Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e Computação
Departamento de Sistemas de Energia Elétrica

# Predição do Desempenho de Programas Paralelos Por Simulação do Grafo de Execução

candidato : Aleardo Manacero Júnior
orientador: Prof. Dr. André L. Morelato França

Tese submetida à Faculdade de Engenharia Elétrica e Computação da Universidade Estadual de Campinas, para preenchimento dos pré-requisitos parciais para obtenção do Título de Doutor em Engenharia Elétrica.

8 de Setembro de 1997

# Resumo

O desenvolvimento de programas para ambientes de programação paralela exige do projetista uma atenção especial quanto ao desempenho obtido pelo conjunto programa-máquina. Os custos elevados de processamento fazem com que seja necessário obter o melhor desempenho possível para reduzir custos e tempos de processamento. O problema passa a ser como definir medidas de desempenho e como realizar as medições para verificar se o sistema é eficiente ou não.

Existem diversas ferramentas de análise ou predição de desempenho, que procuram fornecer ao usuário dados sobre o programa, Para auxiliar o projetista a localizar pontos críticos do mesmo e fazer acertos para melhorar o desempenho do sistema. Infelizmente a maioria dessas ferramentas trabalha com grandes aproximações no modelo do ambiente paralelo, fazendo com que os resultados obtidos não sejam totalmente precisos. Além disso, quando essas ferramentas fazem uso de medidas experimentais para a realização da análise, elas acabam introduzindo erros experimentais pela necessidade de inserir código adicional ao programa analisado.

Neste trabalho é apresentada uma nova metodologia para realizar medidas de desempenho sem a necessidade de código adicional ao programa e, além disso, sem a necessidade de usar a máquina alvo do programa durante todo o processo. A metodologia faz a reescrita do código executável para um grafo de execução. Este é um grafo dirigido que armazena informações sobre o tempo de processamento de cada instrução de máquina incluída no código do programa. As instruções são agrupadas em blocos seqüenciais, que são os vértices do grafo. Os arcos entre qualquer par de vértices indica a relação de precedência entre eles.

A informação contida no grafo é usada por um simulador que o percorre para fazer as medições necessárias para a predição ou análise de desempenho. A simulação pode ser feita em estações de trabalho comuns, preservando o ambiente paralelo para processamento útil apenas.

Esta tese está organizada de modo a fornecer os conhecimentos básicos sobre medidas de desempenho em sistemas paralelos e uma discussão detalhada sobre o método proposto para medição de desempenho. Inclui também uma descrição sobre um protótipo implementado usando essa abordagem e sua avaliação em testes comparativos com um sistema real. Esse sistema é descrito em dois apêndices, enquanto um terceiro apêndice faz uma rápida descrição do processador usado nos "benchmarks".

# Abstract

The process of writing a new code for a parallel programming environment demands from its designer a lot of attention on the performance achieved by the pair program-machine. The high processing costs justify the efforts to reduce processing time and costs, which leads to the problem of defining performance metrics and approaches to measure the system performance.

Several performance analysis and prediction tools are available to help designers in such a task. With those tools the designer can locate critical points in the code and look for solutions to improve the program's performance. Unfortunately, most of those tools make so many approximations in the parallel environment model, that make themselves somewhat inaccurate. Moreover method of measurement usually adopted get performance data through an instrumented program's run. This approach affects the measures since additional code is inserted into the program under analysis.

This work introduces a new method to make such measurements without including additional code nor requiring runs on the target parallel machine. The proposed approach is to rewrite the executable code into an execution graph, which is a directed graph that keeps the information about the processing time of each machine instruction included in the code under analysis. These instructions are grouped into sequential blocks, which constitute the vertexes of the execution graph. The links between any pair of vertexes give their precedence relationship.

The information stored into the execution graph is used by a simulator that runs the graph in order to make the necessary measurement for the performance prediction or analysis. The simulation can be run on ordinary workstations, saving the parallel environment for useful processing only.

This thesis provides some background on performance metrics in parallel systems and also presents a detailed discussion about the proposed method for performance measurements, including a description of a prototype implemented using such approach. Next this prototype is used to evaluate the method comparing against a benchmark system, which is described along two appendixes. The processor used in the benchmark is briefly described in a third appendix.

# Agradecimentos

Durante a realização deste trabalho tive contato com muitas pessoas, várias delas merecendo um agradecimento especial nesta página. Como é difícil citar todos nominalmente, deixo indicados apenas os que tiveram um maior envolvimento com alguma parte do trabalho. Aos demais deixo registrado um muito obrigado pelo que tiverem me ajudado. A seguir deixo meu agradecimentos aos mais envolvidos com este trabalho.

Ao meu orientador André Morelato pela paciência em desenvolver um trabalho muitas vezes interrompido por circunstâncias alheias à nossa vontade.

Aos colegas da E791, em especial, Cat James, Jeff Appel e Jean Slaughter, pelo auxílio para o entendimento do trabalho envolvido com física de partículas e ao Prof. Augusto Brandão por ter dado a oportunidade de realizar o trabalho no Fermilab.

Aos colegas do DSEE, principalmente pelo incentivo para a conclusão deste trabalho.

Aos colegas do DCCE/Ibilce, pelo apoio durante meu afastamento do departamento e, em especial, ao Prof. José A. Cordeiro por suas contribuições para a geração de números aleatórios.

Aos meus pais por terem permitido que eu chegasse até aqui.

Em especial à minha esposa (Ciça) e meu filho por suportarem dias de péssimo humor e também minha ausência em vários momentos.

À CAPES e ao Fermilab pelo suporte financeiro durante parte do trabalho.

Dedico este trabalho à Ciça e ao Gustavo

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Capítulo 1

# Introdução

## 1.1 Predição e análise de desempenho

Um dos grandes problemas com computadores é fazer um uso eficiente dos mesmos de forma a obter uma relação custo/benefício favorável ao usuário/provedor do sistema. Para equipamentos de pequeno porte essa questão não é crítica dado o baixo custo dessas máquinas, mas o mesmo não ocorre em sistemas de grande porte e, atualmente, sistemas de computação de alto desempenho. Nesses casos o custo do equipamento é grande demais para que se permita um uso ineficiente do mesmo ou as necessidades de esforço computacional são elevadas para desprezar possíveis ganhos de eficiência. Quando se fala em computação paralela é inevitável que se determine pontos ótimos de operação para um dado programa. Pequenas diferenças de eficiência podem significar grandes variações nos custos de processamento.

Desse modo é importante que existam mecanismos para determinar se um programa está ou não sendo executado de forma eficiente em uma dada máquina. Em outras palavras, é necessário que se tenha um mecanismo para fazer a medição de desempenho desses programas. Entretanto, a simples medição de desempenho pode ser uma ação ineficaz se não houver possibilidade de mudança do programa e/ou máquina em que ele é executado. Para facilitar essas mudanças é preciso fazer uma predição/análise de desempenho durante o processo de desenvolvimento do programa e definição do equipamento. O processo de predição/análise pode ser feito empiricamente pelo projetista do programa ou através de alguma ferramenta desenvolvida para tanto. Como a primeira possibilidade exige um grau de experiência elevado é preferível que se tenha uma ferramenta para auxiliar o projetista nessa tarefa.

Com isso a construção de ferramentas para predição/análise de desempenho de programas é considerada uma área de grande importância em engenharia de computação. Os

problemas com o desenvolvimento de tais ferramentas são a definição de uma estratégia adequada para a realização das medidas de desempenho e a especificação de um modelo correto para o sistema sob análise. Ambos apresentam diversas complicações na sua resolução, portanto qualquer esforço nessa direção é de grande importância, mesmo que sejam direcionados para algum equipamento ou caso específico.

## 1.2 Computação de alto desempenho e sistemas de energia elétrica

Os grandes sistemas de energia elétrica são operados através de centros de controle que realizam suas tarefas por meio de sistemas hierarquizados de computadores em tempo real. As tarefas a serem executadas automaticamente por um centro de controle são de natureza múltipla, indo desde supervisionar a aquisição de dados das unidades remotas de medição, transformar as medidas em informações de engenharia, realizar análises que fornecem as condições de operação e segurança da rede elétrica, até a implementação de ações de controle necessárias para manter o sistema elétrico operando e atendendo aos requisitos de qualidade.

Na atual geração de centros de controle diversos tipos de análise não são realizadas em tempo real, mesmo nos centros de controle mundialmente mais desenvolvidos, por falta de capacidade computacional, pois os problemas envolvidos são realmente formidáveis. Por exemplo, pode-se citar o cálculo de fluxo de carga ótimo com restrições de segurança (que é necessário para operar a rede elétrica de forma segura e otimizada), a análise de segurança estática, e a análise de segurança dinâmica com redespacho de geraçao e determinação de limites operativos (que são necessárias para manter a rede operando, sem blecautes, mesmo após a ocorrência de perturbações severas). A solução desses problemas, no caso de redes de grande porte, pode envolver milhares de equações algébricas e diferenciais e alguns milhões de variáveis que precisam ser resolvidas ciclicamente em intervalos de alguns segundos.

A fim de elevar a capacidade computacional dos centros de controle, uma linha de trabalho que vem sendo mundialmente seguida é utilizar processamento paralelo e distribuído, não só para resolver simultaneamente várias tarefas, bem como para resolver em paralelo um problema específico. Outra linha de pesquisa, que não é incompatível com a anterior, consiste em projetar máquinas dedicadas ("special purpose"), em geral multiprocessadas, para resolver mais eficientemente problemas específicos. Em ambas as abordagens, torna-se fundamental dispor de ferramentas computacionais que auxiliem o desenvolvimento e teste de desempenho de programas aplicativos, relacionando-os com as máquinas-alvo. Esse fato tem motivado

a realização de trabalhos como este que, embora não diretamente envolvendo sistemas de energia elétrica, são extremamente importantes para a evolução tecnológica da área.

## 1.3 Problemas em análise/predição de desempenho

Em linhas gerais o grande fator de complicação na solução dos problemas indicados na seção 1.1 é a diversidade de configurações que um ambiente computacional (máquina e programa) pode assumir. Esse fator torna-se ainda mais importante quando se trata de ambientes paralelos, os quais possuem uma grande gama de arquiteturas e soluções diferentes. Isso dificulta a obtenção de um modelo correto de forma automática e aumenta a complexidade das técnicas de medição no programa.

Gerar modelos para o sistema de forma automática facilitaria o trabalho de análise de desempenho. Entretanto, apenas sistemas que possuam características bastante gerais podem ser gerados dessa forma. Sistemas mais elaborados precisam de técnicas mais rebuscadas para a geração de modelos que os representem de forma equivalente. Isso faz com que a maioria das ferramentas para análise de desempenho tenha problemas de adequação em várias configurações. Disso resulta uma intensa pesquisa no desenvolvimento de ferramentas que possam ser aplicadas a uma gama extensa de ambientes, mesmo com algum prejuízo de precisão.

Já com respeito às técnicas de medição no sistema, um problema é obter mecanismos que não dependam do "hardware" em que se faz a medida. Isso é complicado uma vez que a maior parte das medidas é invasiva, logo é necessário que a ferramenta tenha sido desenvolvida também para o processador em que se realizarão as medidas. Outro problema é conseguir medidas precisas e sem a interferência de quem as obtém. Como são invasivas elas acabam por disputar o acesso ao processador com o programa em medição. Logo é preciso que o tempo consumido pela medição seja desconsiderado de alguma forma antes que se analisem os dados brutos de desempenho.

## 1.4 Objetivos do trabalho

Como exposto anteriormente a predição/análise de desempenho é um processo complexo, sem soluções que sejam ao mesmo tempo simples e precisas. Também comentou-se que os resultados de desempenho podem representar ganhos significativos tanto para quem usa um sistema como para quem o fornece. Dessa forma o objetivo principal deste trabalho é apresentar uma nova metodologia para fazer a predição de desempenho de programas paralelos

em que sejam minimizados os problemas aqui descritos.

O principal aspecto dessa metodologia é a forma como são realizadas as medições para a análise de desempenho. Nela é necessário apenas que se conheça a configuração da máquina na qual se deseja medir o desempenho, incluindo-se informações sobre instruções de máquina e detalhes sobre a topologia de conexão entre processadores, e o código do programa compilado para aquele tipo de processador. Com esses dados de entrada é feita a construção do grafo de execução do programa e o mesmo é simulado para o modelo de máquina desejado. Os resultados da simulação são usados então para a análise/predição de desempenho do programa.

Com este tipo de metodologia evita-se uma medição invasiva no programa pois como as medidas são tomadas pela simulação do grafo de execução acabam por não sofrerem interferência do simulador. Além disso, a construção do modelo do programa através da geração de seu grafo de execução faz com que modelos de programas complexos possam ser obtidos de forma automática, sem interferência do projetista.

## 1.5 Descrição do texto

Para um bom entendimento dos problemas envolvidos com o desenvolvimento da metodologia aqui proposta, dedica-se o Capítulo Dois à descrição de conceitos preliminares sobre paralelismo, medidas de desempenho e desempenho em sistemas paralelos. Em seguida faz-se uma revisão sobre trabalhos desenvolvidos na área para situar o leitor no contexto deste tema.

No Capítulo Três é feita uma descrição da metodologia proposta, com detalhes sobre a sua formulação e justificativas da abordagem escolhida. Também se descreve a implementação de um protótipo construído para verificar se esta metodologia é capaz de produzir os resultados esperados para predição e análise de desempenho.

Os resultados dos testes de validação são apresentados no Capítulo Quatro. Nele aparecem também "benchmarks" realizados com o programa e a máquina reais, o que possibilita uma comparação de precisão do protótipo. Também são apresentadas nesse capítulo simulações feitas para testar o efeito da variação de granulação do programa original sobre seu desempenho.

A partir dos resultados obtidos com o protótipo é apresentado no Capítulo Cinco uma análise sobre a viabilidade de aplicação da metodologia para a construção de uma ferramenta completa para análise de desempenho, aperfeiçoando-se o presente protótipo. Como um complemento aparecem ainda algumas perspectivas para trabalho futuro na área de análise

de desempenho.

Por fim, existem três apêndices para a descrição mais detalhada do caso estudado no Capítulo Quatro. Primeiro é feita uma descrição do CPS, que é o ambiente de programação paralela utilizada pelo programa testado. Depois, faz-se uma breve descrição sobre detetores de partículas e de programas usados para a reconstrução de eventos em experimentos de física de altas energias, uma vez que foi exatamente um desses programas o utilizado para testes. O último apêndice descreve o processador MIPS R3000, que é o processador usado nas máquinas paralelas em que se executou o programa aqui testado.

# Capítulo 2

# Métodos de predição e análise de desempenho

Em qualquer sistema, computacional ou não, o desempenho é um fator que deve ser buscado durante todo o seu desenvolvimento. Mas nem sempre isso ocorre, muitas vezes o projetista não se preocupa em fazer com que seu produto tenha um desempenho ótimo, bastando-lhe que esse desempenho não seja sofrível (em alguns casos nem isso é verdade). Quando se fala em programas de computadores, um melhor ou pior desempenho significa menos ou mais dinheiro gasto com manutenção de profissionais e máquinas na execução do serviço para o qual o programa foi desenvolvido.

Na realidade esse custo é proporcional ao tempo necessário para executar a tarefa e ao custo do ambiente em que se fará o processamento. Dessa forma, um programa de desempenho ruim representa mais tempo para o fornecimento das respostas ou a exigência de máquinas mais potentes e potencialmente mais caras ou ainda ambos. Desse modo, a busca pelo desempenho ótimo é crucial para programas que são muito utilizados ou que devem ser executados durante muito tempo a cada ativação. Programas executando em sistemas paralelos e/ou distribuídos pertencem a essa categoria pois, em geral, são programas com uma carga computacional bastante elevada.

Dada a importância de obter-se o melhor desempenho possível para um determinado programa, é necessário que esse desempenho seja definido e medido de alguma forma. Logo, para que o projetista tenha condições de melhorar seu programa, alguém (ou algo) tem de lhe fornecer dados que balizem o trabalho. Tais dados podem vir de uma ferramenta para análise de desempenho ou de um especialista humano na área. A segunda solução é menos encontrada porque tais profissionais são raros e é mais fácil manter dois projetistas que também

saibam utilizar uma ferramenta de análise do que apenas um projetista e um especialista em desempenho. Diante disso, normalmente são usadas ferramentas para medição e análise de desempenho.

O projeto dessas ferramentas envolve questões bastante diversas. Como a ferramenta será usada por pessoas que não são especialistas na área, ela tem de ser fácil de usar, o que significa que deve dar ao usuário a oportunidade de medir o que desejar e, ainda mais, ter uma visão simplificada da resposta que for interessante. Atender esses objetivos é uma tarefa bastante complexa, principalmente porque são muitas as medidas de desempenho possíveis e na maioria das vezes o usuário está interessado em detalhes específicos do sistema, o que dificulta a coleta dessas medidas. Da mesma forma, se existe uma grande quantidade de medidas é difícil estabelecer critérios para apresentar os resultados de modo simplificado.

Outro problema a ser resolvido é a forma como serão obtidas as medidas necessárias para o usuário. A instrumentação dessas medidas é uma tarefa bastante complexa, ainda sem uma solução que possa ser considerada definitiva. Durante este capítulo serão examinadas diversas maneiras de realizar medições no conjunto programa-máquina, com indicação de suas capacidades e problemas. Serão apresentadas em conjunto algumas ferramentas para análise/estimativa de desempenho existentes.

## 2.1 Conceitos fundamentais

Antes de descrever ferramentas para análise de desempenho é necessário que alguns conceitos fundamentais sejam apresentados para que o texto fique mais claro e completo. Assim, nessa seção serão abordados os conceitos sobre medidas de desempenho, para que se determine quais possam interessar dentro de um sistema computacional e quais medições devem ser feitas para obtê-las. Além disso, será feita uma rápida discussão sobre sistemas de computação paralela e distribuída, examinando-se as medidas de desempenho desses sistemas e como algumas de suas características as influenciam.

### 2.1.1 Medidas de desempenho

Para que seja possível fazer qualquer análise de desempenho de um programa é preciso que se tenham medidas sobre como será o seu comportamento no sistema para o qual está sendo projetado. Mesmo quando se quer apenas uma estimativa de desempenho, algum tipo de medida tem de ser utilizada. Conseqüentemente, as ferramentas usadas para análise/estimativa de desempenho devem incluir algum mecanismo de instrumentação que

forneça tais medidas. De modo simplificado, as medições podem ser divididas em teóricas, quando são criadas por modelos analíticos, e experimentais, quando são realizadas durante a execução do programa.

Medidas de desempenho geradas através de modelos analíticos apresentam como vantagem fundamental a simplicidade da realização das medidas, uma vez que na realidade existem apenas estimativas dos tempos de execução nos vários trechos de um programa. Essa simplicidade, no entanto, faz com que os resultados tenham sua precisão dependente da qualidade das estimativas. Se estas forem precisas, então o modelo analítico resultante também será preciso. Mas, se as estimativas não forem precisas ou as relações entre elas não forem adequadas, então o resultado será provavelmente ruim. Apesar de seus problemas de precisão, esse tipo de medição é muito útil quando ainda não existe código para ser executado experimentalmente.

Quando se tem o código do programa disponível, existem várias técnicas para a obtenção das medidas experimentais, com farta literatura descrevendo-as, analisando-as e criticando-as [17, 26, 46, 49, 50]. Os autores desses textos procuram classificar cada técnica segundo a estratégia usada para a instrumentação da medida, porém não existe uniformidade nas escalas usadas nessas classificações. Neste texto será usada a classificação dada por Pierce e Mudge em [46], descrita a seguir.

1. **Monitoração por "hardware"**, usando analisadores lógicos diretamente acoplados ao(s) processador(es) ou "hardware" especializado para gravação dos eventos ocorridos nos barramentos do sistema. Embora possam obter medidas precisas essas ferramentas não são adequadas para o desenvolvimento de ferramentas de análise de desempenho devido ao alto custo envolvido em sua implementação e pela exigência de pessoal especializado no "hardware" utilizado.
2. **Monitoração pelo sistema operacional**, usando interrupções do sistema para a gravação de informações sobre a execução. Trata-se de uma técnica primitiva, precisa, mas muito lenta pelo elevado número de interrupções geradas durante as medições.
3. **Modificação do código fonte**, através da inserção de comandos especiais no código fonte, os quais farão as medições desejadas. Métodos dessa classe sofrem por serem invasivos (alterando as medidas obtidas) e também relativamente imprecisos.
4. **Modificação do código objeto**, inserindo os comandos para as medições no código objeto. Apresentam os mesmos problemas encontrados com a modificação do código fonte.

5. **Modificação do código executável**, através da inserção de comandos já no executável. Também apresentam os problemas de serem invasivos e imprecisos.

Examinando essa classificação percebe-se que as duas primeiras técnicas não devem ser usadas para fazer as medidas de desempenho de programas. A primeira delas é excelente se o objetivo for o desenvolvimento da arquitetura da máquina que estiver sendo medida, com aplicação imediata também no desenvolvimento de compiladores para essas máquinas, uma vez que viabilizam dados sobre a ocupação de registradores, falta de dados em "cache", movimentações com a memória, etc., os quais são necessários para o desenvolvimento desses sistemas. Já a segunda categoria apenas aparece para ilustrar a evolução da técnica ali utilizada para as técnicas relacionadas com a modificação de código.

Já as técnicas baseadas na modificação do código, independentemente do momento em que é realizada essa modificação, são bastante utilizadas e fornecem informações para a maior parte das ferramentas de análise/predição de desempenho. A importância de tais técnicas pode ser notada a partir de trabalhos como o de Eustace e Srivastava [57], que apresenta uma ferramenta para construir ferramentas de medição que seriam inseridas no código fonte do programa a ser analisado.

Técnicas de modificação de código podem ainda ser classificadas segundo o tipo de informação resultante da medição e a forma como ela é realizada. Reed [49] descreve quatro categorias de técnicas de medição por modificação do código, a saber, "profiling", contagem de eventos, medição de intervalos de tempo e extração de traços dos eventos.

- A técnica mais usada é a de **"profiling"**, a qual já era indicada desde o início da década de 70, inclusive como parte integrante de sistemas de depuração de código [54]. A idéia principal no funcionamento de ferramentas desse tipo é obter informações sobre quanto cada trecho do programa ocupa do tempo total de execução. Os resultados são apresentados geralmente na forma de tabelas de funções, com estatísticas do uso de cada função (histograma). De posse dessas informações é possível fazer a detecção de pontos críticos de desempenho.

  "Profilers" obtêm os dados sobre o uso do processador através da amostragem do contador de programas em intervalos fixos. Isso reduz a quantidade de informações que precisam ser armazenadas mas causa sérios problemas quanto à precisão das medidas realizadas, pois o intervalo de amostragem fica, em geral, entre 10 e 20 milissegundos, o que é um tempo exageradamente grande se existirem funções muito pequenas mas que sejam muito executadas. Além disso, existem problemas de precisão oriundos de falhas conceituais na implementação dessas ferramentas, principalmente no tratamento

de chamadas recursivas de funções, como é indicado por Ponder e Fateman em [48]. Apesar desses problemas, ferramentas como o gprof [26], dpat, prof e jprof [50] entre outras, são utilizadas para medidas de desempenho de programas.

- Um método mais preciso, por eliminar a necessidade de amostragem, porém mais invasivo, é o de **contagem de eventos**, cujo funcionamento está baseado em modificar o código para que sejam contadas as ocorrências de determinados eventos de interesse (chamadas de uma rotina por exemplo). Isso significa um acréscimo no tempo de execução do programa e, também, das instruções que estiverem sendo "contadas". Essa carga computacional adicionada e o excessivo volume de dados gerado acabam por fazer com que essa técnica não seja muito usada.
- Uma variação pouco usada de "profiling" é a **medição de intervalos de tempos**, que troca as amostragens do primeiro por chamadas para medir o relógio do sistema inseridas no código do programa. Acumulando os tempos medidos pode-se gerar os mesmos resultados estatísticos. Os problemas dessa técnica são a precisão do relógio do sistema, que é atualizado numa baixa freqüência, e a possibilidade de se medir intervalos espúrios em sistemas com compartilhamento de tempo da cpu.
- Finalmente, outra técnica bastante utilizada é a **extração de traços dos eventos** - "event tracing" - presente em ferramentas como IDTrace [46] e ALPES [37], por exemplo. É a técnica que fornece os resultados mais detalhados, pois baseia-se em registros datados da ocorrência de cada evento no sistema. Infelizmente, paga-se um preço muito alto por essa precisão, pois o volume de dados é demasiadamente grande pela alta freqüência em que eventos ocorrem. A necessidade de se datar a ocorrência de cada evento também faz com que o código seja modificado, alterando então o tempo de execução. Essa alteração pode ser controlada na apresentação do resultado final se as marcas de tempo introduzidas em cada evento puderem ser alteradas para descontar o tempo gasto em sua manipulação. Isso porém é difícil de obter.

Das técnicas de instrumentação de código indicadas, tem-se que "profilers" e "event tracing" são as mais usadas nas ferramentas de análise/predição de desempenho existentes. Como dito anteriormente algumas são aplicadas nos estágios iniciais de compilação enquanto outras alteram o código já pronto ou quase pronto para execução. Existentes diversas propostas para esse tipo de instrumentação, algumas já integrantes de sistemas comercias, como pixie [43], criado por Earl Killian para sistemas MIPS, prof e gprof, disponíveis em sistemas UNIX. Várias outras têm sido propostas, principalmente fazendo a instrumentação no código executável, tais como *nixie* e *epoxie* [67], desenvolvidas pela Digital, *qp* e *qpt* [38], desenvolvidas

na Universidade de Wisconsin, ou ainda *jprof*, pela Mentor Graphics.

As propostas indicadas no parágrafo anterior não serão detalhadas pois a base delas é comum, isso é, todas obtêm seus dados a partir da instrumentação do código do programa. O objetivo da discussão realizada é indicar os principais problemas envolvidos para a obtenção de medidas para analisar o desempenho de qualquer programa, paralelo ou não. Para sistemas paralelos surgem outros problemas pois as medidas de interesse também são diversas. A seguir são apresentados os conceitos essenciais para a definição das medidas de interesse para a estimativa/análise de desempenho de programas.

### 2.1.2 Predição e análise de desempenho

Predizer e analisar o desempenho de um programa são atividades distintas, embora sejam usadas com objetivos bastante semelhantes. A predição de desempenho se preocupa basicamente em fornecer ao usuário interessado um valor que indique se o desempenho esperado para o programa é adequado ou não. Já a análise de desempenho busca dar informações que permitam ao usuário alterar o seu programa para atingir um melhor desempenho. Com isso, as medidas necessárias em cada caso são diferentes.

Na prática as medidas usadas para análise de desempenho devem ser mais detalhadas do que as usadas para a predição. Isso porque quando se busca refinar o desempenho do programa tornam-se necessários detalhes que podem ser omitidos quando o objetivo for simplesmente saber se o rendimento é adequado. Na Tabela 2.1, a seguir, são indicadas algumas das diferentes medidas de desempenho e o uso das mesmas em predição e análise de programas.

Da tabela nota-se que as medidas de interesse para predição de desempenho são

| Medida | Uso | Significado |
|---|---|---|
| Carga no sistema | análise | carga computacional sobre o sistema |
| Espera por entrada/saída | análise | tempo de bloqueio em entrada/saída |
| Espera por sincronismo | análise | tempo de bloqueio em sincronismo |
| Faltas de página | predição/análise | número de páginas buscadas em disco |
| Tempo de processamento | predição/análise | tempo para executar o programa |
| Tempo de relógio | predição/análise | tempo entre início e fim do programa |
| Tempo em comunicação | predição/análise | tempo gasto com tarefas de comunicação |
| Tempo por tarefa | análise | execução em cada tarefa do programa |

**Tabela 2.1: Medidas de desempenho e seu uso.**

simples e provavelmente mais fáceis de serem obtidas. De fato, se tanto o programa como a máquina que irá executá-lo estiverem disponíveis, as medidas relacionadas na tabela são imediatas. Infelizmente, quando o objetivo é estimar o desempenho, nem sempre a máquina e o programa estão disponíveis ou ainda o custo de tomar essas medidas diretamente na máquina pode ser proibitivo.

As medidas indicadas apenas para a análise de desempenho podem ser obtidas através dos mecanismos indicados na seção 2.1.1. Com alguma adaptação pode-se aplicar os mesmos mecanismos na obtenção das medidas de interesse para predição, as quais podem ser menos precisas. Mas se o objetivo for a análise de desempenho os dados devem ser precisos (quando vindos de um modelo analítico) ou em grande quantidade (se obtidos experimentalmente) para que o usuário possa ter confiança nos resultados, o que dificulta sua obtenção. A seção 2.3 mostra como algumas ferramentas de análise e predição de desempenho conseguem as informações necessárias.

Deve-se antecipar que toda essa argumentação é válida quando se fala de sistemas com processador único, mesmo se este for compartilhado. Quando o programa em desenvolvimento for executado em sistemas multiprocessados, além das medidas indicadas, a métrica de desempenho deve adotar outra filosofia, procurando inserir nas medidas o fato do sistema não obedecer mais aos critérios seqüenciais existentes em sistemas com um processador. Surgem então as medidas de desempenho de sistemas paralelos descritas a seguir.

### 2.1.3 Medidas de desempenho em sistemas paralelos

Para sistemas paralelos[1] a principal medida de desempenho é o **ganho de velocidade** ("speedup"), ou seja, quão mais rápido é um programa executando em paralelo em relação a um programa de referência. A partir desse conceito uma série de diferentes abordagens na definição desse número foi criada. A primeira das abordagens é a **Lei de Amdahl** [9], que procura determinar qual o máximo ganho de velocidade possível considerando o potencial de paralelismo existente no programa.

A argumentação de Amdahl procurava demonstrar que o ganho de velocidade não seria infinito mesmo se existissem infinitos processadores em paralelo. Na realidade, o ganho seria proporcional à razão entre as componentes paralela e serial do programa, como mostra a equação 2.1:

[1] Aqui o termo "sistemas paralelos" engloba desde máquinas massivamente paralelas até redes de estações de trabalho executando um programa paralelo.

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

onde $N$ é o número de processadores em paralelo, $s$ representa a porção serial do programa e $p$ sua porção paralela, com $s + p = 1$. Pela Lei de Amdahl quando o número de processadores tender a infinito, o ganho será no máximo de $1/s$. Essa conclusão é correta se se assumir que o tamanho do problema não varia com o número de processadores. Esse fato faz com que aparentemente todo programa tenha um grau de paralelismo bastante baixo, o que não é verdade. Por isso a equação 2.1 é conhecida como "speedup" para tamanho fixo. Uma forma de explicar esse problema foi apresentada por Gustafson [27], que sugere manter o tempo de execução fixo e não o tamanho do problema. Para que o tempo de execução paralela se mantenha fixo é necessário aumentar o tamanho do problema quando se aumenta o tamanho da máquina. Assim, o ganho de velocidade é calculado em relação ao tempo necessário para se executar o novo problema seqüencialmente. A equação 2.2 apresenta a nova formulação, conhecida como "speedup de tempo fixo" ou escalonado:

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

O objetivo principal do "speedup" de tempo fixo é medir o desempenho de um programa paralelo partindo do princípio de que se existirem mais processadores então o problema a ser resolvido será proporcionalmente maior. Quando um algoritmo ou máquina permitem que isso seja obtido, então se diz que ele é **escalável**. A medida de escalabilidade de um sistema é importante para que se saiba se é vantajoso aumentar o número de processadores de um determinado sistema paralelo. Quando ela for alta, o sistema pode ter mais elementos de processamento e, conseqüentemente, resolver problemas maiores. Se ela for baixa, dá uma indicação do limite de expansão da máquina e do problema.

Existem algumas variações sobre a forma de se calcular o "speedup", considerando memória constante [32, 60], valores assintóticos, valores absolutos e valores relativos de "speedup" [52], por exemplo. Todas essas medidas têm como objetivo verificar se o desempenho atingido pelo sistema é favorável ou não. Como o "speedup" pode ser medido de várias maneiras, é possível escolher aquela que forneça o melhor resultado de escalabilidade.

Alpern e Carter [7] fazem uma interessante discussão sobre o perigo de não existirem padrões bem definidos para a escolha e apresentação dos resultados de escalabilidade. Com a falta de padrão e a importância talvez exagerada que é dada para a escalabilidade, eles dizem que resultados podem ser distorcidos, na realidade apresentados segundo a métrica que for mais interessante, para que pareçam bastante bons.

Na mesma linha de raciocínio, Sahni [52] advoga que a métrica que se deve ter sempre em mente é o tempo de execução. Para ele não importa se o algoritmo é ou não escalável, importa se o programa executa ou não mais rápido. Sua argumentação é válida do ponto de vista do usuário do programa paralelo, o qual prefere, é claro, resultados mais rápidos. Já do ponto de vista do administrador do sistema (ou de quem paga por ele) nem sempre tempo de execução menor significa um melhor desempenho do sistema. Segundo essa ótica o que se procura é ter velocidade adequada com o menor custo de processamento, num casamento perfeito entre necessidade e disponibilidade.

É a partir dessa situação que outras medidas de desempenho são formuladas para sistemas paralelos. Hwang reúne em seu livro sobre arquiteturas avançadas de computador ([32]-capítulo 3) várias dessas medidas de desempenho em sistemas paralelos. As equações 2.3, 2.4, 2.5 e 2.6 a seguir mostram como esses valores podem ser calculados a partir do "speedup" relativo e do número de operações realizadas no sistema, assumindo que $O(n)$ é o número total de operações realizada num sistema com $n$ processadores e que $T(n)$ é o tempo de execução em unidades arbitrárias de tempo. Além disso, $T(1)$ é o tempo de execução do mesmo problema usando apenas um processador. Assume-se ainda que $T(1) = O(1)$, e que o "speedup" $S(n)$ é igual a $T(1)/T(n)$.

- **Eficiência**

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

  A eficiência de um programa executando numa máquina paralela indica se o sistema está sendo bem aproveitado para resolver o problema. Esse aproveitamento dá uma medida da escalabilidade do programa, ou seja, se ele for escalável, então seu ganho de velocidade com $n$ processadores deve ser muito próximo de $n$, ou eficiência próxima de um.

