Daniel Cardoso de Leão

Implementação do paradigma SPMD em ferramenta para avaliação de desempenho de programas paralelos

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
Ano 2011

Daniel Cardoso de Leão

Implementação do paradigma SPMD em ferramenta para avaliação de desempenho de programas paralelos

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 Jr.

São José do Rio Preto
Ano 2011

Daniel Cardoso de Leão

Implementação do paradigma SPMD em ferramenta para avaliação de desempenho de programas paralelos

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.

_____
Prof. Dr. Aleardo Manacero Jr.

_____
Daniel Cardoso de Leão

Banca Examinadora:
Prof. Dr. Alex Sandro Roschildt Pinto
Prof. Dr. Norian Marranghello

São José do Rio Preto
Ano 2011

À minha família e amigos.

Agradecimentos

Agradeço primeiramente a Deus pela vida que me concedeu e por estar comigo nos momentos mais difíceis.

Aos meus pais, pelo amor, apoio, motivação, confiança e pelo esforço realizado para que eu chegasse até aqui.

Ao meu orientador, Prof. Dr. Aleardo Manacero Jr., por ter me confiado este projeto, pela sua orientação e também pelo apoio ao longo do curso.

A todos os amigos e pessoas que conviveram comigo durante esse período.

Resumo

Sistemas computacionais de alto desempenho possuem altos custos associados ao *hardware*. Isso faz com que os programadores tentem obter o maior aproveitamento possível dos recursos disponíveis, concentrando esforços na otimização de seus programas.

Neste contexto a análise de desempenho possui um 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. Essas ferramentas podem ou fazer aproximações do ambiente paralelo, proporcionando um custo reduzido, embora diminuam a precisão dos resultados obtidos, ou fazer uso do sistema real para suas medições, o que nem sempre é viável devido ao alto custo associado.

Este trabalho complementa a implementação de uma ferramenta de medição de desempenho baseada em simulação, o Grasptool (*GRAph Simulator Performance TOOL*), acrescentando o paradigma SPMD aos modelos utilizados para simular o comportamento do programa em uma execução real. No Grasptool a simulação é realizada utilizando as informações do grafo de execução obtido a partir da reescrita do código executável, compilado para a arquitetura x86, das trocas de mensagens MPI e do ambiente de execução, tais como número de processos, velocidade de processamento e largura de banda.

Abstract

High-performance computing systems have high costs associated with hardware. This makes programmers try to get the best possible use of available resources by focusing efforts on optimizing your programs.

In this context the performance analysis has a key role in cost reduction. There are several performance analysis tools that help the designer to identify critical points of the system. These tools either make approximations about the parallel environment, which provides a reduced cost, but also decreases the accuracy of results, or make use of the real system for their measurements, that is not always feasible due to the high associated cost.

This work complements the implementation of a performance measurement tool based on simulation, the Grasptool (*GRAph Simulator Performance TOOL*), adding the SPMD paradigm to the models used to simulate the behavior of the program in an actual execution. In Grasptool the simulation is performed using information from the execution graph obtained from the rewriting of the executable code, compiled for the x86 architecture, the MPI message exchanges and the execution environment, such as number of processes, processing speed and width bandwidth.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1 - Introdução

A eficiência de um sistema computacional está intrinsicamente relacionada ao seu tempo de execução, o qual pode ser reduzido com a utilização de algoritmos mais otimizados ou com aumento do poder computacional do ambiente. A segunda alternativa, embora mais simples, implica em custos, como aquisição de novos equipamentos e contratação de pessoal qualificado para sua manutenção. Além disso, se o sistema não utiliza todo seu poder computacional isso implica em um investimento desnecessário.

A utilização do sistema de modo a obter o maior aproveitamento possível é um dos objetivos da computação de alto desempenho, tendo em vista que o custo de novos equipamentos é muito elevado. Desta forma surge a necessidade do uso de ferramentas para predição de desempenho de sistemas, de modo a avaliar a viabilidade da 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.

O Grasptool é uma ferramenta para predição baseada na simulação de grafos obtidos a partir do código executável do programa a ser medido.

1.1 Objetivo

O objetivo desse trabalho é, fundamentalmente, a implementação do paradigma SPMD (*Single Program – Multiple Data*) entre os modelos de simulação

da ferramenta de avaliação do Grasptool. Esse modelo, assim como o modelo mestre/escravo implementado, irá utilizar as informações do grafo de execução gerado pela ferramenta para fazer a simulação de um programa implementado no paradigma SPMD.

1.2 Organização da monografia

No capítulo dois é apresentada a base teórica que fundamenta o projeto, apresentando as diferentes metodologias para análise de desempenho em sistemas paralelos. Além disso, apresenta o embasamento formal da metodologia utilizada.

Em seguida, no capítulo três, é apresentada a ferramenta *Grasptool* e detalhes de implementação do modelo, incluindo pseudocódigos. Já no capítulo quatro são descritos os testes e resultados que validam a eficiência da ferramenta proposta. Finalmente, no capítulo cinco, são expostas as conclusões sobre o projeto.

Capítulo 2 - Fundamentação Teórica

O desempenho de sistemas computacionais torna-se uma questão importante à medida que cresce a necessidade do usuário em obter resultados de forma cada vez mais rápida, sem desperdiçar recursos computacionais e a utilidade da informação após seu processamento. A análise de desempenho oferece ferramentas para avaliação de sistemas, identificando se os recursos estão sendo utilizados de forma ótima ou então localizando seus gargalos. Estas ferramentas podem ser classificadas em dois grupos segundo a forma de atuação: 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 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 Manacero [5], que é a base deste projeto.

2.1 Medidas de Desempenho

A escolha do conjunto de medidas de desempenho deve ser feita levando em consideração o sistema que será avaliado. Uma maneira de preparar esse conjunto é

listar os serviços oferecidos pelo sistema. Para cada requisição de serviço feita pelo sistema, há um número possível de saídas. Essas saídas podem ser classificadas de acordo com a figura 2.1, na qual se pode ver que o sistema pode realizar a requisição do serviço corretamente, incorretamente ou recusar a realização do serviço. As medidas associadas com esses três tipos de saída, com sucesso, com erro e indisponível são também chamadas de medidas de velocidade, confiabilidade e disponibilidade. Muitos sistemas oferecem mais de um tipo de serviço e, com isso, o número de medidas cresce proporcionalmente.

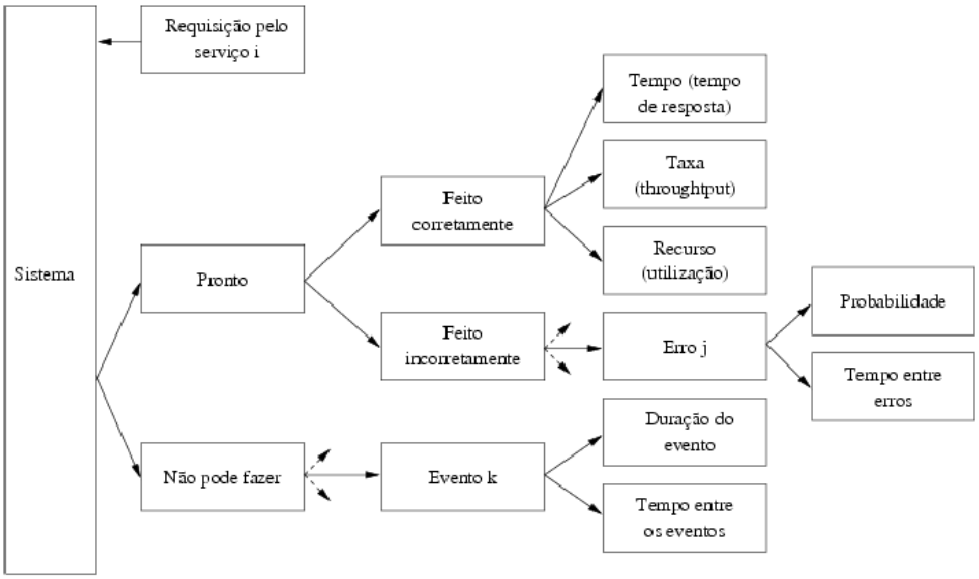


Figura 2.1: Saídas das requisições do sistema. [28]

É muito importante escolher corretamente as medidas para avaliar o desempenho do sistema. Depois de selecionar um número razoável de medidas importantes, deve-se utilizar algumas considerações para selecionar um subconjunto dessas medidas a fim de se obter: variabilidade baixa, ausência de redundância e integridade. Por exemplo, se duas medidas fornecem essencialmente a mesma informação, é mais viável que se considere apenas uma para o estudo do desempenho do sistema.

2.1.1 Medidas mais utilizadas

Nesta seção serão apresentadas algumas das medidas de desempenho mais utilizadas, deixando claro que estas medidas não são as únicas.

- **Tempo de resposta:** É definido como o tempo que o sistema demora a responder a partir do momento que o usuário requisitou determinado serviço. Esta é uma versão bastante simplista do que pode ser tempo de resposta. Em outro caso, pode-se calcular o tempo de resposta considerando o que o usuário gasta para digitar a requisição pelo serviço e o intervalo de tempo que o sistema gasta para enviar a resposta ao usuário.

- ***Throughput*:** É definido como a taxa de requisições que o sistema pode atender em uma determinada unidade de tempo. Para UCP's o *throughput* é medido em milhões de instruções por segundo (MIPS) ou milhões de operações de ponto flutuante por segundo (MFLOPS). Para redes de computadores o *throughput* é medido em pacotes por segundo (pps) ou em bits por segundo (bps).

  O maior *throughput* alcançável em um sistema é denominada eficiência. Por exemplo, se em uma rede de 10 Mbps (megabits por segundo) o maior *throughput* alcançável é de 5 Mpbs, então a eficiência dessa rede é de 50%.

- **Utilização:** É medida como sendo a fração do tempo em que os recursos ficam ocupados atendendo as requisições dos usuários.

- **Confiabilidade:** É medida como sendo a probabilidade de que ocorram erros no sistema ou pelo tempo entre a ocorrência de um erro e outro. Nesse último caso, é denominada como segundos livres de erro (*error-free seconds*).

- **Disponibilidade:** É definida como o intervalo de tempo em que o sistema está disponível para atender as requisições dos usuários.

2.1.2 Medidas de desempenho em sistemas paralelos

Em sistemas paralelos além das medidas já citadas são introduzidas novas medidas, particulares ao ambiente de paralelismo. A seguir será apresentada a definição de uma dessas medidas.

- ***Speedup*:** é a medida mais importante em sistemas paralelos. Ela verifica o ganho de velocidade de um programa ao ser executado em paralelo. Uma das abordagens para medição do *speedup* é definida pela Lei de Amdahl [26], que relaciona o ganho de velocidade ao desempenho em paralelo e sequencial do programa, denotada pela equação 2.1 onde s é a porção sequencial do código e p sua porção paralela, portanto s + p = 1 e N é o número de processadores em paralelo.

$$\text{speedup} = \frac{(s + p)}{(s + \frac{p}{N})} \qquad (2.1)$$

A equação 2.1 é válida somente quando o tamanho do problema não varia com o número de processadores. Sendo conhecida como *speedup* para tamanho fixo. Outra abordagem, apresentada por Gustafson[27], considera intervalos de tempo fixos e não o tamanho do problema, para isso o tamanho do problema aumenta de acordo com a capacidade das máquinas. Assim, a equação 2.2 relaciona o ganho de velocidade em função de sua execução sequencial.

