. Esse formato é o mesmo para laços *for*, *while* e *do...while*. Com uma pequena diferença em laços *do...while*, pelo menos uma iteração é realizada com este comando.

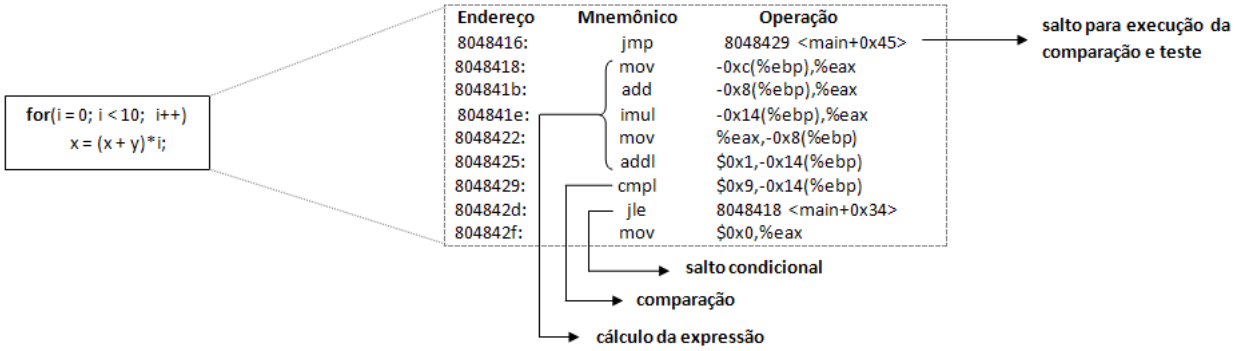


Figura 3.13: Desmonte de um laço de iteração.

6. **Comunicação**

   Os vértices de comunicação estão relacionados a primitivas MPI como send, recv e sendrecv. A função *send* é não-bloqueante, por exemplo, um processo A envia uma mensagem ao processo B, a execução de A prossegue normalmente após o envio da mensagem. Já as funções *recv* e *sendrecv* são bloqueantes, caso um processo A esteja em *recv*, sua execução só prossegue com a chegada de um *send* de B. Já no caso de *sendrecv*, o processo A envia uma mensagem e fica em *recv* aguardando uma mensagem de B.

   As características de envio e recebimento de mensagens entre os nós de computação estão fortemente ligadas ao paradigma adotado para a simulação. Os detalhes de implementação dos paradigmas implementados na seqüência.

3.3.3 Simulação do paradigma Mestre/Escravo

Assim como mencionado na subseção 3.3.1, este paradigma divide as tarefas entre dois tipos de processos: mestre e escravo. O primeiro é encarregado de enviar as tarefas aos

demais processos, estes que realizam o processamento de fato, em seguida os escravos retornam os resultados ao mestre, e este cuida da junção dos dados e apresentação ao usuário.

Um dos pontos cruciais da simulação sob este paradigma é a identificação dos vértices que separam o código do mestre para o código do escravo, pois durante a simulação o código mestre deve executar estritamente suas funções, o mesmo vale para o código do escravo. Para isso devem ser identificados os vértices de decisão que representam a separação do códigos.

O outro fator fundamental está relacionado aos vértices de comunicação contidos em laços. No caso do código do mestre, esses laços devem executar para a quantidade de escravos, haja vista que se mais mensagens forem enviadas/recebidas o resultado da simulação pode ser inesperado.

Os detalhes de implementação desses fatores fundamentais utilizados na simulação serão apresentados a seguir.

Identificação do vértice de separação dos códigos

O mapeamento dos vértices que separam os códigos é feito utilizando um percurso único pelo grafo, através de algumas modificações no algoritmo BFS[7]. Uma das vantagens desse algoritmo é sua passagem única pelos nós do grafo, ao contrário da simulação que pode passar por um vértice inúmeras vezes para representar sua execução.

O algoritmo a seguir demonstra sua execução, a princípio é passado o vértice de início da simulação, ou seja, o primeiro vértice da função principal do programa. Na linha 2, uma fila é criada para o controle dos vértices a serem visitados. Nas linhas 3 e 4 são declarados dois tipos de vértices para controle interno. Já as listas declaradas na linha 5, irão armazenar os vértices de comunicação do mestre e escravo, respectivamente. Na seqüência, o vértice inicial é adicionado à fila para o percurso no grafo. Este percurso será avaliado enquanto houver vértices não visitados, ou seja, enquanto a fila não estiver vazia, conforme mostrado na linha 9.

Em seguida, o primeiro vértice da fila é removido e para cada vértice adjacente é verificado se este já foi visitado anteriormente. Caso este não tenha sido visitado, este é marcado como visitado, isso garante o percurso único pelo vértice. Além disso, a relação de parentesco do vértice é armazenada, isso será vital para a identificação dos vértices que separam o código do mestre para o escravo. Na linha 15, o vértice adjacente ao extraído é adicionado na fila, isso garante a passagem por seus descendentes. Por fim, na linha 17 é verificado se o vértice é de comunicação, em caso afirmativo este é adicionado em sua lista específica.

[7] *Breadth First Search,* utilizado em teoria dos grafos

```
1  BfsModificado( verticeInicial )
2     Fila F
3     Vertice S
4     Vertice T
5     Lista L1, L2
6
7     coloque verticeInicial na fila F
8
9     ENQUANTO a fila F não estiver vazia FAÇA
10           S recebe primeiro vértice da fila F
11           PARA CADA T adjacente a S FAÇA
12               SE T ainda não foi visitado ENTÃO
13                    marque T como visitado
14                    marque S como pai de T
15                    coloque T na fila F
16
17                    SE T for um vértice de comunicação FAÇA
18                        SE for comunicação do mestre FAÇA
19                             adicione T na lista L1
20                        SENÃO
21                             adicione T na lista L2
```

Em posse das relações de parentesco e dos vértices de comunicação, basta percorrer o caminho traçado a partir dos nós de comunicação do mestre e escravos e será encontrado o vértice em comum. Desta forma são obtidos os vértices que separam os códigos mestre e escravo.

Ao término desta etapa serão armazenados os vértices que dividem o código entre mestre e escravo, assim durante a simulação de um teste condicional *if* também será levado em consideração se o teste divide ou não o código, caso isso aconteça cada processo mestre ou escravo é levado de forma determinística a sua porção do programa.

Identificação de vértices de comunicação contidos em laços

Para identificação dos vértices de comunicação contidos em laços é possível utilizar o mesmo algoritmo de percurso usado no item anterior, com uma alteração, a inserção em uma lista dos vértices de laços de repetição. Na figura 3.14 é mostrada, a partir do desmonte de um programa, uma troca de mensagem dentro de um laço de repetição. Neste algoritmo deve ser armazenado o vértice de salto condicional, com mnemônico JLE.

Em seguida, para todos os laços de repetição é aplicado um algoritmo de percurso, cujo

critério de parada é o encontro de um vértice de comunicação ou então o fim do laço. Note que este percurso não é para o grafo todo, é apenas para os vértices contidos no laço de repetição, portanto, sua execução é menos custosa do ponto de vista computacional.

Ao término desta etapa são armazenados os vértices de laços de repetição que contém nós de comunicação. Com isso, durante a simulação de um laço sob este paradigma, é avaliado se o laço possui um vértice de comunicação. Em caso positivo é verificado se a iteração irá exceder o número de escravos, pois as iterações, neste caso, estão condicionadas ao número de processos escravos.