- **Redundância**

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

  A redundância de um sistema é uma indicação do ajuste existente entre o paralelismo do programa e o paralelismo da máquina. Quanto menor a redundância melhor esse ajuste, indicando que não existe uma sobrecarga de computação ao se explorar o paralelismo.

- **Utilização**

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

Compondo-se as medidas de redundância e eficiência tem-se uma medida mais significante da real ocupação do sistema. A utilização representa quanto dos recursos do sistema foram mantidos em uso. Assim, sistemas com boa utilização são extremamente desejáveis para quem paga por seu uso.

- **Qualidade de paralelismo**

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

Uma última composição é dada pela qualidade de paralelismo, que apresenta melhores valores quando a eficiência e o "speedup" do sistema são maiores e a redundância menor. Portanto, a qualidade de paralelismo pode ser entendida como uma medida global da vantagem obtida com a paralelização de um programa.

Finalizando, as medidas de desempenho em sistemas paralelos podem ser vistas de dois pontos de vista diferentes: o do usuário e o do provedor do sistema. Ao primeiro interessa basicamente a aceleração obtida com o programa em paralelo, os seja, alguma medida de "speedup". Ao usuário interessa também, se for possível, obter uma boa medida de eficiência, mas como seu interesse maior é quanto ao tempo necessário para fazer seu processamento essa medida não é essencial.

Sob a ótica do provedor do sistema paralelo o tempo gasto para executar a tarefa não é essencial, exceto se o usuário estiver escolhendo quem lhe prestará o serviço. Ao provedor interessa saber se o sistema está sendo usado de modo eficiente e se não existe desperdício de recursos. Desse modo as medidas de seu interesse passam a ser as de eficiência, redundância, utilização e qualidade.

## 2.2 Parâmetros de perturbação em ambientes multiprocessados

Durante a seção anterior foram apresentados os conceitos de predição/análise de desempenho de programas. Também indicou-se que para sistemas paralelos várias das métricas usadas - e comprovadamente eficazes - em sistemas com apenas um processador deixam de ser válidas, o que faz com que esses sistemas tenham um tratamento diferenciado durante a análise de desempenho. Também foram definidas diversas métricas de desempenho de um sistema paralelo, tais como "speedup", eficiência e redundância, por exemplo. Falta, entretanto, enunciar mais claramente os principais fatores de perturbação de desempenho em sistemas paralelos e como ocorrem essas perturbações.

Sobre os três fatores discutidos a seguir pode-se dizer que a relação entre os tempos consumidos com comunicação e processamento efetivo é uma característica mais ligada ao problema que está sendo programado, enquanto o dimensionamento da granulação do programa é orientada pelas características da máquina em que ocorrerá o processamento, ficando então o tráfego no canal de comunicação determinada pelo relacionamento entre programa e máquina. A seguir tem-se uma descrição mais detalhada desses fatores.

### 2.2.1 Relação comunicação X processamento

A relação entre os tempos consumidos com a comunicação entre tarefas e com o meio exterior e o tempo efetivo de processamento afeta o desempenho de um sistema paralelo na exata proporção prevista pela Lei de Amdahl. Isso quer dizer que para sistemas paralelos o processamento de comunicação é uma forma de seqüenciamento das atividades que podem ser executadas em paralelo. Embora não existam restrições sobre a realização paralela das interações entre tarefas, deve ficar claro que o acesso às informações é feito na maioria das vezes de modo estritamente seqüencial, como nos modelos EREW, ERCW e CREW (ver [32] por ex.).

Como as operações seqüenciais impõem um limite superior no grau de paralelismo que pode ser obtido com um determinado programa, fica fácil de perceber que quanto mais tempo um programa gasta com comunicação, menor será o seu grau de paralelismo e, comparativamente, pior será o seu desempenho em sistemas paralelos que exijam acesso exclusivo aos canais de comunicação. A equação 2.1 mostra claramente que se a porção seqüencial de um programa for relativamente alta, então o "speedup" terá um crescimento pequeno comparativamente ao crescimento no grau de paralelismo da máquina.

Apesar dessa lei não ser válida do ponto de vista da escalabilidade de um sistema, tem-se que o processo de comunicação tem um comportamento próximo de linear com o crescimento do tamanho do problema, isso é, ao aumentar-se o volume de processamento também se aumenta o volume de comunicação dentro do programa. Se parte dessa comunicação tiver que ser feita com o acesso exclusivo aos canais de comunicação, tem-se que a parte serial do problema não é proporcionalmente reduzida, diminuindo a escalabilidade do mesmo.

A Figura 2.1 ilustra como a relação entre comunicação e processamento afeta o "speedup" e conseqüentemente o desempenho de um sistema paralelo. Nessa figura o eixo horizontal representa o número de processadores **N** no sistema e o vertical o "speedup" obtido. As curvas **A**, **B** e **C** mostradas possuem valores diferentes para a relação comunicação/processamento, sendo $A < B < C$.

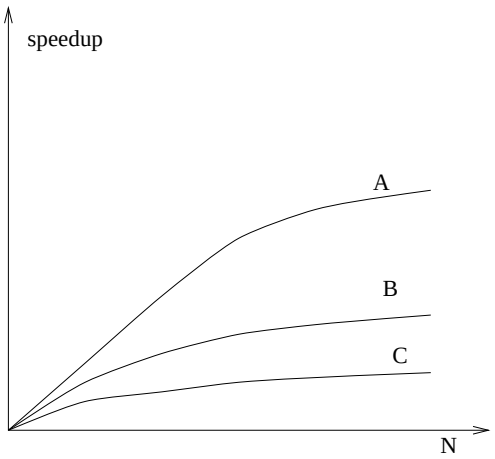



**Figura 2.1: Influência da comunicação no "speedup".**

### 2.2.2 Dimensionamento da granulação

O conceito de granulação em sistemas paralelos é usado para dar uma medida do grau de paralelismo que pode ser atingido na execução de um programa. Assim, cada programa pode ser paralelizado com diferentes tamanhos de grão[2], que são definidos segundo características do programa e orientados pelo tipo de máquina a ser utilizada. De modo resumido, a Tabela 2.2 define tamanhos de grão em programas paralelos, com indicações sobre a escala de instruções que podem ser agrupadas:

| Nível | Tamanho do grão | Tipo de instruções agrupadas | Instruções por grão |
|---|---|---|---|
| 1 | fino | instruções ou comandos | até 20 |
| 2 | fino | ciclos não recursivos | até 500 |
| 3 | médio | procedimentos ou tarefas | até 2000 |
| 4 | médio ou grosso | subprogramas | +2000 |
| 5 | grosso | programas | +2000 |

**Tabela 2.2: Níveis de granulação em programas paralelos**

A escolha por um ou outro tamanho de grão depende do tipo de problema que está sendo resolvido, da arquitetura do sistema onde o programa será executado e da disponibilidade de compiladores ou bibliotecas especiais para paralelização. Essa escolha vai afetar em muito o desempenho atingido por um programa pois tanto o grau máximo de paralelismo quanto o volume de comunicação entre processadores dependem do tamanho do grão.

[2]Grão é o agrupamento de diversas instruções seqüenciais em um único bloco que pode ser executado em paralelo com outros blocos.

Para a determinação do grau de paralelismo, tem-se que quanto maior o grão, menor é o grau de paralelismo, uma vez que, por exemplo, o número de subprogramas (grão de nível quatro) de um determinado problema que podem ser executados em paralelo é claramente menor que o número de instruções de máquina (nível um) que podem ser paralelizadas.

O desempenho obtido com cada granulação depende da arquitetura da máquina, pois esse é um fator preponderante na capacidade de comunicação entre os vários processadores. Isso ocorre porque o tamanho do grão é inversamente proporcional ao volume de comunicação no sistema, isto é, quanto menor for o grão mais dados terão que ser trocados entre cada processador. Assim, se uma determinada máquina possuir uma arquitetura que permita uma rápida troca de dados, então a granulação poderá ser do nível de instruções sem que o desempenho se deteriore de modo acentuado, como ocorreria em uma máquina com baixa taxa de transmissão de dados.

Para o projetista de programas, a variação de granulação segundo a arquitetura da máquina faz com que a adaptação de um programa de um sistema para outro tenha que ser acompanhada de uma análise sobre o tamanho ideal do grão para que o desempenho possa ser mantido ou até mesmo melhorado. Isso significa que a paralelização do programa deve ser refeita para atender ao modelo computacional indicado para a nova arquitetura.

Dessa forma percebe-se claramente que o desempenho de um programa paralelo pode ser modificado ao variar-se o tamanho do grão. Portanto a análise de desempenho também deve levar em conta essa possibilidade, verificando como a execução de um programa varia para diferentes tamanhos de grão.

### 2.2.3 Tráfego no canal de comunicação

A discussão dos outros dois fatores de perturbação do desempenho em sistemas paralelos tornou explícita a importância da comunicação para a determinação desse desempenho. Em linhas gerais, quanto mais comunicação existir pior será o desempenho, mas a definição sobre qual volume de comunicação deve ser considerado alto e qual é baixo depende de condições relativas aos métodos e canais de comunicação usados no sistema ou, em última análise, das condições de tráfego no canal de comunicação.

Do gráfico apresentado na Figura 2.1 tem-se que o desempenho de um programa paralelo piora quando a relação entre comunicação e processamento aumenta. Se for considerado um volume de processamento constante em cada uma das curvas daquela figura, então pode-se concluir que o que faz a relação aumentar é exatamente o volume de comunicação. Desse modo, quanto maior o volume de comunicação em um dado programa pior será o seu

desempenho.

Em 2.2.1 mostrou-se que esse pior desempenho era conseqüência direta da Lei de Amdahl, usando para tanto o conceito de que parte da comunicação tem que ser feita seqüencialmente. Entretanto, apenas esse conceito é insuficiente para mostrar a relação entre o tamanho do grão e o desempenho de um programa paralelo. Nesse caso, o que vai determinar realmente o desempenho é a associação entre a necessidade de acesso exclusivo ao canal de comunicação e o volume de comunicação existente.

Também já se mostrou que é necessário garantir o acesso exclusivo às informações nos modelos EREW, ERCW e CREW. Para o modelo CRCW não se faz nenhuma restrição quanto ao acesso à memória, entretanto mesmo nesse modelo tem-se que fazer algumas restrições de exclusão mútua para o acesso a dispositivos físicos, que tanto podem ser discos como canais de comunicação. Desse modo, qualquer um dos modelos de computação paralela existentes apresenta em algum momento a necessidade de acesso restrito. Nesses momentos o que vai determinar o desempenho do programa é o volume de carga concorrente.

O caso mais crítico do ponto de vista de paralelismo é o acesso aos canais de comunicação. O tráfego nesses canais vai determinar em última instância qual a dinâmica da porção serial dos programas em execução. Um aumento no tráfego no canal possivelmente representa uma diminuição na velocidade de comunicação. Isso implica no aumento da porção serial do programa executado e, portanto, uma diminuição no valor do "speedup" do sistema pela lei de Amdahl.

## 2.3 Métodos para predição e/ou análise de desempenho

Tendo examinado os conceitos básicos sobre medidas de desempenho, tanto em sistemas com um processador quanto em sistemas paralelos, é necessário um estudo sobre o estado da arte em análise e estimativa de desempenho de sistemas. Nas próximas páginas apresenta-se uma revisão dos trabalhos divulgados na área, com uma crítica sobre suas qualidades e deficiências, em especial sobre a metodologia adotada em cada proposta para obter as medidas necessárias para análise/predição.

Nessa revisão os trabalhos estão divididos segundo a abordagem usada para a obtenção dos resultados do desempenho, que são: métodos analíticos, métodos baseados em "benchmarking" e métodos baseados em simulação. Dentro dessas categorias ainda é possível

distinguir os métodos segundo a forma como é feita a instrumentação dos dados a serem analisados. Em vários dos métodos o que se usa são dados obtidos a partir da modificação do programa através de "profilers" ou de extração de traços de eventos. Em ambos os casos pode ser questionado se a informação obtida a partir de uma execução de um programa reflete outras execuções do mesmo programa. Isso é verdade em várias situações como indica Wall em [66] ao analisar a precisão de diversas formas de "profiling". Segundo Wall, nas poucas vezes em que os dados obtidos por "profiling" não representam o programa adequadamente é possível alterar a forma em que os parâmetros do programa são calculados e, conseqüentemente, melhorar a sua precisão.

Deve-se salientar que, apesar de abrangente, essa revisão não apresenta todos os trabalhos na área, mesmo porque muitos deles diferem apenas em questões superficiais, sem alterações conceituais sobre a metodologia usada. Desse modo, neste texto serão discutidos apenas alguns trabalhos em cada uma das categorias listadas acima, com citações para trabalhos similares a cada um deles. Também não será feita nenhuma distinção entre métodos usados para sistemas seqüenciais e métodos usados em sistemas paralelos, muito embora esteja claro que as técnicas usadas em sistemas seqüenciais não possam ser aplicadas de modo equivalente quando existir paralelismo. Quando for necessário apenas será indicado se um dado método é aplicado em sistemas paralelos ou seqüenciais.

### 2.3.1 Métodos analíticos

Dentro dessa categoria se colocam os métodos de análise e predição de desempenho com técnicas de medição baseadas em modelos analíticos, em que o uso de medições experimentais fica restrito a uma execução para 'profiling" ou muitas vezes nem ocorrem. Métodos analíticos utilizam-se de técnicas de análise fundamentadas essencialmente em modelos de redes de Petri e cadeias de Markov, nas quais o programa é definido como uma seqüência de estados.

A criação de um modelo analítico para um programa é uma tarefa árdua. Na maioria das vezes o programa é grande e complexo, em especial quando se trata de sistemas paralelos. Isso dificulta a criação de um modelo preciso para o programa, diminuindo a precisão dos resultados de desempenho que serão analisados. Apesar do problema de precisão, essa é a única técnica que permite a predição do desempenho de um programa em um sistema sem que se tenha o programa e o sistema disponíveis. Isso representa uma vantagem considerável nas primeiras fases de desenvolvimento de um projeto.

Nos métodos analíticos em uso existe uma característica comum, que é o uso de

probabilidades para decidir a mudança de um estado para outro. Tanto os modelos baseados em rede de Petri quanto os de cadeia de Markov usam a teoria de processos estocásticos de forma intensa pois a criação de um modelo determinístico é difícil e indesejável pela característica inerentemente aleatória da execução de um programa. Apenas alguns raros métodos usam modelos determinísticos.

Outro aspecto favorável aos métodos analíticos é a sua velocidade de resposta. Como esses métodos podem ser reduzidos em sistemas de equações, vários métodos de resolução podem ser aplicados de modo eficiente e elegante, desde que o caso em estudo não seja de porte elevado, quando o número de estados criados cresceria de forma exponencial.

A seguir são apresentados alguns métodos analíticos examinados:

### Modelo de rede de Petri estocástica generalizada (GSPN)

Como mencionado, uma das técnicas usadas em métodos analíticos faz uso de redes de Petri (PN daqui por diante). Uma das formas de se aplicar PN em análise de desempenho é através do uso de PN estocástica generalizada, as quais são uma das variações criadas a partir da definição básica de PN feita por Carl Petri em 1962. Como aqui a preocupação é com a aplicação e não com a teoria das PN, não será feita uma descrição detalhada da mesma, que pode ser encontrada em [45], por exemplo.

Entretanto, é necessário que se caracterize PN estocástica para que se possa correlacionar a mesma com análise de desempenho. Inicialmente tem-se que uma PN é um grafo bipartido composto de lugares, transições e um conjunto de arcos ligando lugares e transições. Os lugares podem conter "tokens", que passam de um lugar a outro através do disparo de alguma transição. Examinando-se o número de "tokens" a cada disparo de uma transição é possível determinar se uma PN é ilimitada (possui lugares ilimitados) e também sua alcançabilidade. Essa definição para PN ilimitada é útil para a aplicação de GSPN ilimitadas em análise de desempenho.

Prosseguindo com essas definições, uma GSPN é formalmente definida como uma quíntupla $(P, T, A, M, W)$, onde $P$ são lugares, $T$ transições, $A$ arcos e $M$ é um conjunto de marcas ("tokens"), definidos como em PN. Já $W$ é um conjunto de funções dependentes das marcações que mapeam transições em números reais não negativos, ou $W : T \times M \rightarrow R$. Além disso, as transições nesse tipo de rede são divididas em dois grupos: *imediatas e temporizadas*. Para as transições temporizadas as funções $W$ retornam a taxa média de disparo da transição, com tempos de disparo distribuídos exponencialmente. Já as transições imediatas disparam segundo uma probabilidade que depende da marcação atual determinada pelas funções $W$,

em tempo igual a zero.

Uma característica fundamental de GSPN's é que as mesmas são isomórficas às cadeias de Markov, podendo portanto serem tratadas de modo idêntico. Assim, Gandra, Drake e Gregorio [22] aproveitando-se dessa característica e das condições listadas a seguir, elaboram um modelo para GSPN ilimitada para análise de desempenho:

(a) Todos os lugares e transições são conectados por arcos únicos.

(b) A rede tem $n$ lugares ilimitados ($n \geq 1$).

(c) Nenhuma transição tem mais do que um lugar de saída ilimitado.

(d) As freqüências de disparo são independentes das marcações.

Essas condições permitem limitar a cadeia de Markov gerada através da GSPN ilimitada que as obedeça. Logo, o modelo para análise de desempenho pode ser obtido e resolvido de forma simples através do cálculo da distribuição de equilíbrio da cadeia de Markov equivalente. O processo de redução da cadeia de Markov, originalmente ilimitada para uma cadeia limitada, é feito através de um algoritmo de agregação que procura unir dois ou mais estados da cadeia em um único estado ("lumping").

A agregação pode ser **horizontal**, em que os estados podem ser unidos se e somente se eles diferirem somente no número não nulo de "tokens" contidos em seus lugares ilimitados. Esse tipo de agregação permite o cálculo de vários índices de desempenho. Uma segunda forma de agregação é a **vertical**, em que a união de estados da cadeia ocorre se e somente se esses estados obedeçam a regra para a agregação horizontal e ainda tenham o mesmo número de "tokens" em um dos estados ilimitados. A partir desse tipo de agregação é possível obter medidas mais refinadas de desempenho, como a quantidade de memória necessária para comunicação entre processos. O algoritmo para agregação, descrito em [22], não será discutido aqui, sendo necessário apenas uma discussão sobre seus resultados.

A aplicação desse método é feita através da criação de um modelo GSPN para o sistema em análise. Nesse modelo são especificadas as transições possíveis e suas probabilidades de disparo. Identificam-se os lugares ilimitados, que servirão de referência para o restante do processo. Na seqüência é gerada a cadeia de Markov equivalente e nela são aplicadas as regras de agregação horizontal e vertical, em momentos e com objetivos distintos. Para a cadeia resultante é resolvido um sistema de equações lineares (usando funções de probabilidades) cujos resultados são as medidas de equivalência desejadas.

Em [22] apresenta-se um exemplo para esse tipo de análise. No caso é feito um estudo sobre um sistema com dois processos do tipo leitor-escritor, que podem ser represen-

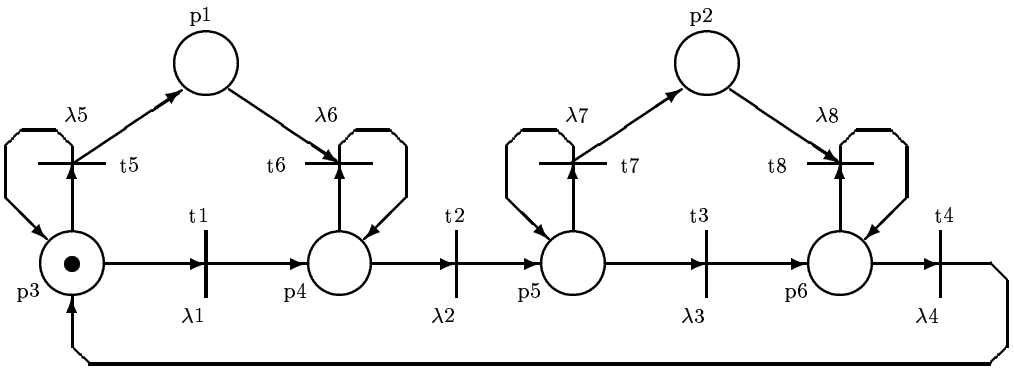



**Figura 2.2: GSPN ilimitada examinada por Gandra et alii.**

tados pela rede da Figura 2.2, com os lugares $p1$ e $p2$ ilimitados pela diferença de velocidade entre produção (transições $t5$ e $t7$) e consumo ($t6$ e $t8$) de mensagens causada pelas funções de distribuição de probabilidades $\lambda_i$. Aplicando-se os algoritmos de aglomeração horizontal chega-se até a cadeia de Markov apresentada na Figura 2.3, a qual pode ser resolvida de forma simples para a obtenção de informações como tempo de execução de um ciclo do sistema, intervalos entre eventos, tempo médio de execução de cada tarefa, etc., ou ainda necessidade de memória para transferência de mensagens (quando é aplicado o algoritmo de aglomeração vertical).

Quanto a resultados práticos, Gandra apresenta a análise de um programa para controle de um robô móvel. Para esse caso são apresentados os resultados obtidos para tempo médio por tarefa, tempo de ciclo do programa, tempo de execução em cada processador paralelo (o sistema é testado para até três processadores) e memória necessária para a sua execução. Não é feita nenhuma comparação com resultados do sistema real, o que impede a verificação da precisão do método. Mostra-se entretanto que com o mesmo se pode obter uma primeira medida de desempenho com bastante rapidez.

Em linhas semelhantes a esse trabalho podem ser citados os de Marsan, Balbo e Conte [42], que faz a avaliação de desempenho de sistemas multiprocessados usando GSPN's, ou ainda o de Zuberek [68], que usa PN's temporizadas ilimitadas. Um outro trabalho, contendo uma revisão bastante extensa da aplicação de PN's em análise de desempenho, é o de Browne e Adiga, que parte de GSPN's e GTPN's (PN's temporizadas) até chegar ao PCM (Parallel Computation Model) que procura construir uma estrutura comum para modelos baseados em grafos.

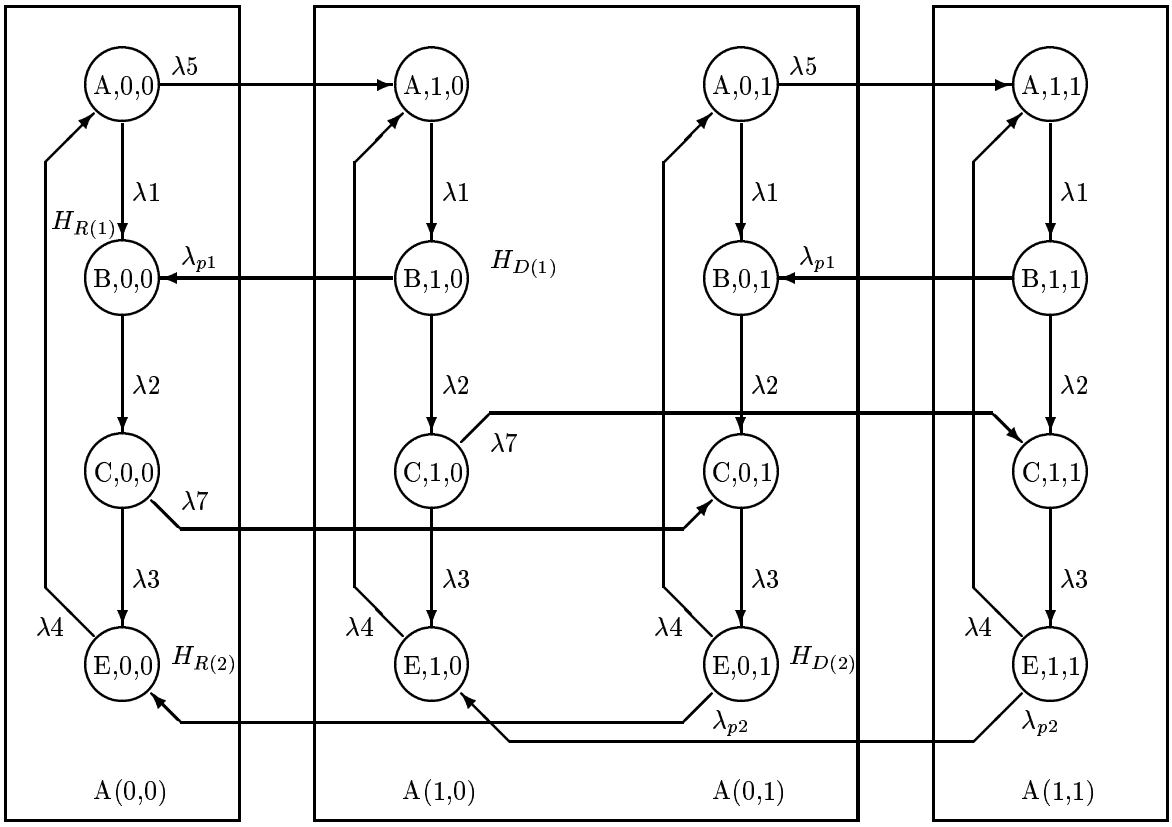



**Figura 2.3: Cadeia de Markov horizontal para o exemplo em Gandra et alli.**

### Modelo gerado em tempo de compilação

Um método analítico bem mais simples é obtido através da compilação das especificações do programa através de uma linguagem de descrição. O processo de análise é feito em duas etapas: uma etapa de geração das especificações e outra de obtenção de dados para análise. A etapa de geração de especificações consiste na escrita das especificações (em alguma linguagem de descrição) e sua compilação. Já a obtenção de dados pode ser feita diretamente pela compilação do programa modelado.

A maior vantagem dessa abordagem, quando comparada com métodos baseados em PN's por exemplo, é a relativa simplicidade do modelo inicial, que pode ser obtido através da escrita das especificações do programa em uma dada linguagem de descrição. Entretanto, é exatamente essa simplicidade que causa sua pequena precisão. Como esse tipo de método pode ser aplicado mesmo sem a existência de um protótipo para o programa, tem-se que a crítica a sua relativa imprecisão pode ser atenuada em favor do seu baixo custo e do bom nível de parametrização que pode ser obtido.

Existem várias técnicas diferentes para a obtenção da descrição do programa, assim como acontece com métodos baseados em redes de Petri, sendo que a principal diferença entre eles é o formato da linguagem de descrição, a qual pode ter uma orientação mais algébrica ou mais comportamental. Essa orientação é determinada pela linguagem de descrição usada. Além disso, a linguagem de descrição é responsável também pela precisão dos resultados obtidos na análise de desempenho, uma vez que a especificação das relações temporais entre tarefas (sincronismo e contenção de recursos) assim como as medidas de tempo de execução em cada trecho da tarefa é feita através da linguagem.

Do ponto de vista das relações entre tarefas uma linguagem pode assumir dois modelos para representação de paralelismo: série-paralelo e não-série-paralelo. O primeiro é o mais simples deles, no qual se pode reduzir o número de estados/trechos do programa até um único estado através de convoluções e produtos, o que não é possível para o segundo modelo [56].

Para o modelo série-paralelo van Gemund apresenta uma linguagem de descrição chamada PAMELA (**P**erform**A**nce **M**od**E**ling **LA**nguage) [24], na qual são definidos alguns operadores para a descrição de eventos estocásticos, calculados durante a fase de compilação, e de redutores série-paralelo.

A abordagem de van Gemund faz sua fundamentação formal através da linguagem de descrição, gerando modelos separados para o programa e a máquina, que são combinados em um modelo para o sistema completo. Esse modelo é então compilado em um modelo parametrizado de desempenho, que calcula o tempo esperado de execução do sistema com um custo computacional inferior ao de simulação do mesmo.

Em [24] são apresentados testes feitos com PAMELA, com indicações de uma razoável precisão para os valores de tempo de execução calculados em relação aos obtidos através de "benchmarking". No texto afirma-se que o erro está limitado em 50% do valor experimental, independentemente dos parâmetros de programa e máquina. Nos resultados do exemplo apresentado esse erro é ainda menor, o que indica a sua utilidade em etapas iniciais do desenvolvimento de um programa paralelo.

### Métodos baseados em Teoria de Filas

Uma outra forma de se obter valores aproximados para tempos de execução é através do uso de teoria de filas, que pode ser igualmente aplicada em modelos série-paralelo e não-série-paralelo. A vantagem de sua aplicação é a existência de todo um formalismo matemático para a análise. O problema é que a especificação das filas depende dos modelos

estocásticos usados e mais ainda tem sua aplicação limitada pela explosão no número de estados em sistemas mais complexos.

A análise com um modelo de filas é feita de forma bastante simples, bastando que o modelo paralelo (SP ou NSP) especifique as distribuições de probabilidade para o tempo de execução de cada trecho do programa. O cálculo dos tempos de execução para o sistema podem então ser realizados usando os modelos de filas adequados para cada caso. Tais modelos são conhecidos e de solução trivial, desconsiderando-se a explosão de estados é claro.

Uma forma de evitar a explosão de estados é reduzir o número de eventos do modelo paralelo através de algum modelo de aproximação. Sötz [56] apresenta um método de redução do número de fases série do modelo aproximando-os por dois estados, um com tempo de execução com distribuição exponencial e outro com tempo determinístico. Com isso, é possível reduzir significativamente o número de estados para análise.

Sötz realiza alguns testes com o modelo, com resultados consideráveis de precisão quando comparados com métodos baseados em redes de Petri e cadeias de Markov. Apesar de indicar que os resultados markovianos são exatos, sabe-se que eles também apresentam problemas de precisão em relação aos valores reais de execução. A vantagem é que com a aproximação realizada é possível fazer a análise de sistemas de maior porte (impossíveis com cadeias de Markov) e com razoável precisão.

Outros modelos de filas podem ser encontrados em [62] (por Thomasian e Bay) e em [23] (por Gelenbe e Liu). No trabalho de Thomasian e Bay usam-se apenas tempos de execução com distribuição exponencial para o modelo, que pode ser resolvido através de algum método de espaço de estados. O problema com esse tipo de modelo paralelo é que o comportamento da maior parte dos programas paralelos não apresenta a característica de distribuição exponencial.

Já o trabalho de Gelenbe e Liu faz a análise através de um modelo de filas simplificado onde os estados do sistema são identificados por um vetor com as tarefas em execução classificadas de acordo com o número de predecessores ainda não encerrados. Em [23] são apresentados alguns exemplos, com resultados comparados com os resultados obtidos por um simulador. A precisão do modelo simplificado se manteve inferior a 40% para todos os casos examinados.

### Métodos baseados em Cadeia de Markov

Métodos baseados em cadeias de Markov dominam as técnicas de análise de desempenho analíticas em conjunto com os modelos de rede de Petri. Em última análise, boa

parte dos modelos de rede de Petri e de filas podem ser transformados em cadeias de Markov. Como já foi apontado, a resolução de processos markovianos não é trivial para qualquer caso. Muitas vezes o número de estados da cadeia é elevado, o que inviabiliza a sua resolução analítica. Entretanto, como os métodos para essa resolução são precisos e matematicamente sólidos, o uso de cadeias de Markov (CM) permanece atrativo e com muitas variantes que procuram evitar a explosão do número de estados.

O princípio básico dos modelos markovianos é o mesmo que rege os de filas, isto é, a execução de um programa é dividida em um conjunto de etapas para as quais se definem recursos necessários e relações de precedência, usando para tanto funções de distribuição de probabilidade na descrição da ocorrência de cada etapa. Isso permite a geração de um conjunto de estados que o sistema pode assumir, conjunto esse que pode ou não ser infinito.

Para tornar a análise precisa é essencial que as funções de distribuição mantenham uma curta distância do sistema real. Logo, outra grande dificuldade para a implementação de ferramentas de análise de desempenho baseadas em CM é o dimensionamento adequado dessas funções para cada estado do modelo. Entretanto, assim como nos casos anteriores, a precisão pode ser minimizada quando a análise for feita em etapas iniciais do desenvolvimento do sistema.

Dentre os vários métodos de análise de desempenho baseados em CM's podem ser incluídos os que usam Markov em algum momento da análise, tal como o de Gandra et alii [22] que trabalha com GSPN ilimitadas, o de Kim e Shi [36], cujo modelo de filas é intrinsicamente ligado ao modelo markoviano, ou ainda o de Malony et alii [39], que usa um pacote para análise de modelos de grafos estocásticos, gerados a partir de especificações de modelos separados para a máquina, programa e sistema, usando a metodologia de três passos de Herzog [3].

Por outro lado existem métodos com base apenas em cadeias de Markov, usualmente com aproximações feitas para permitir o cálculo do processo markoviano sem que ocorra uma explosão no número de estados da cadeia. Nessa categoria aparecem, entre outros, o trabalho de Kapelnikov, Muntz e Ercegovac [34], o de Trivedi, et alii. [63] e o de Iazeolla e Marinuzzi [33].

Kapelnikov faz a redução da CM aplicando o teorema de Norton para a decomposição de redes fechadas. Embora Norton apenas possa ser aplicado para redes que sejam do tipo *forma-produto*, ele acaba sendo aproximado para a aplicação em outras redes, sem um grande prejuízo em sua precisão, que os autores indicam ser menor que 2% do valor obtido

[3]No Capítulo Três descreve-se a metodologia de Herzog com mais detalhes.

por simulação.

Já Trivedi usa cadeias de Markov com recompensa para fazer a aproximação de estados repetidos da cadeia original (possivelmente infinita). Em seu trabalho são apresentadas também várias outras possibilidades para análise de desempenho, usando redes de Petri, teoria de filas e cadeias de Markov comuns. Mas como sua preocupação era com a apresentação de técnicas de avaliação de desempenho não são indicadas medidas de comparação entre as mesmas.

Finalmente, o método de Iazeolla/Marinuzzi faz reduções na cadeia de Markov original através de manipulação de "strings", aglomeração e eliminação recursiva. Com isso são obtidas matrizes de blocos que podem ser resolvidas de forma eficiente pelo seu tamanho reduzido em relação ao tamanho inicial da cadeia de Markov. Os autores apenas apresentam dados que provam a relativa velocidade do método, em especial o fato de obter uma solução com complexidade proporcional ao tamanho da matriz de blocos reduzida. Não são apresentados testes sobre a precisão do método.