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

2.2 Técnicas para avaliação de 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 no *hardware* dos equipamentos, é possível capturar os eventos do sistema. Técnica custosa devido à necessidade de todo o ambiente de análise, dos equipamentos 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 ligeiramente imprecisos devido a inserção desses códigos.

- **Modificação do código objeto:** insere comandos no código objeto, possuindo os mesmos problemas de precisão da modificação do código fonte.

- **Modificação do código executável:** insere comandos no código executável, apresentando 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 [10], dpat e jprof [11].

- **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 que são inseridas no código do programa.

- **Extração de traços dos eventos:** também conhecida como *event tracing*, presente em ferramentas como IDTrace [12] e ALPES [13], 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. Outras técnicas, como modelagem analítica ou simulação, são menos custosas e os resultados obtidos variam de acordo com sua modelagem.

2.3 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.

2.3.1 Métodos Analíticos

Este método emprega tanto o modelo de redes quanto o modelo de grafos, em que são definidos os estados do sistema durante a execução do programa. Alguns

exemplos dessa categoria são: o modelo de rede de Petri estocástica generalizada (GSPN) [19, 20], métodos baseado em teoria de filas [21], métodos baseados em cadeia de Markov [22, 23] e modelos determinísticos [24].

Basicamente os métodos analíticos utilizam um conjunto de equações e funções que determinam o comportamento do sistema, incluindo todos os fatores que influenciam no comportamento do sistema como parâmetros do modelo. Isso permite que, possa ser obtida uma avaliação mais precisa do desempenho.

A modelagem correta desses valores permite uma análise bem ampla e aprofundada de características sobre o sistema e ainda permite criar relacionamentos entre os diversos parâmetros. A modelagem analítica possui um menor custo de execução quando comparada com as outras técnicas de avaliação de desempenho e os resultados obtidos através do modelo analítico podem ser validados através de comparações com testes reais executados a partir do próprio programa. Porém, criar um modelo analítico para um problema pode ser uma tarefa custosa, pois geralmente os programas são grandes e complexos, principalmente em sistemas paralelos.

2.3.2 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. São métodos caros, pois necessitam da máquina real para a obtenção dos resultados. Sua estratégia é baseada na execução do programa na máquina alvo e a coleta das métricas para a análise de desempenho do sistema. Este método é frequentemente utilizado para avaliar o desempenho da máquina, independentemente do programa a ser executado nela.

Alguns problemas podem ser encontrados para a elaboração de um *benchmark*, como caracterização da carga aplicada ao sistema computacional, Calzarossa e Serezzi [4], a estruturação de procedimentos para fazer as medições [1] e também o projeto de programas que executam testes sobre a capacidade real de escalabilidade do sistema [2]. Os resultados obtidos através deste método poderiam ser dados em tempo gasto para executar a tarefa, contudo, dado a heterogeneidade

das aplicações, estas medidas são convertidas em unidades relativas como, por exemplo, Mips ou Flops.

Em um programa específico de usuário, a medida de desempenho deve 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 seu desempenho analisado 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.

2.3.3 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, em simuladores, isto é variável devido às condições de sua simulação e dos 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 se deseja fazer a análise de novos programas, quando 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 usados como ferramentas de análise de desempenho encontrados são PDL [25], Axe [16] e PAWS [17], Rsim [18] e Simics [19].

2.4 Predição do desempenho através da simulação

Esta seção irá apresentar uma breve descrição da técnica proposta em [5], que consiste na análise de desempenho de programas paralelos através de simulação. Esta técnica gera um grafo de execução a partir de um código executável, o que permite a obtenção de dados de uma forma não invasiva.

2.4.1 Descrição da Metodologia

O fundamento básico deste método é a "metodologia de três passos" de Herzog [6] 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 ultimo um modelo de interação entre os dois primeiros no momento da execução.

Na técnica proposta em [5] 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 ultimo, 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.

2.4.2 Componentes para implementação do método

O modelo proposto em [5] 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. Finalmente, com as informações do ambiente de simulação, o grafo é simulado. Esses módulos serão detalhados 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. Para melhor modelar um programa os vértices podem ser diferenciados por determinadas características. A seguir serão descritas cinco categorias básicas de vértices definidas para o modelo.

- **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, representam os vértices sem desvios condicionais.

- **Decisão:** representam os desvios condicionais, possuem uma aresta de entrada e duas ou mais arestas de saída.

- **Agrupamento:** possuem duas ou mais arestas de entrada e uma aresta de saída representando o fim de um desvio condicional, onde são agrupados os caminhos criados no seu início.

- **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 adicionalmente atribuídas aos vértices:

- **Vértice de Execução:** representa 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 ou de agrupamento.

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

Otimização do grafo de execução

Não é possível 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**

Pode-se 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 2.2 ilustra o procedimento mencionado acima.

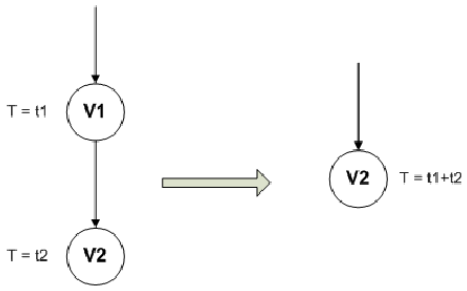
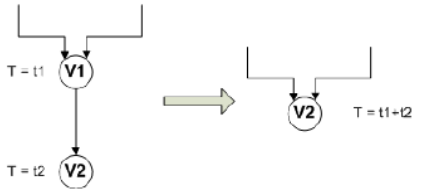


Figura 2.2: Aglutinação de vértices passagem.

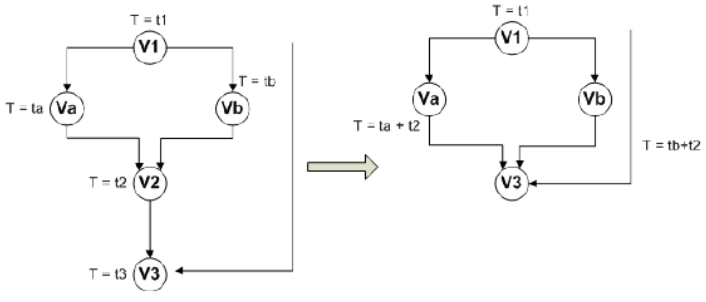
2. Aglutinação de vértices de agrupamento

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 2.3(a).

Também é possível reduzir um grafo mesmo quando mais de uma aresta incide sobre o nó de origem, como ilustrado na figura 2.3(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 2.3: 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 de comunicação.

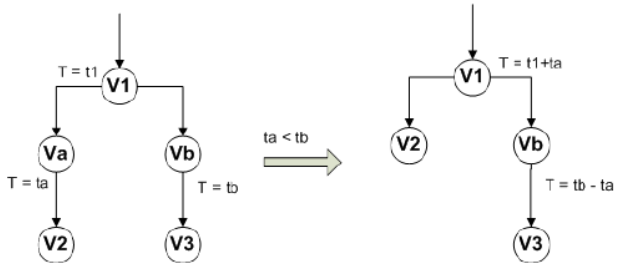


Figura 2.4: Redução de vértices comuns.

Como é mostrado na figura 2.4, 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 é o módulo final do método e 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.

- **Estratégia de operação**

  A figura 2.5 mostra o algoritmo utilizado pelo simulador, cujos passos são descritos em seguida.

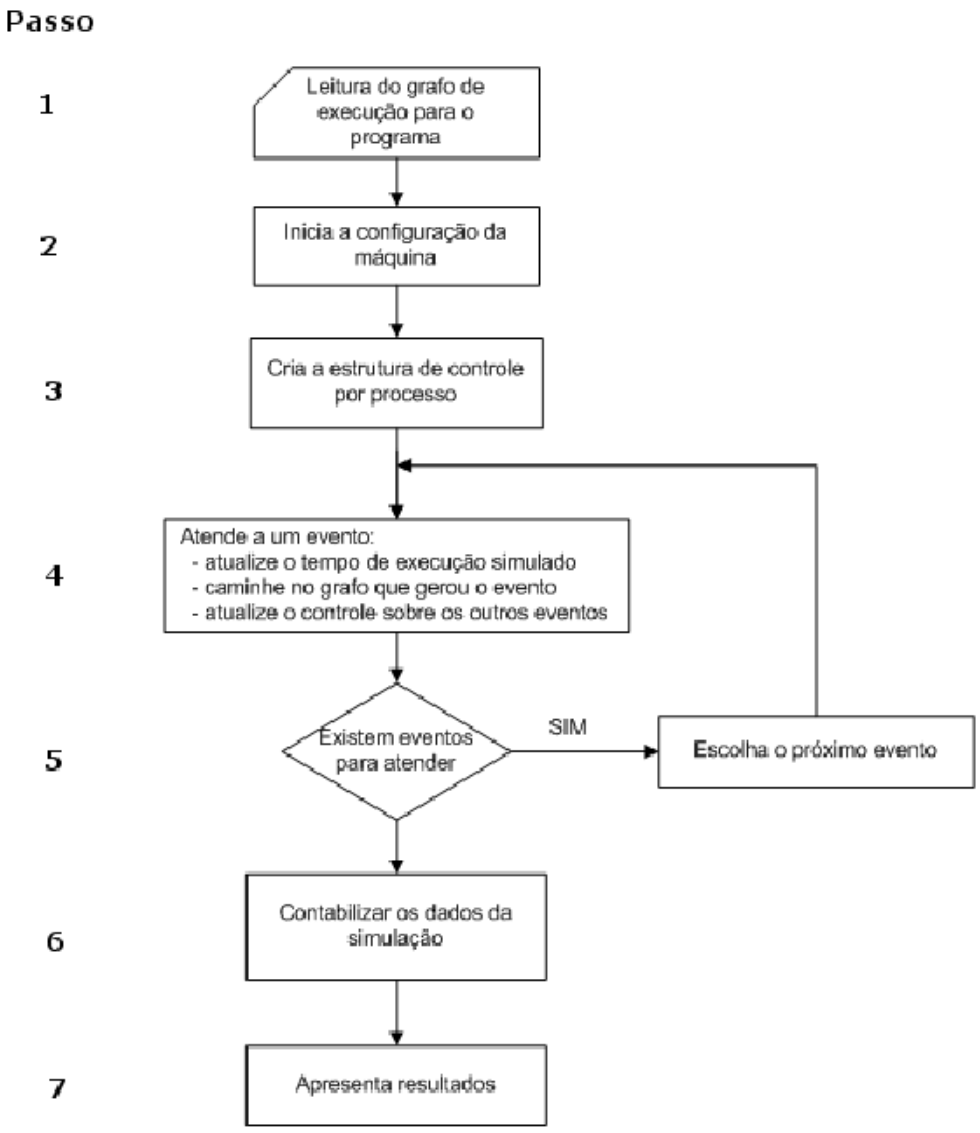


Figura 2.5: Algoritmo de Simulação. [29]

- **Passo 1**

  Trata da leitura do grafo de execução, o qual pode ser de um único programa do tipo SPMD ou múltiplos programas do tipo mestre-escravo.