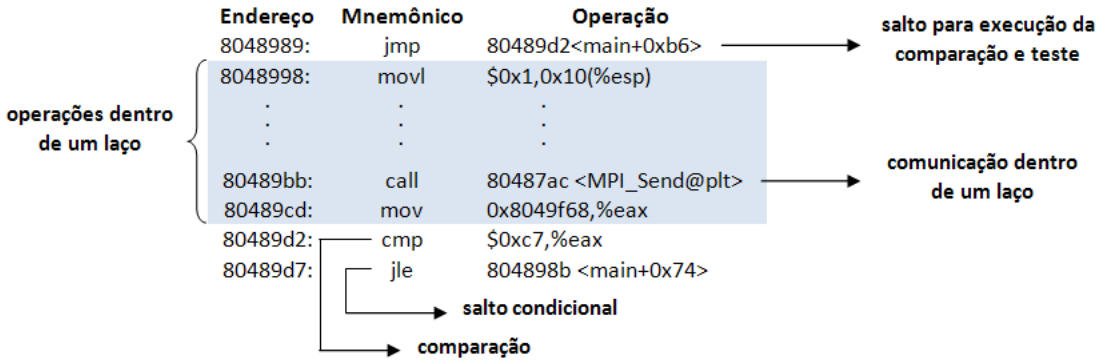


Figura 3.14: Desmonte de um laço com troca de mensagem.

3.3.4 Simulação do paradigma SPMD

Além da implementação da arquitetura mestre/escravo para simulação de programas paralelos, este trabalho teve uma abordagem inicial no desenvolvimento da simulação sob o paradigma SPMD . Este padrão compreende diversos modelos de comunicação entre os processos e conseqüentemente diversos estilos de programação, devido a essa heterogeneidade foi explorado um modelo específico, mostrado na figura 3.15, que representa a comunicação em malha de uma dimensão.

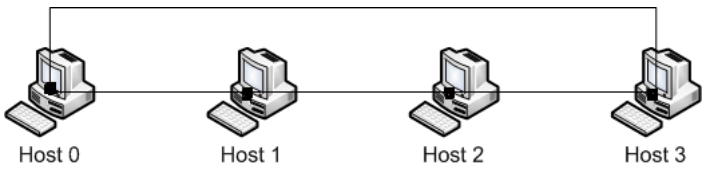


Figura 3.15: Modelo de malha em uma dimensão.

Como pode ser visto, as máquinas que compõem este padrão estão conectadas em pares, ou seja, cada máquina se comunica somente com seu vizinho da esquerda e da direita. Há também a figura de uma máquina central neste paradigma, essa por sua vez tem como objetivo a centralização das informações, configurações iniciais do trabalho e também tem papel no processamento dos dados. A figura 3.16 ilustra esta máquina central.

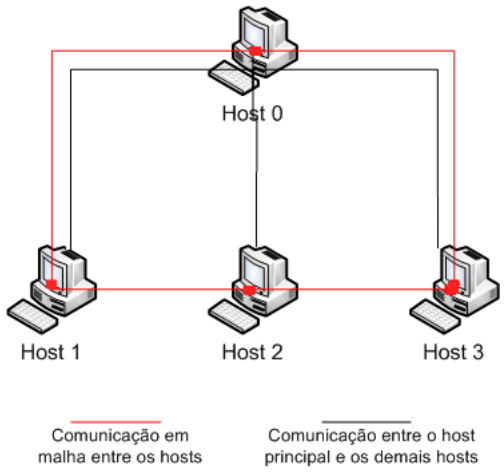


Figura 3.16: Máquina principal no modelo malha em uma dimensão.

Assim como foi citado anteriormente, não é possível saber em tempo de compilação o destinatário e o remetente de uma troca de mensagem. Diferentemente do modelo mestre/escravo é preciso obter algumas informações adicionais, estes dados estão relacionados a origem e destino da mensagem, se esta é para o vizinho da esquerda/direita, se a mensagem pertence a máquina central (envio de configurações, etc.) ou até mesmo se é uma mensagem de resposta para a máquina central (devolução dos resultados).

São criadas duas estruturas de configuração, uma delas é endereçada ao processo principal e a segunda aos demais processos, portanto no momento da nomeação das instruções de troca de mensagem será necessário fazer este passo duas vezes conforme dito anteriormente.

Com base nestas informações passa-se a configuração do grafo de execução de tal forma que cada processo possa executar sua respectiva mensagem de comunicação. Esta configuração baseia-se na procura de instruções de teste de decisão que contenham em seu corpo uma instrução de troca de mensagem, por exemplo, quando é encontrado uma instrução de troca de mensagem, é verificado primeiramente se esta pertence ao processo, caso esta verificação seja verdadeira é feita a busca por instruções de teste de decisão anteriores a esta, tendo encontrado, este vértice é marcado como verdadeiro, ou seja, este processo precisa executar o corpo verdadeiro do teste de decisão. Já se a instrução de troca de mensagem não pertence ao processo fazem-se os mesmos passos anteriores, mas desta vez o vértice de teste de decisão é marcado como falso, ou seja, o processo tem que executar a parte falsa do teste de decisão. A figura 3.17 e o pseudocódigo a seguir irão ilustrar o que foi discutido.

```
1   iteração recebe zero
2   ENQUANTO iteração for menor que dois {
3      SE a pilha de funções estiver vazia {
4         incrementa iteração
5         grafo volta ao início
6         limpa pilha de testes de decisão
7      }
8      ESCOLHA tipo de vértice {
9         salto condicional para frente {
10           coloca o vértice na pilha de testes de decisão
11           inicializa vértice na tabela hash
12        }
13        comunicação {
14           SE este vértice pertence ao processo {
15              ENQUANTO a pilha de teste de decisão não estiver vazia {
16                 Se o vértice de comunicação está no corpo de instruções
17                  do teste de decisão do topo da pilha de teste de decisão {
18                     marcar como verdadeiro este vértice de teste de decisão
                        na tabela hash
19                 }
20                 desempilhar o topo da pilha de teste de decisão
21              }
22           }
23           SENÃO {
24              ENQUANTO a pilha de teste de decisão não estiver vazia {
16                 SE o vértice de comunicação está no corpo de instruções
17                  do teste de decisão do topo da pilha de teste de decisão {
18                     marcar como falso este vértice de teste de decisão na
                        tabela hash
19                 }
20                 desempilhar o topo da pilha de teste de decisão
21              }
25           }
26        }
27        retorno de função {
28           retorna para a função chamadora
29        }
30     }
31     Vá para o próximo vértice
32  }
```

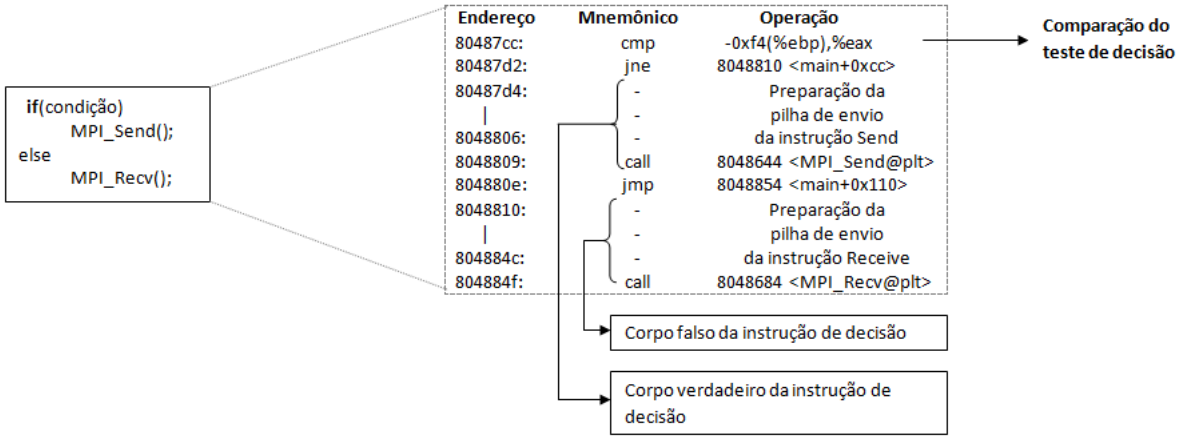


Figura 3.17: Desmonte de uma instrução de decisão.