### Modelos determinísticos

Uma forma de evitar a complexidade dos modelos estocásticos quando aplicados em sistemas paralelos é assumir que tempos de execução e espera em pontos de comunicação e/ou sincronismo são constantes ao longo da execução do programa. Isto é equivalente a tornar o programa previsível em seu comportamento, ou seja determinístico, o que é possível apenas dentro de certos limites. A argumentação em favor desses modelos é que os modelos estocásticos apresentam problemas computacionais (explosão no número de estados) e que suas descrições raramente incluem dados de comparação usando programas reais, o que impede qualquer afirmação no sentido de que eles seriam a solução para o problema de análise de desempenho de programas paralelos.

Isso permite que se procurem modelos determinísticos para a realização da análise de desempenho, mesmo com o comprometimento da precisão ao assumir que os programas teriam sua execução sempre previsível. A grande vantagem dos modelos determinísticos é a sua simplicidade, que quando levada para a implementação de uma ferramenta de análise resulta em maior velocidade e capacidade de processamento para a mesma.

Adve mostra em sua tese de doutorado [1] que a pressuposição de tempos constantes na execução das tarefas de um programa paralelo não traz um prejuízo grande para o modelo, pelo menos em programas de memória compartilhada ou com granulação equivalente. Em sua tese ele cria um modelo determinístico para tais programas que pode ser resolvido

simplesmente através de um conjunto de equações dirigidas por valores máximos de tempo entre processos que estejam se sincronizando. A metodologia adotada é bastante simples e produziu resultados interessantes, com erros inferiores a 10% segundo seu autor.

Uma outra formulação para análise de desempenho através de modelos determinísticos foi proposta por Sun e Zhu [61]. Nessa formulação a preocupação é com a determinação de escalabilidade de sistemas paralelos usando apenas uma fórmula simples que relacione degradação do paralelismo, escalabilidade e capacidade de computação em um processador. Os resultados de escalabilidade apresentados para essa formulação são significativos, atingindo uma precisão considerável para os testes realizados. Infelizmente os autores não indicam se a mesma formulação poderia ser usada em máquinas que não sejam de memória compartilhada.

### 2.3.2 Métodos baseados em "benchmarking"

Dentre os métodos para análise de desempenho esses são os que apresentam a menor complexidade de implementação pois suas medições são feitas diretamente sobre o conjunto cujo desempenho se deseja medir. Por outro lado, eles também são os mais caros ao exigir o uso da máquina real para a obtenção de medidas. Aqui a estratégia de medição é executar o programa na máquina alvo e medir tudo que for desejável para que se faça uma análise do desempenho do programa.

Esses métodos são mais usados quando se pretende verificar o desempenho de uma máquina, independente do programa que será executado. Entretanto, isso representa um problema de difícil solução que é o de encontrar "benchmarks" (os programas que serão executados para a tomada de medidas) que possam ser considerados eficazes, isto é, que proporcionem medidas confiáveis e que não permitam alterações que beneficiem uma ou outra arquitetura.

Os problemas com a criação de "benchmarks" vão desde a caracterização da carga aplicada ao sistema computacional, como indicado por Calzarossa e Serazzi [15], até a própria estruturação de procedimentos para fazer as medições (Bradley e Larson em [13]), passando é claro pelo projeto de programas de forma a permitirem testes sobre a capacidade real de escalabilidade do sistema (Gustafson et alii [28]). Em outra linha de trabalho encontram-se organizações que sugerem conjuntos de "benchmark" padronizados, como *Linpack*, *SPEC*, *Perfect Benchmarks* e *NAS Parallel Benchmarks*. Em geral, os resultados produzidos por esses conjuntos de medidas poderiam ser dados em tempo gasto para executar uma tarefa. Mas, para permitir a comparação entre máquinas diferentes, essas medidas são transformadas

em unidades relativas como Specmarks, Mips, Mflops, Dhrystones e Whestones por exemplo.

Entretanto, para a medição de desempenho de um programa específico de um usuário os conjuntos listados acima não podem ser considerados. Nesse caso, a medida de desempenho tem que ser tomada diretamente sobre a máquina, implicando em gastos diretos com a compra do equipamento e indiretos pela manutenção do mesmo em atividades diferentes da produção real. Isso representa um custo relativamente alto, fazendo com que testes desse tipo só ocorram quando o equipamento já estiver disponível de antemão.

Apesar desses problemas, a medição de desempenho através de "benchmarking" tem sido bastante utilizada. Sua relativa precisão é um forte argumento em seu favor. Além disso, o fato de que em muitas das ocasiões em que se quer analisar o desempenho de um novo programa já se tem a máquina sobre a qual ele executará reduz bastante o custo de sua aplicação. Portanto, esses dois fatores tornam métodos baseados em "benchmarking" atrativos para a medição de desempenho. Um outro fator bastante relevante é que a maioria das técnicas de medição faz uso de "profilers" ou traços de eventos, os quais necessitam do código executável e também da máquina para que possam ser executados. Desse modo, grande parte das ferramentas para análise de desempenho acabam por fazer uso de alguma forma de "benchmarking" durante alguma fase de análise.

**"Benchmarks" de equipamentos**

Embora essa categoria de "benchmarks" não ofereça a possibilidade de testes sobre o desempenho de um dado programa em uma máquina específica, é preciso que se faça uma rápida apresentação sobre o estado da arte desses testes. O que se descreve a seguir é um conjunto de ferramentas para "benchmarking" disponíveis para o teste de "hardware", mostrando principalmente os conceitos usados em sua concepção para que isso possa ser transferido para a análise de desempenho por "benchmarking".

1. SLALOM [28]

   É um "benchmark" desenvolvido no Ames Laboratory por Gustafson et alii, que procura definir um programa para o teste de escalabilidade do sistema mantendo o tempo de execução fixo, isto é, calcula-se o "speedup" de tempo fixo. No caso do SLALOM o programa utilizado resolve o problema de radiosidade (equilíbrio de radiação emitida/absorvida) de um corpo. A escolha por esse programa foi feita com base em alguns objetivos que deveriam ser atingidos por um programa de teste, tais como independência de precisão, independência de linguagem/arquitetura e possibilidade de

execução em tempo fixo para pequenas variações no grau de paralelismo do equipamento.

Essas restrições são importantes e devem ser levadas em consideração no momento de fazer a análise de desempenho de um programa específico, isto é, a realização das medidas para análise precisa ser feita com o objetivo de atender tais restrições ou sabendo-se previamente que elas não podem ser atendidas e quais as implicações disso no desempenho do mesmo.

2. Matriz de paralelismo [13]

   Bradley e Larson apresentam uma metodologia para a realização dos testes de "benchmark" que procura sistematizar o desenvolvimento da análise de desempenho para que se possa coordenar o tipo de medida realizado e o local em que isso é feito. Essa sistemática para a análise de desempenho é um fator importante para que o projetista possa definir quais são e como devem ser obtidas as medidas de seu interesse.

3. LINPACK [19]

   Trata-se de um "benchmark" tradicional, que usa um conjunto de funções para a resolução de problemas básicos de álgebra linear (**BLAS**). A resposta de desempenho é dada em termos de Mflops pois as rotinas internas do LINPACK fazem uso intenso de operações de ponto flutuante. Embora hoje existam conjuntos de teste melhores é necessário indicar a importância do pioneirismo deste teste.

4. SPEC [64]

   O problema do LINPACK é o fato de o mesmo oferecer apenas medidas com operações de ponto flutuante. Quando se quer analisar o desempenho de uma máquina para operações com inteiros é necessário adotar um outro programa como referência. Para resolver esse tipo de problema alguns consórcios foram formados na busca por conjuntos de programas de teste que permitissem a avaliação do equipamento tanto para operações com ponto flutuante como para inteiros. Um dos mais bem sucedidos consórcios é o SPEC (Standard Performance Evaluation Corporation), que apresenta um total de oito programas para avaliar o desempenho com inteiros e dez para programas de ponto flutuante. Os resultados aparecem na forma de SPECmarks, divididos em SPECint e SPECfp com a indicação da versão, como SPECint95 por exemplo.

   O grande mérito do SPEC no caso de equipamentos paralelos é definir que os programas podem ser implementados (inclusive seus algoritmos) pelo usuário que estiver testando o equipamento. Isso permite adequar o programa para a arquitetura da máquina, a qual pode ter melhor desempenho com um algoritmo diferente daquele normalmente usado

em outras máquinas para resolver o mesmo problema. Assim, o desempenho obtido com um sistema passa a depender da capacidade de análise e programação de quem o usa. Essa possibilidade de ajustar o desempenho de um programa de acordo com a máquina é um dos fatores que impulsionam a análise de desempenho de programas durante o seu desenvolvimento.

**"Benchmarks" para obtenção de traços de eventos**

Uma das aplicações típicas para o "benchmarking" é a obtenção de traços de eventos na execução de programas. A maior parte das ferramentas para análise de desempenho faz uso de execução do programa numa máquina real para obter as informações necessárias para seu trabalho. Claro que isso não caracteriza a realização de "benchmarks" exaustivos sobre o sistema, como seria a medida ideal de desempenho. Entretanto, por exigir a utilização da máquina em algum instante tais ferramentas podem ser descritas como fazendo "benchmarking" para fazer a análise de desempenho. A seguir apresentam-se algumas das ferramentas para análise de desempenho que fazem esse tipo de uso:

1. Medea [16]

   É uma ferramenta bastante versátil para a análise de desempenho, que permite uma fácil visualização dos resultados. Os dados de entrada para avaliação são obtidos pela execução do programa convenientemente modificado para o levantamento de traços de execução. A versão apresentada por Calzarossa et alii admitia quatro estruturas diferentes de traços (PARMON, PICL, VFCS e ALOG) e gerava informações como "speedup", assinaturas (tempos consumidos com execução, comunicação, etc.), distribuição de tempo consumido por cada função, etc..

2. VISPAT [31]

   É uma ferramenta menos versátil mas criada com o objetivo específico de fazer avaliações em sistemas que usam o padrão de troca de mensagens MPI, desenvolvido em Edinburgh. Com isso fica mais fácil o tratamento dos traços obtidos, uma vez que esses obedecem sempre uma única estrutura. Segundo seus autores (Hondroudakis et alii) a vantagem do VISPAT é a sua proximidade com o código fonte, o que facilita a visualização e interpretação dos resultados da análise.

3. Pablo [49]

   Pablo é uma ferramenta desenvolvida na Universidade de Illinois e que apresenta uma diversidade de medidas de avaliação bastante elevada. Apesar de não permitir vários

tipos de traços de entrada como Medea, Pablo é reconhecidamente mais difundido, principalmente graças ao seu formato para dados (SDDF - Self-Describing Data Format), que é o padrão nativo para a ferramenta proprietária para análise de desempenho ParAide usada nas máquinas Paragon XP/S da Intel. Além disso, a estrutura para interface com o usuário de Pablo é muito amigável, permitindo uma fácil identificação das medidas disponíveis e seus detalhes.

Além das ferramentas listadas aqui podem ser citadas ainda outras ferramentas que fazem uso de "benchmarks" na obtenção dos traços de eventos, tais como Seeplex [17] e ParAide. Como dito antes, o uso de "benchmarking" nessas ferramentas fica restrito à obtenção de traços e, portanto, não caracterizam plenamente o uso desta técnica para fazer a análise.

**"Benchmarks" para a análise**

Enquanto as ferramentas listadas no grupo anterior apenas obtêm os dados para análise através de "benchmarking", existem abordagens que o usam para obter as avaliações de desempenho do sistema. Nesse caso não existe uma grande diversidade de ferramentas mas sim de iniciativas isoladas que procuram fazer a análise de desempenho de uma configuração qualquer. Nesse caminho podem ser citados os trabalhos de Manacero [40], que examina o desempenho de programas aplicados em física de partículas executando em redes de estações de trabalho com diferentes cargas de tráfego, ou o de Gill, Zhou e Sandhu [25], que faz o estudo sobre a carga do sistema de arquivos em ambientes distribuídos de grande escala.

Um dos poucos trabalhos que procura criar uma ferramenta de avaliação de desempenho mais versátil é o apresentado por Kitajima e Plateau ao descreverem o ambiente ALPES [37]. Em ALPES se faz a predição de desempenho de um programa a partir da execução de um programa artificial contendo apenas os traços do programa original. Essa execução é realizada na máquina real, ocupando-a de fato.

O programa artificial, definido como sintético, é gerado a partir da modelagem do programa real na linguagem de descrição ANDES. O programa sintético é um modelo quantitativo do programa real descrito por um grafo dirigido. Cada vértice desse grafo detalha as interdependências entre vértices e também a computação nele realizada. O valor dessa computação é definido por uma função que pode ser constante, aleatória ou dependente de outros atributos. Desse modo, a precisão das avaliações de desempenho realizadas com ALPES depende fundamentalmente da atribuição de funções corretas para cada um dos vértices do programa sintético.

Um aspecto interessante de ALPES é que, apesar de ser projetado para execução na máquina Meganode[4], é possível fazer a emulação de máquinas distintas através da alteração do programa sintético a partir de uma descrição da máquina.

Do ponto de vista prático, Kitajima e Plateau apresentam testes feitos com um programa que implementa o modelo de Bellman-Ford para cálculo do caminho mínimo entre um vértice qualquer de um grafo e um vértice do tipo sorvedouro. Para esse programa foram realizados testes para a determinação do grau ótimo de paralelismo, considerando-se variações no número de processadores, grau de multiprogramação e velocidade do processador. Nos testes foi medida a utilização da CPU em termos de tempo ocioso, trabalho útil e "overhead" introduzido pelo paralelismo. Entretanto não são apresentados dados de comparação com a execução real do programa, o que impede a verificação de sua precisão.

### 2.3.3 Métodos baseados em simulação

O terceiro grupo de métodos para análise de desempenho envolve aqueles baseados em técnicas de simulação. Nessa categoria aparecem vários tipos de simuladores, grande parte deles baseados na simulação de eventos em redes de Petri, o que se assemelha ao descrito para os métodos analíticos. A grande diferença entre métodos analíticos e de simulação está na forma como os resultados e o modelo para o sistema são obtidos. Enquanto no primeiro caso os modelos são compostos por sistemas de equações que representem o sistema em análise, no segundo essas equações são substituídas por regras de comportamento que ditam como eventos ocorrem e modificam o estado do sistema. Essa diferença é a principal razão das vantagens relativas de cada um desses métodos.

A precisão dos métodos analíticos é garantida pela certeza de que se o equacionamento do sistema estiver correto, então os resultados obtidos serão também corretos, desconsiderando-se as hipóteses estocásticas levantadas, é claro. Já para os simuladores, mesmo que o modelo esteja correto ainda existe a incerteza sobre as condições em que se realizou a simulação, as quais apenas podem ser repetidas se o simulador for determinístico.

Por outro lado, é muito mais simples gerar um modelo de comportamento para simulação do que um modelo de equações analíticas. Assim, caso se deseje usar a mesma abordagem para a análise de outro programa, é muito mais simples fazer as alterações necessárias no modelo para o simulador do que nas equações que representam o sistema. Isso é importante se for lembrado que ferramentas de análise de desempenho devem ser simples de usar, logo a adaptação entre um sistema e outro deve minimizar a intervenção do usuário na

[4]Meganode é uma máquina paralela de memória distribuída com 128 transputers.

geração do modelo.

Um aspecto relevante no uso de simuladores para análise de desempenho é que várias ferramentas baseadas em métodos analíticos acabam fazendo uso dos mesmos para efeito de comparação entre as aproximações usadas para reduzir o número de estados. Dessa forma, vários simuladores usando redes de Petri e outros modelos são citados como referência para os resultados obtidos pelos métodos analíticos, tais como o QNAP2 usado em [23] ou GSPN em [56]. Esses simuladores não serão analisados aqui por serem ferramentas de uso mais geral. Na mesma categoria aparece o ambiente para modelagem, análise e simulação de sistemas concorrentes SARA [20], desenvolvido na UCLA. A seguir podem ser vistos alguns simuladores dedicados ao problema de análise de desempenho:

1. **PDL** [65]

   Trata-se de um simulador baseado na linguagem de descrição de desempenho PDL (Performance Description Language). O objetivo dessa linguagem é fornecer um método simples de fazer a especificação de um sistema (programa e máquina) para realizar a análise de seu desempenho através de simulação, tal como ocorre com linguagens de descrição de *hardware* como o VHDL.

   Um modelo em PDL é composto por portas, módulos e transportadores, fixando uma estrutura típica de *hardware* na qual as portas podem representar sinais lógicos, os módulos seriam portas lógicas e os transportadores seriam as ligações entre portas lógicas. A especificação de um sistema seria feita através de um programa PDL usando tais estruturas segundo gramática própria. Esse programa é compilado para uma forma intermediária que pode então ser analisado através de simulação.

   Embora o método tenha sido desenvolvido para a análise da máquina, o mesmo pode ser usado para uma análise conjunta *software/hardware*, incluindo-se custos relativos para as tecnologias empregadas na máquina. Isso permite que a análise de desempenho seja feita também em termos de custo associado ao processamento, o que é bastante desejável do ponto de vista de quem vai custear o sistema.

   Embora seja um tratamento interessante do ponto de vista de modelagem, não foram apresentados dados sobre a precisão dos resultados obtidos com PDL. No trabalho de Vemuri aparece apenas um exemplo em que são verificadas várias propostas para a implementação de um sistema embarcado, com a indicação daquelas que tenham melhor relação custo $x$ desempenho.

2. **Axe** [53]

   O método desenvolvido por Sarukkai, Mehra e Block preocupa-se mais com a análise de escalabilidade de programas em máquinas com comunicação por troca de mensagens, o que é uma limitação. Seu funcionamento está baseado na ferramenta MK, que gera um modelo para o programa baseado em sua árvore de derivação sintática.

   A geração da árvore de derivação sintática exige que o programa fonte esteja disponível e também exige compiladores dedicados para diferentes linguagens. No trabalho em que Axe é apresentado a linguagem que aparece nos exemplos é o Fortran. Além da árvore sintática é preciso obter dados sobre tempos de execução a partir de traços de eventos determinados por execução do programa devidamente instrumentado. A exigência de uma máquina real é outra restrição que pode ser feita ao método, mas isso acaba por facilitar também a sua comparação com medidas reais de "benchmarking".

   Quanto aos resultados obtidos por Axe tem-se que sua precisão é razoável, principalmente se for considerado que as comparações são feitas contra medidas de execuções reais dos programas testados nas máquinas. Os erros ficam abaixo de 15% nas piores situações, quando o número de processadores chega a 512. Deve-se observar também que em todas ocasiões o tempo de execução simulado ficou abaixo do tempo de execução real, o que indica que as simulações podem ser tomadas como um limite superior para a escalabilidade.

3. **PAWS** [44]

   O funcionamento de PAWS (Parallel Assessment Window System) é diferente de Axe e PDL no que diz respeito ao tipo de aplicação. PAWS permite que seja feita a análise de desempenho de máquinas e programas ainda em desenvolvimento. O sistema é composto por quatro ferramentas que fazem a descrição da aplicação (programa), da arquitetura (máquina), de desempenho e interface de visualização gráfica.

   A ferramenta para descrição da aplicação transcreve um programa escrito em linguagem de alto nível em um grafo de dependência de dados (fluxo de dados), o qual pode ser mapeado para diferentes arquiteturas. Já a ferramenta para caracterização da arquitetura permite ao usuário descrever máquinas distintas, mesmo que elas ainda não existam de fato. Os modelos gerados por essas duas ferramentas podem ser aplicados à ferramenta de análise de desempenho, que realizaria simulações para determinar medidas de interesse do usuário, podendo ser apresentadas graficamente pela ferramenta de visualização.

   Do ponto de vista de formulação a abordagem usada em PAWS é bastante interessante

por separar os modelos de máquina e de programa. Essa tem sido a abordagem mais comum em várias ferramentas de análise de desempenho e permite a comparação de desempenho entre diversas arquiteturas ou algoritmos. O maior inconveniente nessa proposta é a exigência dos programas fontes para serem testados, muito embora isso seja realmente necessário se o processador da máquina hipotética ainda não estiver disponível.

Foram apresentados alguns testes de simulação, procurando determinar principalmente o grau de paralelismo do programa em cada instante de execução. Sobre sua precisão não foram indicados valores de comparação, o que impede qualquer opinião definitiva sobre o mesmo.

# Capítulo 3

# Simulação do código executável

Neste capítulo será feita a apresentação de um novo método para predição e análise de desempenho de programas paralelizados. Na seção 3.1 serão apresentados os motivos que levaram até esse novo método, enquanto que uma descrição formal do mesmo, com todas as suas características, vantagens e desvantagens em relação às estratégias já existentes, pode ser encontrada ao longo da seção 3.2. O capítulo é encerrado, na seção 3.3 com uma descrição detalhada de um protótipo implementado para exemplificar experimentalmente a funcionalidade da abordagem, tanto do ponto de vista da eficiência quanto da validação dos resultados obtidos.

## 3.1 Predição de desempenho através da simulação do código executável

Nesta seção serão apontadas as razões para a utilização de uma nova metodologia para estimativa de desempenho de programas executando em máquinas paralelas. Como se poderá examinar com mais detalhes adiante, existem várias abordagens diferentes para realizar a predição, embora nenhuma delas consiga ser superior às demais em todos os tipos de ambiente de programação paralela. Ao final desta seção ainda serão descritos informalmente tanto o método para predição aqui proposto, como também algumas de suas vantagens em relação às demais metodologias.

### 3.1.1 Por que um novo método para predição de desempenho

No capítulo anterior foram apresentados diversos métodos para se fazer não apenas a estimativa de desempenho de um sistema paralelo, como também a determinação dos pontos ótimos de operação para cada aplicação. Viu-se também que, em geral, esses métodos podem ser classificados em três grandes grupos, de acordo com a estratégia usada para se chegar aos resultados. De forma resumida podem ser adicionados os seguintes comentários para cada um desses grupos:

1. Métodos analíticos: embora procurem fazer uma determinação analítica e provavelmente exata do ponto ótimo de operação, tais métodos tem sua aplicabilidade restrita à obtenção de um equacionamento preciso e computacionalmente factível do sistema, no qual devem ser levados em consideração uma quantidade elevada de parâmetros, muitos dos quais de difícil dimensionamento, tais como velocidade de transmissão, probabilidades de atraso, repetições em laços de execução ("loops"), variedade na forma dos dados, etc..

2. Métodos baseados em "benchmarking": apesar de apresentarem resultados precisos sem a necessidade de modelos elaborados, são métodos com elevado custo operacional. Isso porque para a determinação do ponto ótimo tem-se que operar sobre o sistema real repetidas vezes, o que significa a sua ocupação por tarefas que não são produtivas no sentido formal da palavra. Além disso, os resultados obtidos para um determinado problema apenas são úteis naquela instância, não podendo ser extrapolados para outros problemas.

3. Métodos baseados em simulação: assim como os métodos analíticos, o grande problema no uso de simulação é a obtenção de um modelo que seja fiel ao problema real. A seu favor tem-se que os parâmetros difíceis de dimensionar analiticamente podem ser tratados mais facilmente por simulação. Outro aspecto relevante é que as simulações podem ser feitas num ambiente diferente daquele em que se vai de fato processar em paralelo, o que reduz bastante o custo operacional da predição de desempenho.

Dentre essas estratégias a que se mostra mais atrativa em termos de precisão, facilidade de modelagem e custos computacionais é a de simulação, uma vez que o modelo a ser simulado pode ser refinado posteriormente através de "benchmarks" reais, com um custo menor do que o do "benchmarking" propriamente dito. Claro que essa mesma argumentação poderia ser aplicada aos métodos analíticos, porém neste caso a transposição dos resultados de "benchmarking" para o modelo analítico são de complexidade muito maior do que

para simulação, principalmente pelos parâmetros de difícil dimensionamento, como indicado anteriormente.

Outros argumentos favoráveis ao uso de simuladores estão ligados a fatores como flexibilidade no modelo de simulação, em que se pode passar mais rapidamente de um problema a outro quando o modelo utilizado no simulador for suficientemente genérico, e também pela velocidade em que se pode fazer as medições para a predição do desempenho.

Quanto à flexibilidade, tem-se que qualquer método analítico depende de sua formulação matemática, a qual em geral é orientada para um sistema específico e portanto inflexível. Mudar o modelo neste caso significa alterar toda uma estrutura de equações, o que não é uma tarefa trivial. Já para os métodos baseados em "benchmarking", a flexibilidade é grande desde que se tenha sistemas e programas disponíveis para teste. Os métodos baseados em simulação encontram-se num patamar intermediário, com sua flexibilidade dependente de como for especificado o modelo para o conjunto sistema-programas, ou seja, de como é o ambiente sob simulação.

Já com relação à velocidade, tem-se que o uso de técnicas analíticas pode levar a uma rapidez maior do que os demais métodos, uma vez que a partir do equacionamento correto do sistema é necessário apenas que se execute uma vez um método de otimização, resolução de equações diferenciais ou estocásticas para os quais existem soluções elegantes em termos de velocidade e precisão. O pior caso é o uso de "benchmarking", quando a velocidade dos testes depende da velocidade real do sistema e da quantidade de repetições necessárias para que se tenham dados estatisticamente confiáveis. Entretanto, obter dados em quantidade suficiente e em grande velocidade é inviável nesse caso, devido ao custo elevado na paralelização do teste (implica em multiplicação do "hardware" de teste, que é a máquina real do sistema).

Desse modo, pode-se afirmar que o uso de simulação apresenta, em termos gerais, bom potencial de utilização, uma vez que possui uma boa velocidade aliada a um baixo custo e fácil adaptabilidade para mudanças no sistema e/ou programas. O problema no uso de simuladores está na obtenção de um modelo que represente de modo fidedigno o que acontece no mundo real. Isso pode ser obtido por diferentes técnicas, muitas das quais foram apresentadas no capítulo anterior, inclusive com bons resultados para os testes indicados nos documentos que as descrevem. Surge então o questionamento sobre a necessidade de se propor uma nova técnica para fazer a predição de desempenho de sistemas paralelos/distribuídos. Os próximos parágrafos procuram responder a essa questão.

Inicialmente deve ser levado em consideração o fato de que nenhum dos métodos existentes é definitivo ou perfeito. Aliás, a imperfeição deles está principalmente relacionada

com a forma como são obtidos os dados relativos à execução do programa. No Capítulo Dois foram apresentados os principais mecanismos de medição ("profiling", traços, etc.) e várias ferramentas de análise de desempenho (ALPES, Pablo, etc.). Lá percebeu-se que as ferramentas são eficientes e bem formuladas mas dependem fundamentalmente das medições realizadas. Desse modo, é preciso buscar uma forma de medição que apresente melhores resultados do que aqueles obtidos atualmente.

Neste trabalho apresenta-se uma nova estratégia de obtenção das medidas necessárias para ferramentas de análise de desempenho baseada na reescrita do código executável transformado em um grafo de execução. A metodologia aqui descrita envolve também a estimativa de desempenho através da simulação desse grafo, mas o principal objetivo do trabalho é obter uma forma precisa de medir os tempos de execução de um programa paralelo que não necessite de métodos invasivos nem uma quantidade excessiva de informações. Isso é obtido com a simulação do grafo de execução, o que significa medições não invasivas e uma quantidade de dados perfeitamente controlável.

Com essa abordagem é possível construir uma ferramenta de análise/predição de desempenho que seja flexível e eficiente. No protótipo descrito a seguir mostra-se como atingir esses objetivos. Outros objetivos desse tipo de ferramenta, como simplicidade e produção de resultados de fácil visualização, também podem ser atingidos com a implementação de uma ferramenta mais complexa seguindo a mesma abordagem. No protótipo tais objetivos adicionais estarão apenas previstos, com breve descrição de como poderiam ser incorporados ao mesmo.

### 3.1.2 Descrição da metodologia

Na próxima seção será feita uma descrição detalhada do método aqui proposto. Porém, para que possam ser apontadas informalmente suas vantagens e desvantagens em relação a outros métodos de predição de desempenho, é necessário que se apresente, mesmo que com alguma superficialidade, sua filosofia de funcionamento, assim como os módulos funcionais necessários à sua implementação.

O princípio básico no qual se fundamenta esse método é o da "metodologia de três passos" de Herzog [29], em especial na forma como a mesma é descrita por Malony, Mertsiotakis e Quick [39]. Essa metodologia procura separar o modelo para o sistema em análise em três diferentes modelos (Figura 3.1): um deles detalhando o programa que será analisado, outro com detalhes da máquina sobre a qual se fará o processamento futuro do programa e finalmente um terceiro modelo versando sobre a interação entre os dois primeiros

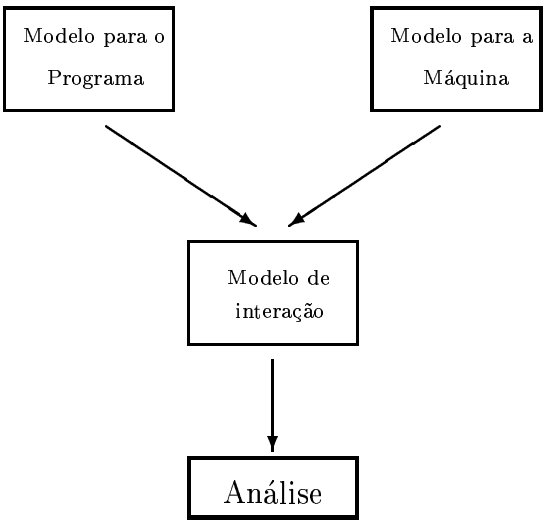



**Figura 3.1: Metodologia de três passos de Herzog.**

no momento de execução.

Na abordagem aqui proposta o que se faz é obter um modelo para o programa através da reescrita do código executável, transformando-o em um grafo que representa os possíveis caminhos numa instância de sua execução. É a forma em que se obtém o modelo do programa que diferencia essa abordagem daquela proposta em [39], que usa uma linguagem de descrição de paralelismo para obter seu modelo de programa. Quanto aos outros dois modelos (o da máquina e o da interação programa-máquina), as duas abordagens apresentam apenas diferenças circunstanciais de modelagem, irrelevantes numa primeira análise, apesar de usarem métodos diferentes para a execução da análise de desempenho, que é feita através de processos markovianos no caso de Malony e por simulação nesta proposta. O modelo da máquina é especificado através de um conjunto de parâmetros para o simulador, tais como velocidade dos processadores envolvidos, taxa de transferência de dados entre unidades de processamento, tamanho de memória, etc.. Por fim, o modelo para a interação entre programa e máquina, que compreende taxas de acerto em memórias "cache", carga nos processadores e sobre o suporte de comunicação, entre outros, também é definido através de parâmetros passados ao simulador. A Figura 3.2 ilustra como são obtidos cada um desses modelos na proposta. Nela, os blocos com linhas tracejadas indicam a forma de obtenção dos modelos imediatamente ligados a eles, ou seja, o modelo de programa é o resultado da geração do grafo de execução e é o simulador quem implementa o modelo de interação.

Como se pode observar, o método aqui apresentado tem a sua implementação

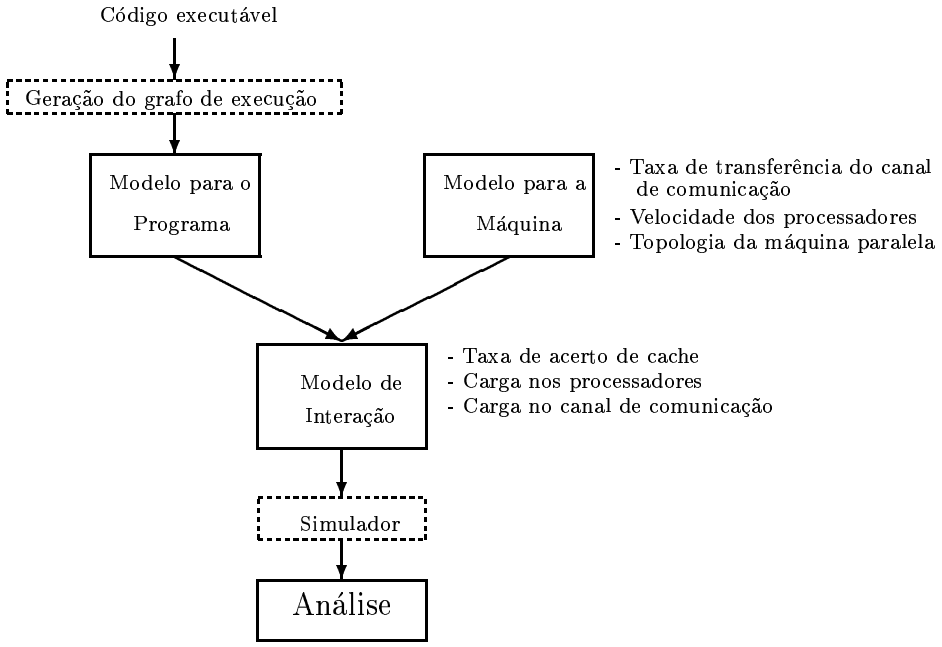



**Figura 3.2: Obtenção dos modelos da metodologia de Herzog nesta proposta.**

computacional constituida de dois grandes módulos, um em que se faz de fato a simulação do conjunto programa-máquina, para a obtenção de dados que permitam a predição de desempenho, e outro responsável pela obtenção do modelo para o programa em análise. Um terceiro módulo, intermediário, é acrescentado para fazer uma otimização (redução do número de vértices) no grafo de execução, com o objetivo único de melhorar o desempenho do simulador.

De forma resumida, o método proposto exige do usuário um programa executável, o qual é reescrito na forma de um grafo de execução, reduzido para um grafo mínimo que ainda descreva de forma exata os caminhos possíveis de execução. Esse grafo é posteriormente simulado, para que sejam gerados os dados sobre o desempenho do conjunto programa-máquina.