- **Passo 2**

  Neste passo obtém-se a configuração da máquina e do ambiente de execução como descrito na subseção 2.4.1. Estas informações têm grande

utilidade para a composição dos modelos de máquina e interação programa-máquina dentro da metodologia de três passos de Herzog.

- **Passo 3**

  Após os passos 1 e 2 serem realizados, implementa-se uma estrutura de controle para cada um dos processos paralelos que serão simulados. 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.

- **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.

- **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 à 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.

- **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.

- **Passo7**

  Este passo se refere a uma interface entre o simulador e o usuário onde serão apresentadas as medidas coletadas pelo simulador.

2.5 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 e desvantagens. Finalmente, foi exposto o método proposto em [5] através da predição de desempenho por meio da simulação do grafo de execução.

Atualmente existe um protótipo de ferramenta, implementado em C, dirigido à arquitetura de processadores MIPS e voltada para a simulação de programas paralelos em PVM e uma implementação em Java deste método utilizando a plataforma x86 e voltado para predição de desempenho em um ambiente MPI. No próximo capítulo será abordada a implementação do paradigma SPMD no protótipo em Java.

Capítulo 3 - Detalhamento e desenvolvimento do projeto

Neste capítulo é feita a descrição do projeto executado, iniciando pelos conceitos que guiaram a adoção das tecnologias usadas no projeto. Detalhes da especificação e sua implementação completam o capítulo.

3.1 Conceitos Básicos

A seguir serão apresentados alguns conceitos sobre as tecnologias utilizadas ao longo deste projeto, bem como razões para 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 Por que x86

No final dos anos 70 a Intel© desenvolveu uma arquitetura de processadores que se tornou popular em computadores pessoais ao longo dos anos, os primeiros modelos foram nomeados com a terminação 86 (8086, 80186, 80286, 80386), surgindo assim 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 embutido em linguagens largamente utilizadas não foi necessário desenvolver uma nova linguagem, e consequentemente um novo compilador, para aplicações paralelas.

O sucesso deste padrão se deve à 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 portabilidade desses programas a uma nova linguagem. Basta adaptar os trechos de comunicação dos códigos legados à biblioteca de MPI.

Existem diversas implementações do padrão MPI. Entre as mais comuns estão MPICH[7] e OpenMPI[8], além de diversos esforços de desenvolvimento comerciais como IntelMPI[9] e HP-MPI[3].

3.1.3 Java

A linguagem de programação Java foi escolhida devido à portabilidade que ela oferece aos programas. Ao compilar um programa em Java são gerados *bytecodes* e não códigos binários nativos do processador, como acontece em outras linguagens. Esses *bytecodes* são interpretados por uma Java Virtual Machine (JVM), que os traduzem para a arquitetura onde o programa será executado. Dessa forma os programas podem funcionar em qualquer plataforma de hardware e software que possua uma versão da JVM, tornando essas aplicações independentes da plataforma onde executam.

Além das características que uma linguagem orientada a objetos proporciona, como herança, encapsulamento e polimorfismo, Java fornece 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, sendo isso feito automaticamente pela JVM quando um objeto não é referenciado. Devido a essas e outras características torna-se mais simples para o programador desenvolver projetos mais robustos e confiáveis.

3.1.4 *Disassembler*

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

De 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 [14] 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

No capítulo anterior foram apresentadas as três etapas básicas da geração do grafo de execução a partir de um código executável. A seguir serão apresentados os detalhes da implementação de cada uma dessas etapas.

3.2.1 Leitura do código executável

Esta é a mais simples das três etapas. É utilizado um *disassembler* no código executável e ao término é gerado um arquivo único com o nome das funções contidas no código binário, o endereço lógico das instruções, seus *opcodes* e os mnemônicos. A figura 3.1 mostra um exemplo do trecho de um arquivo gerado pelo objdump.

```
00000000004028f0 <main>:
  4028f0:       48 83 ec 08             sub    $0x8,%rsp
  4028f4:       0f 1f 40 00             nopl   0x0(%rax)
  4028f8:       31 c0                   xor    %eax,%eax
  4028fa:       e8 55 e2 ff ff          callq  400b54 <yylex>
  4028ff:       85 c0                   test   %eax,%eax
  402901:       75 f5                   jne    4028f8 <main+0x8>
  402903:       48 83 c4 08             add    $0x8,%rsp
```

Figura 3.1: Trecho do arquivo gerado pelo objdump.

Em seguida as informações de cada função são armazenadas em arquivos separados nomeados com a data atual (seguindo a convenção de datas em Java) seguido do endereço lógico da função e com a extensão ".gttxt" (GraspTool TeXT). A adição da data é importante para impedir que os arquivos sejam sobrescritos caso haja duas instâncias do gerador sendo executadas simultaneamente, enquanto que 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 para utilização em fases posteriores.

3.2.2 Interpretação das instruções de máquina

Com as funções separadas em arquivos é possível analisar cada uma isoladamente, facilitando a reutilização de trechos analisados anteriormente. Os arquivos são lidos linha por linha, extraindo os *opcodes* que 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 sua execução e a classifica em um dos tipos básicos definidos 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 dividida 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 dividida 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.

Para algumas instruções são necessários dados relevantes sobre sua operação. No caso de uma instrução de salto é preciso calcular o endereço de salto, contido no código da 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. O cálculo do endereço de salto é feito invertendo os códigos que representam o endereço e verificando se o primeiro número é maior ou igual a 128, ou seja, se o primeiro bit está ativado, em caso afirmativo o salto é para trás, neste caso é necessário fazer uma negação desse endereço para obter o seu deslocamento. Caso contrário, ou seja, no salto 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.

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 sequência 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 e, consequentemente, o consumo de memória, além de

acelerar o processo de simulação, devido a diminuição de vértices a serem visitados. Este agrupamento é possível, pois as instruções de execução em sequência formam um bloco de código que sempre será executado. O agrupamento é feito até que uma instrução diferente de execução seja encontrada, o que pode ser um salto, uma chamada/retorno de função ou de comunicação. Quando isso ocorre são gerados dois vértices: o primeiro para as instruções de execução acumuladas e o segundo para a instrução diferente encontrada.

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*, 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* que mapeia um endereço para um vértice) se o vértice v1 existe, em caso afirmativo basta v2 referenciar v1, caso contrário este endereço está no meio de v1, pois os demais tipos de vértices anteriores já foram criados. Portanto será necessário dividir v1 em dois v1,1 será a primeira metade contendo os endereços anteriores ao endereço referenciado e estará ligado a segunda metade v1,2 contendo os endereços restantes, no cálculo de ciclos de v1,1 e v1,2 é feita uma proporção de acordo com a quantidade de endereços de v1,1 e v1,2 , de forma a distribuir os ciclos de v1. Feito isso o vértice de desvio v2 terá uma referência a v1,2. Esse processo pode ser visto na figura 3.2.

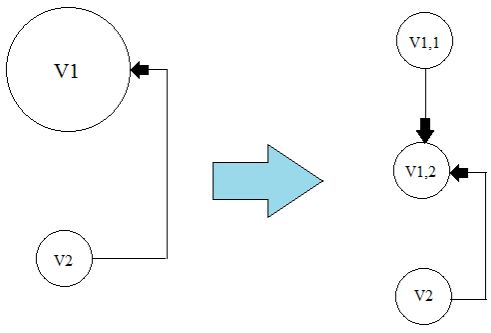


Figura 3.2: Divisão de um vértice de execução.

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, consequentemente, referenciado. A estrutura utilizada para o armazenamento de um vértice é constituída por 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 conectividade 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 de uma estrutura contendo dados como nome da função, seus endereços inicial/final da função e uma referência para o primeiro vértice da função. Esta lista de grafos será a estrutura mais utilizada durante a simulação do grafo de execução.

3.3 O Simulador

Nesta seção será apresentado o 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 proposta por Manacero [5]. A seguir tem-se a descrição do simulador e suas funcionalidades, a figura 3.3 mostra um diagrama de estados do simulador.

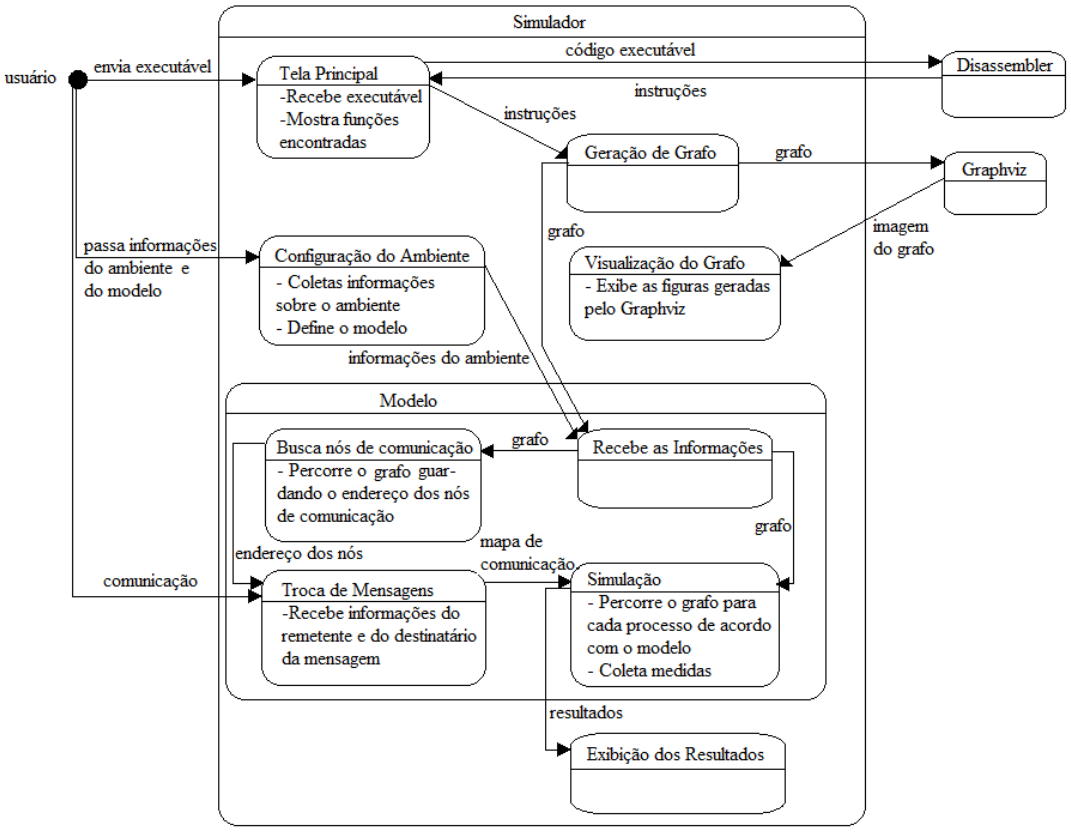


Figura 3.3: Diagrama de estados do simulador.