Como é possível verificar, o algoritmo monta duas configurações baseadas no teste de decisão que contém instruções de troca de mensagem. A primeira está relacionada ao *host* principal e a segunda para os demais processos. Deste modo os processos executarão suas respectivas instruções de troca de mensagem, sem o risco de um teste de decisão levá-los a execução de trechos de outros processos. O algoritmo apresentado ainda precisa de alguns ajustes, como por exemplo, a limitação de iterações em laços que contém primitivas de comunicação.

3.3.5 Medidas de Desempenho Coletadas

Nesta subseção serão abordadas as medidas de desempenho coletadas pelo simulador para a análise de desempenho, as estratégias utilizadas para a coleta e a maneira como estas medidas serão exibidas ao usuário. São quatro os tipos de medidas que o simulador coletará do programa executável, estas medidas irão ajudar o usuário na identificação dos gargalos de seu programa como também seus pontos críticos, facilitando assim o diagnóstico da situação.

A primeira medida a ser retirada na simulação é responsável por mostrar ao usuário o tempo total que o processo gastou para encerrar sua execução. Esta medida é obtida através do acumulador de ciclos descrito na subseção anterior. Com esta medida o usuário poderá comparar a variação do tempo entre todos os processos. Assim pode ser levado em consideração o tempo mínimo e máximo de execução do programa.

Também são coletados em todos os processos as trocas de mensagens MPI, em função disso é obtida a quantidade e o tamanho total das mensagens trocadas entre os processos. E a partir da largura de banda fornecida pelo usuário é possível obter o tempo total gasto para a transmissão dessas mensagens.

Outra medida importante explorada está relacionada às funções presentes no programa

simulado. Esta medida é fundamental pois provê um contador de execução de determinada função, além disso fornece uma medição de quantos ciclos foram gastos neste trecho de código. Desta forma, o usuário poderá verificar se esta função foi um gargalo na simulação. Para coletar esta informação é utilizada uma pilha que contém as funções chamadas pelo processo. Desta forma cada vez que esta função é chamada incrementa-se seu contador e, enquanto esta mesma função estiver no topo da pilha, os ciclos de máquina executados por ela são guardados em outro acumulador de ciclos, específico para cada função.

Por fim, a última medida explorada pelo simulador está relacionado aos laços de repetição, que tem como princípio mostrar ao usuário a quantidade de vezes que um determinado laço de repetição foi executado. Esta medida é importante para o usuário verificar se um dado laço de repetição está sendo o gargalo do seu programa. Vale ressaltar que uma das entradas fornecidas pelo usuário é o número máximo de vezes que um laço pode ser executado, portanto, os laços que alcançam este limite superior são possíveis gargalos do programa.

3.4 Considerações finais

Este capítulo apresentou as principais etapas e estratégias para o desenvolvimento do projeto. Foram descritos alguns algoritmos relevantes para seu funcionamento além de elementos visuais para melhor ilustração. Esses passos têm como objetivo apresentar as características mais relevantes na implementação das especificações do modelo.

Os testes efetuados para validação da eficiência do projeto serão apresentados no próximo capítulo. Estes testes serão seguidos por seus resultados e a análise das informações obtidas.

Capítulo 4

Testes e Resultados

Este capítulo irá apresentar alguns testes, resultados e suas análises de programas paralelos MPI. Para tanto foram utilizados programas escritos na linguagem C usando as bibliotecas do OpenMPI. O sistema operacional utilizado é baseado na plataforma Linux. Os testes reais foram realizados no cluster do GSPD[1], este é constituído de 8 máquinas e um *front-end*. Todas as máquinas possuem processador Intel© *Dual Core* de 1.8Ghz de *clock*.

Os testes foram realizados sob o paradigma mestre/escravo devido a necessidade de ajustes no modelo SPMD apresentado anteriormente.

4.1 Testes sobre o modelo mestre/escravo

A seguir serão apresentados os testes realizados para o paradigma mestre/escravo. Esses testes incluem simulações e *benchmarks* reais de avaliação do sistema.

O primeiro programa calcula um integral em um intervalo definido. Já o segundo realiza as operações necessárias para uma multiplicação de matrizes paralela.

4.1.1 Cálculo de uma integral

O primeiro teste é um programa que calcula a integral de uma função. Uma integral pode ser definida como a área ocupada sob uma função f qualquer. Integrais podem ser obtidas por métodos analíticos, através de expressões matemáticas ou métodos iterativos com o auxílio computacional. Devido a complexidade de algumas integrais o modelo analítico nem sempre é de trivial solução, portanto pode-se recorrer aos modelos iterativos para obter uma aproximação do resultado.

O algoritmo desenvolvido divide a função f em intervalos regulares e passa aos escravos o cálculo de cada trecho. Este cálculo é feito por cada escravo ao dividir seu trechos em

[1]Grupo de Pesquisas em Sistemas Paralelos e Distribuídos

partes ínfimas e assim aproximar a região a um trapézio. Após o cálculo de seu intervalo, cada escravo envia a resposta ao mestre de sua solução.

Trocas de mensagens

No teste real foram colocados em execução oito processos (um mestre e sete escravos). A primeira medida avaliada está relacionada ao número de trocas de mensagens entre os processos, a figura 4.1 exibe os resultados obtidos na simulação, no entanto esses valores são os mesmos para o teste real. Esses resultados são idênticos devido ao mapeamento de laços com trocas de mensagens. Conforme explicado no capítulo anterior, esses laços executam de modo determinístico no modelo mestre/escravo implementado. A comunicação é perfeitamente válida, haja vista que inicialmente o algoritmo distribui as tarefas aos escravos (7 *Sends*), estes recebem sua informação, realizam seus cálculos e retornam a resposta ao mestre (7 *Recvs*). O tempo decorrente das trocas de mensagens pode ser considerado irrelevante neste teste, devido a alta taxa de transmissão de dados associada a inatividade da rede no momento da execução.