Pelo que se pode observar, uma das vantagens desta proposta é a precisão obtida com a reescrita do código executável na forma de grafo de execução. Outra vantagem evidente é o baixo custo de simulação pois a proposta não precisa da máquina paralela para ter suas medições. Desse modo, torna-se plenamente justificável o desenvolvimento mais refinado do método proposto.

Neste trabalho estão indicados os caminhos para que se construa uma ferramenta mais completa para a análise de desempenho, o que inclui uma interface mais amigável com o usuário e módulos de análise qualitativa dos resultados, por exemplo. Já o protótipo implementado se preocupa apenas com a predição do desempenho, em especial com a granulação

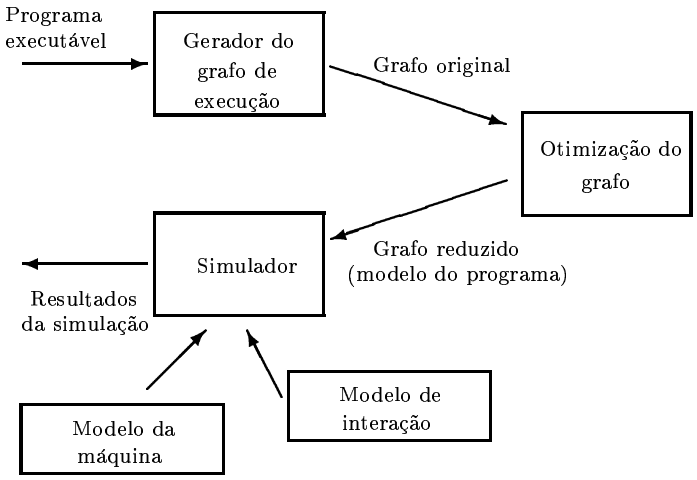



**Figura 3.3: Visão global do funcionamento do método.**

e escalabilidade do programa, que são informações essenciais na fase de dimensionamento do programa e da máquina paralela em que será executado, tanto com o objetivo de otimizar a velocidade global de processamento quanto de reduzir os custos de instalação e manutenção do sistema físico.

## 3.2 Módulos funcionais do método

O método para a estimativa/análise de desempenho proposto é composto por três módulos fundamentais: um primeiro encarregado de reescrever o código executável, transformando-o num grafo de execução; um segundo módulo que faz a redução desse grafo para um grafo mínimo e, finalmente, um módulo para simular a execução do grafo mínimo e obter as medidas de interesse na predição/análise de desempenho. Nesta seção esses módulos serão descritos em detalhe, mostrando a estratégia de funcionamento, restrições impostas e o formalismo utilizado em cada um deles. A Figura 3.3 mostra como esses módulos se relacionam e também como a metodologia de Herzog é mapeada neles.

### 3.2.1 Geração do grafo de execução

Antes de discutir como é gerado um grafo de execução é necessário definir o que esse grafo de fato representa e como o mesmo é composto. O grafo de execução de um programa é um grafo orientado que apresenta todos os possíveis caminhos que o programa pode seguir

durante uma instância de execução, como pode ser visto na Figura 3.4. Os vértices desse grafo definem pontos de processamento e as arestas indicam os caminhos que podem ser seguidos durante a execução do programa.

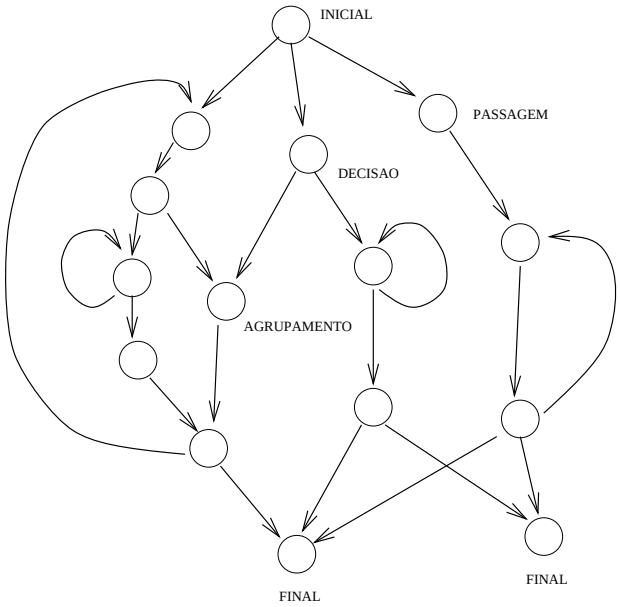



**Figura 3.4: Grafo de execução de um programa**

Fazer a análise de desempenho usando um grafo de execução é uma técnica bem conhecida, sendo utilizada na maioria das ferramentas da área, principalmente naquelas baseadas em "profiling" [26, 30, 50]. A grande diferença entre o grafo de execução utilizado aqui e aqueles utilizados em outras ferramentas é a forma em que eles são gerados. Enquanto nas demais técnicas os grafos são obtidos por medições invasivas realizadas durante a execução do programa real numa máquina real, ou pela geração de um modelo de execução para o programa com tempos de execução em cada tarefa também estimados, a idéia aqui é reescrever o código executável a partir de suas instruções de máquina, medindo então os tempos de execução a partir do tempo consumido por cada instrução de máquina.

Obter medidas sobre os tempos de execução através da reescrita do código executável apresenta algumas vantagens em relação às técnicas de "profiling" e extração de traços de eventos. Basicamente, estas técnicas necessitam da adição de código adicional ao programa para que as medidas de interesse possam ser obtidas. Esse código adicional faz com que os tempos medidos tenham alguma imprecisão, que pode ser maior ou menor dependendo do grau de invasão da medição. Já quando se faz a reescrita do código para um grafo de execução e

medem-se os tempos com a simulação desse grafo, não existe a necessidade de código adicional para fazer qualquer medida, logo os tempos medidos podem ser considerados exatos, exceto por um indeterminismo inerente à execução de qualquer programa.

A reescrita do código de máquina nada mais é do que uma descompilação do código numa outra linguagem. Bowen em [11] e Breuer e Bowen em [12] fazem um estudo formal de como descompilar um código executável para código fonte, com o objetivo de verificar a correção de um programa compilado, isso é, checar se o programa compilado é funcionalmente equivalente ao seu fonte. Embora a verificação de correção não seja um objetivo da predição de desempenho, o formalismo por eles apresentado permite que se utilize a descompilação como ferramenta para a reescrita do código executável numa linguagem orientada para medidas de desempenho, ou seja, uma linguagem que descreva como o computador executaria o programa.

Uma das linguagens que podem resultar da descompilação é um grafo de execução. Este é um grafo orientado $\mathcal{G}$, com um conjunto de vértices $\mathcal{V}$ e um conjunto de arestas $\mathcal{A}$, representado por $\mathcal{G} = (\mathcal{V},\mathcal{A})$. Como já mencionado, nesse grafo os vértices representam conjuntos de instruções ou tarefas que são executadas seqüencialmente, enquanto as arestas representam o modo como essas instruções são interdependentes, isto é, se existe uma aresta ligando os vértices *v1* e *v2*, partindo de *v1* e indo até *v2*, como na Figura 3.5, tem-se que as instruções representadas por *v2* somente são executadas após a conclusão das instruções em *v1*.

Desse modo, segundo os diferentes tipos de conexão com os vizinhos, podem-se classificar os vértices em cinco categorias, a saber: INICIAL, PASSAGEM, DECISÃO, AGRUPAMENTO e FINAL. A seguir definem-se essas categorias, representadas de modo gráfico na Figura 3.6:

- INICIAL: cada grafo de execução possui apenas um único vértice dessa categoria, sendo que ele é caracterizado por não possuir arestas incidentes, ou seja, não possuir vértices ascendentes. Este tipo de vértice indica o ponto no qual se inicia a execução do programa;
- PASSAGEM: são os vértices com apenas uma aresta chegando e uma aresta partindo,

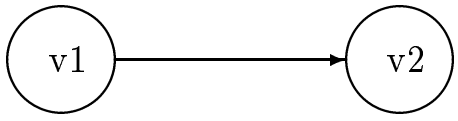



**Figura 3.5: Interdependência entre dois vértices**

representando portando trechos do programa em que não existem desvios condicionais. Esses vértices são utilizados, por exemplo, para indicar os pontos de sincronismo dentro de um programa.

- DECISÃO: trechos do programa contendo o início de desvios condicionais são representados por vértices com um ascendente e dois ou mais descendentes. Cada descendente indica um dos possíveis caminhos a serem seguidos durante a execução, dependendo do valor da condição testada no desvio.
- AGRUPAMENTO: ao final de cada desvio condicional ocorre o agrupamento dos vários ramos criados em seu início. Essa situação é identificada por um vértice de agrupamento, que possui duas ou mais arestas incidentes.
- FINAL: são vértices que representam o final da execução do programa, quer isso ocorra com sucesso ou não. Esses vértices são caracterizados por não possuírem descendentes, ou seja, são terminais.

Embora apenas essas categorias básicas estejam sendo definidas, deve ficar claro que é possível a combinação, em um mesmo vértice, de duas categorias, ou seja, um determinado vértice pode ser a composição de um vértice de decisão com outro de agrupamento, como pode ser visto na Figura 3.4. A única restrição é que apenas exista um vértice do tipo INICIAL, quer ele tenha um único ou vários descendentes (vértice de DECISÃO).

Da forma como foi feita, a classificação dos vértices não leva em consideração características dinâmicas da execução de um programa. É necessário que cada vértice possa determinar qual o tipo de operação que está sendo executada nele, para que o simulador opere convenientemente. Os tipos de operação considerados são os de execução, sincronismo e comunicação, que apresentam o seguinte significado semântico:

1. Vértices de execução: indicam que o tempo de execução para o vértice não depende de fatores externos ao processador, podendo ser de qualquer dos tipos listados anteriormente;
2. Vértices de sincronismo: indicam que a passagem para o vértice seguinte ocorrerá apenas após o atendimento das restrições de sincronismo para o vértice em questão. Desse modo o tempo consumido no vértice depende do instante em que essas restrições forem atendidas;
3. Vértices de comunicação: indicam que a passagem para o próximo vértice depende do processo de comunicação definido no vértice. Assim, o tempo nele consumido depen-

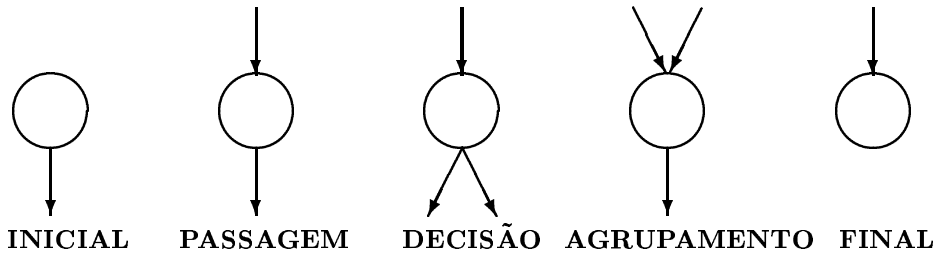



**Figura 3.6: Tipos básicos de vértices**

derá da disponibilidade dos canais de comunicação e do tamanho do que estiver sendo transferido.

Uma vez definido o formato do grafo de execução pode-se abordar o modo de obtê-lo a partir do código executável, as restrições formais a serem resolvidas e as técnicas de descompilação do programa, o que é feito na seqüência.

**A. Estratégia de descompilação**

A descompilação de um executável deve levar em conta informações sobre o conjunto de instruções da máquina para a qual foi compilado, a estrutura usada na sua criação, isso é, sua segmentação, e, principalmente, quantos ciclos de máquina se espera consumir com cada instrução. Tais informações, embora fáceis de se obter, são distintas para cada conjunto processador/compilador, fato que deve ser levado em conta na portabilidade de uma ferramenta que use a metodologia aqui apresentada. Com essa informação em mãos, a obtenção do grafo de execução segue as seguintes etapas:

1. Leitura do código executável;
2. Interpretação das instruções de máquina;
3. Agrupamento de instruções em blocos de fluxo contínuo;

Durante a primeira etapa - leitura do código executável - gera-se uma descrição em linguagem "assembly" do programa, separando-se as sub-rotinas que o compõe, bem como sua tabela de símbolos, com um mapeamento dos endereços lógicos dessas rotinas ao longo do programa. Essa fase está fortemente ligada ao processador para o qual o executável foi criado, o que não poderia ser diferente partindo-se de um código binário. A dependência pode ser atenuada fazendo-se uma especificação conveniente dos módulos da ferramenta de

análise de desempenho, melhorando a portabilidade visto que a grande maioria dos ambientes já possui pequenas ferramentas que fazem o desmonte (do termo em inglês "disassembly") de programas executáveis e de suas tabelas de símbolos.

Cumprida a etapa de separação das sub-rotinas, em "assembly", passa-se à segunda etapa do processo de geração do grafo, que é realizada juntamente com a terceira etapa, até que todas as rotinas sejam examinadas. A interpretação das instruções de máquina pode ser feita com um mapeamento entre cada instrução e o seu significado semântico, isso é, qual o tipo de ação tomada pela instrução.

As ações que podem ser tomadas ao se executar uma determinada instrução são classificadas em três grupos básicos: instruções de saltos incondicionais ("jump"), instruções de saltos condicionais ("branch") e instruções computacionais, em que estão incluídas as instruções que não são de saltos. Essa divisão facilita a implementação pois para cada significado semântico definem-se tipos diferentes de vértices no grafo de execução. De forma prática, uma instrução de salto condicional gera um vértice de DECISÃO, enquanto que saltos incondicionais e instruções computacionais geram vértices PASSAGEM, FINAL e AGRUPAMENTO, segundo o endereço lógico em que se encontra cada instrução.

Na prática, a interpretação das instruções tem que ser feita de maneira estritamente seqüencial. Isso significa que ao se encontrar uma chamada de sub-rotina (função) durante a interpretação de um trecho de código deve-se fazer a interpretação do corpo da sub-rotina chamada antes de se passar para a instrução seguinte. Desse modo garante-se a obtenção de um grafo conexo, embora traga outros problemas, discutidos à frente. A conexidade do grafo é necessária pois programas são conexos.

Na última etapa para a geração do grafo de execução, a mais difícil de todas, deve-se agrupar dentro de um vértice seqüências de instruções em que não ocorram desvios condicionais ou agrupamento de ramos de execução. A maior dificuldade surge quando se tenta definir quais instruções podem ser agrupadas num único vértice, quando se tem que passar para outro vértice e como os vários vértices podem ser conectados através de arestas do grafo orientado. Deve ficar claro que parte dessas decisões são tomadas a partir do significado semântico das instruções estabelecido na etapa anterior. Infelizmente, nem sempre isso é suficiente para o estabelecimento dos vértices e arestas do grafo. Existem situações, como ciclos de repetição e chamadas recursivas de funções, por exemplo, nas quais o tratamento a ser dado não pode ser tão simples como aquele dispensado às instruções computacionais e desvios condicionais.

**B. Principais restrições para a geração do grafo**

As principais restrições na geração do grafo se referem ao controle de ciclos e de recursão. Além desses importantes aspectos no processo, pode-se acrescentar também o tratamento de desvios e o tratamento para chamadas de funções, muito embora esses problemas tenham uma solução relativamente mais simples.

No caso dos desvios condicionais, o problema é como estabelecer o número de arestas que partem de um único vértice e como determinar quais são os endereços lógicos iniciais para cada um dos vértices-destino dessas arestas. Já quando se fala em desvios incondicionais, o problema é como identificar o endereço lógico para o qual o processador retornaria ao final do desvio, deixando-se implícito aqui que esse tipo de instrução é usado, na maioria das vezes, durante chamadas e retornos de sub-rotinas[5]. Por outro lado, um aspecto relevante para o tratamento de vértices que representem desvios é o dimensionamento formal do tempo a ser consumido em cada um de seus ramos e como decidir qual a aresta a ser seguida em cada instante.

Os problemas de decisão do caminho no grafo e da quantificação dos tempos nos ramos influenciam de modo acentuado a eficácia do método. Portanto se pode afirmar que os resultados da simulação são dependentes da técnica de decisão utilizada para escolher um caminho e da técnica usada para medir o tempo gasto nesse caminho. Essa influência tornar-se-á ainda maior quando os tempos de execução em cada caminho forem substancialmente distintos. Nesses casos, a escolha por um ou outro caminho pode significar uma simulação com resultados absurdos em relação ao que poderia ser esperado.

Quanto ao tratamento de ciclos, a grande dificuldade é como estimar a quantidade de vezes em que um determinado ciclo será executado, antes de se passar para a instrução seguinte. Aqui também, a escolha por um determinado número de repetições faz com que os tempos de execução simulados se tornem muito distintos para cada simulação e, portanto, difíceis de serem analisados. Sob outra ótica, a de técnicas de programação estruturada, um ciclo pode assumir duas formas distintas - testes *a priori* e *a posteriori* - de acordo com o momento em que ocorre a decisão pela continuidade ou não da execução do corpo do ciclo. Isso representa um outro problema para o estabelecimento do ciclo, ou seja, como fazer para diferenciar o início de um ciclo de um desvio condicional simples ou ainda como determinar quais vértices do grafo representam o início e o fim de um ciclo.

Já o tratamento de chamadas de funções (sub-rotinas) tem como dificuldade o

[5]De fato, esses desvios também podem ser usados em ciclos de repetição, para o retorno ao início de seu corpo, embora na maioria das implementações encontradas opte-se por desvios condicionais testando valores constantes.

estabelecimento, no grafo, dos endereços de início da função e de retorno para quem a chamou. Identificar tais endereços significa criar arestas ligando o vértice em que ocorreu a chamada ao vértice no qual se inicia a execução da função e também entre o(s) vértice(s) final(is) da função e o vértice seguinte àquele que ocasionou sua ativação. A primeira aresta é simples e não apresenta nenhum empecilho na criação do grafo, bastando criar-se uma aresta do vértice que chama ao vértice chamado. O problema é a criação da(s) aresta(s) de retorno pois uma determinada função pode ser chamada de diversos pontos do programa, ou de modo equivalente, ter arestas incidentes partindo de pontos distintos. Isso implica em se ter também vários vértices para retorno. A identificação do vértice de retorno correto se torna um problema a mais e terá sua solução indicada quando da discussão sobre o tratamento dado às chamadas de funções, inclusive recursivas.

Finalmente, as dificuldades para o tratamento de recursão, além das inerentes às funções de modo geral, surgem da necessidade de se dimensionar qual vai ser o grau de aprofundamento das chamadas recursivas, isto é, quantos níveis de recursão existem de fato no momento de se executar o programa real. Esse é um problema de tratamento bastante difícil, que é simplificado nas ferramentas de "profiling" mais utilizadas, [26, 50] por exemplo. A dificuldade é ainda maior quando a recursão for indireta, isto é, ocorrer uma seqüência de chamadas do tipo $\mathbf{A} \rightarrow \mathbf{B} \rightarrow \mathbf{A} \rightarrow \mathbf{B}$, por exemplo. Nesses casos, a identificação da recursão pode ser confundida pela seqüência das chamadas. Mais uma vez, a quantidade de recursões simuladas pode representar a obtenção ou não de uma resposta correta para o desempenho de um dado programa em um dado sistema.

### C. Tratamento de desvios

Como mencionado, o tratamento de desvios precisa levar em consideração os seguintes aspectos:

- Estabelecimento da quantidade de arestas que saem do vértice de decisão;
- Determinação dos vértices para os quais essas arestas são dirigidas (endereço inicial de cada ramo);
- Determinação do endereço de retorno para desvios em chamadas de sub-rotinas;
- Dimensionamento do tempo de execução de cada ramo;
- Estabelecimento de uma política de decisão para o caminho a ser seguido;

O primeiro problema é resolvido no momento de fazer o agrupamento de instruções em um único bloco seqüencial. O que se faz é agrupar vértices de decisão sempre que isso

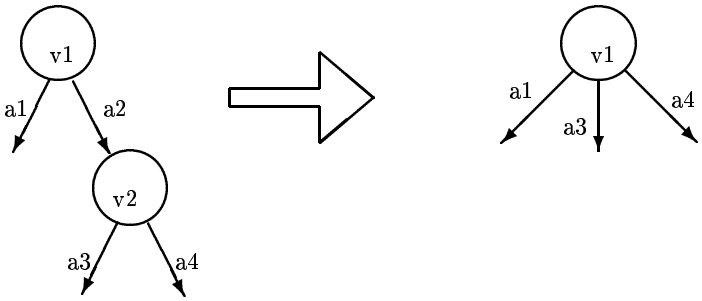



**Figura 3.7: Agrupamento de testes de decisão**

for possível. Assim, quando se atingir o agrupamento máximo tem-se automaticamente a quantidade de arestas que parte de um vértice de decisão. Graficamente, o que ocorre pode ser visto na Figura 3.7, onde se mostra um grafo com um vértice de decisão *v1* contendo apenas duas arestas emergentes, *a1* e *a2*, (para os casos em que a condição testada for verdadeira ou falsa). Se uma dessas arestas levar a outro vértice de decisão, *v2*, sem outros tipos de vértices entre eles, pode-se incorporar *v2* em *v1*, eliminando-se a aresta *a2* e acrescentando-se as arestas que partem de *v2* em *v1*. Se existirem outros vértices entre dois vértices de decisão eles não podem ser aglutinados.

Desse mesmo modo se resolve também o problema de determinar para quais vértices as arestas emergentes de um dado vértice devem ir.

O terceiro problema é resolvido através da forma como se tratam sub-rotinas dentro do grafo de execução. A técnica utilizada aqui consiste em considerar o grafo de um programa como sendo a união dos grafos individuais de cada sub-rotina. Desse modo, a chamada de uma sub-rotina é representada no grafo por uma aresta saindo do grafo corrente para o grafo que representa a sub-rotina chamada e o seu retorno é representado por arestas entre os pontos de saída da sub-rotina e o vértice que contenha a instrução seguinte dentro do grafo corrente. Com isso mantém-se o grafo conexo, como ilustrado na Figura 3.8.

Em relação aos dois últimos aspectos do tratamento de desvios tem-se de estabelecer uma solução de compromisso, principalmente pelo fato da política de decisão adotada ter grande influência sobre os tempos de execução dimensionados. Primeiro, uma possível política é adotar limites máximos e mínimos de tempo para a execução dos vários ramos que partem de um vértice de decisão. Essa solução, no entanto, apresenta um grave problema quando existem ramificações mais internas ao primeiro vértice de decisão. A propagação dos limites de tempo das ramificações internas para o exterior não pode ser feita de forma ingênua, simplesmente acrescentando-se limites mínimos e limites máximos e mantendo-se apenas um

par de valores limites.

Uma outra política é deixar para o simulador decidir por qual aresta vai caminhar. Essa decisão pode ser tomada com o uso de técnicas de números aleatórios com distribuições de probabilidade definidas segundo algum parâmetro externo ao simulador. Dessa forma, o que se faz é calcular os tempos consumidos nos trechos em que não ocorram desvios e usar esses tempos, associados aos vértices do grafo de alguma forma, durante a simulação. Isso evita a propagação de tempos de execução entre os vários vértices do grafo, reduzindo a complexidade de sua implementação.

### D. Tratamento de ciclos de repetição

Mencionou-se anteriormente que os ciclos de repetição impunham dois problemas ao método. Primeiro, como determinar o número de vezes que o corpo de um ciclo seria executado. Segundo, como diferenciar o teste de decisão de um ciclo de um teste de decisão para um desvio. A seguir são apresentadas as técnicas que permitem resolver ambos os problemas.

Inicialmente, a determinação do número de repetições em um dado ciclo é deixada também para o momento de simulação do programa. O número de repetições é determinado através de uma função de geração de números aleatórios com função de distribuição de probabilidades (*fdp*) definida previamente. Logo, o tempo que será consumido dentro de um ciclo é equivalente ao somatório dos tempos consumido durante cada uma das repetições executadas. Isso evita o uso de tempos médios de execução dentro de um ciclo pois cada repetição é simulada individualmente.

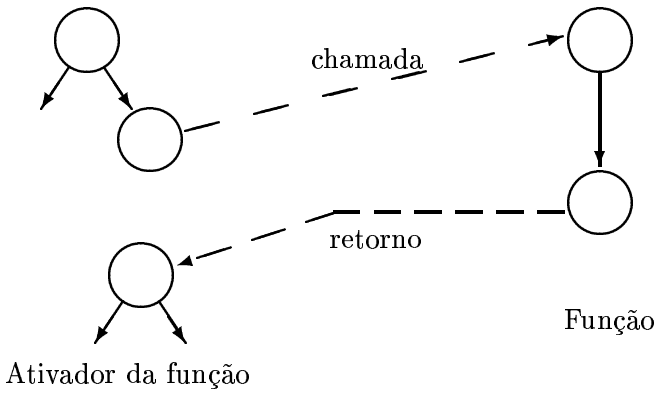



**Figura 3.8: Chamada e retorno de uma sub-rotina**

Quanto ao segundo problema, a solução passa pela determinação de quais são os vértices inicial e final do ciclo. O mapeamento desses vértices pode ser feito de forma bastante simples ao se considerar que em todo e qualquer ciclo, em linguagem "assembly", vai existir uma instrução de desvio na qual o endereço para o salto é anterior ao endereço da instrução. Em outras palavras, isso significa que em qualquer ciclo vai existir um momento em que o contador de programas deve ser decrescido em algum valor para que se possa voltar ao início do corpo do ciclo.

A determinação dos vértices em que o ciclo se inicia e termina é feita com a identificação da instrução em que isso ocorre e dos endereços dessa instrução e da instrução para a qual o contador de programas é desviado. Essas informações são facilmente obtidas a partir dos códigos de máquina para o processador que supostamente executará o programa paralelo. Infelizmente, a determinação da existência de um ciclo e de seu corpo, realizada pela identificação da instrução de desvio em seu final, acarreta na maioria das vezes um novo problema, que é o de diferenciar entre as duas formas possíveis de estruturação para um ciclo, isto é, distinguir entre ciclos "repeat-until" e "while".

Esse problema surge sempre que a instrução de desvio no final do ciclo for uma instrução de desvio condicional. A Figura 3.9 apresenta como ficaria o grafo para "while" encerrado por desvio incondicional, para "while" encerrado por desvio condicional e para "repeat-until". Deve se observar que no primeiro e terceiro casos não existem problemas na identificação do ciclo, seu tipo e em qual vértice devem ser colocadas as informações para que o vértice de decisão seja marcado como um ciclo. Já no caso de "while" com desvio condicional, que é uma construção muito utilizada pelos compiladores, na qual o teste final sempre tem um valor constante, fica mais difícil determinar qual dos dois vértices de decisão deve ser usado como teste de decisão para o ciclo e qual deles deve ter um de seus ramos eliminados.

Nesse caso, para que se possa determinar que o primeiro teste é quem de fato faz a decisão no ciclo e não o segundo, é necessário verificar se em ambos os vértices existe uma aresta para um vértice em comum, que seria a continuação do programa, e também o tipo de vértice para o qual vai a aresta de retorno. Um vértice de decisão em que uma das arestas aponte para um endereço lógico exterior ao ciclo caracteriza um ciclo tipo "while" com desvio condicional em seu final. Outros tipos de vértices caracterizam um ciclo tipo "repeat-until". Quando o ciclo for do tipo "while" deve-se também eliminar do grafo gerado a aresta que nunca será percorrida (indicada na Figura 3.9 por ser um teste sobre um valor constante). Essa transformação acaba por tornar os testes condicionais no final de ciclos tipo "while" em simples desvios incondicionais para o início do seu corpo.

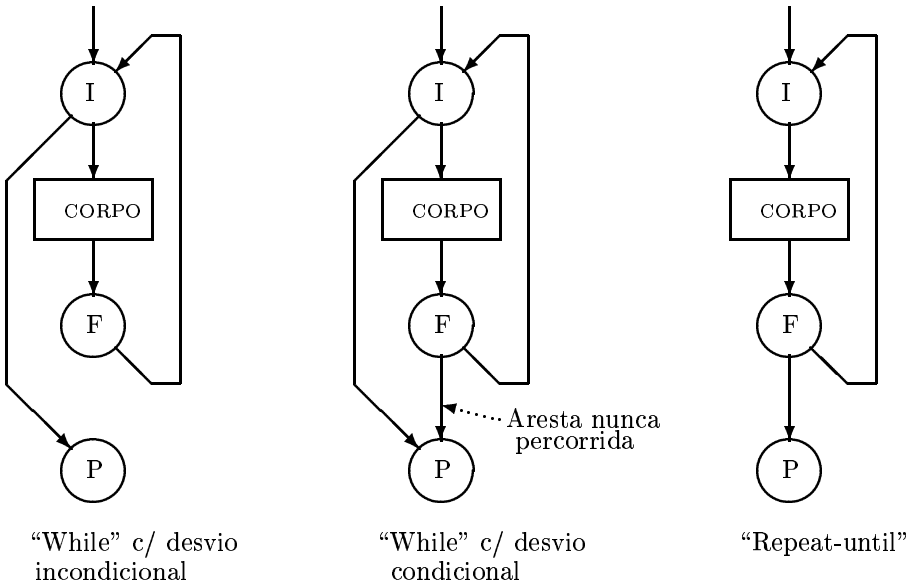



**Figura 3.9: Modos diferentes para construção de um ciclo**

Deve ficar claro aqui que toda essa discussão vale apenas quando os ciclos, mesmo em linguagem de máquina, obedeçam uma estruturação rígida. Programas que usem GOTO's para a quebra dessa estrutura, como indicado na Figura 3.10, complicam a identificação do corpo de um ciclo e de qual vértice deve ser marcado como local para tomadas de decisão. Nesses casos, a estratégia de identificar ciclos a partir da existência de uma instrução com desvio para endereço anterior, usada para os ciclos estruturados, fará com que todos os possíveis ciclos não-estruturados sejam definidos como sendo ciclos "repeat-until" com a adição de caminhos de desvio do ciclo nos pontos onde ocorre o GOTO. Deste modo, apesar de não existir um corpo definido para o ciclo, consegue-se determinar o vértice em que ocorre a decisão sobre sua continuidade.

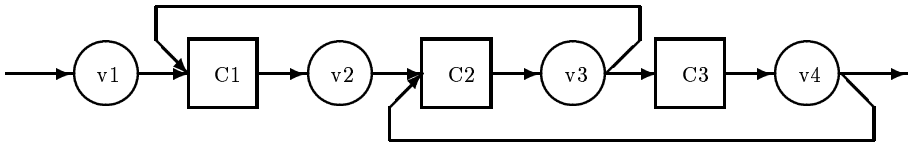



**Figura 3.10: Ciclos não-estruturados usando GOTO**

**E. Funções simples e funções recursivas**

Alguns dos problemas relacionados com chamadas de funções, recursivas ou não, já foram abordados no momento da discussão feita sobre o tratamento de desvios. Naquele momento indicou-se que funções dentro de um programa seriam representadas como grafos de execução isolados, os quais seriam conectados através de arestas criadas nas chamadas de função (ver Figura 3.8). A aresta que liga o vértice que faz a ativação da função com o seu início é trivial. O grande problema é como criar arestas ligando um vértice de retorno da função com o vértice que representa a próxima instrução no programa que ocasionou a chamada da função.

A solução encontrada foi fazer com que vértices de retorno em qualquer função sejam representados como vértices do tipo FINAL, isto é, sem arestas emergentes. Em paralelo, adiciona-se uma informação no vértice que fez a chamada da função indicando qual é o vértice seguinte dentro do programa. Essa informação é passada ao simulador, que irá usar o vértice indicado para dar continuidade ao processo de simulação até que se chegue a um vértice FINAL e não existam vértices anotados como continuação.

Essa estratégia resolve de forma simples o problema de como identificar para qual ponto do programa uma função deve retornar, quando a mesma for chamada em diversos pontos. Isso resolve também um problema típico de "profiling" que é o estabelecimento de um tempo médio de execução para cada função, independente dos parâmetros de cada ativação. Com a possibilidade de se simular a função em cada uma de suas ativações deixa de ser necessário o estabelecimento de um tempo fixo de execução para a mesma. Aliás, também o problema de quantificação de chamadas recursivas é resolvido através dessa estratégia. O que se faz é simplesmente executar o corpo da função e, no caso de ocorrer uma recursão (que faria parte de um ramo de algum desvio condicional) basta executá-la do mesmo modo, mesmo se a chamada recursiva for direta ou indireta.

Durante o processo de descompilação um problema no tratamento de funções recursivas é evitar que a mesma função seja transformada num grafo de execução recursivamente. Isso criaria um ciclo de repetição infinito, uma vez que o processo a ser seguido é de criar um grafo de execução para uma função toda vez que a mesma for ativada por uma instrução qualquer. Esse problema existe para qualquer função que faça parte do programa, mas é crítico no caso de funções recursivas, quando ao se tentar criar um grafo de execução para a função chega-se até sua chamada recursiva, o que faz com que se tente montar um novo grafo de execução para a mesma função, caindo-se em um processo de repetição infinita.

Para resolver esse problema, o que se faz é criar marcações para os grafos indi-

viduais de cada função. Assim, ao se iniciar a geração do grafo de uma determinada função esta é marcada como já tendo sido examinada. Na continuação do processo, sempre que a partir de uma função for necessário construir o grafo de uma segunda função, isto só ocorrerá quando a segunda função não estiver marcada. Caso contrário, apenas é criada uma aresta partindo do vértice que a chama em direção ao início de seu grafo. Com isso, tanto chamadas recursivas como múltiplas chamadas para uma mesma função não implicam a construção de seu grafo várias vezes.

### 3.2.2 Otimização do grafo de execução

Muito embora o grafo gerado usando o método da seção anterior seja relativamente pequeno, ainda é desejável reduzir o número de vértices do mesmo, uma vez que serão estes os pontos de controle para o simulador do programa. A redução do grafo não pode ser feita sem critérios, isto é, sem levar em conta condições quanto ao tempo gasto por vértice para fazer a simulação e quanto à precisão pretendida com a simulação. Essas condições são antagônicas pois menos vértices implicam resultados mais rápidos, porém menos precisos.