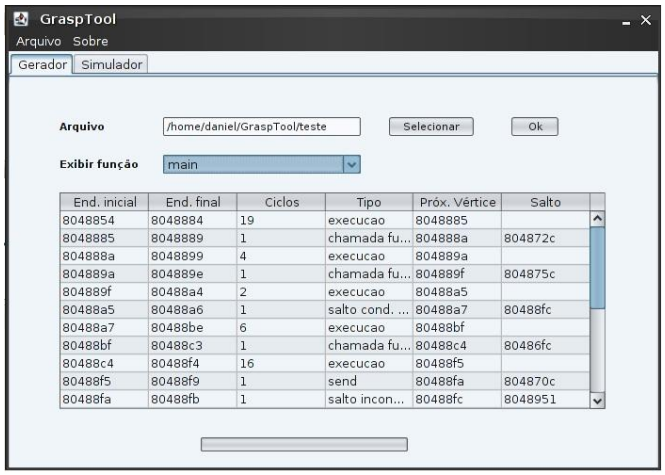
3.3.1 Interface gráfica do Gerador do grafo de execução

A interface gráfica do Gerador do grafo de execução é bastante simples e intuitiva, como mostrado na figura 3.4(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”. Durante a geração do grafo o usuário pode navegar nos menus do programa e inclusive interrompê-lo, graças a sua execução em uma *thread* separada da interface gráfica.

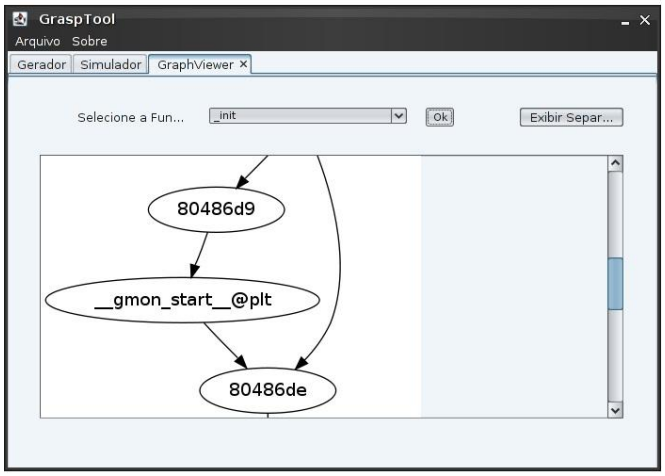
Após a obtenção do grafo de execução é possível verificar detalhes dos vértices adicionados em um grafo, como por exemplo, seus endereços inicial/final, número de ciclos gastos no vértice, tipo de instrução e o próximo endereço. Isso também pode ser visto na figura 3.4(a).

Outra funcionalidade adotada é a exibição dos grafos na forma de DAGs, como mostrado na figura 3.4(b). Utilizou-se o programa gratuito Graphviz [15] 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.



(a) Tela principal do Gerador.



(b) Exibição de um grafo de uma função.

Figura 3.4: Interface do gerador do grafo de execução.

3.3.2 Inicialização do Simulador

O simulador recebe duas entradas de dados para realizar suas tarefas. A primeira entrada é o grafo de execução criado pelo Gerador e a segunda entrada é a descrição do ambiente onde o sistema será simulado. Desta forma, o simulador poderá criar o modelo de interação do sistema como descrito em [6]. A seguir serão abordadas as entradas de dados.

Grafo de Execução

É a primeira entrada de dados 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, outras são desenvolvidas pelo programador, como a entrada/saída de dados, a comunicação entre as máquinas, entre outras.

Dados do Usuário

Esta entrada está relacionada à máquina alvo, ou seja, o usuário irá fornecer detalhes do ambiente que seria usado pelo programa a ser simulado em uma execução real. A figura 3.5 mostra a tela de entrada de dados do usuário.

A primeira informação que o usuário deve fornecer ao simulador é a largura de banda, em Mbps. Esta informação é relevante para a estimativa do atraso durante as trocas de mensagens MPI.

A seguir tem-se 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.

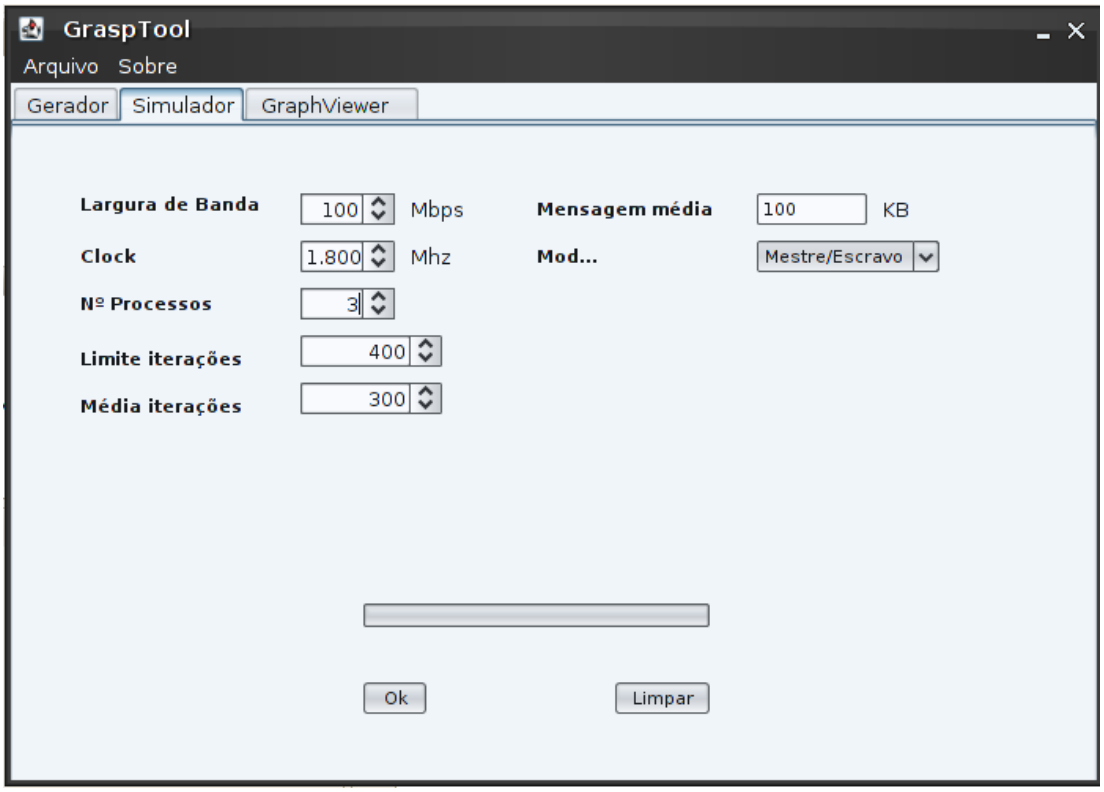


Figura 3.5: Tela principal do simulador.

Também deve ser informado ao simulador o número de processos paralelos a serem simulados. Em seguida informa-se o tamanho médio das mensagens, em Kb, 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 de vezes que um laço deve ser executado. Este valor atua como um limitante superior ao número de iterações, já que nenhum laço será executado eternamente.

Por fim o usuário deverá escolher o modelo de programa que será simulado. Inicialmente o único modelo disponível era o mestre/escravo, portanto ele será apresentado nesta seção. Detalhes da implementação do modelo SPMD, que é o alvo deste trabalho, serão abordados na seção 3.4.

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 no simulador, pois não é possível identificar durante a geração do grafo qual o sentido da troca de mensagens. A figura 3.6 mostra como a solicitação do tipo de vértice de comunicação ocorre no modelo mestre/escravo.

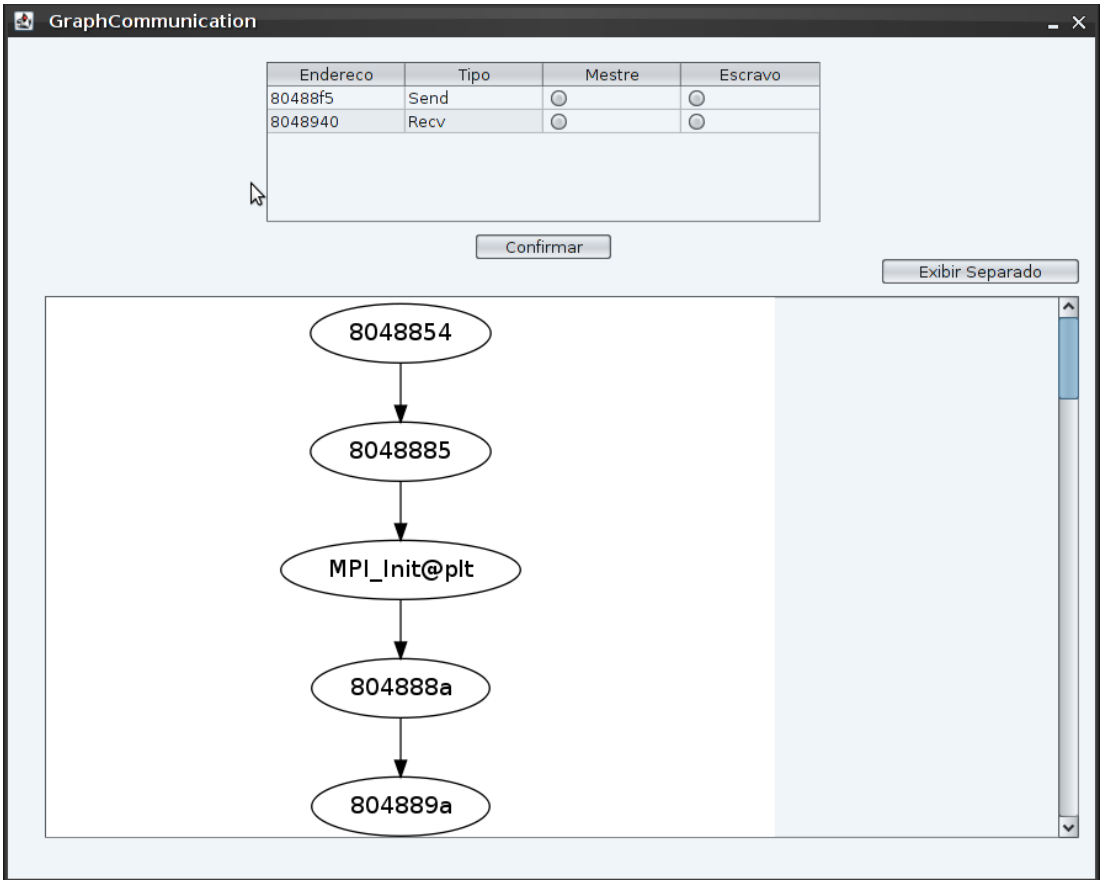


Figura 3.6: Solicitação de informações sobre os vértices no modelo mestre/escravo.