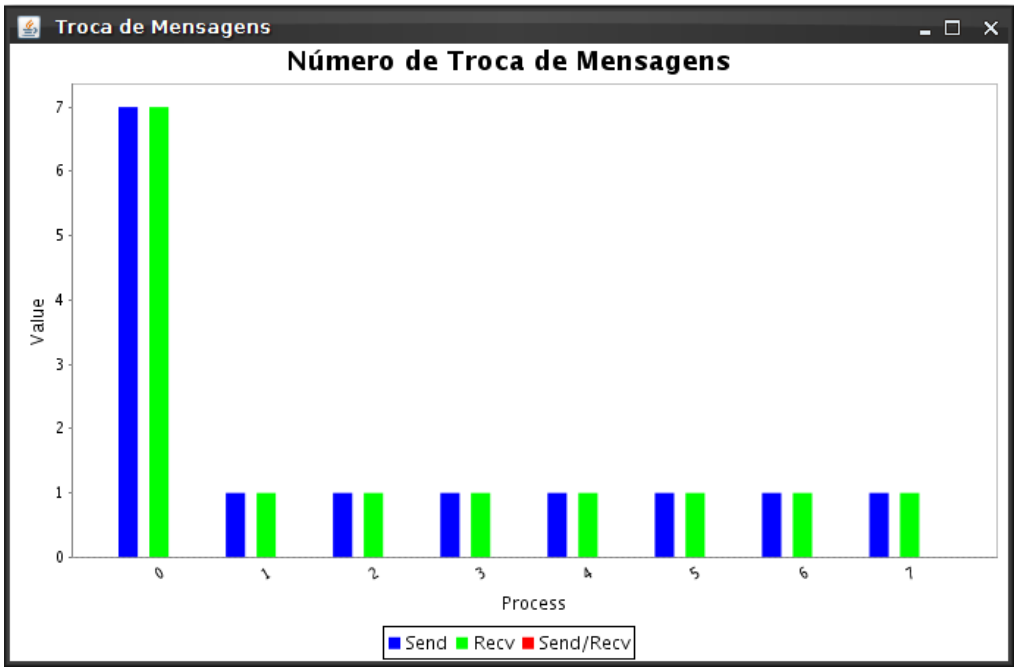


Figura 4.1: Trocas de mensagens entre os processos simulados.

Iterações de laços

A principal vantagem na contagem de iterações de laços está relacionada a identificação de gargalos do sistema. Haja vista que a maior parte do processamento dos programas está em laços de repetição, portanto, esta é uma estatística fundamental para identificação de trechos de códigos a serem otimizados.

Na figura 4.2 são mostradas estatísticas de execução de um laço do programa para cada processo simulado, em um dos cinco testes realizados. Sua execução segue o comportamento da função de distribuição exponencial, em razão dos parâmetros passados como limite e média de iterações.

Pela figura 4.2 notamos a ausência de iterações no processo zero, isso ocorre pois este laço está no corpo do código escravo, portanto, não há nenhuma execução no código mestre.

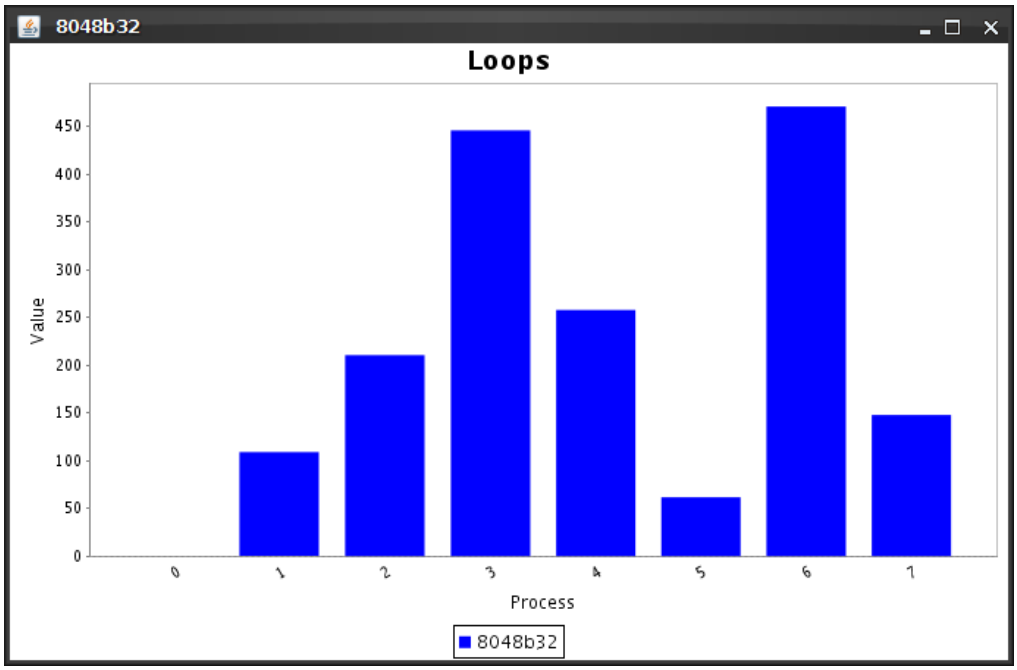


Figura 4.2: Estatísticas sobre a execução de um laço no simulador.

Tempo de execução

O tempo de execução é uma das principais métricas para avaliação da viabilidade de execução do programa no ambiente simulado. Esse valor pode ser obtido pela razão entre os ciclos executados e o *clock* da máquina em questão. A tabela 4.1 mostra a média dos tempos obtidos, em segundos, para cada processo em cinco simulações e testes reais. É suposto que as máquinas simuladas tenham o mesmo *clock* encontrado no *cluster*, e seu valor real obtido na execução paralela.

A precisão desses valores varia de acordo com as iterações dos laços, e estes estão fortemente relacionados a eficiência da função de distribuição de probabilidade exponencial. Os tempos de execução se tornam cada vez mais próximos do real de acordo com os ajustes do limite de iterações e seu valor médio. A eficiência da ferramenta pode ser observada no tempo médio obtido na simulação em relação ao tempo real. Na simulação foi obtido um tempo médio, que envolve todos processos nos cinco testes, de 2.01 segundos. Já o tempo

médio real foi de 1.90 segundos. Isso fornece um erro de aproximadamente 5%, o que é perfeitamente factível em uma simulação.

Estes valores asseguram a eficiência da metodologia, de acordo com o ajuste da função de distribuição de probabilidade exponencial e os parâmetros fornecidos pelo usuário.

Tabela 4.1: Tempo em segundos gastos na simulação e no ambiente real

ID processo	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**t. médio**
t. simulado	4.24	1.79	1.42	2.97	0.05	1.42	2.56	1.65	2.01
t. real	3.11	2.95	2.45	2.09	1.76	1.33	0.93	0.64	1.90

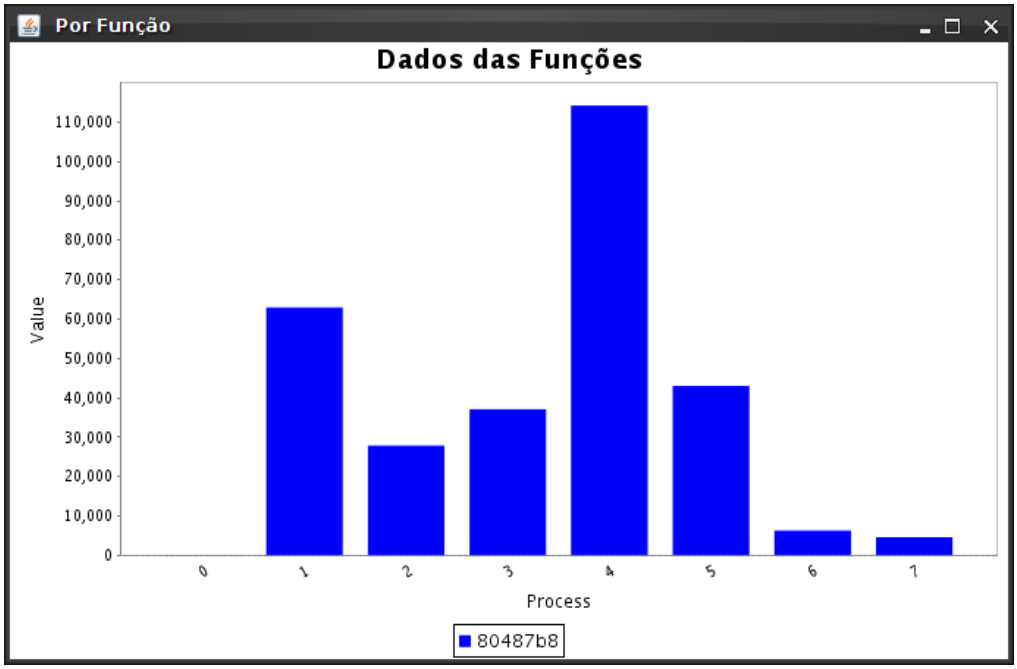
Chamadas de função

Algumas chamadas de função neste programa estão relacionadas aos laços, pois são chamadas para funções que realizam alguns cálculos da integral. A figura 4.3(a) mostra o número de chamadas para a função que calcula uma raiz quadrada em uma iteração da integral. Assim como ocorreu no item anterior, as chamadas para esta função estão no corpo do código escravo, portanto, a ausência de chamadas para o processo zero é correta.