Diante disso, o que se faz é buscar uma solução de compromisso entre tempo de simulação e precisão dos resultados. Isso pode ser obtido através do estabelecimento de níveis de otimização, começando de um sistema que com a máxima redução no grafo, com resultados pouco precisos é claro, e indo até um nível mínimo de redução (nenhuma redução), com os resultados mais precisos possíveis, passando por níveis intermediários de redução. Desse modo, a solução de compromisso atenderia aos interesses do usuário, o qual poderia estar interessado numa primeira aproximação para o problema, com resultados ainda grosseiros, ou numa simulação final, buscando detalhes que possam ser melhorados no conjunto programa-máquina. O nível de otimização pode ser ajustado pelo usuário com a definição de quais tipos de redução vão ser habilitadas e quais não.

Algumas das técnicas utilizadas neste trabalho se basearam em textos sobre otimização de código em compiladores, [2, 47] por exemplo, principalmente aquelas que trabalham sobre a árvore de derivação sintática ou sobre a árvore de transformação do programa. Outras técnicas surgiram devido às características peculiares do modelo de vértices usado no grafo de execução. Todas, entretanto, têm em comum o fato de só poderem ser aplicadas para um determinado conjunto de vértices. Deve-se notar, porém, que aqui o objetivo da otimização é a diminuição do número de vértices no grafo de execução, sem descaracterizar o tempo consumido para percorrer o grafo. Assim, algumas das técnicas de otimização de código não podem ser usadas, como, por exemplo, a retirada de constantes para fora de ciclos

de repetição. A seguir são apresentadas as três principais técnicas de otimização que podem ser habilitadas.

**A. Aglutinação de vértices PASSAGEM:**

Um vértice PASSAGEM pode ser incorporado ao vértice destino da aresta que parte dele, como mostra a Figura 3.11. Essa operação não pode ser feita quando o vértice de passagem estiver representando um ponto de sincronismo no programa, exceto quando o nível máximo de redução estiver habilitado. Outro impedimento para essa operação surge quando o vértice destino for do tipo AGRUPAMENTO. O que se faz nesta operação é determinar qual o tempo que o vértice a ser aglutinado consome na execução e acrescentá-lo ao tempo consumido pelo vértice destino.

A operação de aglutinação faz com que o tempo que seria consumido pelo vértice PASSAGEM *V1* seja acrescido ao tempo total gasto no vértice *V2*, qualquer que seja o seu tipo. Com isso, o vértice *V1* pode ser eliminado, fazendo-se com que a aresta que nele incidia passe a incidir em *V2*. Portanto, o grafo passa a ter um ponto de controle a menos durante a simulação, tornando-a mais rápida.

Pode-se provar que a operação acima não deturpa o resultado da simulação, desde que atenda às restrições impostas. Informalmente, com o auxílio da Figura 3.11, percebe-se claramente que se o simulador considerar um tempo *t1* para o primeiro vértice e *t2* para o segundo, como esses vértices seriam executados seqüencialmente, sem bloqueio externo entre ambos, o tempo total seria igual a *t1 + t2*, que é exatamente o tempo atribuído para o vértice *V2* após a redução do grafo.

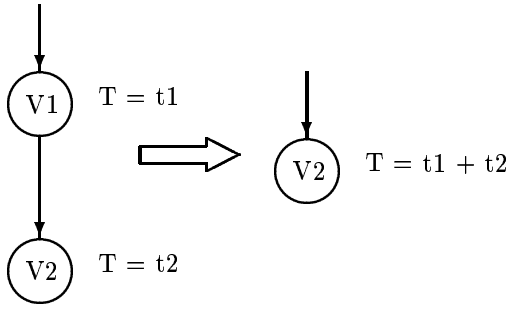
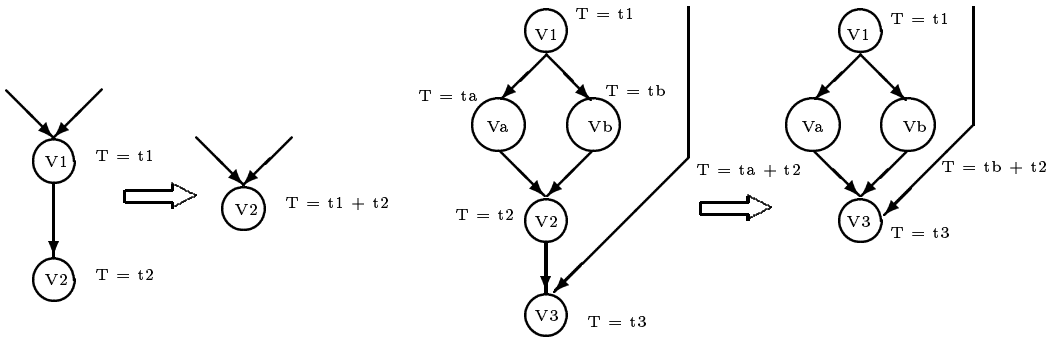



**Figura 3.11: Aglutinação de vértices PASSAGEM**




(a) Incorporação para frente (b) Incorporação para trás

**Figura 3.12: Aglutinação de vértices AGRUPAMENTO**

**B. Aglutinação de vértices AGRUPAMENTO:**

Assim como no caso anterior podem-se incorporar vértices de aglutinamento ao vértice destino de sua aresta emergente. Essa operação é válida desde que o destino não possua outras arestas incidentes, como mostra a Figura 3.12.a. Nesse caso, a operação de aglutinação de vértices é feita de modo semelhante ao que foi feito para os vértices PASSAGEM.

No entanto, existe um caso em que é possível reduzir-se o grafo, mesmo com mais de uma aresta incidindo no nó origem. A Figura 3.12.b ilustra essa situação, na qual o vértice AGRUPAMENTO *V2* finaliza todas as arestas que tenham partido de um vértice DECISÃO *V1* e que as outras arestas incidindo no vértice destino *V3* venham de pontos do grafo fora dos caminhos entre *V1* e *V2*.

Essa situação exige uma análise e tratamento diferenciado. Inicialmente, o processo de remoção do vértice *V2* é feito passando-se suas informações para seus antecessores. Especificamente os antecessores passam a apontar para o sucessor de *V2* e tem seu tempo de execução acrescido do tempo que seria consumido por ele. Como dito anteriormente, isso apenas é possível quando as demais arestas que incidem em *V3* venham de fora dos ramos entre *V1* e *V2*. Isso porque é apenas nessa situação que a propagação do tempo gasto no vértice pode ser realizada. A verificação dessa afirmação também pode ser feita de modo intuitivo.

Antes de se reduzir o grafo o tempo total gasto entre *V1* e *V3* pode ser igual ao tempo consumido em um dos caminhos possíveis entre esses vértices, os quais contêm *V2* obrigatoriamente. Assim, tem-se que o tempo total para se chegar até *V3* será igual a *t1 + ... + ta + t2* se passar por *Va*, ou *t1 + ... + tb + t2* se passar por *Vb*. A mesma computação

poderia ser feita para outros caminhos que incidissem em *V2*.

Com a eliminação do vértice *V2* e a propagação de seu tempo na forma indicada, tem-se que os tempos consumidos em cada um dos caminhos entre *V1* e *V2* continuam a ser os mesmos que foram calculados antes de sua redução, sendo que agora os vértices imediatamente antecedentes a *V2* tiveram seus tempos aumentados.

A restrição de que todos os caminhos entre *V1* e *V2* estejam fechados até se chegar a esse último e que as demais arestas que incidam sobre *V3* venham de outros pontos, o que é decorrente da primeira restrição, é necessária para simplificar o processo de redução do grafo. Caso essa restrição não seja imposta poderiam existir situações nas quais um ou mais caminhos possíveis entre *V1* e *V3* fossem perdidos, principalmente quando o ramo não fechado até *V2* partisse de um de seus antecessores imediatos,.

### C. Redução de vértices comuns em ramos distintos

Uma outra forma de redução no tamanho do grafo envolve um técnica semelhante a de eliminação de código comum aos ramos de um teste de decisão ("hoisting"). Nesse caso, o que se procura fazer é diminuir o número de vértices dentro dos vários ramos que partem de um vértice DECISÃO. Isso é possível quando os vértices iniciais de cada ramo não envolvem instruções de sincronismo ou comunicação. A Figura 3.13 mostra como é possível fazer a redução do grafo dentro dessa estratégia.

Como se pode perceber, ao aplicar-se a estratégia consegue-se, pelo menos, a eliminação de um dos vértices iniciais. É possível obter uma maior redução quando os tempos consumidos por vários dos sucessores do vértice DECISÃO forem iguais ao de menor tempo. Nesse caso, todos os vértices que atenderem à condição serão eliminados, mantendo-se apenas aqueles cujos tempos forem maiores que esse tempo mínimo.

O procedimento para fazer essa redução é relativamente simples e pode ser inferido diretamente da Figura 3.13. Ao se verificar que os sucessores de *V1* não envolvem operações de sincronismo basta procurar qual deles possui o menor tempo de execução. Na figura esse vértice é dado por *Va* com tempo igual a *ta*. Elimina-se esse vértice, fazendo com que a aresta que nele incidia seja substituída pelas arestas que dele partiam. A mesma operação pode ser feita para todos os outros vértices que tenham tempo de execução igual a *ta*. Para manter a equivalência dos tempos de execução em todos os caminhos, *ta* é acrescido ao tempo de *V1* e decrescido dos tempos em cada um dos outros caminhos. Aliás, isso reforça a possibilidade de eliminarem-se vários vértices quando seus tempos forem iguais, uma vez que o tempo resultante, caso não fossem eliminados, seria igual a zero.

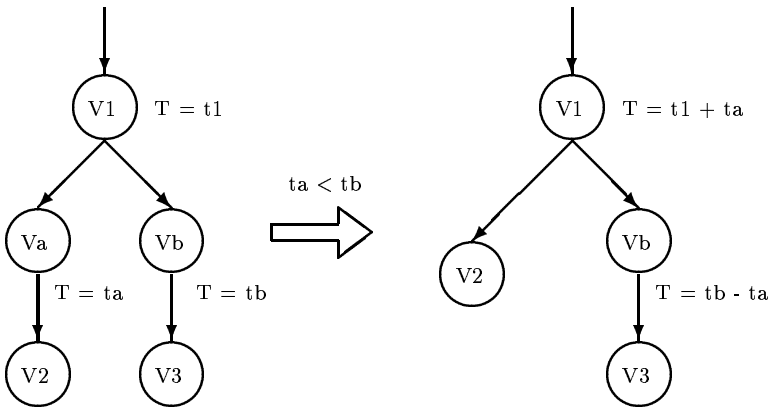



**Figura 3.13: Redução de vértices comuns**

Assim como nos casos anteriores, a prova da validade da operação pode ser feita com uma análise intuitiva. Aqui é preciso mostrar que os tempos gastos para se chegar aos sucessores dos sucessores de *V1* não são alterados com a redução no grafo. A partir da Figura 3.13 tem-se que para se chegar até *V2* o tempo gasto é igual a *t1 + ta*, antes da redução. Para *V3* a forma de cálculo é idêntica, resultando em *t1 + tb*.

Com a eliminação de *Va* e a propagação de seu tempo de execução pelo grafo, tem-se que para se chegar ao vértice *V2* só é preciso passar por *V1*, consumindo-se um tempo total de *t1 + ta*, que é igual ao anterior.

Para atingir o vértice *V3* o tempo é igual a soma dos tempos gastos em *V1* e *Vb*, que são iguais a (*t1 + ta*) e (*tb - ta*) respectivamente. Assim, o tempo resultante será igual a *t1 + tb*, que é igual ao tempo obtido antes da redução. Se houvesse mais caminhos possíveis, a análise seguiria da mesma forma, sem deturpar o tempo de qualquer dos caminhos partindo de *V1*.

Como dito acima, a única restrição para a aplicação dessa técnica é de que não existam vértices de sincronismo ou comunicação entre os descendentes de *V1*. Essa restrição é necessária pois para esses dois tipos de vértice não se pode determinar, *a priori*, qual será o tempo gasto em sua execução. Nos dois casos o tempo de execução depende de fatores que apenas serão conhecidos durante a simulação, tais como carga nos processadores, tráfego no meio de comunicação, concorrência por recursos compartilhados, etc., sendo impossível determinar qual dos vértices descendentes tem o menor tempo de execução.

### 3.2.3 Simulação do grafo de execução

O último módulo necessário ao método aqui apresentado é um simulador do grafo de execução. O funcionamento desse módulo está baseado numa estrutura centralizada de controle sobre a ocorrência de eventos. Os eventos são passagens de um vértice para outro dentro do grafo de execução, ou seja, sempre que se caminha por uma aresta é gerado um evento. Seguem-se as descrições do funcionamento do simulador, das técnicas estatísticas utilizadas e da estratégia de coleta e tratamento dos resultados das simulações.

#### A. Estratégia de operação

Como dito acima, o simulador é dirigido pela ocorrência de eventos. Sempre que ocorrer um evento o simulador atualiza sua estrutura de controle e "espera" pela ocorrência de um novo evento. Lembrando que eventos representam simplesmente a passagem por arestas do grafo de execução, é preciso especificar detalhadamente como se caminha nas arestas do grafo e como é feito o controle sobre a ocorrência de eventos no simulador. Antes disso é interessante examinar o algoritmo do funcionamento global do simulador, que é apresentado na Figura 3.14.

Não serão examinados os passos **1** e **7** por se tratarem de operações simples. O primeiro consistindo simplesmente na leitura do grafo de execução, o qual pode ser único para programas com apenas um código (SPMD[6] por exemplo) ou múltiplo para programas tipo mestre-escravo. Já a apresentação de resultados pode ser entendida como sendo uma simples interface entre o simulador e seu usuário, a qual depende obviamente do esforço despendido na criação de uma interface amigável, cuja discussão não é o propósito deste trabalho. Posto isso, pode-se passar à descrição dos demais passos.

1. **Passo 2**:

   A configuração da máquina deve ser feita para que se saibam quantos processadores estarão disponíveis, quantos processos devem ser criados (na hipótese de o número de processos não ser igual ao de processadores), qual a velocidade (ciclos de máquina/segundo) do processador, qual a carga aplicada sobre cada processador, qual a taxa de acertos em memória "cache", qual a velocidade de transmissão de dados no meio de comunicação, etc..

   Essas informações serão úteis para a composição dos modelos de máquina e de interação

[6] O modelo SPMD é aquele no qual um mesmo código é executado em paralelo para várias instâncias de dados.

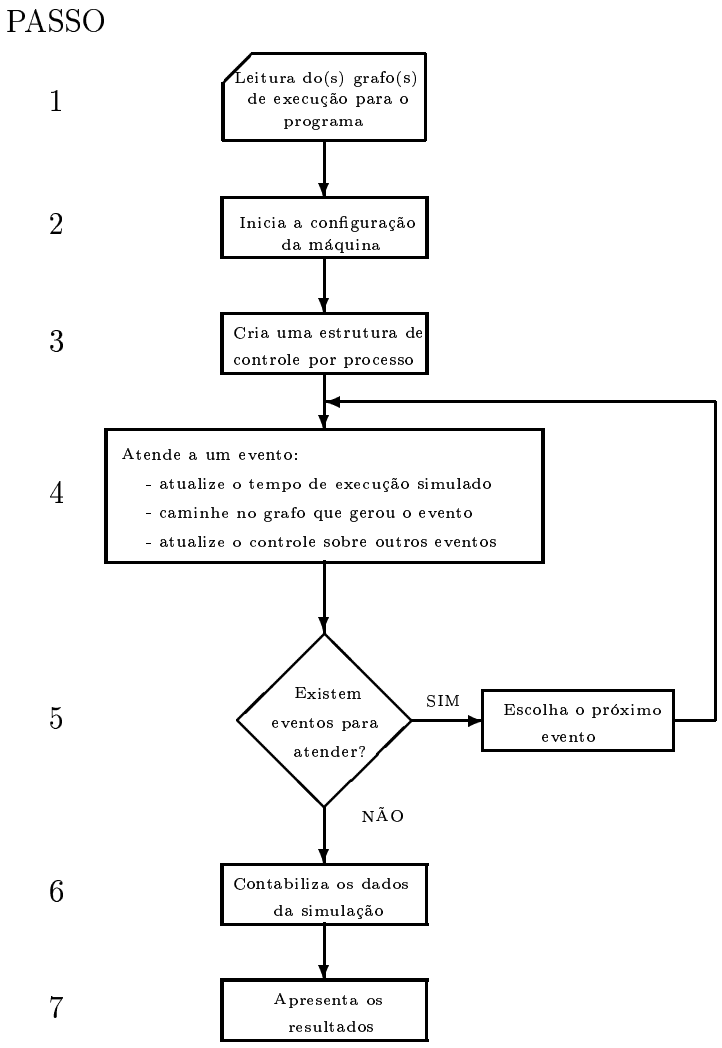



**Figura 3.14: Algoritmo do funcionamento do simulador**

programa-máquina, dentro da metodologia dos três passos de Herzog.

2. **Passo 3**:

   Com as informações obtidas nos passos **1** e **2** é possível a criação de uma estrutura de controle para cada um dos processos a serem executados. Nessa estrutura vão estar informações sobre o estado do processo, isto é, se está executando ou esperando por sincronismo ou comunicação, qual a próxima aresta a ser caminhada (um apontador para o próximo vértice) e quando deve ocorrer o próximo evento. Esses dados permitem ao simulador o controle sobre a evolução da simulação, principalmente através do estado e do instante em que deve ocorrer o próximo evento.

   Além das estruturas referentes aos processos, são criadas duas outras. A primeira

emula um processo para o meio de comunicação disponível, ou melhor cria um processo equivalente aos demais processos para cada ligação entre processadores. Para uma máquina de memória compartilhada com barramento comum seria criado apenas um processo. Em um hipercubo de dimensão $n$ seriam $\frac{n \times 2^n}{2}$ processos (estruturas).

A segunda estrutura criada é usada para o controle das demais estruturas pelo simulador. Essa estrutura é do tipo "blackboard", na qual ficam armazenadas as informações correntes sobre o conjunto de processos em simulação. Isso significa ter apontadores para as estruturas individuais de cada processo e, principalmente, listas dos processos ativos e bloqueados e um apontador para qual evento deve ser o próximo a ocorrer.

3. **Passo 4**:

   O atendimento a um evento provoca uma série de ações por parte do simulador. A primeira delas é a atualização do tempo de execução simulado, isto é, a passagem por um determinado vértice consome algum tempo do processador. Esse tempo é acrescido ao tempo já decorrido, para que a ocorrência de outros eventos possa ser habilitada.

   A seguir, o processo que gerou esse evento caminha em seu grafo, indo para o vértice indicado pela aresta percorrida. Essa ação implica em verificar o tipo do novo vértice e atualizar o estado do processo segundo as informações no vértice. Essa operação consiste em calcular o tempo para a ocorrência do próximo evento (somando-se o tempo de execução do vértice ao tempo atual) e verificar qual será o novo estado do processo (que pode ser "executando", "sincronizando", "comunicando" ou "encerrado"). Simultaneamente é escolhido o caminho pelo qual ocorrerá o próximo passo no grafo dentre as arestas emergentes do novo vértice. A estratégia de escolha por uma das arestas é descrita adiante.

   Finalmente, os demais processos têm suas estruturas atualizadas, caso estivessem esperando por uma comunicação ou sincronização. Isso implica em atualizar o estado e o tempo esperado para o próximo evento dos processos que estavam na lista de processos bloqueados e tenham sido liberados pelo evento recém-ocorrido.

4. **Passo 5**:

   Nesse passo verifica-se o critério de parada da simulação. A simulação termina quando todos os processos estejam no estado "encerrado", exceto, é claro, os processos relativos aos meios de comunicação, que nunca atingem esse estado. Caso algum processo ainda esteja executando é necessário que a simulação tenha continuidade, o que é feito com a escolha do próximo evento a ocorrer. A escolha é feita de forma trivial, isto é, escolhe-se o evento com menor instante de ocorrência entre todos os próximos eventos

de processos que estejam no estado "executando", ou seja, processos contidos na lista de processos ativos. Os processos que estejam na lista de bloqueados, quer seja por comunicação quer seja por sincronismo, não entram na disputa pois não podem ter seus instantes de ocorrência determinados.

5. **Passo 6**:

   Uma vez concluída a simulação é necessário que se faça a coleta e quantificação dos dados sobre a simulação. Essa etapa depende daquilo que for considerado como objetivo do processo de simulação. As informações típicas a serem coletadas aqui incluem o tempo total de execução, "speedup" relativo, os tempos gastos com processamento e comunicação, as taxas de ocupação dos processadores e dos suportes de comunicação, pontos ótimos de operação para o sistema, considerando-se variações na granulação e escalabilidade do programa.

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

Durante a descrição do algoritmo não foram dados detalhes sobre o modo usado para decidir por um dado caminho em desvios nem sobre a quantidade de vezes que um determinado ciclo será repetido. Anteriormente mencionou-se que essas decisões seriam tomadas a partir de números aleatórios gerados segundo determinadas funções de distribuição de probabilidade. Essas funções não podem ser idênticas em simulações diferentes nem a mesma em qualquer ponto de uma simulação. A necessidade de várias funções se prende basicamente às características distintas entre um ciclo e um desvio e entre dois programas quaisquer.

Primeiro, é necessário observar que um vértice DECISÃO usado no início de um desvio condicional, possui um número limitado de vértices descendentes imediatos. Esse limite faz com que o número aleatório a ser gerado tenha seu módulo equivalente a esse limite. Além disso, sabe-se que na maioria das vezes nem todos os caminhos são equiprováveis, ou seja, a probabilidade de que, numa execução real do programa, um caminho seja escolhido é quase sempre diferente da probabilidade de escolha por outro caminho.

Além disso, vértices que decidem pela continuidade ou não de um ciclo, devem ter um mecanismo distinto para a geração do número aleatório. Nesse caso, o número gerado (0 para sair do ciclo e 1 para continuar nele) deve ter diferentes probabilidades para cada evento amostral, sendo que inicialmente a probabilidade de ocorrência de um 1 é maior do que a de um 0, probabilidade essa que vai diminuindo a cada 1 obtido (o ciclo tende a terminar). Essa estrutura admite implicitamente que o programa não possui ciclos infinitos, ou seja, todo ciclo tem um final, exceto na improvável ocorrência de uma seqüência infinita de sorteios do

número 1.

Em ambos os casos deixa-se aberta a possibilidade de se usar funções de distribuição de probabilidade desenhadas para o programa em análise. Particularmente, para o protótipo implementado foram usadas funções com distribuição uniforme, normal e exponencial. Numa rápida análise, cada uma das funções se adapta melhor a uma das situações aqui descritas, como se pode ver na Tabela 3.1 dada a seguir:

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

**Tabela 3.1: Funções de distribuição de probabilidade e suas aplicações**

Feita esta apresentação, percebe-se que a precisão do simulador está relacionada basicamente com a adaptação coerente entre os geradores de números aleatórios e os pontos em que eles serão aplicados. Esse casamento pode ser alterado para uma melhor sintonia do modelo de interação programa-máquina, ou seja, os geradores de números aleatórios podem ter seus limites ajustados segundo expectativas do usuário quanto ao conjunto de valores possíveis para os testes durante a execução real do programa.

### 3.2.4 Vantagens e desvantagens desta proposta

Nesse momento é aconselhável que se examine qualitativamente o método proposto para que sejam discutidos os aspectos que potencialmente constituem vantagens ou desvantagens do mesmo. Assim, listam-se primeiramente as desvantagens:

- É necessário um programa compilado para a máquina. Mas isso também ocorre com a maioria das ferramentas analisadas, logo a desvantagem comparativa não é grande;
- Não fornece argumentos analíticos, isso é, comprovação através de um modelo matemático, para justificar os resultados. Qualquer tentativa nessa direção precisa demonstrar que os modelos de geração de números aleatórios é correta e perfeitamente

adequada ao problema;

- O uso de bibliotecas de reescrita de código para diferentes processadores restringe a portabilidade do método. Para obter a portabilidade desejada é necessária a reprogramação de um conjunto grande de operações na geração do grafo;
- O processo de simulação pode ser demorado, principalmente quando se pensa em máquinas massivamente paralelas, as quais exigiriam uma quantidade muito grande de processos controlados pelo simulador.

Das desvantagens apontadas a mais crítica é a possível lentidão da simulação. A velocidade do simulador depende fundamentalmente da quantidade de vértices contidos no(s) grafo(s) de execução e na quantidade de processos ligados a cada grafo. Portanto é fundamental que se examine como sua velocidade varia segundo o tamanho do grafo e o número de processos.

Do algoritmo da Figura 3.14 percebe-se que os pontos críticos para determinar a sua velocidade são a decisão por qual caminho a seguir, a decisão sobre qual evento irá ocorrer e as atualizações nos dados sobre cada processo desprezados os tempos para iniciação e apresentação dos resultados, que supostamente variam linearmente com o tamanho do problema. Nenhum desses pontos depende simultaneamente do tamanho do grafo e do número de processos, o que facilita bastante a análise da complexidade de suas operações.

A velocidade da operação de escolha por um caminho no grafo depende do número de arestas que partem do vértice para o qual se destina a aresta percorrida pelo evento atual. Isso cria uma dependência quanto ao tamanho do grafo e quanto ao algoritmo de geração de números aleatórios. Este último é independente do problema, logo não será considerado. Quanto ao tempo necessário para decidir qual o caminho a seguir pode-se considerá-lo diretamente proporcional ao tamanho do grafo, tendo relação linear com constante de proporcionalidade menor que um, já que para cada aumento no tamanho do grafo é gerada uma quantidade proporcional de arestas, mas muitas delas ficam fora do caminho seguido durante a simulação, não aumentando na mesma proporção o número de arestas realmente testadas.

Quanto ao tempo gasto para decidir qual o próximo evento a ser executado, tem-se que o mesmo depende apenas do número de processos presentes na simulação. Como a escolha é feita pegando-se o evento com menor instante de ocorrência dentre os processos ativos, tem-se que esse tempo é diretamente proporcional ao número de processos com constante igual a um.

Já em relação ao problema de atualização dos dados sobre cada processo, tem-se que o tempo gasto nessa tarefa depende apenas do número de processos no simulador. Como

essa atualização depende basicamente de buscas na lista de processos bloqueados, para verificar se o evento pelo qual estavam esperando já ocorreu, tem-se que o tempo necessário para essa atividade depende da estrutura utilizada na implementação da lista de processos. Mais uma vez, também aqui existem excelentes implementações de algoritmos de busca e ordenação, que permitem manter o tempo dentro de limites aceitáveis mesmo para uma quantidade elevada de processos.

Uma vez analisadas as desvantagens, passa-se a examinar as vantagens da abordagem proposta:

- O modelo do programa é mais preciso.

  Isso é verdade, principalmente quando se percebe que o grafo simulado tem seus tempos de execução (em cada vértice) determinados de forma precisa, sem interferência de mecanismos de extração de traços como na maioria das ferramentas existentes. Se for necessária a extração de traços de eventos, para análises mais detalhadas de desempenho e afinação do programa, eles podem ser obtidos sem deformar os tempos consumidos de fato no processamento do programa.

  A única restrição a ser feita quanto à precisão do modelo é o uso de números aleatórios para tomadas de decisão em desvios condicionais e ciclos. Embora esse fato seja realmente relevante, é sabido que a maioria das demais metodologias também fazem simplificações que reduzem sua precisão. Mesmo os "benchmarks" dependem fundamentalmente dos dados de entrada usados, os quais podem não representar situações reais.

- Tem um baixo custo de utilização.

  Como o volume de dados necessários ao simulador pode ser perfeitamente acomodado numa única estação de trabalho, não é preciso que se gaste em maquinário. Esse baixo volume de informações vem do fato que apenas uns poucos grafos de execução são necessários para simular centenas de processos, pois para cada processo é necessário apenas um apontador privativo para o grafo que estiver executando, o qual pode ser compartilhado por vários processos.

- Sua capacidade de adaptação depende apenas da existência de bibliotecas para reescrita de código.

  Embora essa característica possa ser vista também como uma desvantagem, isto é, seria necessário uma biblioteca para que o método possa ser executado para outra máquina, é preferível ver este aspecto como uma vantagem, uma vez que se o sistema for devidamente modulado, a construção de diferentes bibliotecas pode ser obtida com um baixo custo de implementação.

- O modelo da máquina depende basicamente de detalhes técnicos.

  Os detalhes necessários para o modelo de máquina são, de fato, fartamente disponíveis e são suficientes para que se faça a simulação com boa precisão. Esse fato torna o método fácil de ser usado.

- O modelo de interação programa-máquina depende apenas de condições operacionais sobre o sistema.

  Isso facilita a utilização do sistema por um usuário que tenha algum conhecimento sobre o programa que está sendo analisado e sobre o ambiente em que ele deverá ser utilizado posteriormente. Na realidade, as informações necessárias aqui são facilmente determinadas por esse tipo de usuário, pois em sua maioria dizem respeito ao modo como o programa será executado no sistema.

Em linhas gerais, existe uma maior preponderância de fatores favoráveis a esse método. Suas vantagens suplantam em muito suas desvantagens. A única desvantagem realmente crítica é uma eventual baixa velocidade do simulador. Porém, isso depende da forma de implementação e do tipo de aplicação.

## 3.3 Implementação de um protótipo do método

Como indicado no final da seção anterior a determinação da velocidade do simulador, assim como do gerador do grafo de execução, só pode ser feita mediante a implementação desses sistemas. O que se descreve a seguir é a implementação de um protótipo de uma ferramenta que realiza a geração de um grafo de execução e sua simulação, com o objetivo de dimensionar a escalabilidade do conjunto programa-máquina especificado no próximo capítulo. Nas próximas páginas será feita uma descrição geral da implementação do protótipo, seguida de discussão mais detalhada sobre os principais problemas encontrados na implementação de cada um dos seus três módulos.

### 3.3.1 Descrição geral do protótipo

Já se mencionou que um protótipo para o método formulado é composto por três módulos básicos: gerador do grafo de execução, módulo para a otimização do grafo e o simulador. Nesta seção descreve-se quais foram as técnicas utilizadas na implementação desses módulos.

O protótipo foi implementado para ser executado em estações de trabalho usando o sistema operacional UNIX[7]. O mesmo não foi implementado para máquinas DOS por ser mais simples a manipulação de grandes quantidades de dados em máquinas UNIX e também por se saber que grande parte do desenvolvimento de programas paralelos é feito nesses ambientes. Assim, a mesma plataforma usada no desenvolvimento do programa poderia ser usada para fazer a predição de seu desempenho.

A linguagem de programação utilizada é o ANSI-C. A opção por essa linguagem foi feita pela sua portabilidade e também pela sua capacidade de manipulação em baixo nível. O compilador usado foi o **gcc**[8] versão 2.7.2, executando numa estação **Sun sparc-5**, rodando **Solaris**[9] versão 3.2. A máquina foi escolhida por sua disponibilidade de acesso e por sua capacidade de processamento. Testes preliminares feitos entre as várias máquinas disponíveis revelaram que a mesma era, em geral, duas vezes mais rápida do que as demais. Quanto ao compilador, optou-se pelo **gcc** por sua disponibilidade em um grande número de sistemas e por ter se revelado um compilador mais preciso que os demais, detectando pontos potenciais de erros de execução que não eram acusados por outros compiladores, em especial o **cc** disponível no ambiente **SunOS**.

Determinado o ambiente de implementação do protótipo falta explicitar o que está realmente nele implementado e que medidas de desempenho podem ser tomadas com sua utilização. As respostas para essas questões estão condicionadas ao objetivo principal da implementação do protótipo. Como o que se pretende aqui é exemplificar e testar a proposta de uma nova metodologia para se estimar o desempenho de um programa paralelo, a partir de sua execução simulada, e não a implementação de uma nova ferramenta para a predição de desempenho, o protótipo tem algumas limitações inerentes a todo protótipo.

Por exemplo, o protótipo implementado não possui uma interface amigável, o que seria necessário numa ferramenta. Ele também não está disponível para vários processadores, sendo implementado apenas para a família R2000/R3000 fabricados pela MIPS, pois o caso estudado no Capítulo Quatro usa máquinas equipadas com esse processador. O protótipo também não faz a preparação de traços de eventos, que poderiam ser usados numa ferramenta de análise de desempenho. A grande preocupação na implementação do protótipo foi obter dados que permitissem verificar:

1. Qual o tamanho ótimo do grão numa dada configuração de máquinas;
2. Qual o grau ótimo de paralelismo para um dado tamanho de grão;

[7]UNIX é marca registrada da AT&T.
[8]gcc é publicamente disponível, com copyright da Free Software Foundation, Inc.
[9]Solaris e SunOS são marcas registradas da Sun Microsystems Inc.

3. Qual a combinação ótima entre tamanho do grão e grau de paralelismo.

Confrontando-se essas medições com valores obtidos em testes sobre o sistema real, pode-se atingir plenamente o objetivo do protótipo, que é verificar experimentalmente a validade da metodologia proposta. As características que não foram implementadas servem apenas para aperfeiçoar o protótipo na direção de uma ferramenta mais poderosa para a análise prévia do desempenho de sistemas paralelos. Nenhuma delas é essencial para que se verifique a validade da metodologia, tanto no que se refere à sua eficiência, quanto à sua precisão.

Embora não tenha sido implementada a organização geral de uma ferramenta de predição/análise de desempenho usando a metodologia proposta é mostrada na Figura 3.15. Nela observa-se com clareza como seria seu funcionamento para vários modelos de processadores e máquinas. Mostra-se também a manipulação de dados realizada para que mais informações possam ser obtidas durante a simulação. As bibliotecas de processadores contêm todas as funções de interpretação de código específicas para cada processador. Já os vários modelos de máquina descrevem arquiteturas paralelas reais ou hipotéticas. Para máquinas hipotéticas a simulação é possível desde que o processador usado tenha um compilador disponível. Finalmente, o usuário também pode definir quais são os resultados que lhe interessam através da interface com o usuário, que passaria ao simulador a relação de medidas que teriam que ser tomadas e armazenadas em um banco de dados de execução.

### 3.3.2 Implementação do gerador do grafo de execução

A geração do grafo de execução é feita executando-se as etapas de leitura do código, interpretação das instruções de máquina e agrupamento de instruções em blocos seqüenciais. As tarefas executadas em cada uma dessas etapas já foram indicadas na seção 3.2 e, portanto, apenas serão apresentados aqui os detalhes relativos à implementação de cada um deles.