3.3.3 Motor de Simulação

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

A seguir as funções de probabilidade são carregadas. Este projeto utiliza as funções de distribuição de probabilidade normal e exponencial. A função de distribuição de probabilidade normal é encarregada de tratar testes de decisão 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 exponencial. Essas funções foram inicialmente implementadas na linguagem C por Zafalon e Manacero [18], e portadas para Java. Terminada esta primeira etapa, a simulação propriamente dita é iniciada. O pseudocódigo 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 FAÇA
2       PARA todos os processos ativos FAÇA
3               verifique se a pilha de funções chamadas está vazia
4       SE todas as pilhas estão vazias FAÇA
5               saia do enquanto
6       PARA todos os prcessos ativos FAÇA
7               selecione o processo i com menos número de ciclos
8       executar o vértice do processo i
9 mostrar dados coletados ao usuário
```

Figura 3.7: Algoritmo de simulação.

Critério de Parada

Como é possível verificar no algoritmo mostrado na figura 3.7, a primeira parte, que compreende as linhas 2-5, 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 6-7 do algoritmo e busca pelo processo com vértice que possui menor tempo de execução, isso é verificado pelo 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 com menor tempo de execução. 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 pelo processo. Após atualizar os dados de ciclos, o vértice em execução avança para o próximo nó e a simulação prossegue.

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 à 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.

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, pois este é o retorno da função principal do programa, ou seja, a função chamadora dos demais procedimentos. 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 real 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 que a predição de execução seja fiel ao 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*.

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. 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, sendo 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.

6. Comunicação

Os vértices de comunicação estão relacionados a primitivas MPI como *send*, *recv* e *sendrecv*. A função *send* é não-bloqueante, ou seja, 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, ou seja, 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 serão descritos na sequência.

3.3.4 Simulação do paradigma Mestre/Escravo

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

No caso de MPI, que usa o modelo SPMD, 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 dos 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, pois se mais mensagens forem enviadas/recebidas o resultado da simulação pode ser inesperado.

Identificação do vértice de separação mestre-escravo

O mapeamento dos vértices que separam os códigos do mestre e escravos é feito utilizando um percurso único pelo grafo, através de algumas modificações no algoritmo BFS. 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 da figura 3.8 mostra 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 sequência, 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, garantindo o percurso único pelo vértice. Além disso, a relação de parentesco do vértice é armazenada, sendo vital para a identificação dos vértices que separam o código do mestre do escravo. Na linha 15, o vértice adjacente ao extraído é adicionado na fila, isso garante a passagem por seus descendentes. Por fim, na linha 16 é verificado se o vértice é de comunicação, em caso afirmativo este é adicionado em sua lista específica.