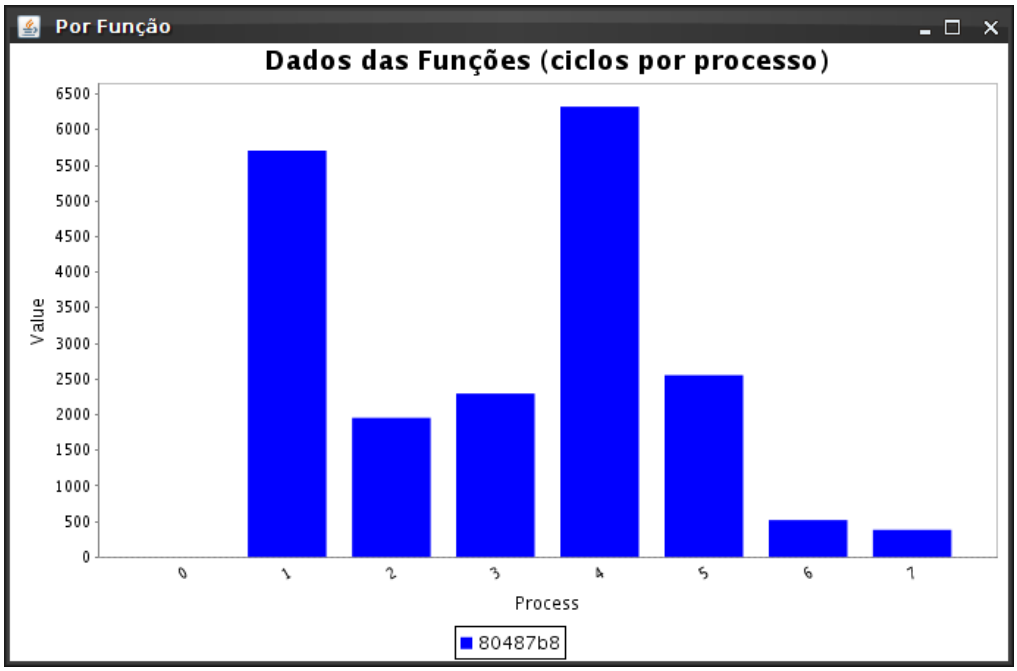
Como é possível perceber, o número de chamadas de função é expressivo, isso garante a qualidade da aproximação da integral e mostra a eficiência do simulador diante de situações extremas.

Outra informação relevante nas chamadas de função está vinculada ao número de ciclos de cpu gastos para execução da função. Esses valores são importantes para identificação das funções que gastam maior parte do tempo de processamento.

A figura 4.3(b) mostra os ciclos de cpu gastos na execução da função mostrada. Com essa informação é possível obter uma média de ciclos gastos por chamada de função e assim estimar o tempo de execução gasto no ambiente simulado.



(a) Número de chamadas para a função.



(b) Ciclos de cpu simulados na função.

Figura 4.3: Estatísticas sobre a execução de uma função.

4.1.2 Múltiplicação de matrizes

Este é um problema clássico em paralelismo. Dada uma matriz MxN de números reais é feita sua multiplicação. O algoritmo divide a matriz original em submatrizes e repassa aos escravos seu cálculo. Estes realizam todo o processamento e armazenam a matriz de resposta em um arquivo, devido a grande quantidade de dados que seria transmitida por uma troca de mensagem. Portanto, o gasto com comunicação, assim como no exemplo anterior, é desprezível.

Foram realizados cinco simulações e testes reais, cada um com 2, 4 e 8 processos em execução. Para todos os testes foi fornecida uma matriz de dimensão 400x400, o que resulta em pelo menos 64000000 operações matemáticas.

Trocas de mensagens

A tabela 4.2 mostra o número de mensagens trocadas durante os testes, esses valores são iguais na simulação e no teste real, por motivos já explicados anteriormente. As trocas de mensagens refletem a operação inicial e final do programa. Por exemplo, com dois processos o processo mestre envia o trabalho ao escravo, este processa o pedido e retorna o resultado (2 *Sends*). O mesmo raciocínio vale para os demais testes.

Tabela 4.2: Trocas de mensagens entre os processos.

Processos	**2**	**4**	**8**
Sends	2	6	14
Recvs	2	6	14

Iterações de laços

Na simulação com dois processos, o número de iterações para os laços envolvidos na multiplicação das matrizes, tanto na simulação quanto no teste real foram iguais. Isso ocorreu devido ao número máximo de iterações ser igual a sua média. Neste caso a simulação foi determinística, pois esse valor é constante.

Nos demais casos o número máximo de iterações é igual ao número de colunas da matriz, comum para todos os processos escravos, e a média de repetições é o número de linhas da submatriz passada pelo processo mestre. Estes ajustes têm papel fundamental no resultado final da simulação, pois apesar das variações no número de iterações entre a simulação e sua real execução, o fator mais relevante é o tempo total de execução. A figura a seguir ilustra a execução de um laço com 16 processos, esta mesma simulação gerou outros resultados discutidos adiante.

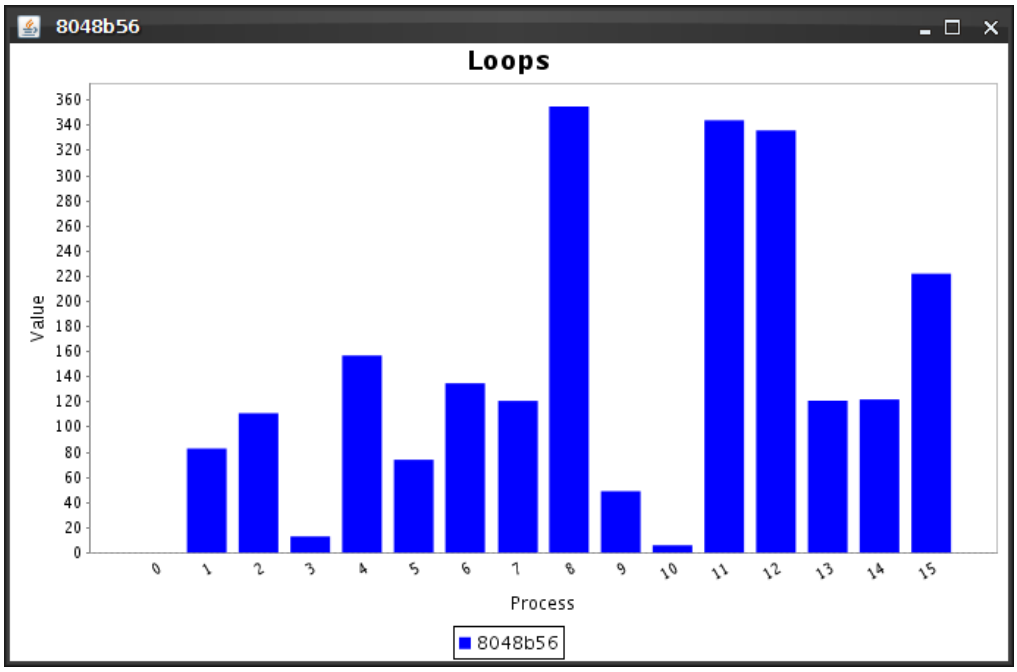


Figura 4.4: Simulação da execução de um loop com 16 processos.