#### A. Leitura do código executável

A etapa de leitura do código é a mais simples de todas e consiste basicamente em ler todo o código executável, instrução por instrução, partindo o mesmo em conjuntos de rotinas, depois deve-se separar cada uma das rotinas em arquivos individuais, criando uma tabela de nomes que associa cada uma das rotinas a um dado arquivo. Nesse processo padronizam-se os seguintes formatos para os arquivos mencionados:

- Arquivos contendo rotinas:

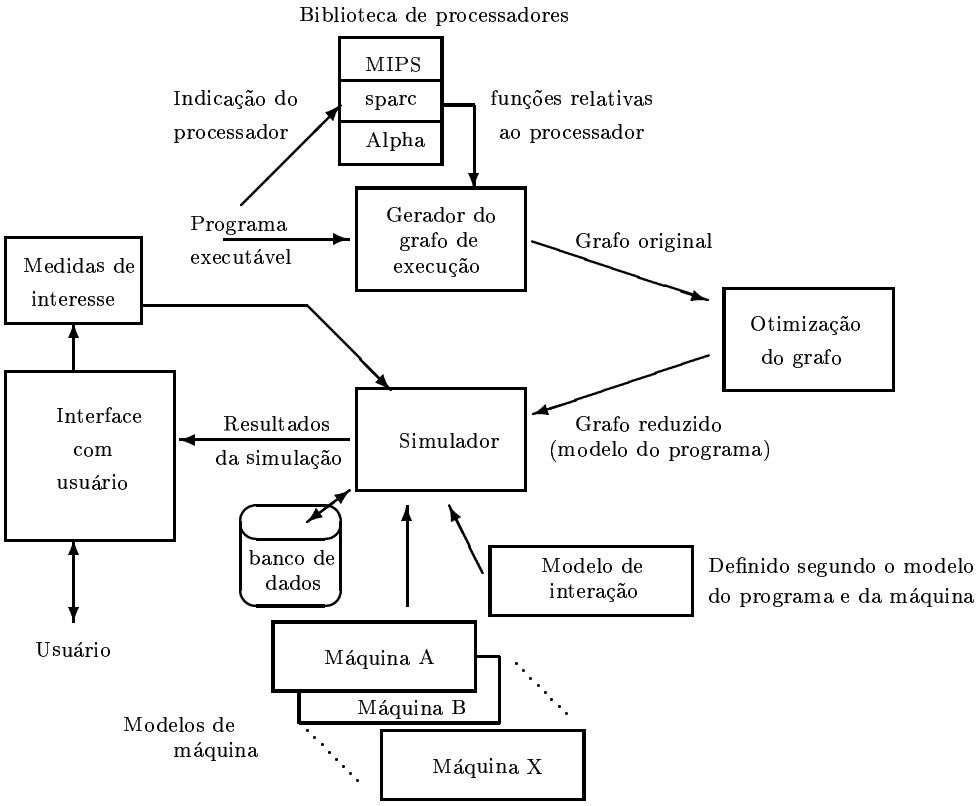



**Figura 3.15: Organização de uma ferramenta baseada na metodologia proposta**

  - Endereço lógico da instrução
  - Código hexadecimal da instrução
  - Mnemônico para a instrução

- Tabela de nomes:

  - Nome da rotina
  - Nome do arquivo, que é dado pelo endereço lógico da rotina no programa por ser um valor único para cada peça de informação

Além disso, é gerada uma tabela ligando nomes de arquivos a seus endereços lógicos, que é uma cópia exata da tabela de símbolos contida no programa. Essas duas tabelas (a de nomes e a de símbolos) são lidas pelo gerador do grafo e armazenadas em estruturas vetoriais. Já os arquivos contendo as rotinas serão lidos seguindo a ordem em que forem necessários numa suposta execução do programa.

Na implementação do protótipo essa etapa foi consideravelmente facilitada com o uso de ferramentas já disponíveis em UNIX. As ferramentas são os comandos "dis" e "elfdump", além de pequenos "scripts" usando o comando "awk".

O comando "dis" faz o desmonte do código executável, gerando um arquivo com as instruções e seus endereços lógicos. As instruções aparecem no formato de palavra (oito dígitos hexadecimais) e também de mnemônico. Os endereços são também palavras de oito dígitos hexadecimais. A Figura 3.16.a apresenta um pequeno trecho de um arquivo produzido pelo "dis".

O comando "elfdump", disponível no sistema operacional IRIX da SGI, equivalente ao "dump" existente no Solaris, retira as informações sobre a tabela de símbolos do programa. Como pode ser visto na Figura 3.16.b, ele apresenta uma tabela contendo informações sobre os símbolos do programa, seus endereços lógicos, tamanho e em que segmento eles se encontram, entre outras coisas. O uso desse comando veio a facilitar a implementação do módulo de interpretação de instruções, principalmente no momento de interpretar endereços de destino em chamadas de função através da instrução JALR [67].

A fragmentação do arquivo obtido com o "dis" é feita com a execução de um pequeno programa que lê esse arquivo e o reescreve em vários outros, criando um arquivo diferente para cada rotina identificada pelo "dis". Enquanto faz a criação dos arquivos contendo as rotinas, esse mesmo programa faz também a criação da tabela de nomes, gerando automaticamente os nomes dados para os arquivos. A estratégia de nomenclatura foi a de denominar cada arquivo através da concatenação dos caracteres **f_** com o endereço inicial da rotina em hexadecimal, como por exemplo **f_004009a0** para a função *main* indicada na Figura 3.16.a

### B. Interpretação das instruções

Uma vez dividido o código em funções pode-se fazer com que cada função seja analisada separadamente, permitindo a reutilização de trechos do grafo de execução que contenham funções previamente analisadas. Isto é útil para o tratamento de chamadas recursivas de rotinas e mesmo rotinas não-recursivas chamadas de pontos distintos do programa. O procedimento básico da interpretação é iniciar a leitura dos arquivos de código a partir daquele contendo a função "start_", que inicia a execução do programa. Ao realizar a geração do grafo de execução dessa função é necessário que se faça também a geração do grafo para outras funções chamadas por "start_", dentre as quais está a função "main" (ou equivalente), onde começa o código escrito pelo programador. Esse processo continua até que se chegue ao final da função pela qual se começou a interpretação.

```
        main:
0x4009a0:   3c1c0fc0        lui     gp,0xfc0
0x4009a4:   279c76d0        addiu   gp,gp,30416
0x4009a8:   0399e021        addu    gp,gp,t9
0x4009ac:   27bdfef8        addiu   sp,sp,-264
0x4009b0:   afbc0018        sw      gp,24(sp)
```

(a) Código produzido pelo dis.

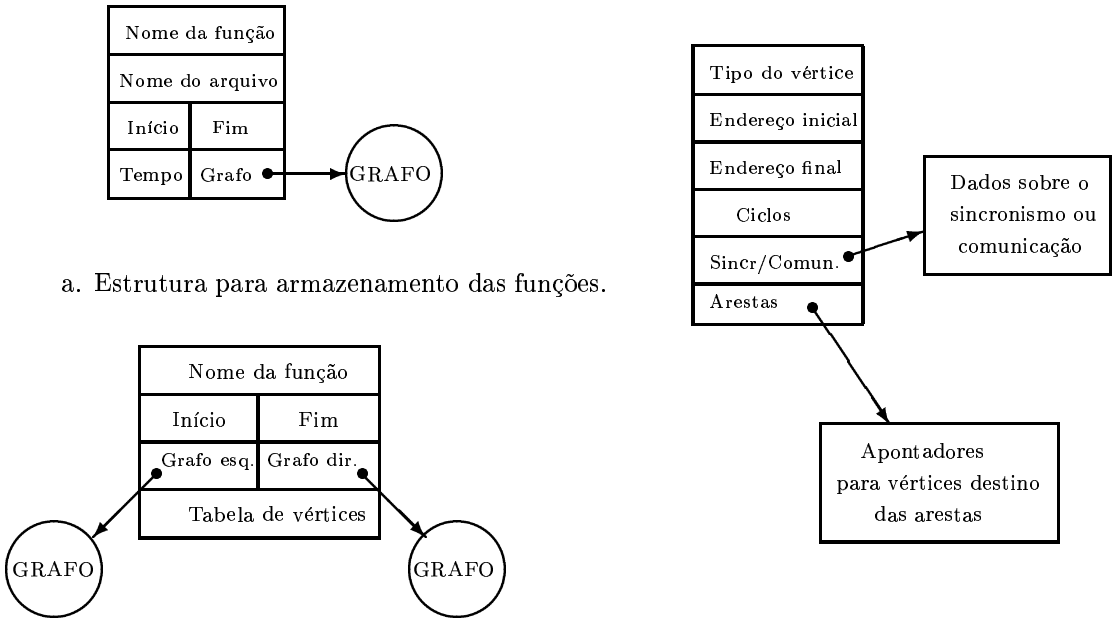
```
[   0]                          0x00000000   0   LOCAL   NOTYPEUNDEF   0
[   1] .text                    0x004073f0   0   LOCAL   NOTYPE.text    0
                                    ⋮
[  20] bump_tape                0x00407d94   0   GLOBALFUNC  .text     0
[  21] cps_get_next_tape_       0x00410c30   0   GLOBALFUNC  .text     0
```

(b) Código produzido pelo elfdump.

**Figura 3.16: Saídas dos comandos *dis* e *elfdump*.**

Para tornar possível a interpretação correta de todo o código é necessário criar algumas estruturas de dados para o armazenamento de informações sobre cada função examinada, sobre o estado dos registradores de uso geral do processador e sobre o grafo que está sendo gerado. A necessidade da estrutura que armazena o grafo e da que armazena informações sobre as funções é claramente visível. Já a estrutura referente aos registradores tem uma finalidade só percebida no momento de tratar as ações que cada instrução indicam ao processador. Essa estrutura de registradores é utilizada para armazenar informações que o programa vai utilizando durante sua execução, principalmente os endereços de rotinas e de retorno de chamadas de rotinas. Esses dados são extremamente necessários para que se possa criar grafos de execução conexos. Apesar de sua importância, a implementação é realizada simplesmente através de um vetor, indexado pelo código do registrador e contendo os "conteúdos" de cada registrador.

Voltando agora para as duas primeiras estruturas, tem-se que a estrutura usada para as funções é basicamente uma tabela "hash", indexada pelo endereço inicial da função, contendo em cada entrada o nome da função, nome do arquivo que a contém, endereços inicial e final da mesma, tempo médio esperado para sua execução (calculado apenas após a simulação) e um apontador para uma estrutura que armazena o seu grafo de execução, como




a. Estrutura para armazenamento das funções.

b. Estrutura dos grafos do programa.

c. Estrutura dos vértices dos grafos.

**Figura 3.17: Estruturas usadas para a geração do grafo de execução.**

mostrado na Figura 3.17.a.

Já a estrutura que armazena o grafo de execução é mais complexa visando facilitar operações de busca no grafo, necessárias no tratamento de desvios condicionais ou não. A Figura 3.17.b apresenta a organização geral dessa estrutura. O essencial aqui é perceber que a mesma está organizada numa forma de árvore de grafos, em que cada nó da árvore contém um grafo de execução com vértices apontados através de uma *Tabela de Vértices*, que contém apontadores para todos os vértices do grafo. Esse nó é formado basicamente por algumas informações gerais sobre a função (nome e seus endereços inicial e final) e a Tabela de Vértices do grafo, além dos apontadores para o grafo a esquerda e a direita desse grafo, ou seja, para os nós vizinhos na árvore de grafos. A indexação na árvore é feita pelo endereço inicial da função representada em cada grafo.

Na tabela de vértices se armazenam todos os vértices de um grafo, sendo que para cada um deles existe uma estrutura de dados (ver Figura 3.17.c) com informações sobre o tipo do vértice (sincronismo, comunicação, desvio interno, chamada de função, retorno de função ou execução), endereços inicial e final do vértice, número de ciclos de máquina consumido pelas instruções do vértice, dados sobre o ponto de sincronismo ou comunicação, quando for o caso, e apontadores para os vértices sucessores.

O processo de geração do grafo passa a ser, portanto, uma seqüência de interpretações de instruções, até que se determine o fim de um bloco, ou seja, se tenha um vértice determinado no grafo. Nesse momento, a estrutura do grafo é atualizada, com a inserção de um ou mais vértices em sua tabela de vértices. A quantidade de vértices é determinada pelo tipo de instrução. Instruções de desvio geram entre zero e duas arestas e as demais uma única aresta. Se o desvio for uma chamada de função (incondicional portanto) os vértices criados representam uma aresta para o grafo da função chamada e uma aresta virtual de retorno daquela função para a instrução seguinte no programa, sendo esta a única exceção no direcionamento das arestas.

### C. Agrupamento em blocos

Da forma como é feita a interpretação de instruções se pode perceber que o agrupamento das mesmas em blocos com execução seqüencial já está prevista no processo. Na realidade todo o processo de interpretação poderia ser feito sem qualquer tipo de agrupamento, desde que se armazenasse em algum lugar qual a seqüência exata de instruções no programa. O problema em fazer isso é que o consumo de memória cresce de forma exagerada e ainda seria necessária a criação de uma linguagem intermediária para o armazenamento. Assim, é muito mais eficiente fazer o agrupamento já durante a interpretação das instruções, sendo a operação conjunta bastante simples. Essa prática diminui o consumo de memória pelo programa sem aumentar o tempo total de processamento .

Ao operar dessa forma não são necessárias novas estruturas de dados, uma vez que todas as estruturas apresentadas na Figura 3.17 são suficientes para esses dois módulos. Na realidade, as estruturas relativas aos grafos e seus vértices são usadas apenas na fase de agrupamento em blocos.

O agrupamento em blocos é controlado pelo tipo de instrução que estiver sendo lida. O resultado da interpretação vai classificar as instruções como sendo de salto ou não. Além disso, existe um controle feito sobre o endereço da instrução corrente para permitir a determinação correta de pontos finais de desvios, o que é feito examinando-se uma lista ordenada contendo todos os endereços iniciais de desvios que ainda não tenham sido percorridos. Quando a instrução for classificada como sendo de salto são determinados os endereços possíveis de desvio. Esses endereços podem ser de vários tipos, segundo o tipo de instrução de desvio utilizada. Assim, para instruções de chamada de função são determinados dois endereços, o da função chamada e outro de retorno da função. Para instruções de retorno de função nenhum endereço é calculado, apenas se recupera um endereço da lista de caminhos

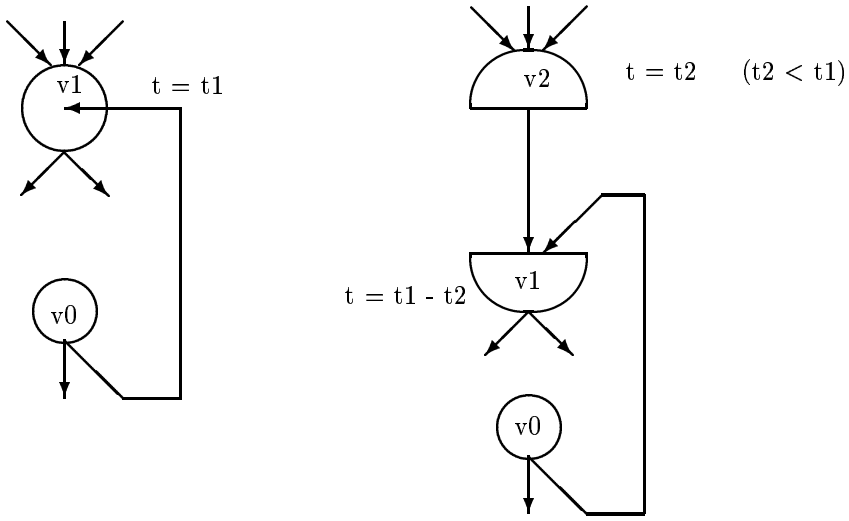



**Figura 3.18: Divisão de um vértice após desvio para trás.**

ainda não percorridos. Agora, se a instrução for de desvio condicional são calculados dois endereços, um sendo o endereço de desvio quando o teste obtiver sucesso e outro com o endereço da instrução seguinte.

Quando a instrução for um desvio condicional ainda se deve prestar atenção ao sentido tomado pelo desvio, isto é, se o desvio é feito para frente ou para trás. Se o desvio for para frente não existe nenhuma dificuldade, bastando acrescentar o endereço de desvio na lista dos caminhos não percorridos e continuar a análise criando um novo bloco que tem início na instrução seguinte à atual. Porém, se o desvio for para trás é necessário que se faça uma análise sobre a configuração atual do grafo, procurando determinar se o desvio está sendo feito para o início de um vértice ou se é para o meio desse vértice. No primeiro caso não existem problemas. No segundo caso esse vértice deve ser repartido em dois outros vértices.

A divisão do vértice (*v1*) é feita da seguinte forma: primeiro copiam-se as arestas que nele incidiam no vértice que está sendo criado (*v2*). Depois, determinam-se quantos ciclos são gastos desde a entrada no vértice *v1* até o ponto de partição. Esse número de ciclos é subtraído do total de ciclos do vértice e também atribuído ao vértice *v2*. O novo vértice passa a ter uma aresta partindo dele até *v1*, que tem todas as demais arestas incidentes eliminadas. Por fim, é estabelecida uma aresta entre o vértice que gerou o desvio para trás (*v0*) e o vértice *v1*. Todo esse procedimento pode ser visualizado na Figura 3.18.

As estruturas descritas nesta seção, bem como o modo de operação para se obter o grafo de execução, incluindo-se a interpretação das instruções e seu agrupamento em

blocos maiores, dão uma clara idéia da forma de implementar esse tipo de ferramenta. Na realidade, embora exijam um certo grau adicional de instrumentação no código executável, os depuradores de programas existentes também fazem a interpretação das instruções presentes no programa, mapeando-as com o código fonte em vez de um grafo de execução. Dessa forma, não será feita aqui uma descrição mais detalhada dos programas envolvidos na geração do grafo.

### 3.3.3 Implementação da redução do grafo de execução

Dando continuidade à descrição do protótipo, passa-se agora ao exame das técnicas usadas no módulo de otimização do grafo. Na seção 3.2.2 foram apresentadas três formas de redução. Para a implementação de qualquer uma das formas é simplesmente necessário percorrer exaustivamente o grafo, procurando por vértices que possam ser reduzidos. Como cada forma se destina a um determinado tipo de vértice, a aplicação de uma delas pode influenciar a aplicação das demais.

Portanto a primeira coisa a resolver na implementação desse módulo é decidir em que ordem a redução será realizada. Dentre os três tipos, aquele no qual se reduzem vértices PASSAGEM é o que mais é influenciado pelos demais, pois esses também consomem vértices PASSAGEM ao fazerem suas reduções. Dessa forma, o trabalho computacional pode ser menor se essa for a última redução a ser realizada. Quanto aos outros dois tipos, a redução de vértices AGRUPAMENTO deve ser feita depois da redução de vértices DECISÃO, segundo argumentação semelhante. Em resumo, a otimização do grafo reduz os vértices na ordem: decisão $\rightarrow$ agrupamento $\rightarrow$ passagem.

#### A. Redução de vértices DECISÃO

A implementação deste tipo de redução vem diretamente das regras de como fazer sua aplicação. Assim, o que se faz é encontrar no grafo de execução todos os vértices desse tipo, partindo sempre do vértice INICIAL de cada grafo. Para cada vértice encontrado, examinam-se os vértices destino de suas arestas, procurando determinar simultaneamente aquele com menor tempo de execução e se nenhum deles é um vértice de comunicação ou sincronismo. Se for possível fazer a redução, aplica-se a mesma e repete-se a análise para o mesmo vértice, até que não seja mais possível reduzir suas arestas. Quando não for possível fazer a redução num dado vértice, procura-se por outro vértice DECISÃO, caminhando-se no grafo por profundidade, até que todos os vértices desse tipo tenham sido examinados.

**B. Redução de vértices AGRUPAMENTO**

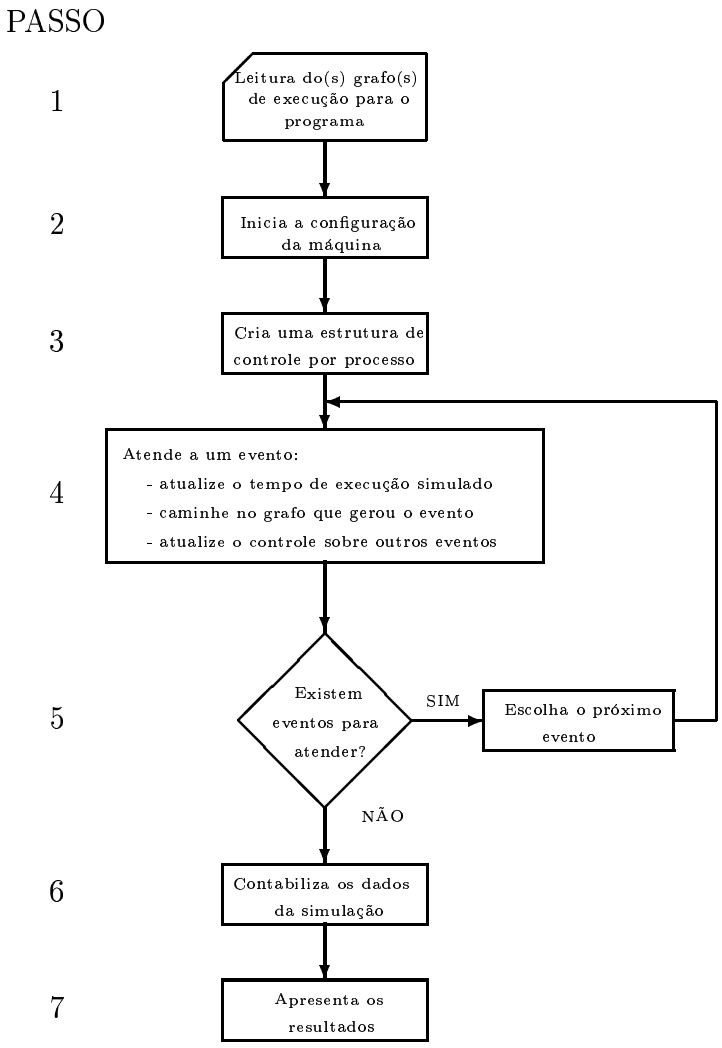
Do mesmo modo, a implementação desse tipo de redução é bastante simples. Aqui os vértices procurados são do tipo AGRUPAMENTO, ou seja, aqueles com mais do que uma aresta incidindo. Aqui também se caminha em profundidade e a aplicação da redução pode ser recursiva. O procedimento consiste em examinar se o destino de um vértice de agrupamento permite que seja aplicada a redução, isso é, se ele atende às restrições impostas em 3.2.2. Quando for possível fazer a redução, o vértice é removido e as arestas que nele incidiam passam a incidir sobre o vértice que era seu destino. Com isso, esse vértice passa a ser do tipo AGRUPAMENTO e também se torna passível de uma redução. Esse processo continua até que não existam mais vértices em que se possa aplicar redução.

**C. Redução de vértices PASSAGEM**

A última das técnicas de redução também é a de aplicação mais simples. Aqui são procurados os vértices PASSAGEM que por alguma circunstância especial ainda não tenham sido reduzidos e que não sejam vértices nem de comunicação nem de sincronismo. Para cada vértice nessa situação é aplicada diretamente a técnica de redução, ou seja, o vértice é eliminado, a aresta que nele incidia passa a incidir no vértice que era seu destino e o tempo que ele consumia é agregado ao tempo do seu vértice destino.

### 3.3.4 Implementação do simulador

Para encerrar a descrição do protótipo falta abordar a implementação do simulador. Os detalhes de sua implementação podem ser mostrados a partir da descrição de seu funcionamento, realizada através do algoritmo da Figura 3.14, repetida aqui para facilitar a leitura. Assim, cada um dos passos indicados foi implementado da seguinte forma:




**A. Passo 1**

É implementado simplesmente com funções para manipulação de arquivos e de estrutura de dados. Aqui, a estrutura de dados usada é matricial para agilizar o processo de leitura dos grafos. Caso fossem usadas as estruturas apresentadas na Figura 3.17 seria necessário um grande esforço de busca no grafo por cada vértice lido, para verificar se o mesmo já estava ou não no grafo. Com a estrutura matricial o processo de busca pode ser feito apenas verificando-se se a posição indicada por um par de índices (linha e coluna) já está ou não preenchida. A estrutura matricial, vista na Figura 3.19.a, permite que em uma única estrutura sejam armazenados todos os grafos de execução de um dado programa, bastando usar uma "linha" da matriz para cada grafo, deixando que as colunas identifiquem os vértices do mesmo.


a. Estrutura para um vértice do grafo

b. Estrutura para a lista de descendentes

**Figura 3.19: Estruturas para vértices dos grafos durante simulação.**

Vale observar nessa figura que as principais informações contidas em cada vértice do grafo são o seu tipo, isto é, uma indicação se o mesmo é de execução, sincronismo, decisão, repetição, comunicação ou ainda uma chamada de função, o tempo consumido pelas instruções contidas no vértice e uma lista dos vértices descendentes, a qual está indicada na Figura 3.19.b. A estrutura da lista de descendentes consiste simplesmente em pares de índices que indicam a posição na matriz em que se encontra o vértice descendente.

### B. Passo 2

Aqui também são feitas apenas leituras de arquivos com dados técnicos sobre a máquina paralela. Além disso, o usuário é questionado sobre alguns detalhes da interação programa-máquina, tais como número de processadores que serão usados, carga em cada processador, etc..

### C. Passo 3

Com as informações já coletadas é possível criar as estruturas de controle para cada processo. Essas estruturas armazenam dados sobre o estado do processo, quanto tempo ele já consumiu durante sua execução simulada, quando deverá ocorrer seu próximo evento e um apontador para o vértice destino da aresta que esse evento representa.

A criação dessas estruturas é simples, bastando que se crie uma lista com todos os processos que farão parte da simulação, inclusive os processos relativos aos suportes de comunicação disponíveis.

Em relação à estrutura usada pelo controlador da simulação, também se percebe que a mesma é simples, consistindo em duas listas de processos, uma com aqueles livres para executar e outra com aqueles bloqueados à espera de algum sincronismo ou comunicação. Dentro das listas já ficam incluídos os apontadores para as estruturas dos respectivos processos,

facilitando o trabalho de busca pelo evento a ser executado.

**D. Passo 4**

Esse é o passo que exige mais atividade do simulador. Para sua implementação é necessário que se resolvam alguns problemas de manipulação dos grafos de cada processo e das listas do controlador da simulação, além de algumas atualizações sobre o estado do processo que ocasionou o evento.

Primeiramente, a ocorrência do evento faz com que o tempo simulado para o processo seja acrescido do tempo que será consumido pelo vértice ao qual a aresta se destina. Além disso, verifica-se qual o tipo daquele vértice. Caso seja um vértice de comunicação ou sincronismo deve-se alterar o estado do processo. Na hipótese de ser um vértice de sincronismo ainda é preciso que se verifique se as condições de sincronismo podem ser satisfeitas ou não, ou ainda, se existe algum processo bloqueado que esperava por esse evento. Essas verificações envolvem buscas na lista de processos bloqueados, o que é simples de se implementar.

Por outro lado, caso o processo continue em execução, este é novamente inserido na lista de processos ativos, a qual é mantida ordenada segundo o tempo previsto para a ocorrência do próximo evento de cada um dos processos. Nesse momento também se verifica se o vértice para o qual o processo caminhou é um vértice de desvio. Se não for nada mais há para fazer, mas se for é preciso que se escolha qual das arestas (caminhos de execução) deve ser seguida.

A decisão do caminho a ser seguido é feita com o uso dos geradores de números aleatórios descritos anteriormente. A definição por um dos geradores é feita a partir da identificação do desvio como sendo uma simples ramificação ou um ciclo. Para o protótipo foram testadas funções de distribuição normais e exponenciais, geradas a partir de transformações sobre números gerados pela função uniforme disponível na biblioteca "math.h" da linguagem C.

**E. Passo 5**

Sua implementação é muito simples, principalmente com a manutenção da lista de processos ativos ordenada pelo tempo de ocorrência dos eventos. Assim, basta verificar se essa lista ainda contém algum processo e, em caso positivo, retirar da lista o seu primeiro elemento, voltando ao passo 4.

### F. Passo 6

Se a lista de processos ativos estiver vazia, inicia-se a coleta dos dados da simulação realizada. Como nesse protótipo não houve a preocupação em obter traços dos eventos, que permitiriam análises mais detalhadas sobre o desempenho e afinação do programa, a coleta de dados também é bastante simples. Os dados que precisam ser levantados são o tempo total de execução, o tempo útil de processamento em cada processador e o tempo consumido com comunicação e sincronismo.

Todos esses valores podem ser obtidos diretamente das estruturas dos processos, bastando que a cada evento ocorrido, se atualize as informações de tempo dessas estruturas. Dessa forma, ao final da execução os tempos estão todos disponíveis.

### G. Passo 7

Como interface de apresentação dos resultados foi implementada uma versão simples de listagem dos tempos obtidos no passo anterior, tanto para tela quanto para arquivo. Obviamente é desejável dispor de formas gráficas de apresentação dos resultados, trabalho esse que fica para a implementação de uma ferramenta completa para análise de desempenho baseada nesta proposta.

# Capítulo 4

# O estudo de um caso

Neste capítulo serão descritos os testes realizados com o protótipo implementado a partir das definições feitas no capítulo anterior. Deve-se salientar que o objetivo desses testes é comprovar experimentalmente se a metodologia proposta neste trabalho é viável e se merece ter um maior desenvolvimento. Partindo-se desse pressuposto, o caso de teste foi escolhido de forma a possibilitar diferentes especificações e também permitir comparações com medidas obtidas em sistemas reais.

Assim, primeiro se descreve o programa analisado e também o ambiente em que ele foi executado na realidade. A seguir são apresentados os resultados obtidos por simulação e as condições em que as mesmas foram realizadas. Apresentam-se também os resultados obtidos através de "benchmarking", possibilitando a comparação desses resultados com os obtidos por simulação para verificar a precisão do protótipo aqui implementado.

## 4.1 Definição do caso

O caso de teste escolhido foi um programa para reconstrução de eventos[10] usado em uma experiência de física de altas energias realizada no Fermilab[11]. Uma descrição superficial dessa experiência pode ser encontrada no Apêndice B. Neste momento a preocupação é com os detalhes do programa e da máquina paralela que serão importantes para a realização das simulações. Assim, nas próximas páginas serão apresentados dados técnicos sobre o programa testado e sobre o ambiente para o qual o programa foi originalmente projetado.

[10] evento significa a medição das partículas resultantes de um choque no alvo.
[11] Fermi National Accelerator Laboratory, localizado em Batavia, Illinois, EUA.

### 4.1.1 O problema analisado

Como mencionado, o programa usado como teste para o protótipo desenvolvido foi um programa para reconstrução de eventos em física de partículas. As características desse programa que devem ser levadas em consideração pelo simulador são as seguintes:

- O programa é composto por três outros programas. Um mestre, que lê os dados de entrada e os passa aos escravos executando o programa cliente, que faz a reconstrução e seleção de eventos, que os passa ao terceiro programa para escrever os resultados obtidos.
- A comunicação entre os vários programas é feita através de primitivas de sincronismo e troca de mensagens usando protocolo TCP/IP.
- Cada nó da máquina paralela executa uma instância desses programas.
- Os programas de leitura e escrita de dados possuem apenas uma instância cada.
- Os clientes possuem tantas instâncias quantos forem os nós disponíveis na máquina.
- O controle de execução dos programas, bem como o fornecimento de funções especializadas para sincronismo e comunicação, é feito pelo CPS[12], que é um sistema desenvolvido no Fermilab e se encontra descrito no Apêndice A.

### 4.1.2 A máquina emulada

O programa de reconstrução era executado em rede de estações de trabalho especialmente configuradas numa topologia em barramento, com cada estação sendo tomada como um nó de uma máquina paralela fracamente acoplada. Cada rede de estações tinha um barramento dedicado, o que evita um tráfego excessivo no canal de comunicação.

A figura 4.1 mostra a configuração típica de uma "farm", que é o nome dessas redes no Fermilab. Nela podem ser vistos os barramentos externo e dedicado à rede. A "farm" é conectada ao mundo externo através de uma estação que atua como "host" da rede. Também é nessa estação que ficam as unidades de fita 8mm, que são as fontes de dados e destino dos resultados da reconstrução. O número de nós em cada "farm" varia de um até vinte, sendo que a "farm" em que se realizaram os "benchmarks" possuía vinte nós, além do "host".

Os programas indicados em 4.1.1 são executados da seguinte forma: programa leitor e programa escritor compartilham o "host" da farm, cada um deles acessando uma fita

[12] ©1992, 1993 Universities Research Association, Inc.

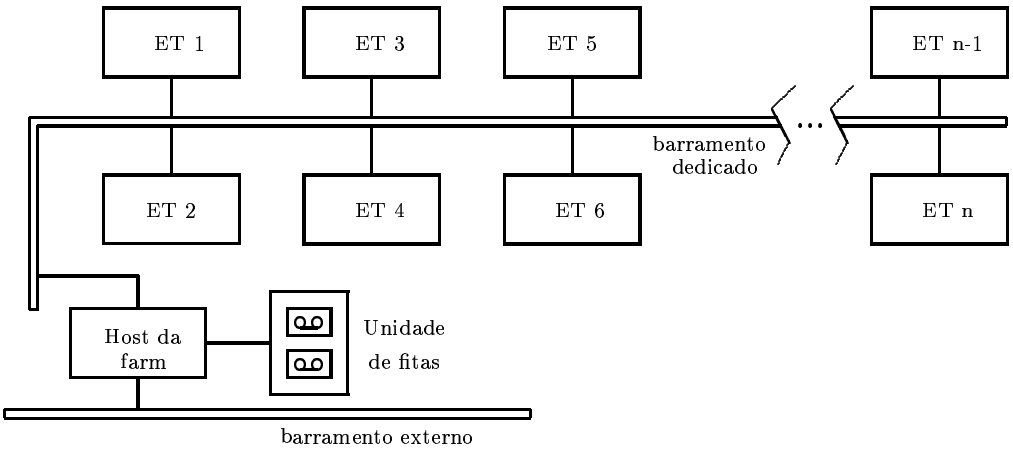



**Figura 4.1: "Farm" com $n$ estações de trabalho.**

da unidade de fitas. O programa cliente tem uma instância criada para cada uma das outras estações de trabalho e, em alguns casos, tem também uma instância criada no "host" para aproveitar parte do tempo de sua CPU, que fica ociosa pois os outros dois programas fazem mais entrada/saída de dados do que processamento de fato. A seguir são apresentados os dados técnicos sobre a "farm" e os três programas.