```
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 o primeiro vértice da fila F
11              PARA cada T adjacente a S FAÇA
12                      SE T ainda não foi visitado FAÇA
13                              marque T como visitado
14                              marque S como pai de T
15                              coloque T na fila F
16                              SE T for um vértice de comunicação FAÇA
17                                      SE for comunicação do mestre FAÇA
18                                              adicione T na lista L1
19                                      SENÃO
20                                              adicione T na lista L2
```

Figura 3.8: Algoritmo para identificação mestre-escravo.

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, fazendo com que cada processo mestre ou escravo seja associado 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, acrescentando-se

uma lista dos vértices de laços de repetição. Para cada laço 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, mas apenas para os vértices contidos no laço de repetição, o que representa uma execução 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.

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 coleta do programa executável, ajudando o usuário na identificação dos gargalos de seu programa como também seus pontos críticos.

A primeira medida retirada na simulação mostra ao usuário o tempo total que o processo gastou para encerrar sua execução. Esta medida é obtida através do acumulador de ciclos descrito na seção anterior. Com esta medida o usuário pode 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 coletadas as trocas de mensagens MPI em todos os processos, possibilitando obter a quantidade e o tamanho total das mensagens trocadas. 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, fornecendo uma medição de quantos ciclos foram gastos neste trecho de código. Desta forma o usuário pode 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 relacionada 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 Implementação do paradigma SPMD

Esta seção apresenta os detalhes da implementação do modelo de simulação sob o paradigma SPMD, que foi o objetivo deste trabalho.

Taxonomia de Flynn

Classificação das arquiteturas de computador proposta em [30]. Ela se baseia nas possíveis unicidade e multiplicidade dos fluxos de instruções e de dados para definir quatro tipos de arquiteturas:

- SISD (*Single Instruction Single Data*): Fluxo único de instruções sobre um único conjunto de dados.

- SIMD (*Single Instruction Multiple Data*): Fluxo único de instruções em múltiplos conjuntos de dados.

- MISD (*Multiple Instruction Single Data*): Fluxo múltiplo de instruções em um único conjunto de dados.

- MIMD (*Multiple Instruction Multiple Data*): Fluxo múltiplo de instruções sobre múltiplos conjuntos de dados.

A arquitetura MIMD pode ser dividida em duas subcategorias de acordo com o modelo de paralelismo utilizado [31]:

- SPMD (*Single Program Multiple Data*): Múltiplos processadores autônomos executando o mesmo programa com diferentes entradas de dados.

- MPMD (*Multiple Program Multiple Data*): Múltiplos processadores autônomos executando pelo menos dois programas independentes.

Este padrão compreende diversos modelos de comunicação entre os processos e, consequentemente, diversos estilos de programação. Devido a essa heterogeneidade foi explorado um modelo específico, mostrado na figura 3.9, que representa a comunicação em malha de uma dimensão.

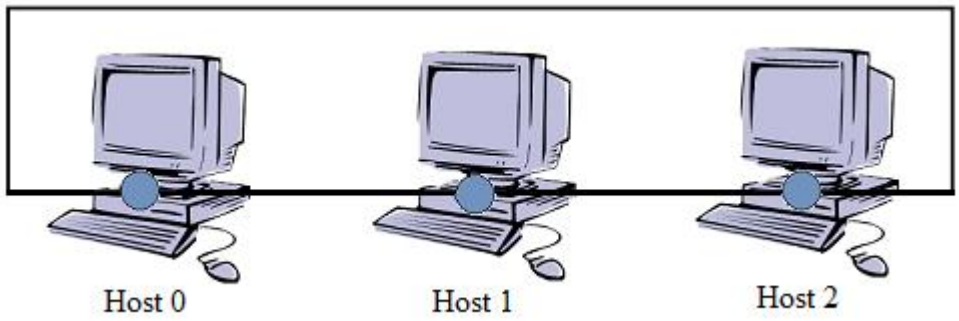


Figura 3.9: Modelo de malha em uma dimensão.

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 3.10 ilustra esta máquina central.

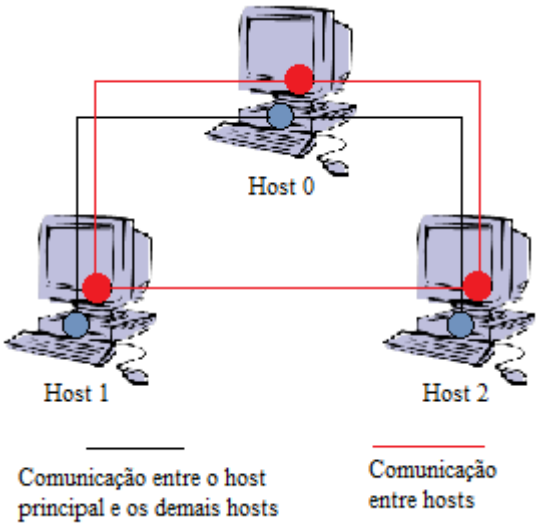


Figura 3.10: Máquina principal no modelo malha em uma dimensão.

Na figura 3.11, foram destacados em vermelho os elementos que apresentam funcionamentos diferentes dependendo do modelo utilizado. A implementação da forma de funcionamento desses elementos para o modelo SPMD foi um dos principais focos deste trabalho.

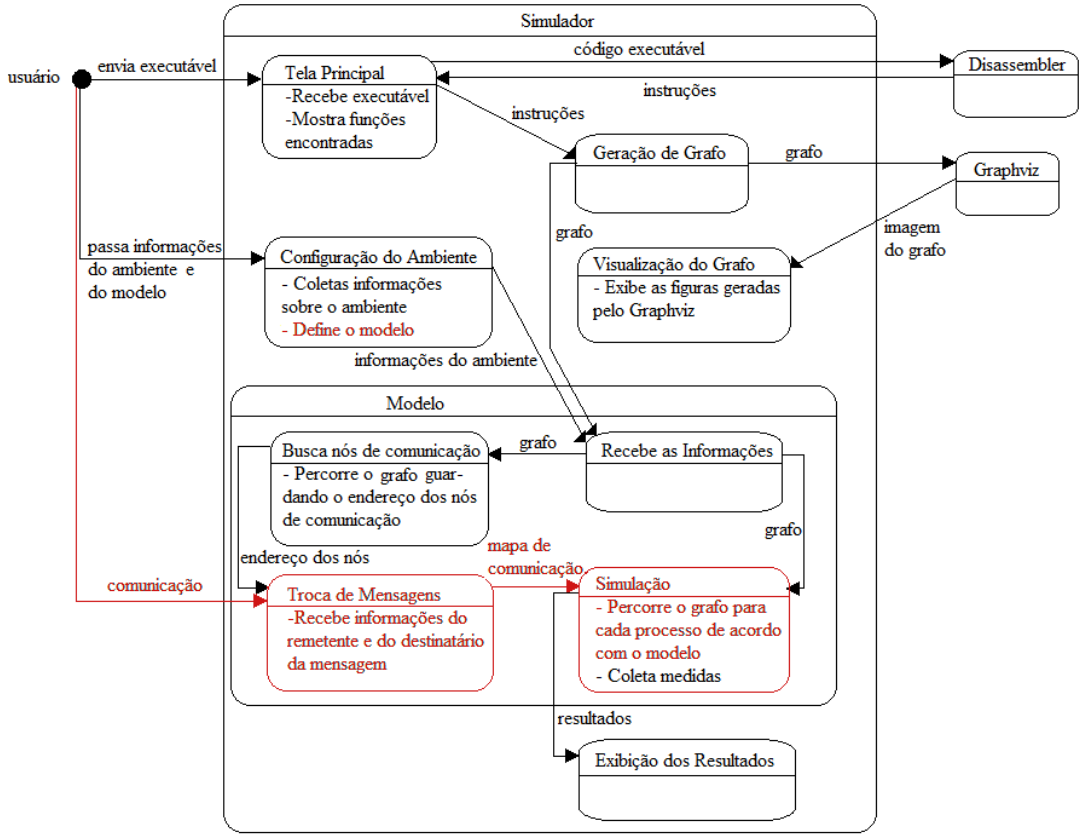


Figura 3.11: Diagrama de estados com elementos destacados.

3.4.1 Identificação dos vértices de comunicação

Como 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 relacionadas à origem e destino da mensagem, se esta é para o vizinho da esquerda ou da direita, se a mensagem pertence à máquina central (envio de configurações, etc.) ou 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.

Passagem de dados pelo Usuário

Na tela de entrada de dados do simulador, onde o usuário configura o ambiente a ser simulado, existe uma caixa de seleção em que é definido o modelo que será usado pelo simulador. O modelo mestre/escravo aparece como opção inicial, para escolher o modelo SPMD basta selecioná-lo como mostra a figura 3.12.

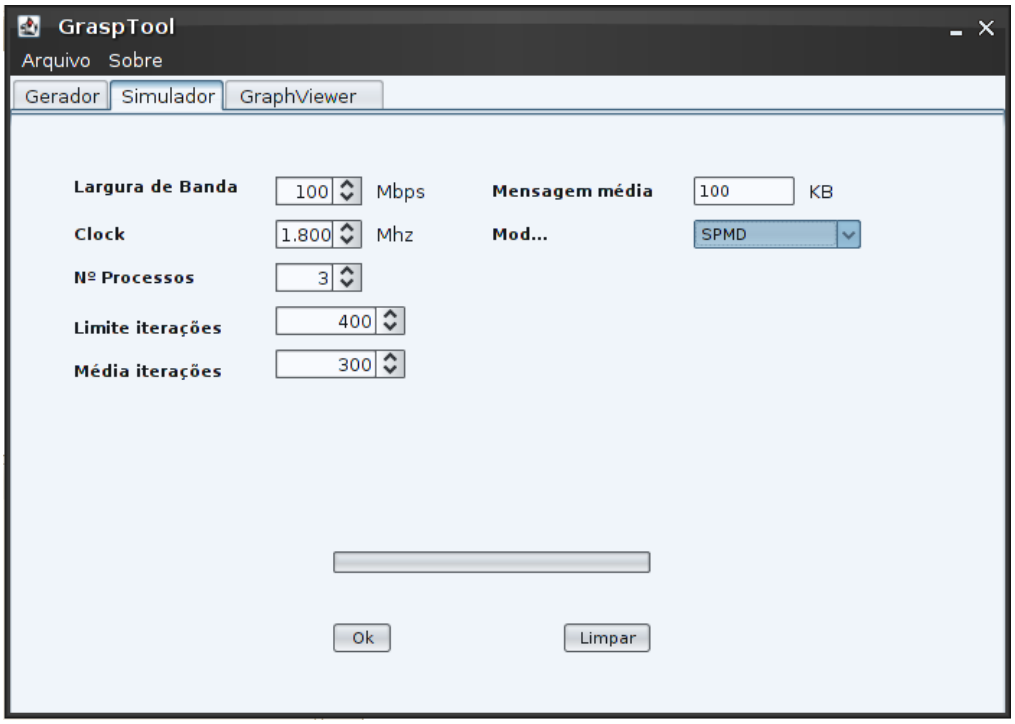


Figura 3.12: Tela principal do simulador com o modelo SPMD selecionado.

Da mesma forma como acontece com o modelo mestre/escravo, antes de iniciar a simulação o usuário deve passar ao simulador as informações necessárias para identificação da forma como a comunicação ocorre em cada instrução de troca de mensagem. A figura 3.13 mostra como a passagem de informações é feita.

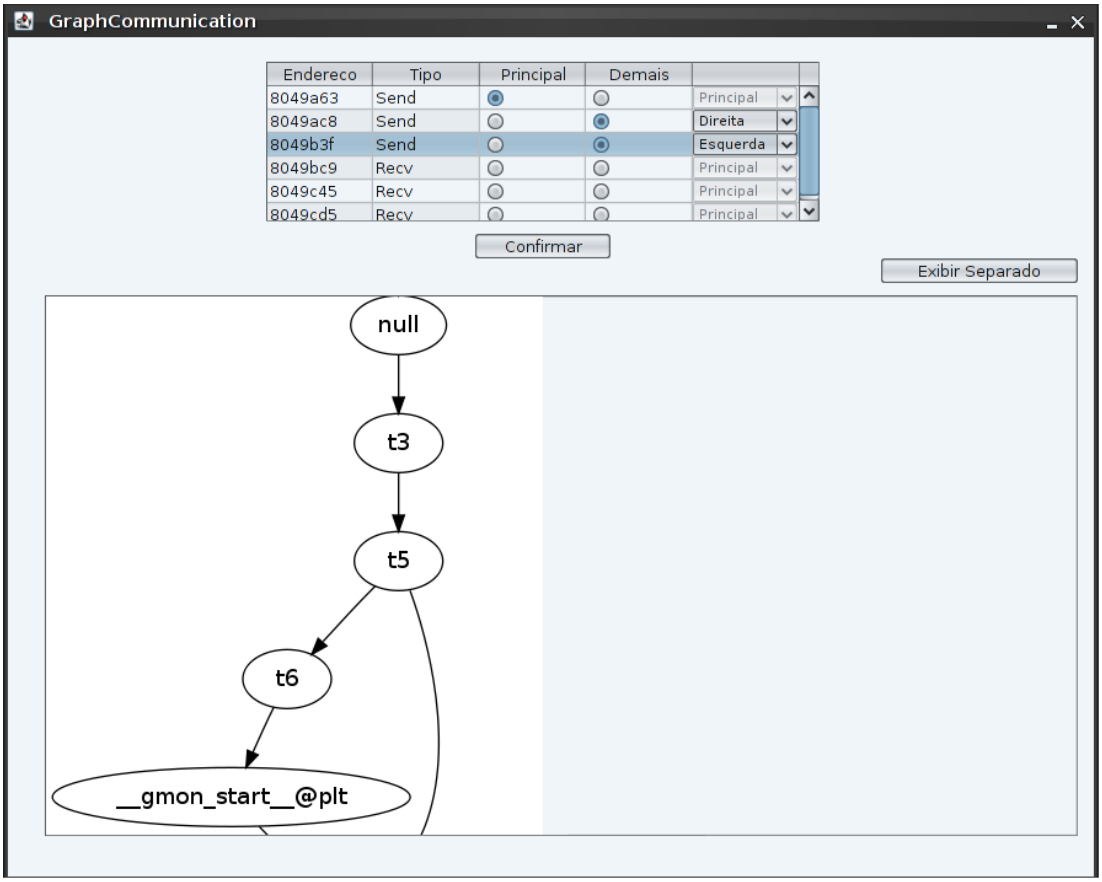


Figura 3.13: Solicitação de informações sobre os vértices no modelo SPMD.

3.4.2 Simulação do modelo SPMD

Com base nestas informações passa-se à configuração do grafo de execução, de tal forma que cada processo possa executar sua respectiva ação 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. Se essa instrução é encontrada verifica-se 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 ramo 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. O pseudocódigo na figura 3.14 ilustra o que foi discutido.

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

Figura 3.14: Algoritmo para identificação SPMD.

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.

3.5 Considerações finais

Este capítulo apresentou a especificação das características mais relevantes na implementação do modelo SPMD na ferramenta de simulação Grasptool. A seguir serão apresentados os testes efetuados para validação da eficiência do projeto e a análise dos resultados.

Capítulo 4 - Testes e Resultados