Tempo de execução

A tabela abaixo mostra os tempos médios obtidos na simulação e na execução real em cinco testes. A eficiência da ferramenta pode ser verificada pelo porcentual de erro do tempo médio de execução, ou seja, por volta de 4%. Na figura 4.5 é mostrado o número de ciclos executados em uma das simulações com 2 processos.

Tabela 4.3: Tempos de execução para 2 processos

ID Processo	**0**	**1**	**t. médio**
t. real	0.7322928	0.7319872	0.73214
t. simulado	0.2832161	1.1097839	0.69650

A tabela 4.4 mostra os tempos médios dos processos, em segundos, das execuções nas simulações e *benchmarks*. Os tempos reais foram próximos entre si devido ao número igual de iterações dos laços. Como a execução dos laços na simulação não é determinística, seu comportamento variou, isso pode ser visto na mudança dos tempos de execução simulados.

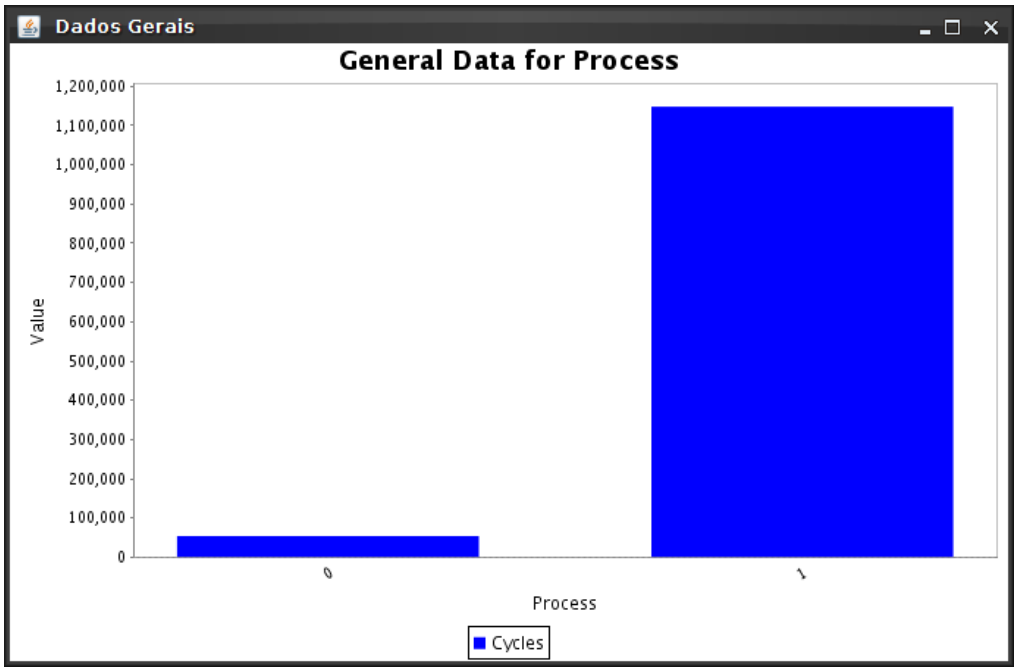


Figura 4.5: Ciclos executados com 2 processos em uma simulação.

Tabela 4.4: Tempos de execução para 4 processos

ID Processo	**0**	**1**	**2**	**3**	**t. médio**
t. real	0.2457	0.2441	0.2408	0.2404	0.2427
t. simulado	0.2721	0.2023	0.3347	0.1929	0.2505

A tabela a seguir mostra os tempos de execução e simulação, em segundos, para 8 processos. Em geral os resultados obtidos foram abaixo do tempo de execução real, isso pode ser explicado pela distribuição de probabilidade utilizada no momento da simulação e aos parâmetros fornecidos. A figura 4.6 mostra os ciclos de execução gastos em uma das cinco simulações executadas, através desses valores são obtidos os seus tempos de execução no ambiente simulado.

Tabela 4.5: Tempos de execução para 8 processos

ID Processo	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**t. médio**
t. real	0.151	0.107	0.103	0.105	0.108	0.104	0.108	0.105	0.11137
t. simulado	0.157	0.087	0.058	0.119	0.096	0.113	0.079	0.114	0.10287

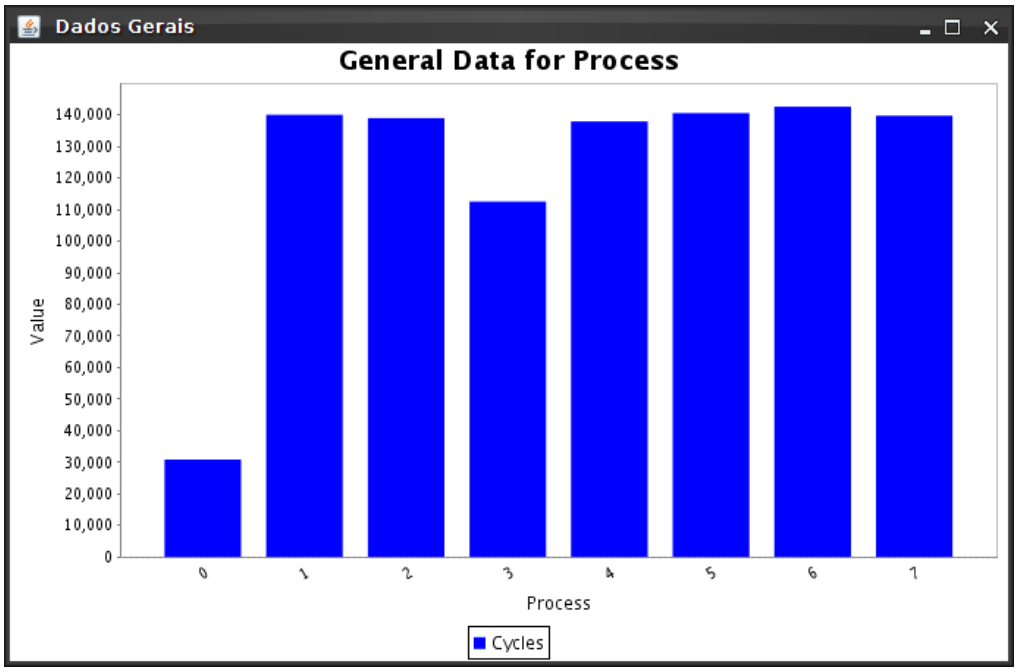


Figura 4.6: Ciclos executados com 8 processos em uma simulação.

Capítulo 5

Conclusões e trabalhos futuros

Este capítulo irá apresentar as conclusões obtidas com o desenvolvimento deste projeto, bem como as experiências acadêmicas, pessoais e profissionais conquistadas para sua elaboração. Em seguida serão discutidas as perspectivas de trabalhos futuros no desenvolvimento desta ferramenta, que foram encontradas durante as fases de desenvolvimento e testes.