### 4.1.3 Detalhamento do sistema

- Programas

  1. Leitor: programa escrito em sua maior parte em Fortran 77, cujo executável tem um tamanho de 669616 bytes.
  2. Cliente: programa escrito em Fortran 77, cujo executável ocupa 4373648 bytes.
  3. Escritor: programa escrito em sua maior parte em Fortran 77, cujo executável ocupa 751208 bytes.

- "Farm"

  A Tabela 4.1 apresenta os principais dados técnicos sobre a "farm" usada como teste para o simulador.

| | |
|---|---|
| Número de estações de trabalho | 21 |
| Estações com capacidade de E/S | 1 |
| Processador usado nas estações | MIPS R3000 |
| Freqüência de relógio | 25 MHz |
| Tipo de barramento | Ethernet (10Mbps) |
| Tamanho de pacote de dados definidos pelos programas | 64 kbytes |
| Volume de dados de entrada | 2 Gbytes |
| Tempo médio de processamento | 6 horas |
| Carga nas CPU's clientes | 95% |
| Carga no "host" (sem cliente) | 40% |
| *Software* de controle | CPS |

**Tabela 4.1: Dados técnicos da "farm".**

## 4.2 Resultados obtidos por simulação

A seguir são apresentados os resultados obtidos por simulação em função de três tipos variáveis: número de processadores, tamanho do grão e modelo estatístico usado pelo simulador. Os dois primeiros testes procuram determinar o ponto ótimo de operação do sistema, enquanto o terceiro faz um estudo sobre qual tipo de função de distribuição de probabilidades modela de forma mais precisa o conjunto de programas em teste. Um quarto conjunto de resultados é acrescentado mostrando como o modelo do programa pode ser alterado para representar de forma mais fiel o sistema sob análise.

### 4.2.1 Parâmetros de simulação

Em todos os casos testados alguns parâmetros foram mantidos constantes. Alguns dizem respeito ao conjunto de dados apresentados na Tabela 4.1 e outros dizem respeito aos grafos de execução gerados para os programas *leitor*, *escritor* e *cliente*. Quanto aos grafos de execução tem-se que o importante é o número de vértices de cada grafo, o que pode ser visto na Tabela 4.2:

Além desses parâmetros, para os dois primeiros testes foi mantido constante o modelo de geração de números aleatórios usados para tomadas de decisão por qual aresta seguir em casos de vértices de decisão e/ou repetição. O modelo usou uma função de distribuição normal para vértices de decisão e uma função de distribuição exponencial para vértices de

| Programa | Vértices |
|---|---|
| Leitor | 1889 |
| Escritor | 3713 |
| Cliente | 74342 |

**Tabela 4.2: Número de vértices nos grafos de execução.**

repetição. Tais distribuições foram obviamente alteradas para permitir o terceiro teste.

### 4.2.2 Variação por número de nós

Como será visto na seção 4.3, apenas foi possível utilizar uma "farm" real para a realização de "benchmarking" variando-se o número de processos clientes executando em processadores isolados. Dessa forma, muito embora o simulador permita a realização de outros tipos de testes, como os apresentados em 4.2.3 e 4.2.4, é necessário que também sejam mostrados os resultados obtidos variando-se o número de processadores paralelos simulados. A Tabela 4.3 apresenta os resultados de desempenho do sistema obtidos através de simulação. Os valores indicados foram calculados a partir de valores médios de várias execuções do simulador, desprezando-se sempre os resultados extremos. Em cada caso o número de repetições foi diferente, procurando obter sempre um mínimo de seis execuções para os casos com maior número de clientes e dez execuções para os casos com menos de oito clientes.

Nessa tabela o tempo de processamento é o tempo total gasto para processar o conjunto de "records", pacotes de eventos em reconstrução (de dez a doze eventos por record), que são as unidades de dados de entrada para o sistema. A quarta coluna apresenta o tempo médio de processamento por record, enquanto a quinta indica esse tempo em cada cliente. É a partir dessas duas colunas que se obtém os valores de "speedup" (a quarta) e da eficiência de cada cliente (a quinta). Por fim, a sexta coluna apresenta o tempo gasto em comunicação por cada cliente.

Antes de analisar os dados constantes da Tabela 4.3 deve ser observado que o simulador ainda fornece o tempo gasto com sincronismo, discretizado por ponto de sincronismo, e uma distinção entre tempo gasto em espera por envio e recebimento de mensagens por cada processo. Essas informações, em conjunto com um "profile" da execução de cada processo, podem ser bastante úteis no momento de avaliar o desempenho de um programa e procurar alternativas para a sua melhoria.

Os valores apresentados na Tabela 4.3 podem ser visualizados através de gráficos

| Clientes | records | tempo de processamento (s) | veloc. média proc. (rec/s) | veloc. média por cliente (rec/s) | tempo médio espera por cli (s/rec) |
|---|---|---|---|---|---|
| 1 | 30 | 292.795 ±158.121 | 0.102 | 0.102 | 0.057 ±0.114 |
| 2 | 30 | 148.361 ±158.121 | 0.202 | 0.101 | 0.000 ±0.000 |
| 3 | 30 | 98.282 ±15.270 | 0.305 | 0.102 | 0.007 ±0.004 |
| 4 | 50 | 124.961 ±17.572 | 0.400 | 0.100 | 0.014 ±0.009 |
| 5 | 50 | 98.240 ±17.499 | 0.509 | 0.108 | 0.018 ±0.014 |
| 6 | 50 | 83.505 ±24.554 | 0.599 | 0.100 | 0.119 ±0.051 |
| 7 | 50 | 69.205 ±25.315 | 0.722 | 0.103 | 0.175 ±0.160 |
| 8 | 80 | 97.943 ±23.489 | 0.817 | 0.102 | 0.174 ±0.089 |
| 9 | 80 | 90.531 ±22.989 | 0.884 | 0.098 | 0.269 ±0.083 |
| 10 | 80 | 76.246 ±18.158 | 1.049 | 0.105 | 0.081 ±0.068 |
| 11 | 80 | 70.775 ±13.182 | 1.130 | 0.103 | 0.203 ±0.089 |
| 12 | 80 | 61.913 ±15.190 | 1.292 | 0.108 | 0.556 ±0.144 |
| 13 | 80 | 63.301 ±18.220 | 1.264 | 0.097 | 0.467 ±0.184 |
| 14 | 80 | 59.257 ±16.095 | 1.350 | 0.096 | 0.693 ±0.298 |
| 15 | 100 | 66.790 ±16.409 | 1.497 | 0.100 | 0.640 ±0.171 |
| 16 | 100 | 65.281 ±23.575 | 1.539 | 0.096 | 0.752 ±0.176 |
| 17 | 100 | 59.755 ±19.824 | 1.674 | 0.098 | 0.810 ±0.251 |
| 18 | 100 | 55.807 ±16.399 | 1.792 | 0.100 | 0.892 ±0.221 |
| 19 | 100 | 53.430 ±18.756 | 1.872 | 0.099 | 0.932 ±0.252 |
| 20 | 120 | 59.393 ±32.570 | 2.020 | 0.101 | 1.251 ±0.305 |
| 21 | 120 | 57.233 ±24.049 | 2.097 | 0.100 | 1.576 ±0.416 |
| 22 | 120 | 55.355 ±24.908 | 2.168 | 0.099 | 1.563 ±0.319 |
| 23 | 120 | 46.528 ±27.883 | 2.579 | 0.112 | 1.722 ±0.382 |
| 24 | 150 | 64.652 ±30.533 | 2.320 | 0.097 | 1.079 ±0.390 |
| 25 | 150 | 56.703 ±15.366 | 2.645 | 0.106 | 0.939 ±0.142 |
| 30 | 180 | 59.756 ±22.918 | 3.012 | 0.100 | 0.881 ±0.250 |
| 35 | 180 | 50.928 ±19.155 | 3.534 | 0.101 | 1.120 ±0.264 |
| 40 | 180 | 51.326 ±25.673 | 3.507 | 0.088 | 1.607 ±0.404 |

**Tabela 4.3: Resultados de simulação em função do número de clientes.**

para a velocidade de processamento por "record", velocidade de processamento de cada cliente, "speedup" e tempo de comunicação. Tais gráficos podem ser vistos nas Figuras 4.2, 4.3, 4.4 e 4.5 respectivamente. Observe-se que os valores de "speedup" foram calculados tomando por base a velocidade de processamento do sistema com apenas um cliente. Uma análise qualitativa desses resultados será feita ao final desse capítulo, quando poderão ser comparados com os resultados obtidos por "benchmarking".

Entretanto, cabe notar aqui que o maior problema desses resultados é o grande desvio padrão observado na Tabela 4.3, o que indica a necessidade de um número maior de simulações até atingir-se valores confiáveis. O baixo número de simulações aqui apresentado se justifica pelo objetivo ser apenas a verificação da funcionalidade da metodologia e não o uso prático da ferramentaOutra informação relevante é a indicação pela curva de "speedup" que o sistema perde muito em eficiência ao serem incorporados mais que trinta e cinco nós

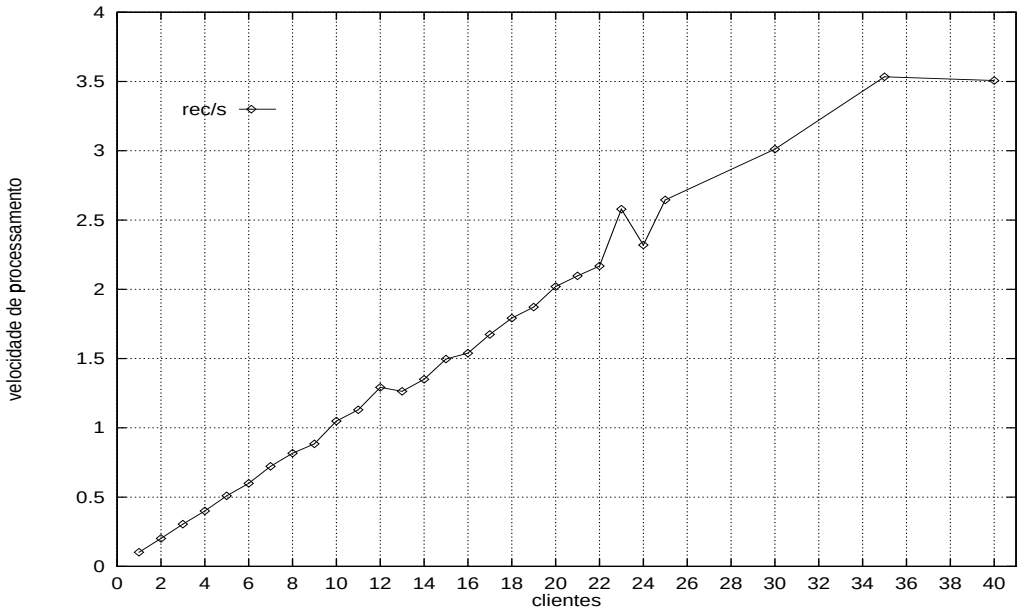



**Figura 4.2: Velocidade de processamento simulado de um "record" no sistema paralelo.**

clientes. Esse resultado pode ser explicado pela Lei de Amdahl se for lembrado que a partir de um certo ponto o tempo gasto com a leitura de novos dados e o seu envio aos clientes supera o tempo gasto para seu processamento efetivo. Isso permite um melhor dimensionamento do sistema paralelo a ser utilizado para a execução do programa, principalmente se for também considerado o custo do "hardware" e do tempo até a conclusão de todo o processamento necessário.

### 4.2.3 Variação por tamanho do grão

Com o simulador também é possível fazer, sem esforços adicionais, testes sobre o desempenho do programa quando se varia o tamanho do grão de dados em uso. Para o programa aqui estudado isso significa variar o tamanho do "record", de forma a fazê-lo conter menos ou mais eventos. Como cada cliente recebe um "record" por vez e o analisa até o final, variar o seu tamanho implica em variar o tamanho do grão de dados.

Para verificar como esse programa tem seu desempenho afetado pelo tamanho do grão, foram realizados alguns testes, apenas com o objetivo de ilustrar essa capacidade do

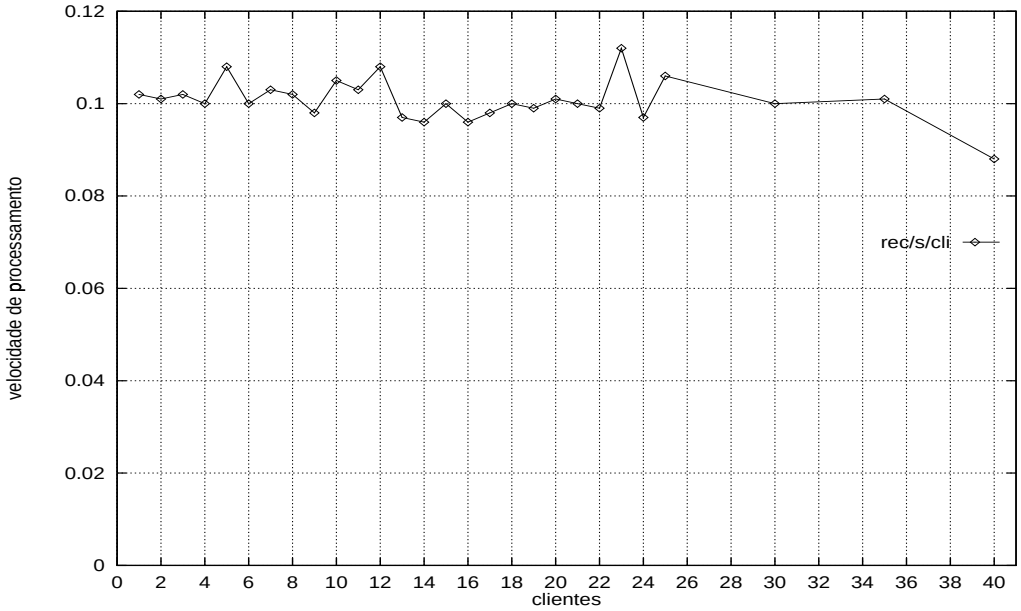



**Figura 4.3: Velocidade de processamento simulado de um "record" por cliente do sistema paralelo.**

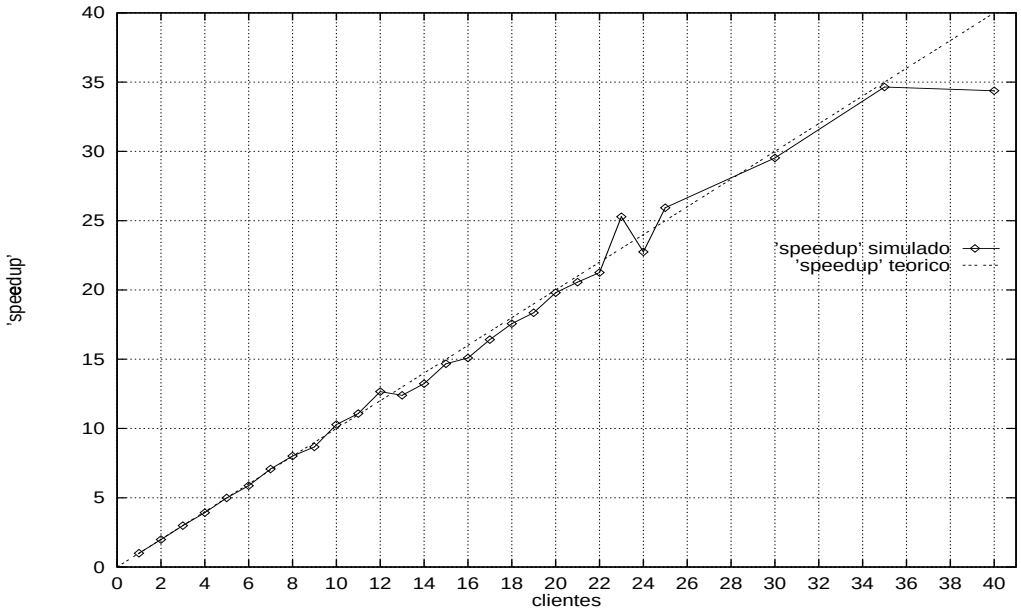



**Figura 4.4: Curva de "speedup" obtida através de simulação.**

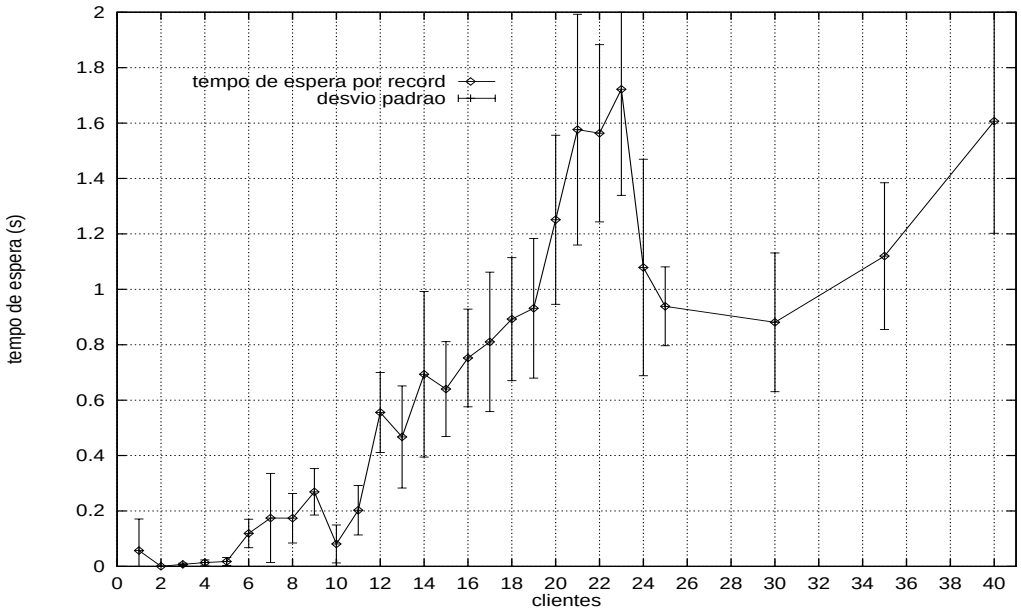



**Figura 4.5: Tempo de espera simulado por um "record" no sistema paralelo.**

simulador e do método de predição de desempenho apresentado. Para tanto, foram escolhidos outros tamanhos determinados de grão, como pacotes de 32 Kbytes e 128 Kbytes. Os resultados obtidos encontram-se na Tabela 4.4. Na Figura 4.6 mostra-se de forma gráfica a variação na velocidade de processamento de cada evento, diferentemente dos gráficos anteriores que apresentavam a velocidade de processamento de cada "record". Como aqui o tamanho dos "records" variou artificialmente, o número de eventos em cada um deles também teve que ser alterado. Os valores apresentados na tabela foram calculados considerando uma média de onze eventos por "record".

| Clientes | Eventos/segundo | | | Espera (segundos/evento) | | |
|---|---|---|---|---|---|---|
| | 32Kbytes | 64Kbytes | 128Kbytes | 32Kbytes | 64Kbytes | 128Kbytes |
| 5 | 1.000257 | 1.119712 | 1.018974 | 0.001616 | 0.001623 | 0.000924 |
| 10 | 1.120762 | 1.154164 | 1.145100 | 0.016461 | 0.007349 | 0.002989 |
| 20 | 1.052645 | 1.111231 | 1.158938 | 0.160508 | 0.113730 | 0.031831 |

**Tabela 4.4: Velocidade de processamento e tempo gasto com comunicação para diferentes tamanhos de grão.**

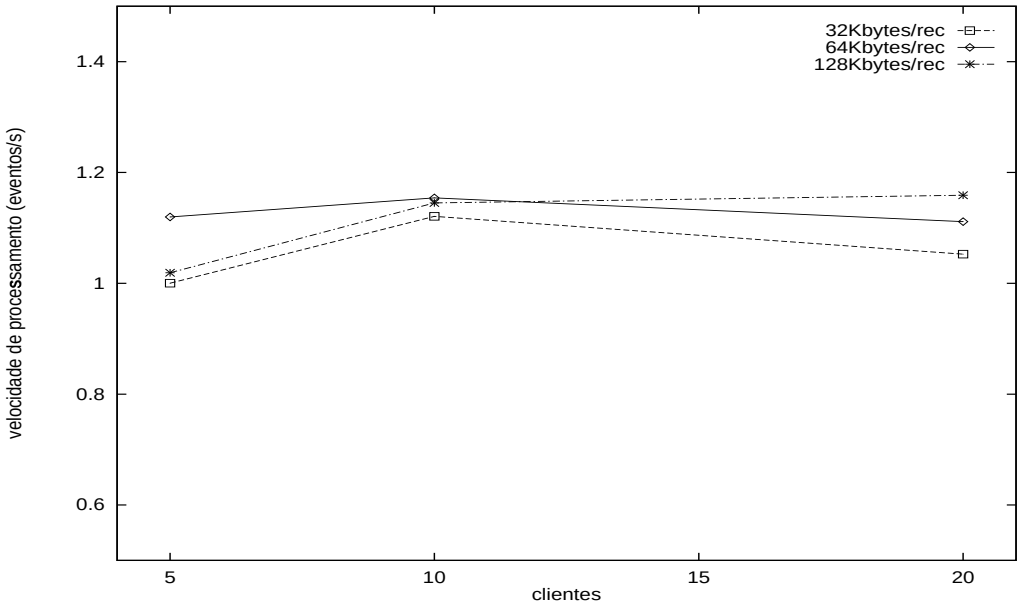



**Figura 4.6: Velocidade de processamento de um evento para diferentes tamanhos de grão.**

### 4.2.4 Variação do modelo estatístico

Um último teste realizado com o simulador procurou determinar a influência do modelo usado para geração de números aleatórios. Esse tipo de teste é útil para verificar se o modelo adotado nos testes anteriores é adequado para o programa em análise.

Assim, além daqueles testes serão realizados também testes com modelos que usam apenas distribuição normal e apenas distribuição exponencial. Em todos os casos será mantido constante o número de clientes (dez) e o tamanho do grão (64 Kbytes). A Tabela 4.5 apresenta os valores obtidos para os três casos aqui avaliados. Em todos os casos o número de "records" simulados foi mantido constante.

Os resultados aqui apresentados mostram que o uso de distribuições normais para a geração de números aleatórios em todos os tipos de decisão resulta em menores velocidades de processamento e razoáveis tempos de comunicação. Já o uso de distribuições exponenciais faz com que a velocidade de processamento seja maior e o tempo com comunicação menor. A combinação dessas duas distribuições, como a usada no restante dos testes produz resultados intermediários. Na seção 4.4 serão discutidas as conseqüências da adoção de uma dessas hipóteses.

| Modelo estatístico | Velocidade de processamento (rec/s) | Tempo de espera (segundos/evento) |
|---|---|---|
| normal + exponencial | 0.105 ±0.033 | 0.808 ±0.684 |
| apenas normal | 0.072 ±0.014 | 0.038 ±0.031 |
| apenas exponencial | 0.121 ±0.075 | 0.002 ±0.002 |

**Tabela 4.5: Velocidade de processamento e tempo gasto com comunicação para diferentes modelos estatísticos.**

### 4.2.5 Aperfeiçoando o modelo de comunicação

Uma análise rápida nos gráficos apresentados nas páginas anteriores permite constatar que o modelo usado para o programa forneceu resultados relativamente homogêneos para velocidade de processamento. Entretanto, os tempos gastos com comunicação observados parecem sofrer uma variação elevada. Examinando-se mais cuidadosamente o modelo para o programa pode-se constatar a razão dessa variação, desde que se conheça o funcionamento real do programa executando na máquina aqui simulada. O que ocorre na realidade é que cada processo cliente apenas envia um "record" para o processo escritor quando esse "record" estiver completamente preenchido, ou seja, contiver 64Kbytes de informação. Isso ocorre, em média, a cada cinco ou seis "records" recebidos pelo cliente.

O modelo para comunicação usado nos testes anteriores previa uma freqüência mais alta para esse evento, simplesmente por usar um vértice de decisão com duas arestas emergentes, que para fdp normal (usada até aqui) escolhe entre os dois caminhos possíveis com igual probabilidade. Dessa forma, para corrigir esse erro de modelagem é necessário que o usuário do protótipo defina probabilidades de ocorrência distintas para cada aresta do vértice em questão, o que pode ser feito de forma bastante simples pois o grafo de cada programa é um arquivo texto de descrição dos vértices.

Os gráficos apresentados a seguir (Figuras 4.7, 4.8) mostram os resultados obtidos com a alteração desse vértice. A primeira mostra os resultados relativos à velocidade de processamento, enquanto a segunda apresenta os tempos gastos com comunicação para as simulações realizadas. Os gráficos mostram apenas os resultados para até vinte clientes. Muito embora não exista impedimento técnico para a realização de testes com uma quantidade maior de clientes, usou-se este limite nesse último teste pois como os testes feitos através de "benchmarking" tinham esse limitante físico não haveria como comparar os resultados para quantidades maiores de clientes.

No gráfico da Figura 4.8 é possível perceber que os tempos gastos com comunicação

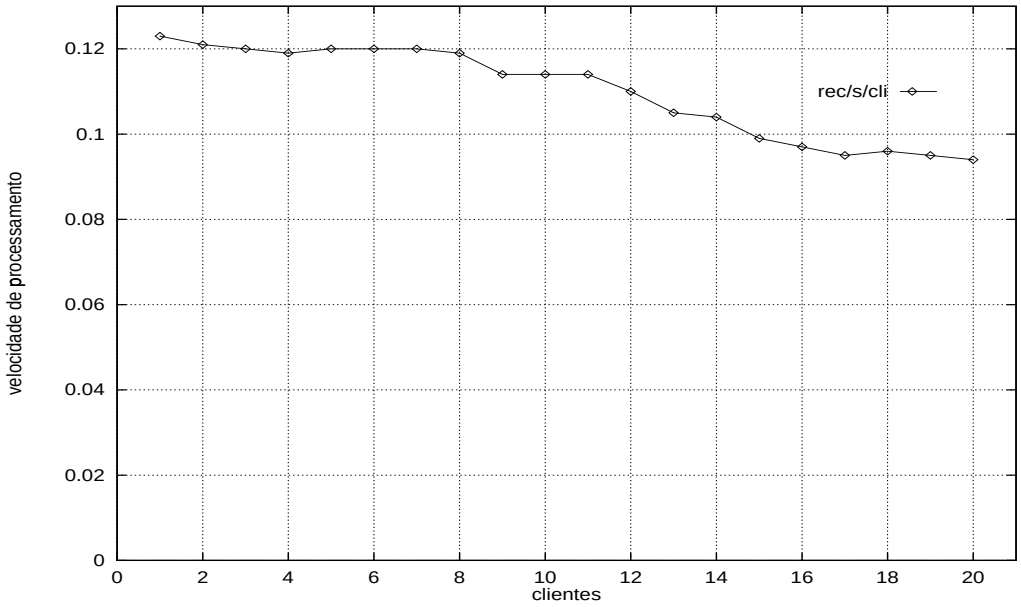



**Figura 4.7: Velocidade de processamento simulado de um "record" por cliente do sistema paralelo modificado.**

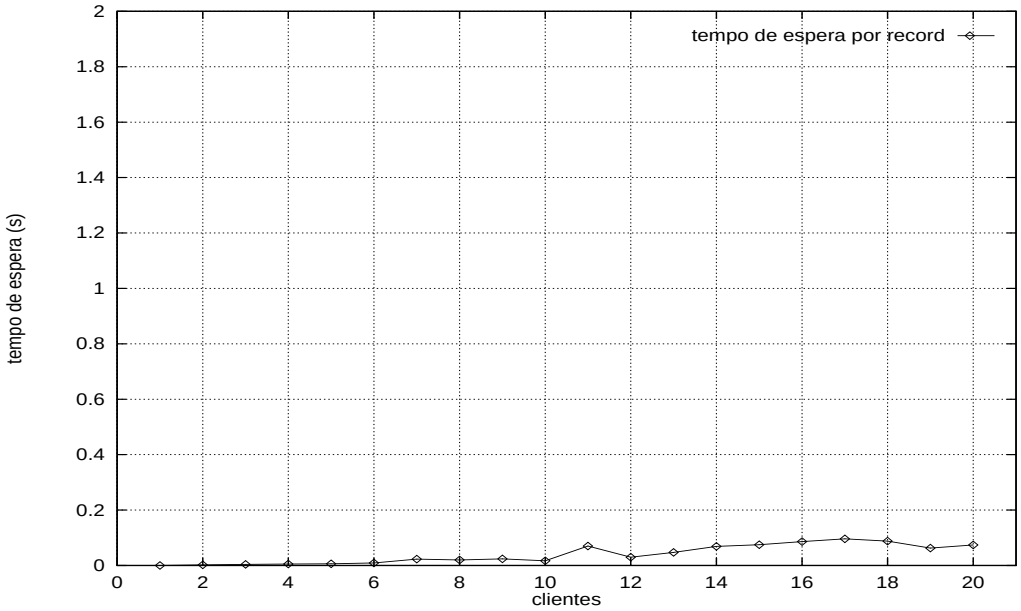



**Figura 4.8: Tempo de espera simulado por um "record" no sistema paralelo modificado.**

são sistematicamente menores e com uma variação mais suave para diferentes números de clientes, o que é um resultado mais fácil de se aceitar do que aquele apresentado anteriormente. Já a curva apresentada na Figura 4.7 também é mais suave, porém apresenta uma tendência de dimunuição da velocidade de processamento mais acentuada do que a observada na Figura 4.3.

## 4.3 Resultados obtidos por "benchmarking"

Os resultados apresentados nas seções anteriores apenas mostram que o protótipo é tecnicamente viável, mas não são suficientes para mostrar a precisão do mesmo. Para tanto foram realizados testes de "benchmarking" em uma das "farms" do Fermilab. Os dados sobre essa "farm" estão na Tabela 4.1 já apresentada. Além daqueles dados é necessário indicar quais os procedimentos usados durante as medições e quais as informações obtidas através das mesmas, o que é feito na Tabela 4.6 a seguir.

As condições de teste listadas na tabela acima foram estabelecidas para que os tempos de execução em cada rodada fossem relativamente constantes e para que se tivesse uma quantidade de informações por nó cliente também uniforme. O número de repetições para cada cliente foi mantido baixo para evitar um excesso de carga não produtiva sobre a "farm" utilizada. Mesmo assim, o período de testes se estendeu por quase um mês. Além disso, não foi possível realizar testes usando mais do que vinte nós por limitação física da "farm", isso é, esse era o número máximo de clientes disponíveis e como cada processo cliente ocupa cerca de 95% da carga de CPU disponível, seria improdutivo colocar-se mais do que um cliente por nó.

| Parâmetro | O que foi adotado |
|---|---|
| Grau de paralelismo | de 1 a 20 clientes (limitação da "farm") |
| Número de repetições | de 6 a 10 p/ cada grau de paralelismo |
| Fonte de dados | fita 8mm padrão, com diferentes quantidades de dados |
| Quantidade de dados | de 1999 a 15899 "records", adequando-se o tempo de execução de forma a uniformizar os tempos de processamento |
| Medidas realizadas | tempo total de processamento,<br>tempo de processamento por evento em cada cliente e<br>tempo gasto em comunicação por cliente |

**Tabela 4.6: Critérios usados nos "benchmark".**

Deve-se salientar que não foram realizadas medições para verificar o desempe-

nho do programa em relação às variações de granulação pois tais mudanças implicariam em alterações significativas no código usado. Além de representar repetidas compilações do programa para cada caso, isso também representava um aumento considerável no tempo que o sistema ficaria improdutivo, o que foi considerado absolutamente inviável para o grupo que fazia uso das "farms" naquele momento.