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

A seguir são apresentados os testes realizados para o paradigma SPMD. Esses testes incluem 50 simulações e benchmarks reais de avaliação do sistema para cada programa a fim de reduzir os desvios padrão.

O primeiro programa faz uma aproximação da integral de uma função, o segundo resolve equações diferenciais de aquecimento de forma paralela e o terceiro determina a temperatura ponto – a – ponto de uma chapa.

4.1 Regra de Quadratura

O primeiro programa possui granularidade grossa, ele calcula a integral de uma função de forma aproximada utilizando um somatório com pesos dos valores assumidos pela função em pontos específicos dentro do domínio de integração. Esse procedimento é conhecido como regra de quadratura. No algoritmo o processo principal divide a função f em intervalos regulares e passa o cálculo de cada trecho a

um processo, incluindo o processo principal. Após o cálculo de seu intervalo, cada processo envia seu resultado ao processo principal.

Trocas de mensagens

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, que coincidem com os valores para o teste real. No teste real foram colocados em execução oito processos.

Esses resultados são idênticos devido ao mapeamento de laços com trocas de mensagens. Conforme explicado no capítulo anterior, esses laços executam de modo determinístico. Inicialmente o processo principal distribui quatro informações necessárias para cada processo (28 *Sends*), estes recebem sua informação, realizam seus cálculos e retornam a resposta ao processo principal (4 *Recvs*). O tempo decorrente das trocas de mensagens pode ser considerado irrelevante neste teste, devido à alta taxa de transmissão de dados associada à inatividade da rede no momento da execução.

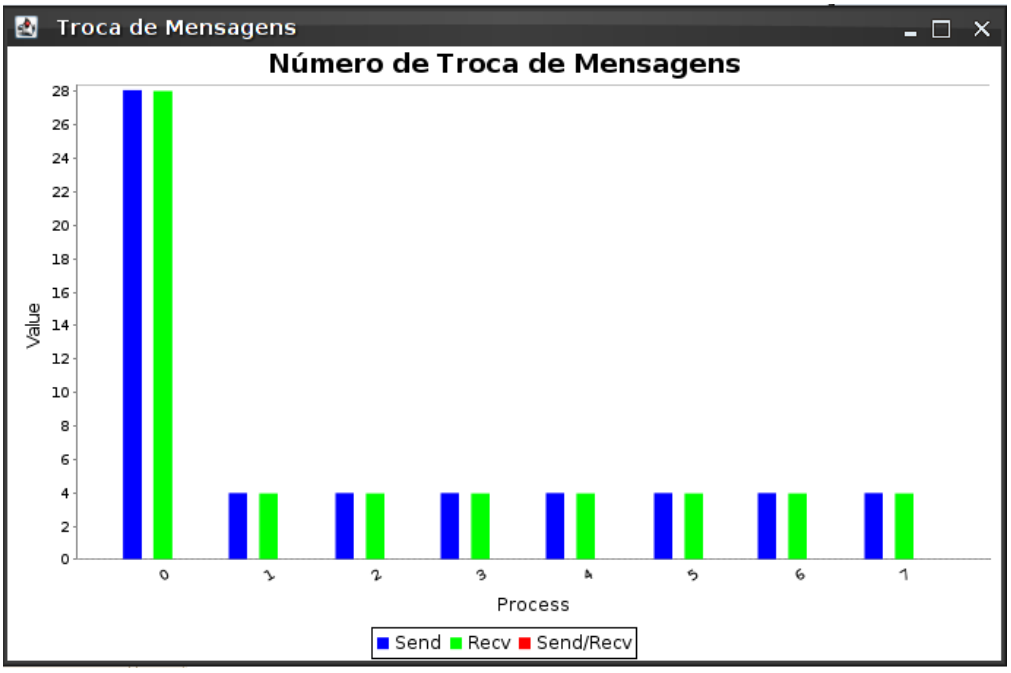


Figura 4.1: Trocas de mensagens entre os processos simulados.

Iterações de laços

A principal vantagem na contagem de iterações de laços está relacionada à identificação de gargalos do sistema. Levando em consideração que a maior parte do processamento dos programas está em laços de repetição, 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 dez testes realizados. Nela notamos a presença de iterações no processo zero. Isso ocorre pois, diferentemente do modelo mestre/escravo, no modelo SPMD o processo zero também participa dos cálculos.

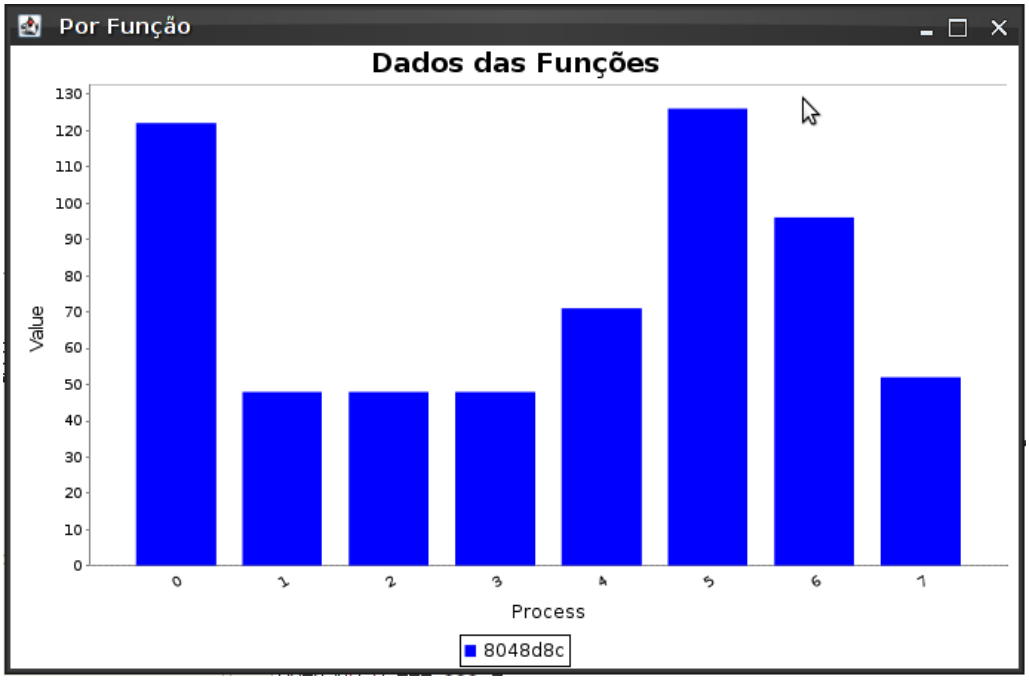


Figura 4.2: Estatísticas sobre a execução de um laço no simulador.

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 para cada processo. 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 relacionados à 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 do modelo pode ser observada no tempo médio obtido na simulação em relação ao tempo real. Na simulação foi obtido um tempo médio, que envolve todos os processos nos dez testes, de $3{,}31 \times 10^{-3}$ segundos com desvio padrão de $0{,}18 \times 10^{-3}$ segundos. Já o tempo médio real foi de $3{,}16 \times 10^{-3}$ segundos. Isso fornece um erro de aproximadamente 4,5%, o que é aceitável 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 gasto na simulação e no ambiente real pelo programa 1.

Id processo	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**t. médio ($x10^{-3}$ s)**
t. simulado ($x10^{-3}$ s)	5,87	3,32	3,27	3,85	2,64	3,11	2,32	2,08	3,31
t. real ($x10^{-3}$ s)	5,47	3,28	2,74	1,91	3,06	2,83	2,60	3,43	3,16

Chamadas de função

Algumas chamadas de função neste programa estão relacionadas aos laços, pois são chamadas para funções que realizam alguns cálculos do somatório. A figura 4.3 mostra o número de chamadas para uma dessas funções. Assim como ocorreu no item anterior, aqui também é possível observar a diferença entre este modelo e o modelo mestre/escravo através das chamadas para esta função feitas pelo processo zero.

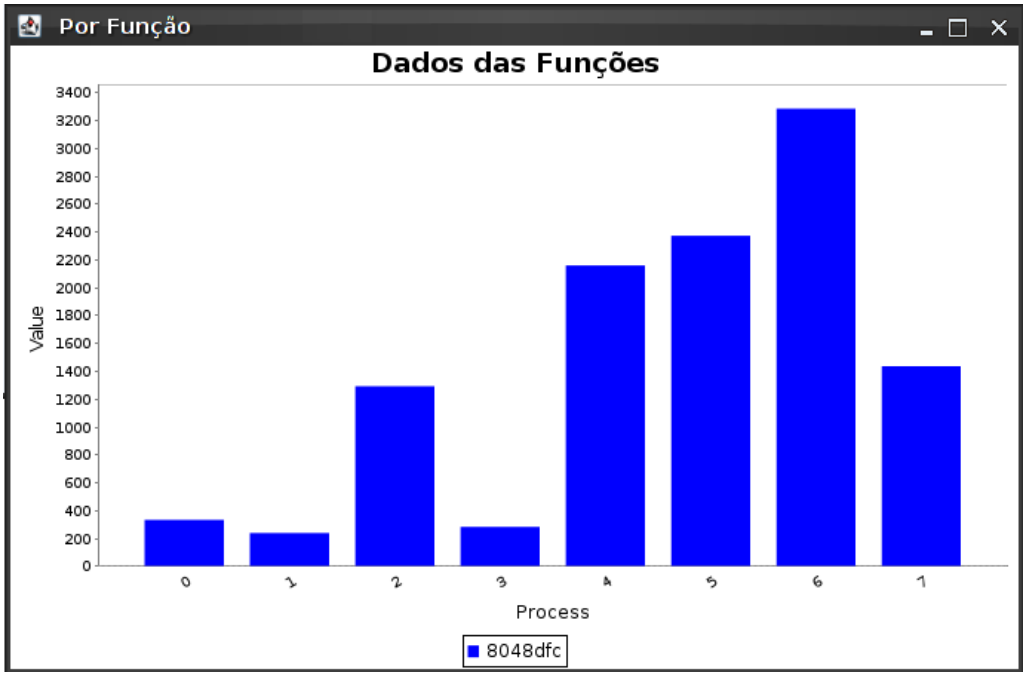


Figura 4.3: Estatísticas sobre a execução de uma função.

4.2 Resolução de uma equação de aquecimento

Este programa resolve equações parciais usadas para calcular o aquecimento em função do tempo, ele também possui granularidade grossa. Tendo P processos, o algoritmo faz uma decomposição do intervalo [A,B] em P subintervalos iguais que são divididos entre todos os processos. A equação diferencial define a relação entre os valores U(x-1,y), U(x,y), U(x+1,y) e o valor "futuro" U(x,y+1). Para achar a solução dos pontos extremos de seus subintervalos cada processo precisa receber o valor do extremo esquerdo do vizinho da direita e o extremo direito do vizinho da esquerda, com isso é possível observar outra característica do modelo SPMD, que é a troca de mensagens entre processos vizinhos.

Trocas de mensagens

A figura 4.4 mostra o número de mensagens trocadas durante os testes, esses valores são iguais na simulação e no teste real, já que as trocas de mensagens não fazem

parte de um laço nem de um teste de decisão. Para este programa também foram usados 8 processos.

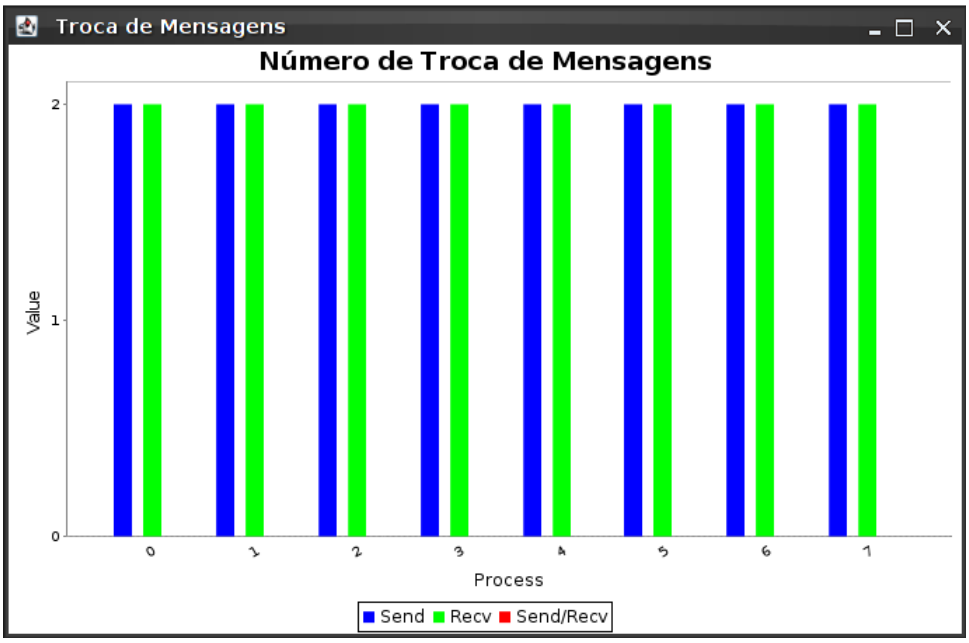


Figura 4.4: Trocas de mensagens entre os processos.

Iterações de laços

A figura 4.5 mostra a execução de um laço do programa para cada processo simulado, em um dos dez testes realizados.

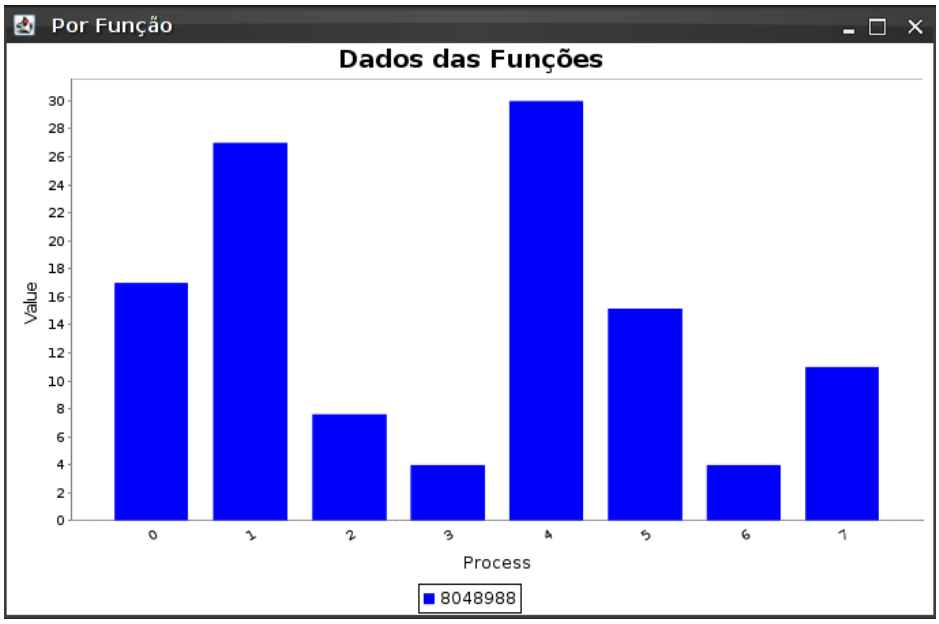


Figura 4.5: Execução de um laço no simulador.

Tempo de execução

A tabela 4.2 abaixo mostra os tempos médios obtidos na simulação e na execução real. A eficiência do modelo pode ser verificada pelo percentual de erro do tempo médio de execução, ou seja, por volta de 3% com desvio padrão de $0{,}02 \times 10^{-3}$ segundos.

Tabela 4.2: Tempo gasto na simulação e no ambiente real pelo programa 2.

Id processo	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**t. médio ($\times 10^{-3}$ s)**
t. simulado ($\times 10^{-3}$ s)	0,45	0,43	0,49	0,49	0,52	0,44	0,64	0,52	0,49
t. real ($\times 10^{-3}$ s)	0,43	0,59	0,45	0,37	0,53	0,68	0,57	0,41	0,51

Chamadas de função

A figura 4.6 mostra o número de chamadas para uma das funções.

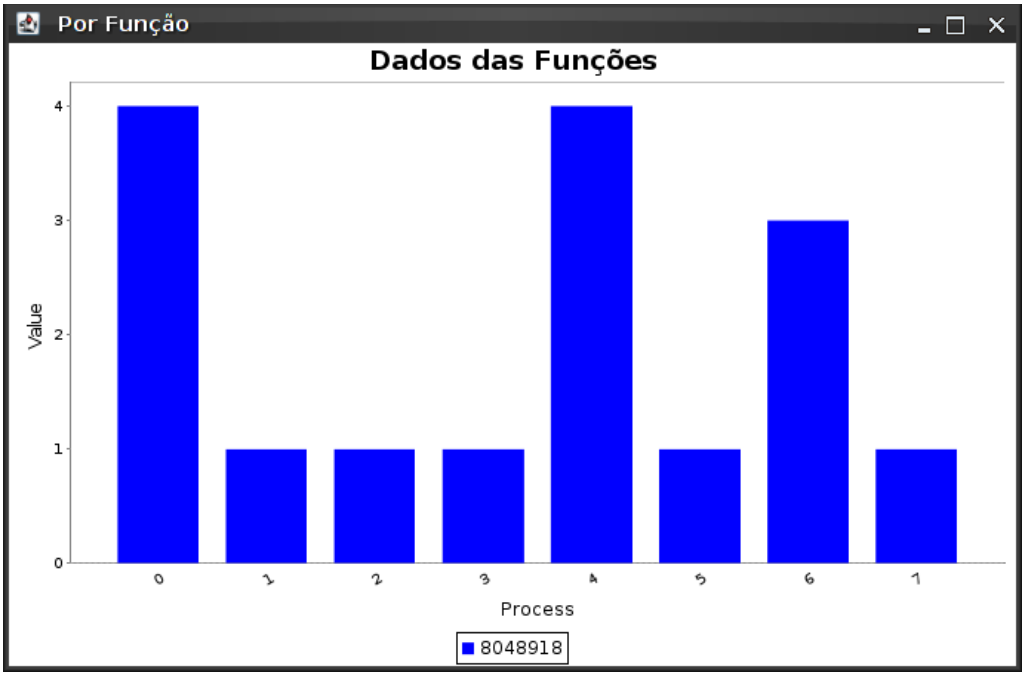


Figura 4.6: Execução de uma função.

4.3 *Red* & *Black*