5.1 Conclusões

A elaboração deste projeto seguiu o cronograma inicialmente estipulado, para as fases de estudos teóricos, desenvolvimento da ferramenta e testes. A eficiência do método foi demonstrada por testes realizados sob os paradigmas implementados em comparação com programas reais. Os resultados obtidos podem ser aperfeiçoados de acordo com os dados de entrada fornecidos, a precisão das funções de distribuição de probabilidades utilizadas e o número de simulações realizadas.

Outra vantagem do modelo é sua flexibilidade quanto ao ambiente simulado, é possível predizer a execução de um algoritmo em posse apenas de seu código executável e com informações sobre o ambiente real de execução.

Além do desenvolvimento da ferramenta, o projeto trouxe desde benefícios intelectuais à concessão de bolsas de iniciação científica pela FAPESP. Em relação ao conhecimento obtido, é notável o aprendizado da arquitetura de processadores x86, o que envolve uma visão do processador para manipulação das suas instruções. Outro acréscimo está relacionado ao aprendizado da linguagem de programação Java, que é visivelmente importante tanto no meio acadêmico quanto empresarial.

Por fim, o projeto também contribuiu para um estudo aprofundado de programação paralela utilizando a biblioteca MPI. Esta foi utilizada desde o início do projeto, na identificação das comunicações entre os processos, até a etapa de simulação com os testes finais mostrados neste trabalho.

5.2 Perspectivas de trabalhos futuros

Apesar do grande esforço empregado no desenvolvimento deste projeto algumas funções podem ser adicionadas para sua maior abrangência. Essas funções podem ser facilmente adicionadas, haja vista a modularidade concebida no projeto.

- Otimização do grafo de execução gerado, para garantir maior velocidade na simulação do programa. Essa característica deve possibilitar o nível de otimização, de forma que o usuário decida sobre a relação tempo de simulação e precisão dos resultados.
- Desenvolvimento de um desmontador em Java, ou seja, independente de plataforma, que realize engenharia reversa em códigos para diversas arquiteturas de processadores. Isso possibilitaria uma gama maior de simulações, em ambientes heterogêneos.
- Adição de novas funções MPI para simulação de sua execução. Isso envolve um estudo do uso da primitivas MPI em funções mais complexas, sob o ponto de vista do desmonte de códigos executáveis.
- Implementação de módulos de simulação de outros paradigmas além do mestre/escravo e spmd. Isso envolve um estudo da estrutura de comunicação dos códigos reais aplicados em MPI.

Referências Bibliográficas

[1] Amdahl, G.M.; "Valid of the single-processor approach to achieving large scale capabilities"; in *AFIPS Conference Proceedings,* vol. 30, p. 483-485, AFIPS Press, Reston, Va., 1967.

[2] Gustafson, J.L.; "Reevaluating Amdahl's law"; *Communications of the ACM*, vol. 31, n.5, p. 532-533, 1988

[3] Calzarossa, M. and Serezzi, G.; "Workload characterization: a survery";*Proceedings of the IEEE*, vol. 81, n. 8, p. 1136-1150, 1993.

[4] Bradley, D.K. and Larson, J.L.; "A parallelism-based analytic approach to performance evaluation using applicaion programs"; *Proceedings of the IEEE*, vol. 81, n. 8, p. 1126-1135, 1993.

[5] Gustafson, J.L. and Rover, D. and Elbert, S. and Carte, M.; "The design of a scalable, fixed-time computer benchmark"; *Journal of Parallel and Distributed Computing*, vol. 12, p. 388-401, 1991.

[6] Gandra, M.; Drake, J. M.; Gregorio, J. A.; "Performance evaluation of parallel systems by using unbounded generalized stochastic petri nets"; *IEEE Trans. on Software Engineering*, 1992.

[7] Manacero, A. J.; "Predição de Desempenho de Programas Paralelos por Simulação do Grafo de execução"; Tese (Doutorado) – Unicamp , 1997.

[8] Herzog, U.; "Formal description, time and performance analysis"; in T. Harder, H. Wedekind and G. Zimmermann (editors), *Entwurf and Betried Verteilter Systeme*, Berlin, 1990, Springer Verlag, Berlin, IFB 264.

[9] Dongarra, J.J.; "Performance of various computers using standard linear equations software"; Technical Report 23, *Argonne Nat. Lab.*, 1988.

[10] Uniejewski, J.; "SPEC benchmark suite: designed for today's advanced systems"; Technical Report 1, *SPEC Newsletter*, 1989.

[11] Vemuri, R., Mandayam, R. and Meduri, V.; "Performance modelling using PDL"; *IEEE Computer*; p. 44-53, april 1996.

[12] Sakukkai, S.R., Mehra, P. and Block, R.J.; "Automated scalability analysis of message-passing parallel programs"; *IEEE Parallel and Distributed Technology*, vol. 3, n. 4, p. 21-32, 1995.

[13] Pease, D. et alii; "PAWS: a performance evaluation tool for parallel computing systems"; *IEEE Computer*, p. 18-29, jan. 1991.

[14] S.L. Graham, P.B. Kessler, and M.K. McKusick; "Gprof: a call graph execution profiler"; *ACM Sigplan Notices*, 17(6):120-126, 1982.

[15] J.F. Reiser and J.P. Skudlarek. "Program profiling problems and a solution via machine language rewriting"; *ACM Sigplan Notices*, 29(1):37-45, 1994.

[16] Jim Pierce and Trevor N. Mudge; "Idtrace - a tracing tool for i486 simulation"; *In MAS-COTS '94: Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, pages 419–420. IEEE Computer Society, 1994.

[17] J.P. Kitajima and B. Plateau; "Modelling parallel program behaviour in alpes"; *Information and Software Technology*, 36(7):457–464, 1994.

[18] GNU Binutils. Disponível em http://www.gnu.org/software/binutils, último acesso em 17/09/2008.

[19] MPICH. Disponível em http://www.mcs.anl.gov/research/projects/mpich2, último acesso em 30/09/2008.

[20] OpenMPI. Disponível em http://www.open-mpi.org, último acesso em 30/09/2008.

[21] IntelMPI. Disponível em http://www.intel.com/cd/software/products/asmo-na/eng/308295.htm, último acesso 30/09/2008.

[22] HP-MPI. Referência em http://docs.hp.com/en/5992-2330/ch10s03.html, último acesso em 30/09/2008.

[23] Graphviz. Referência em http://www.graphviz.org, último acesso em 13/10/2008.

[24] Zafalon, G.F.D. ; Manacero Jr, A. . Construção de Geradores Independentes de Números Aleatórios para Diferentes Distribuições Probabilísticas. In: Workshop em Computação e Aplicações, 2006, Campo Grande, MS. Anais do XXVI Congresso da Sociedade Brasileira de Computação, 2006. v. cd-rom. p. WC1-WC6

Apêndice A

Arquitetura x86

A primeira etapa no desenvolvimento deste projeto fez uso intenso da arquitetura x86 da Intel, para melhor entendimento desta fase serão discutidos alguns detalhes relevantes desta família de processadores, em destaque para os dispositivos de 32 bits.

Essa arquitetura envolve características comuns para dispositivos de 16, 32 e 64 bits da Intel, isso possibilita, por exemplo, a execução de um código compilado para um processador de 16 bits em um modelo de 64 bits.

A.1 Formato das instruções

As instruções de processadores da arquitetura Intel seguem o formato mostrado na figura A.1, extraída da documentação oficial do fabricante. Esse padrão utiliza uma instrução opcional de prefixo de 1 byte, um espaço para o *opcode*, um especificador de endereçamento, caso necessário, consistindo do byte ModR/M e as vezes do campo SIB (*Scale-Index-Base*), em necessário o deslocamento e o campo de dados.

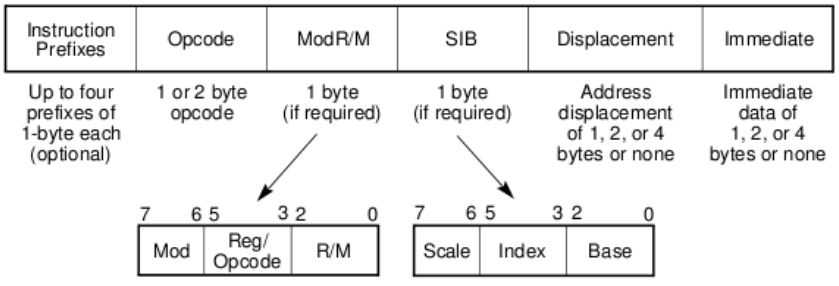


Figura A.1: Formato da instrução na arquitetura Intel.

Prefixos de instruções

São divididos em quatro grupos, cada qual com seu conjunto de códigos permitidos. Os grupos estão divididos entre prefixos de *lock* e repetição, sobrescrita segmento, operador e de endereço. Para cada instrução um prefixo pode ser usado desses grupos e adicionados em qualquer ordem.

Opcode

O *opcode* primário é de 1 ou 2 *bytes*, um *opcode* adicional de 3 *bytes* pode ser codificado no *byte* ModR/M. Codificações menores podem ser definidas usando apenas o *opcode* primário. Esses campos definem a operação da instrução, o seu deslocamento e outras particularidades da instrução.

Bytes ModR/M e SIB

A maioria das instruções com referência a um operando na memória tem um *byte* que especifica o *opcode* primário chamado ModR/M. O *byte* ModR/M contém três campos de informação:

- **Mod**: combina com r/m para formar 32 valores possíveis, oito registradores e 24 modos de endereçamento
- **Reg/Opcode**: especifica um registrador ou 3 *bits* de informação de *opcode*
- **R/M**: pode especificar um registrador como um operando ou pode ser combinado com o campo *mod* para codificar um modo de endereçamento

Algumas codificações do *byte* ModR/M precisam de um segundo *byte* de endereçamento, o *byte* SIB. Os modos de endereçamento *base-plus-index* e *scale-plus-index* precisam deste *byte*. Este campo inclui as seguintes informações:

- **Scale:** especifica o fator de escala
- **Index:** especifica o número do registrador ou seu índice
- **Base:** especifica o número do registrador do registrador base

Campos de deslocamento e imediato

Alguns endereçamentos incluem um deslocamento após o campo ModR/M ou SIB, caso seja necessário esse descolamento pode ser de 1, 2 ou 4 *bytes*. Se a instrução especifica o campo imediato, o operando sempre segue qualquer *byte* de deslocamento.

Apêndice B

MPI

Uma das etapas do desenvolvimento deste projeto envolveu o conhecimento de programação paralela utilizando a biblioteca MPI. A título de detalhamento, serão mostrados a seguir a estrutura geral de programas nesse ambiente e as principais funções, utilizando a linguagem C.

B.1 Estrutura básica

Como mencionando anteriormente, MPI é uma biblioteca que pode ser usada em linguagens de programação como C/C++, Fortran, etc. A primeira instrução necessária para sua compilação é a diretiva de inclusão de biblioteca, mostrada a seguir.

```
#include "mpi.h"
```

Este arquivo, mpi.h, inclui as definições e declarações necessárias para compilação de qualquer programa MPI. Antes de executar qualquer função MPI é necessário invocar a função MPI_Init, e isso deve ocorrer apenas uma vez no código. Esta função recebe como parâmetro os valores recebidos pela função *main* do programa. Após o uso das funções MPI, a função MPI_Finalize deve ser chamada para liberação dos recursos utilizados pelo ambiente. Um programa típico MPI tem a seguinte estrutura:

```
#include "mpi.h"
int main( int argc, char* argv[ ] ){
        ...
        MPI_Init(&argc, &argv);
        ... // chamadas para funções MPI
        MPI_Finalize();
        ...
        return 0;
}
```

B.1.1 Funções

As principais funções utilizadas num ambiente paralelo estão relacionadas a trocas de mensagens, descoberta do identificador do processo e número de processos em execução paralela. A seguir as principais funções.

MPI_Comm_rank

O primeiro parâmetro passado é o comunicador. Este é relacionado aos processos que podem trocar mensagens entre si. O comunicador básico é o MPI_COMM_WORLD. O segundo parâmetro passado recebe o *rank* do processo. No caso de N processos em execução, esse valor pode variar entre 0 e N-1.

```
...
int my_rank;
...
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

MPI_Comm_size

Novamente o primeiro parâmetro passado é o comunicador. Já o segundo parâmetro irá armazenar o número de processos em execução paralela.

```
...
int size;
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI_Send

Função envia uma porção de dados a um processo de *rank* específico. Seus parâmetros são: *buffer* a ser enviado, tamanho do *buffer*, tipo de mensagem (MPI_INT, MPI_FLOAT, MPI_CHAR, etc.), *rank* do processo de destino, uma *tag* que indica o canal de comunicação e o comunicador. A assinatura da função pode ser verificada abaixo:

```
int MPI_Send( void *message,
              int count,
              MPI_Datatype,
              int dest,
              int tag,
              MPI_Comm comm )
```

MPI_Recv

Função recebe uma porção de dados de um processo específico. Seus parâmetros são idênticos a função de envio, exceto pelo fato que agora tem-se a origem da mensagem e não o destino. Outra alteração diz respeito ao status da mensagem recebida, com esta variável é possível obter informações sobre o resultado da comunicação. Uma observação importante é sobre o seu estado, quando a execução atingir esta função, o resto do código só executará após a chegada da mensagem, ou seja, esta função é bloqueante.

```
int MPI_Recv( void* message,
              int count,
              MPI_Datatype datatype,
              int source,
              int tag,
              MPI_Comm comm,
              MPI_Status* status )
```

MPI_Sendrecv

Função realiza ambos MPI_Send e MPI_Recv. Com uso dessa função código fica mais claro e evita duplicação de funções. Sua execução também é bloqueante.

```
int MPI_Sendrecv( void* send_buff,
                  int send_count,
                  MPI_Datatype send_type,
                  int dest,
                  int send_tag,
                  void* recv_buff,
                  int recv_count,
                  int recv_type,
                  int source,
                  int recv_tag,
                  MPI_Comm comm,
                  MPI_Status* status )
```

B.1.2 Exemplo

O exemplo a seguir foi utilizado no primeiro teste do capítulo 5. Como mencionado anteriormente ele calcula a integral de uma função definida. Nos testes realizados o intervalo da integral variou a fim de avaliar as iterações e o tempo de execução.

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

// outras funções implementadas

int main(int argc, char** argv){
double points[2], a=0.0, b=400.0, h, response = 0.0, total = 0.0;
MPI_Status status;
int i, my_rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(my_rank == 0){
       h = (b-a)/(size-1);
       points[0] = a;
       points[1] = a + h;

       for(i=1; i<size; i++){
          MPI_Send(points, 2, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
          points[0] = points[1];
          points[1] = points[1] + h;
       }
       for(i=1; i<size; i++){
          MPI_Recv(&response, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,&status);
          total += response;
        }
        printf("Integral is = [ %lf ]\n",total);
    } else {
       MPI_Recv(points, 2, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
       response = calculateIntegral(points[0], points[1]);
       MPI_Send(&response, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```