Este algoritmo é usado para resolver problemas de transferência de calor em uma chapa através de aproximações sucessivas, ele apresenta granularidade média. Cada ponto da chapa tem sua temperatura, numa iteração, como sendo a média das temperaturas de seus vizinhos na vertical e na horizontal e de sua própria na iteração anterior, assim, a chapa pode ser vista como uma enorme matriz, em que cada elemento é o valor da temperatura da chapa num ponto. Os valores de contorno e constantes também são considerados elementos da matriz

O algoritmo *Red* & *Black* faz o cálculo das temperaturas numa iteração considerando valores da matriz *red*, armazenando os novos resultados na matriz *black,* na iteração seguinte a matriz *red* passa a ser o destino dos resultados calculados a partir da matriz *black.*

A programação desse algoritmo em paralelo foi feita dividindo a matriz em vários grupos de linhas e destinando cada grupo para um dos processos paralelos. Algumas dessas linhas, que estão nas fronteiras entre grupos, são compartilhadas entre pares de processos.

Trocas de mensagens

Nos testes reais foram realizadas 100 iterações com 2 *Sends* e 2 *Recvs* em cada iteração, entretanto como pode ser observado na figura 4.7 isso não ocorreu durante a simulação. Isso se deve ao fato do simulador utilizar uma função de distribuição de probabilidade para determinar a quantidade e iterações nos laços, essa função utiliza informações passadas pelo usuário como descrito no capítulo 3.

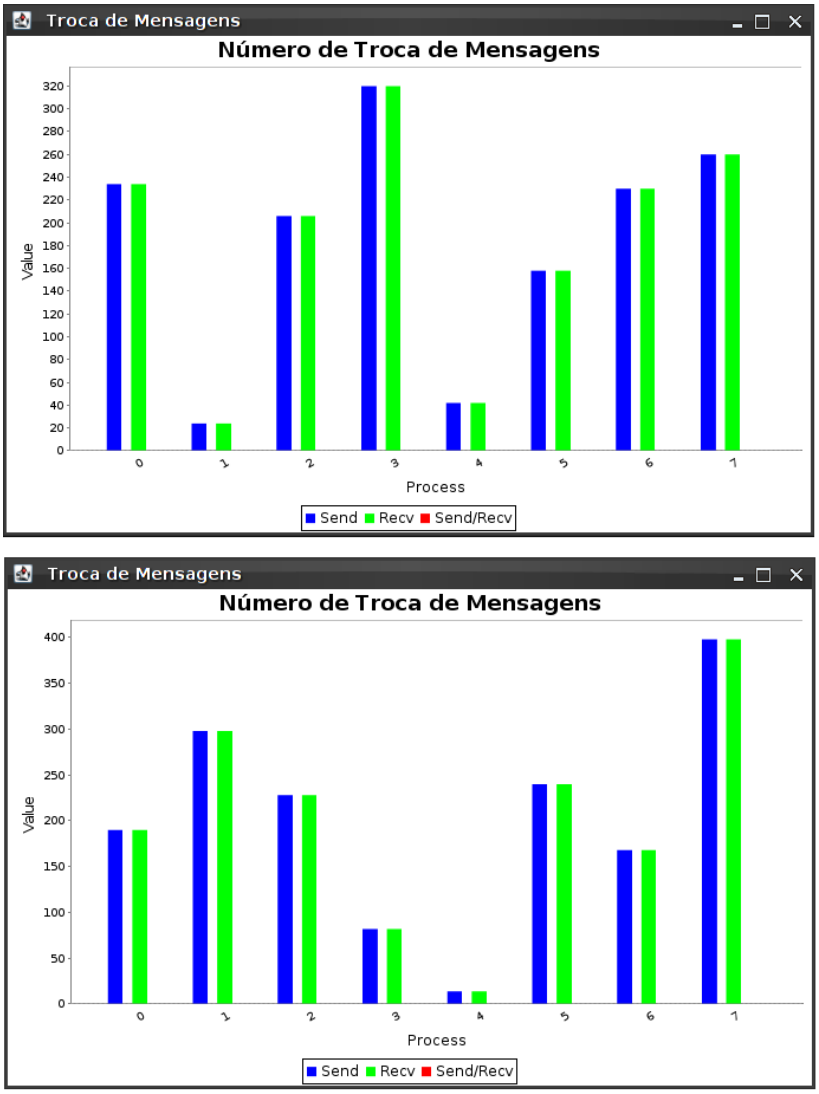


Figura 4.7: Trocas de mensagens entre os processos em dois testes.

Tempo de execução

A tabela 4.3 abaixo mostra os tempos médios obtidos na simulação e na execução real. O percentual de erro do tempo médio de execução é por volta de 1% com desvio padrão de $1{,}6 \times 10^{-3}$ segundos.

Tabela 4.3: Tempo gastos na simulação e no ambiente real pelo programa 3.

Id processo	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**t. médio (s)**
t. simulado (s)	0,130	0,138	0,134	0,135	0,131	0,128	0,125	0,131	0,132
t. real (s)	0,156	0,12	0,12	0,134	0,156	0,114	0,142	0,122	0,133

Capítulo 5 - Conclusão

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 verificada por testes realizados sob o modelo implementado em comparação com programas reais. Os resultados obtidos são influenciados pela qualidade dos parâmetros de entrada fornecidos, a precisão das funções de distribuição de probabilidades utilizadas e o número de simulações realizadas.

Por fim, o projeto também contribuiu para um estudo 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.

Perspectivas

Assim como esse trabalho, outros podem ser realizados com o intuito de adicionar novas características ao Grasptool ou otimizar as existentes. Isso pode ser feito devido à modularidade concebida no projeto. Algumas propostas são citadas abaixo:

- Otimização do grafo de execução gerado para garantir maior velocidade na simulação do programa. Essa característica deve possibilitar uma alteração do

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 que realize engenharia reversa em códigos para diversas arquiteturas de processadores. Isso possibilitaria uma maior desvinculação entre o simulador e a plataforma.

- Adição de novas funções MPI para simulação. Isso envolve um estudo do uso das 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] Bradley, D. K.; Larson, J. L.; “A parallelism-based analytic approach to performance evaluation using application programs”; *Proceedings of the IEEE*, vol. 81, n. 8, p. 1126-1135, 1993.

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

[3] HP-MPI. Disponível em < http://docs.hp.com/en/5992-2330/ch10s03.html >, último acesso em 24/10/2011.

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

[5] Manacero, A. J.; “Predição de Desempenho de Programas Paralelos por Simulação do Grafo de execução”; Tese (Doutorado) - Unicamp, 1997.

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

[7] MPICH. Disponível em < http://www.mcs.anl.gov/research/projects/mpich2 >, último acesso em 24/10/2011.

[8] OpenMPI. Disponível em < http://www.open-mpi.org >, último acesso em 24/10/2011.

[9] IntelMPI. Disponível em < http://www.intel.com/cd/software/products/asmona/eng/308295.htm >, último acesso 24/10/2011.

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

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

[12] Pierce, J.; Mudge T. N.; “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.

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

[14] GNU Binutils. Disponível em < http://www.gnu.org/software/binutils >, último acesso em 19/10/2011.

[15] Graphviz. Disponível em < http://www.graphviz.org >, último acesso em 11/10/2011.

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

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

[18] 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

[19] Marsan, M. A.; Balbo, G.; Conte, G.; “A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems”. [S.l.]: *ACM Trans. Comput. Sys.*, 1984.

[20] Zuberek, W. M.; “Performance evaluation using unbounded timed petri nets”. *Proc. of the Third Intl. Workshop on Petri nets and Performance models*, Kyoto, Japan, 1989.

[21] Thomasian, A.; Bay, P. F.; “Analytic queueing network models for paraller processing of tasks systems”. *IEEE Trans. on Computers*, 1986.

[22] 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.

[23] Jkim, J.; Shin, K. G.; “Execution time analysis of communicating tasks in distributed systems”. *IEEE Trans. on Computers*, 1996.

[24] Adve, V. S.; “Analyzing the behavior and perfomance of paraller programs”. Tese (Doutorado) - Universidade de Wisconsin, 1993.

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

[26] 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.

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

[28] Maciel, F. M.; Oliveira, N. A.; “Geração de um grafo de execução para códigos executando em plataformas baseadas na família x86”; Monografia (Conclusão do curso) - Unesp, 2005.

[29] Calzeta, E. P.; Souza, T. A. D.; “Ferramenta para a predição de desempenho de programas paralelos através da simulação do grafo de execução”; Monografia (Conclusão do curso) - Unesp, 2008.

[30] Flynn, M. J.; “Some Computer Organizations and Their Effectiveness”; *IEEE Transactions on Computers*, v.C-21, n.9, p. 948-960, Set. 1972.

[31] Darema, F.; George, D. A.; Norton, V. A.; Pfister, G. F.; “A single-program-multiple-data computational model for epex fortran”. *Parallel Computing*, n.7, p. 11–24, 1988.