### 4.3.1 Variação por número de nós

A Tabela 4.7 apresenta as principais informações coletadas durante o processo de "benchmarking". Nela o tempo médio de processamento é igual ao tempo de relógio, em segundos, decorrido entre o início e o final da reconstrução de eventos. A partir dele são calculados quantos "records" são processados por segundo na "farm" e também em cada um de seus nós. O tempo médio de espera se refere simplesmente ao tempo que cada cliente consumiu em comunicação.

| Clientes | records | tempo de processamento (s) | veloc. média proc. (rec/s) | veloc. média por cliente (rec/s) | tempo médio espera por cli (s/rec) |
|---|---|---|---|---|---|
| 1 | 1999 | 19643.750 ±13.881 | 0.102 | 0.102 | 0.016 ±0.000 |
| 2 | 1999 | 9823.500 ±5.708 | 0.203 | 0.102 | 0.016 ±0.000 |
| 3 | 1999 | 6553.167 ±3.891 | 0.305 | 0.102 | 0.017 ±0.000 |
| 4 | 2999 | 7364.000 ±3.266 | 0.407 | 0.102 | 0.016 ±0.000 |
| 5 | 3999 | 7874.500 ±3.775 | 0.508 | 0.102 | 0.017 ±0.000 |
| 6 | 4999 | 8203.833 ±16.304 | 0.609 | 0.102 | 0.017 ±0.000 |
| 7 | 5999 | 8459.800 ±6.145 | 0.709 | 0.101 | 0.017 ±0.000 |
| 8 | 6499 | 8024.750 ±8.197 | 0.810 | 0.101 | 0.017 ±0.000 |
| 9 | 7499 | 8254.750 ±4.206 | 0.908 | 0.101 | 0.017 ±0.000 |
| 10 | 8999 | 8893.000 ±6.481 | 1.012 | 0.101 | 0.017 ±0.000 |
| 11 | 9899 | 8884.750 ±5.673 | 1.114 | 0.101 | 0.017 ±0.000 |
| 12 | 10799 | 8900.000 ±9.028 | 1.213 | 0.101 | 0.017 ±0.000 |
| 13 | 11699 | 8905.250 ±3.491 | 1.314 | 0.101 | 0.017 ±0.000 |
| 14 | 12599 | 8937.750 ±11.755 | 1.410 | 0.101 | 0.017 ±0.000 |
| 15 | 13499 | 8949.750 ±9.549 | 1.508 | 0.101 | 0.017 ±0.000 |
| 16 | 13999 | 8704.250 ±4.493 | 1.608 | 0.101 | 0.017 ±0.000 |
| 17 | 13999 | 8218.750 ±18.753 | 1.703 | 0.100 | 0.017 ±0.001 |
| 18 | 14999 | 8407.750 ±21.741 | 1.784 | 0.099 | 0.018 ±0.001 |
| 19 | 14999 | 8043.750 ±69.933 | 1.865 | 0.098 | 0.018 ±0.001 |
| 20 | 15899 | 8184.500 ±100.741 | 1.943 | 0.097 | 0.018 ±0.002 |

**Tabela 4.7: Resultados obtidos com "benchmarks".**

Com os dados da Tabela 4.7 podem ser gerados gráficos mostrando os tempos de processamento e espera com relação ao número de máquinas clientes. O gráfico da Figura 4.9 apresenta a velocidade média de processamento de um record para diferentes números de

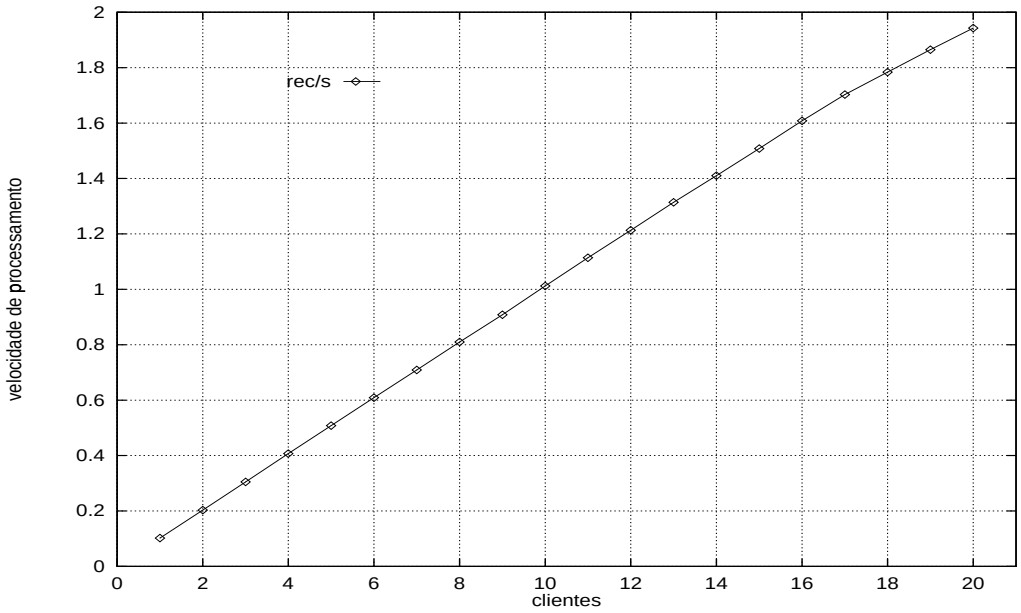



**Figura 4.9: Velocidade de processamento de um "record" no sistema paralelo.**

clientes e acrescenta barras de indicação do desvio padrão entre as várias medidas realizadas para cada valor. Já a Figura 4.10 apresenta a curva com a velocidade de processamento de um record quando feito isoladamente por um cliente.

Já a Figura 4.11 apresenta a curva do "speedup" obtido com o sistema paralelo. Deve-se notar, entretanto, que esse "speedup" é relativo ao tempo de processamento do programa quando executado com apenas uma máquina cliente, além da máquina em que são executados os programas leitor e escritor. Tal resultado pode ser aproximado para o "speedup" real em relação ao programa executando em modo seqüencial pois existe apenas um pequeno custo adicional de comunicação entre cliente e os demais programas, conforme é indicado em [40].

Um rápido exame nessas figuras mostra que existe uma pequena tendência em se perder desempenho após 18 nós paralelos. Embora isso seja facilmente constatado pela Figura 4.11, é impossível afirmar o número de máquinas em que ocorreria o melhor desempenho do sistema, isto é, o ponto em que o tempo de execução passaria a crescer apesar do aumento no grau de paralelismo, ou então o número de máquinas em que a relação custo/desempenho fosse ótima.

Finalmente, a Figura 4.12 apresenta o gráfico com os tempos médios de espera por

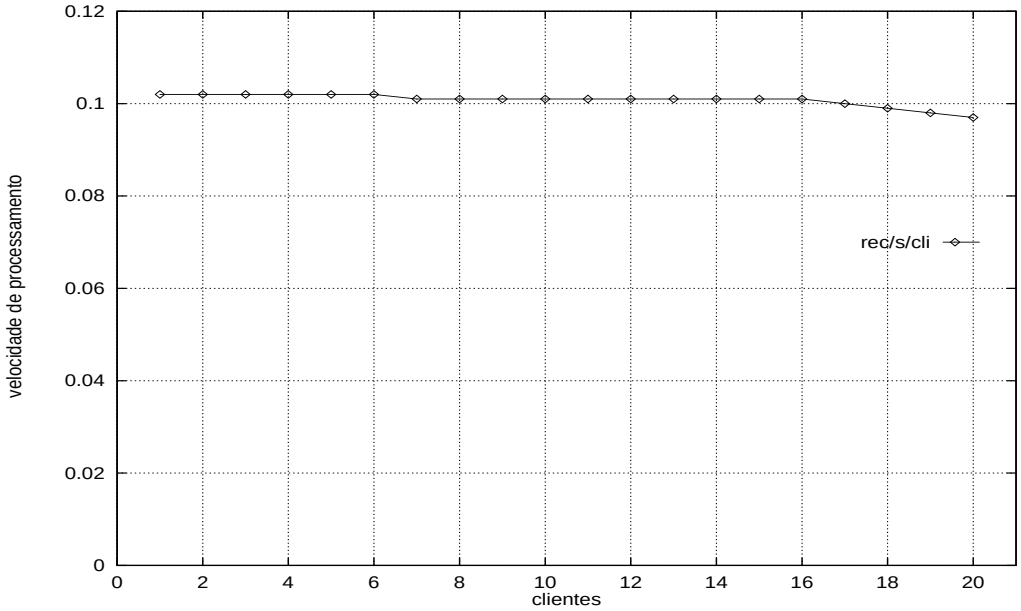



**Figura 4.10: Velocidade de processamento de um "record" por cliente do sistema paralelo.**

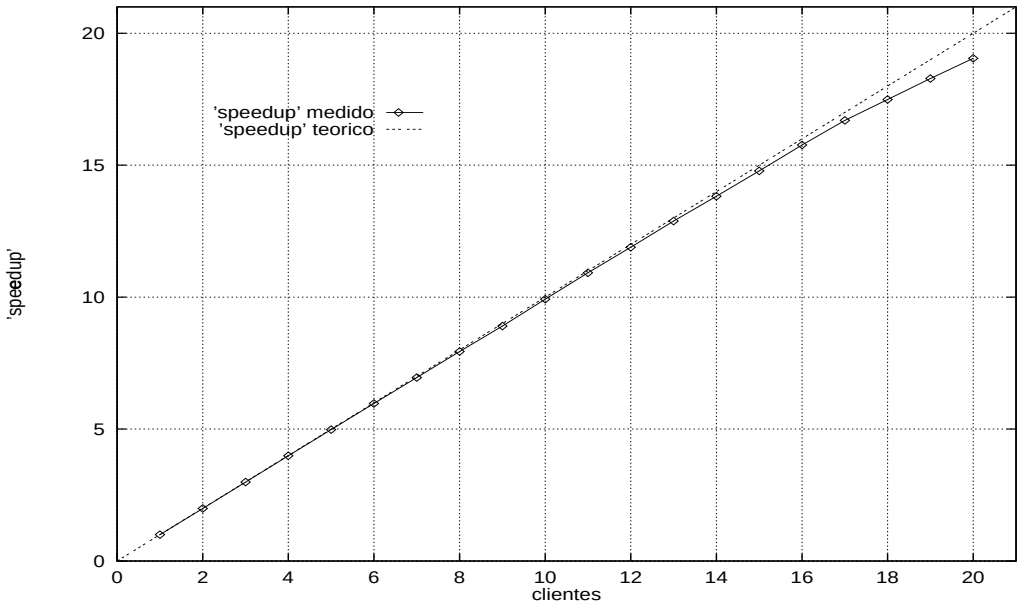



**Figura 4.11: "Speedup" obtido nos "benchmarks" realizados.**

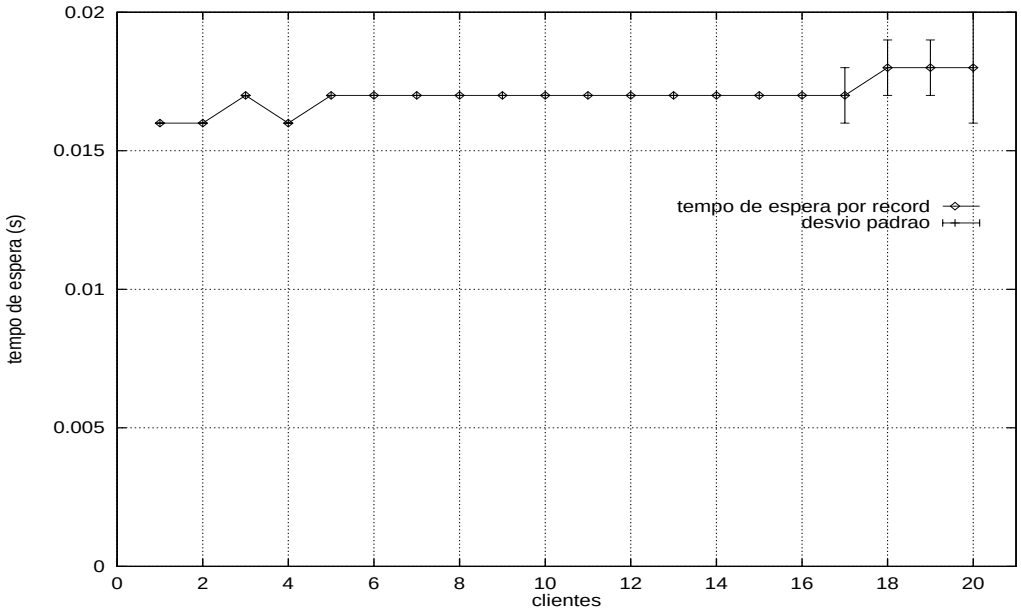



**Figura 4.12: Tempo de espera por um "record" nos "benchmarks" realizados.**

um "record" para cada grau de paralelismo experimentado. Desse gráfico constata-se que o tempo de espera por comunicação aumenta com o número de clientes, o que é muito natural pois o único processo servidor de dados passa a trabalhar para mais clientes. Vale observar que o aumento no tempo de espera é maior a partir de dezoito clientes, comprovando de certo modo o resultado indicado pela diminuição da eficiência calculada através do "speedup".

## 4.4 Análise dos resultados obtidos

Nesta seção será feita uma análise sobre os resultados obtidos com o protótipo para o método de predição de desempenho apresentado neste trabalho. Para uma melhor clareza essa análise será dividida em duas partes. A primeira faz um exame detalhado sobre o desempenho funcional do protótipo, tal como sua velocidade de execução e os testes sobre granulação e modelo de geração de números aleatórios. Já a segunda parte procura analisar a eficácia do protótipo comparando seus resultados com os obtidos através do "benchmarking" apresentado em 4.3.

### 4.4.1 Análise de desempenho do simulador

Inicialmente serão feitos estudos e comentários sobre a velocidade de execução do protótipo e seu comportamento para variações no sistema em análise. Em seguida serão apresentados os resultados para variação no tamanho do grão, do modelo para geração de números aleatórios e também as medidas obtidas pelo simulador que não existiam nos "benchmarks".

#### Velocidade

O capítulo anterior deixou em aberto a questão do desempenho do simulador em termos de velocidade na obtenção dos resultados. Após a execução dos testes é possível fazer um exame sobre a velocidade de execução do simulador.

Antes de examinar a velocidade de execução do simulador é necessário indicar a velocidade de execução dos demais módulos que compõem o protótipo aqui apresentado. Assim, a Tabela 4.8 apresenta os tempos consumidos com a geração e otimização dos grafos usados no caso de teste. Os tempos indicados não diferenciam os tempos de geração e otimização, mas testes preliminares realizados durante o desenvolvimento do protótipo indicam que 30% desse tempo foi gasto com a otimização dos grafos. A máquina em que os testes foram executados é uma Sun sparc-5, com 128 Mbytes de memória.

| Grafo | Tempo (minutos) | vértices/minuto |
|---|---|---|
| Leitor | 15 | 126 |
| Escritor | 18 | 206 |
| Cliente | 105 | 708 |

**Tabela 4.8: Tempo consumido na geração/otimização dos grafos.**

A velocidade maior de números de vértices gerados por minuto para grafos maiores se explica através da quantidade de interações necessárias com o usuário em cada caso, que é aproximadamente constante. Como essas interações tomam mais tempo que o processamento não interativo, elas acabam representando uma maior parte do tempo consumido para os grafos menores.

Examinado-se agora a velocidade do simulador, tem-se que indicar que os tempos de execução variaram de modo bastante irregular ao longo dos testes. Assim, na maioria dos casos em que a simulação obteve sucesso ela durou menos de dois minutos, porém em outros casos, de mesmo tamanho, o simulador consumiu três horas de processamento. Essa variação é a maior desvantagem encontrada para esse método de predição de desempenho, uma vez que é difícil justificar ao usuário de uma ferramenta desse tipo que não se pode saber *a priori*

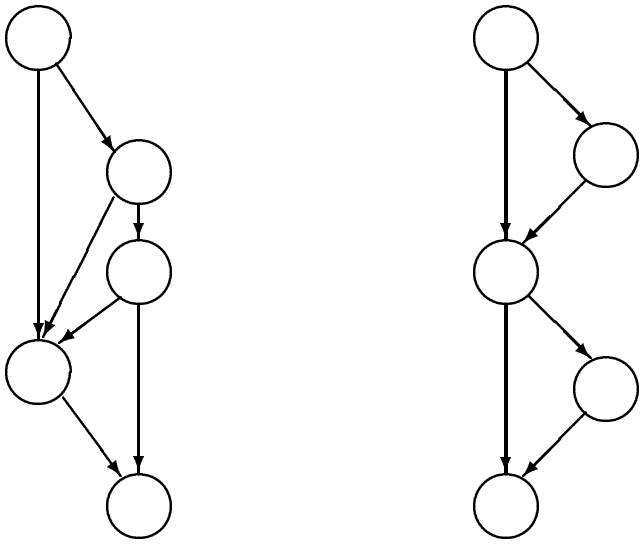


**Figura 4.13: Estruturas de programação problemáticas durante a simulação.**

o tempo consumido para a simulação que ele pretende realizar.

A explicação para tal variação está calcada em duas hipóteses:

- O modelo estatístico usado para decisões dos caminhos a serem seguidos nos grafos pode alterar profundamente o tempo consumido pela simulação caso decida por um caminho com muito mais vértices.
- O protótipo do gerador do grafo de execução é ineficiente em algumas situações, por exemplo cadeias longas da estrutura, como indicado na Figura 4.13.

Essas hipóteses devem ser analisadas em conjunto para verificar se de fato as variações no tempo de simulação podem ocorrer. Isso porque apenas é possível justificar variações entre caminhos escolhidos quando um dos caminhos tiver a estrutura indicada na Figura 4.13. Desse modo é preciso verificar se tais cadeias existem, o que é verdadeiro pelo menos no caso testado para códigos gerados a partir de fontes em Fortran. Em tais códigos o compilador utilizado gera código de tal forma que para cada comando simples de atribuição, do tipo **"var1 = var2 + var3 * var4"** gera uma cadeia dessas estruturas de tamanho equivalente ao número de variáveis e operadores aritméticos/lógicos existentes no comando.

Desse modo, a associação das hipóteses aqui indicadas justifica a grande variação nos tempos de simulação e indica a necessidade de um refinamento do módulo de geração do grafo de execução, o qual precisa conseguir identificar estruturas do tipo cadeia e reduzir o grafo do comando para apenas um vértice, e aí então o tempo de execução seria equivalente

ao tempo gasto para seguir o caminho de execução normal do programa.

Não ocorrendo as hipóteses descritas, o que é perfeitamente possível, pode-se concluir que a velocidade do simulador é adequada para o objetivo de fazer a predição de desempenho de programas paralelos em máquinas seqüenciais. Claro que a velocidade do mesmo pode ser melhorada com o devido refinamento do gerador do grafo de execução, mas para o protótipo aqui desenvolvido a velocidade de simulação não foi problemática. Claro também que a execução em máquinas reais possibilitaria respostas mais rápidas, entretanto com um custo adicional que pode ser exagerado quando ainda se está nas fases iniciais de desenvolvimento do programa paralelo.

**Análise dos testes de granulação**

Um exame sobre os resultados apresentados na Tabela 4.4 mostra que o número de eventos processados por segundo varia consideravelmente para os três casos analisados. Como apenas o caso com pacote de dados (ou tamanho do grão) de 64 Kbytes foi testado em condições reais de execução é necessário cuidado ao examinar-se as curvas obtidas.

Com essa observação em mente é possível indicar que existe uma forte tendência em obter-se melhores resultados com pacotes de maior tamanho para configurações com um maior número de clientes. Isso reflete basicamente as leis de Amdahl e Gustafson por indicarem que o aumento de paralelismo deve ser acompanhado por um aumento da carga a ser processada. Como os tempos gastos com comunicação não aumentam de forma substancial para pacotes com 128 Kbytes tem-se que a carga computacional oferecida por tais pacotes consegue manter as CPUs ativas por mais tempo, conseguindo com isso um desempenho melhor do que o obtido com pacotes de 64 Kbytes.

O desempenho inferior obtido com pacotes de 32 Kbytes se deve basicamente ao espaço ocupado, em bytes, por cada evento. Como a unidade real de processamento é um evento, pacotes menores contém menos eventos e, conseqüentemente, fazem com que os processadores necessitem de novos dados com maior freqüência. Isso aumenta muito o tráfego na rede e piora o desempenho do sistema, como é indicado claramente pelos resultados da seção 4.2.3.

**Análise dos testes sobre os modelos estatísticos**

Os testes realizados com três modelos de geração de números aleatórios produziram resultados que merecem destaque, tanto pelo comportamento do simulador durante a execução de cada um deles quanto pelo padrão de resultados obtidos.

Quanto ao funcionamento do simulador foi observado uma alta taxa de falhas de execução, isto é, o simulador era interrompido pelo sistema com uma freqüência relativamente alta, para as simulações em que o modelo adotava apenas um tipo de função geradora. Nos dois casos examinados o simulador obteve sucesso em apenas 25% das tentativas, o que indica algum problema grave no tratamento de memória, que deve ser resolvido no aperfeiçoamento do protótipo utilizado. A taxa de sucesso para a combinação entre as funções normal e exponencial foi de 75%, merecendo também um maior cuidado nas próximas versões do sistema.

Apesar dos problemas de confiabilidade do protótipo foi possível obter os resultados já apresentados na Tabela 4.5. A partir daqueles resultados e dos obtidos com os "benchmarks" podem-se fazer os seguintes comentários:

- O modelo que usa apenas função de distribuição de probabilidades (fdp) exponencial apresenta um resultado mais coerente para os tempos gastos em comunicação.
- O modelo que usa apenas fdp normal pode ser considerado como um limite inferior para o desempenho do sistema.
- O modelo que combina as duas fdps apresentou o melhor resultado para a velocidade do sistema.
- Apesar de não ter sido testada, o uso da fdp normal apenas para a decisão em vértices que não pertencessem a ciclos e fdp exponencial para testes em ciclos e para modelar atrasos em comunicação deve aparentemente produzir os melhores resultados em ambos os parâmetros (velocidade de execução e tempo gasto com comunicação).

**Flexibilidade de configuração**

Nos testes realizados foram obtidas também medidas sobre os tempos consumidos em cada ponto de sincronismo e diferenciou-se o tempo gasto com o envio de informações daquele gasto com o recebimento das mesmas. Como não é possível confrontar esses resultados com os obtidos através dos "benchmarks", aqui será apresentado o resultado de uma única simulação, apenas para ilustrar o tipo de informação que também estaria disponível ao usuário do protótipo aqui desenvolvido. A Figura 4.14 apresenta a listagem de resultados para a simulação do caso de teste configurado com três clientes.

Os valores que aparecem na figura se referem ao tempo, em segundos, consumido em cada um dos pontos medidos. Vale observar que a numeração de processos segue a ordem de criação deles no simulador, assim, para o caso apresentado o processo 1 é o leitor do sistema, o processo 5 é o escritor e os demais representam os três clientes. Após a identificação do

| | | |
|---|---|---|
| Process 1<br>waiting 66.941564<br>send 66.941564<br>receive 0.000000<br>executing 0.009331<br>waiting sync 1 0.000000<br>waiting sync 2 0.001606<br>waiting sync 3 0.002623<br>waiting sync 4 0.000000<br>waiting sync 5 0.001196<br>waiting sync 6 0.003002<br>waiting sync 7 0.000000<br>sync 0.008427 | Process 2<br>waiting 0.696897<br>send 0.000195<br>receive 0.696703<br>executing 78.189725<br>waiting sync 1 0.002003<br>waiting sync 2 0.003785<br>waiting sync 3 0.002527<br>waiting sync 5 0.021272<br>waiting sync 9 0.000000<br>waiting sync 10 0.001593<br>waiting sync 6 0.002008<br>sync 0.033188 | Process 3<br>waiting 0.399692<br>send 0.180467<br>receive 0.219226<br>executing 65.462266<br>waiting sync 1 0.002339<br>waiting sync 2 0.003783<br>waiting sync 3 0.000000<br>waiting sync 5 0.000000<br>waiting sync 9 0.002000<br>waiting sync 10 0.001591<br>waiting sync 6 0.000000<br>sync 0.009713 |
| Process 4<br>waiting 0.831811<br>send 0.569169<br>receive 0.262642<br>executing 79.195092<br>waiting sync 1 0.002721<br>waiting sync 2 0.000000<br>waiting sync 3 0.002477<br>waiting sync 5 0.007359<br>waiting sync 9 0.001600<br>waiting sync 10 0.000000<br>waiting sync 6 0.002000<br>sync 0.016156 | Process 5<br>waiting 74.915619<br>send 0.158112<br>receive 74.757507<br>executing 0.007717<br>waiting sync 2 0.005363<br>waiting sync 3 0.003426<br>waiting sync 4 0.000813<br>waiting sync 5 0.026396<br>waiting sync 10 0.002155<br>waiting sync 6 0.001792<br>sync 0.039946 | |

**Figura 4.14: Dados adicionais obtidos através do simulador.**

processo aparecem os tempos gastos com espera (total, envio e recebimento), o tempo gasto em execução real e finalmente o tempo gasto com sincronismo (em cada ponto isoladamente e o total). Foram essas as informações usadas na seção 4.2, depois de acumuladas em várias repetições de simulação.

### 4.4.2 Análise comparativa

O último aspecto a ser analisado nos resultados obtidos com o método proposto diz respeito ao seu desempenho. Os resultados apresentados nas seções 4.2 e 4.3 podem ser comparados para verificar se os desempenhos previstos pelo simulador possuem ou não uma precisão adequada e também se a velocidade com que os resultados são produzidos é suportável para o usuário. Para tanto serão comparados os resultados obtidos com o simulador e aqueles obtidos por "benchmarking".

#### Velocidade de execução

A comparação entre o simulador e a execução real do programa no ambiente paralelo é bastante subjetiva. Primeiro por serem situações bastante distintas e depois por ser

necessário considerar também o custo envolvido em cada caso. Apesar disso, é necessário mostrar que a velocidade do simulador é suficiente para que seu uso seja possível.

A Tabela 4.7 apresenta informações sobre o tempo médio de execução dos testes realizados na "farm" disponível no Fermilab. Da mesma tabela é possível retirar informações sobre o tempo consumido por "record" examinado. Já os tempos de execução do simulador foram fornecidos em 4.4.1. Uma comparação direta entre os dois casos apenas é possível se for calculada a velocidade em termos de "records" por segundo, pois o número de "records" analisados em cada caso variou intensamente.

Desse modo, como as simulações consumiram de dois a cinco minutos em média, consumindo um tempo maior para casos com um maior número de clientes, tem-se que a velocidade média foi de **0,25 rec/s**. Observe-se que a velocidade é menor para um maior número de clientes pois o simulador passa a controlar um maior número de estruturas em seu "blackboard".

Já a execução real teve um tempo de execução mais elevado, mas como o número de "records" também foi maior, a sua velocidade acabou sendo maior. Como aqui o tempo de execução depende do grau de paralelismo em cada caso, as velocidades medidas variaram de **0,1 rec/s** para um cliente até **1,9 rec/s** para vinte clientes. Observe-se que aqui a velocidade aumenta proporcionalmente ao número de clientes.

Ocorre entretanto que esses resultados de velocidade não podem ser lidos de forma direta, isto é, olhando-se apenas para os tempos de execução. A forma correta de enxergar tais resultados é levando-se em conta também o custo envolvido com o simulador e com o "benchmarking" e também a capacidade de processamento individual de cada máquina. Assim, é preciso levar em consideração a velocidade e o custo de cada CPU.

O fator velocidade é desfavorável ao simulador, isto é, as simulações ocorreram em um máquina mais rápida do que as máquinas da "farm". Esse fator é compensado quando se examina o custo envolvido em cada configuração. O custo total da "farm" é superior ao da estação de trabalho usada nas simulações. Com isso, apesar do simulador ser proporcionalmente mais lento que a "farm", seu uso é justificado por possuir um custo muito menor e, conseqüentemente, uma relação custo/benefício melhor.

Além disso, um outro aspecto favorável ao uso do simulador é sua maior flexibilidade para testes de granulação, como também no grau máximo de paralelismo atingido, que é limitado fisicamente na "farm".

**Precisão dos resultados do simulador**

Tendo verificado que a velocidade de execução do simulador permite o seu uso no processo de predição de desempenho de um programa paralelo, é necessário verificar se os resultados por ele produzidos são suficientemente precisos para serem úteis ao usuário. Os resultados apresentados nas seções 4.2 e 4.3 podem ser usados para essa análise. Assim, a Figura 4.15 mostra as curvas para o número de "records" analisados por segundo por cliente segundo o simulador, tanto na versão original quanto na modificada, e segundo os "benchmarks" realizados. Na figura se observa que o simulador apresenta um comportamento bastante razoável, com um erro inferior a 3% em média, não passando de 10% para nenhuma quantidade de clientes. A maior discrepância está em não fornecer uma curva tão suave quanto à obtida com a execução na máquina real. Entretanto, como o objetivo do protótipo era verificar a viabilidade e precisão do método, tem-se que a forma da curva parece não ser muito importante uma vez que os dados oscilam em torno do resultado do "benchmark". Portanto, como os valores obtidos por simulação não diferem muito daqueles obtidos por "benchmarking" é possível afirmar que o método é viável e preciso, necessitando apenas de melhorias em sua implementação, tanto para corrigir problemas de suavidade da curva quanto os problemas de variação na velocidade de execução já indicados.

Uma outra comparação possível é quanto aos resultados obtidos para os tempos de comunicação. A Figura 4.16 mostra as curvas dos tempos gastos com comunicação nos dois casos ("becnhmarking" e modelo de comunicação original). Essa figura mostra um aspecto negativo do simulador devido à grande margem de erro nos tempos de comunicação. Essa margem de erro surge principalmente pelo modelo adotado para o canal de comunicação e também para o modo em que a comunicação ocorre. Nos "benchmarks" a comunicação ocorreu em modo assíncrono para o envio de dados entre leitor e clientes e síncrono entre clientes e escritor. Na simulação usou-se apenas o modo síncrono, o que já indica uma pequena elevação nos tempos gastos com comunicação.

Entretanto, o maior erro foi introduzido pelos modelos usados para o canal comunicação e para o modo de comunicação entre clientes e escritor. Para o primeiro caso, o simulador adotou uma postura conservadora quanto ao número de colisões para a transmissão de informações, fazendo com que os tempos gastos com a simples espera pelo canal de comunicação crescesse de modo significativo. Tal erro poderia ser corrigido no modelo, fazendo com que os tempos gastos com comunicação no simulador fossem mais próximos dos tempos reais. Como o objetivo dos testes era primeiramente mostrar a viabilidade do método de predição/análise de desempenho proposto, fez-se uma opção por não modificar o modelo

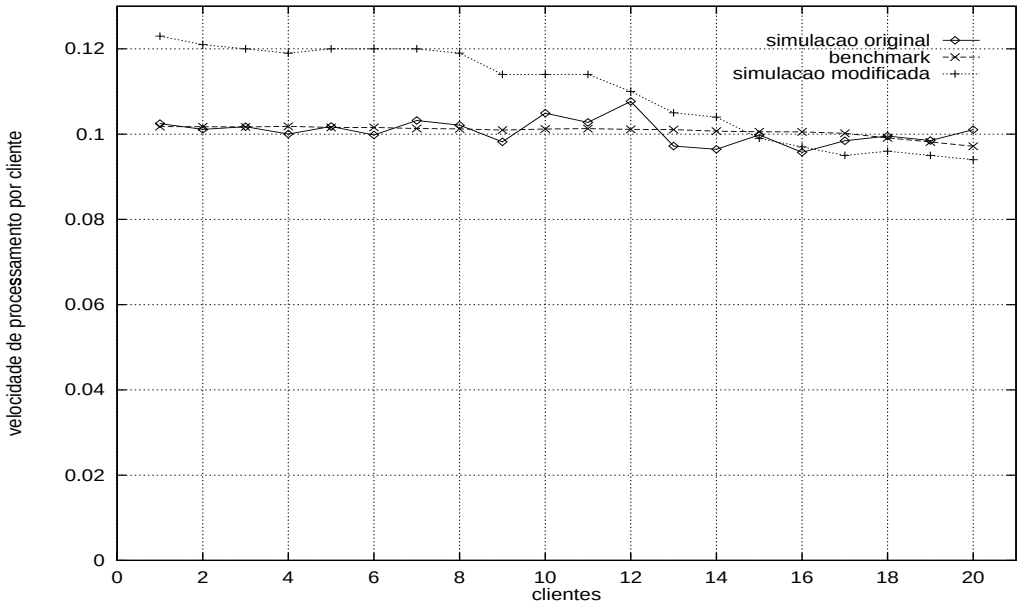



**Figura 4.15: Comparação entre os resultados da simulação (original e modificado) e dos "benchmarks" - velocidade de processamento.**

para o canal de comunicação, deixando-se tal tarefa para aperfeiçoamentos futuros. Aliás, a constatação de que o modelo usado para comunicação foi inadequado permite mostrar que a predição de desempenho pode ser imperfeita, exigindo sempre algum cuidado em seu uso, muito embora os resultados aqui obtidos tenham demonstrado um excelente desempenho para o protótipo nos demais aspectos.

Já para o modo de comunicação entre clientes e escritor foram realizados os testes já indicados ao final da seção 4.2. A Figura 4.17 mostra os tempos gastos com comunicação com o modelo de comunicação modificado, juntamente com os tempos observados por "benchmarking". Através dessa figura é possível perceber que com apenas um pequeno ajuste no grafo de execução do programa cliente, adequando o seu modelo ao que ocorre na prática, foi possível obter resultados bastante mais precisos.

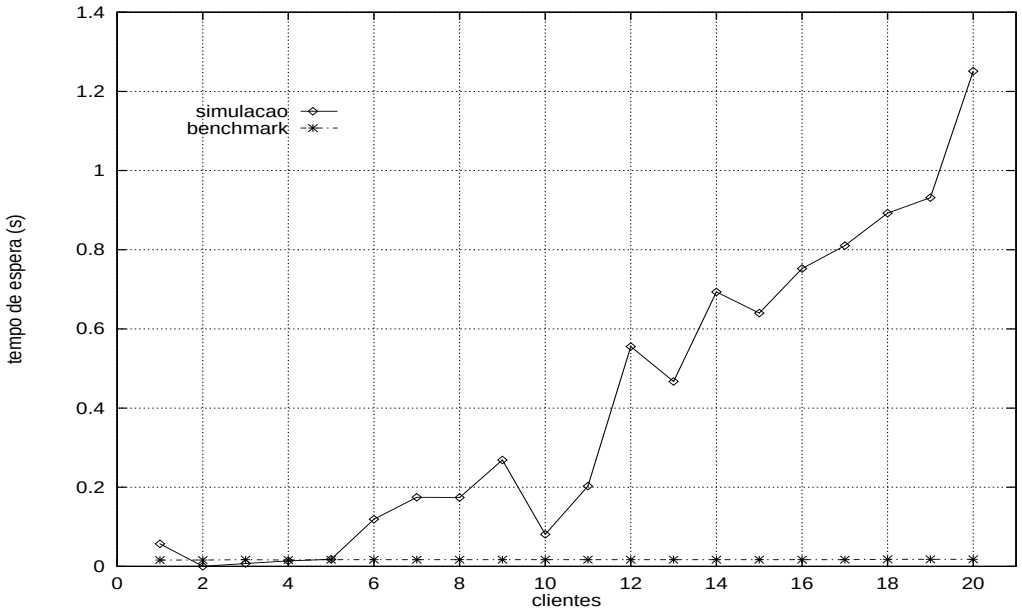



**Figura 4.16: Comparação entre os resultados da simulação e dos "benchmarks" - tempo de comunicação.**

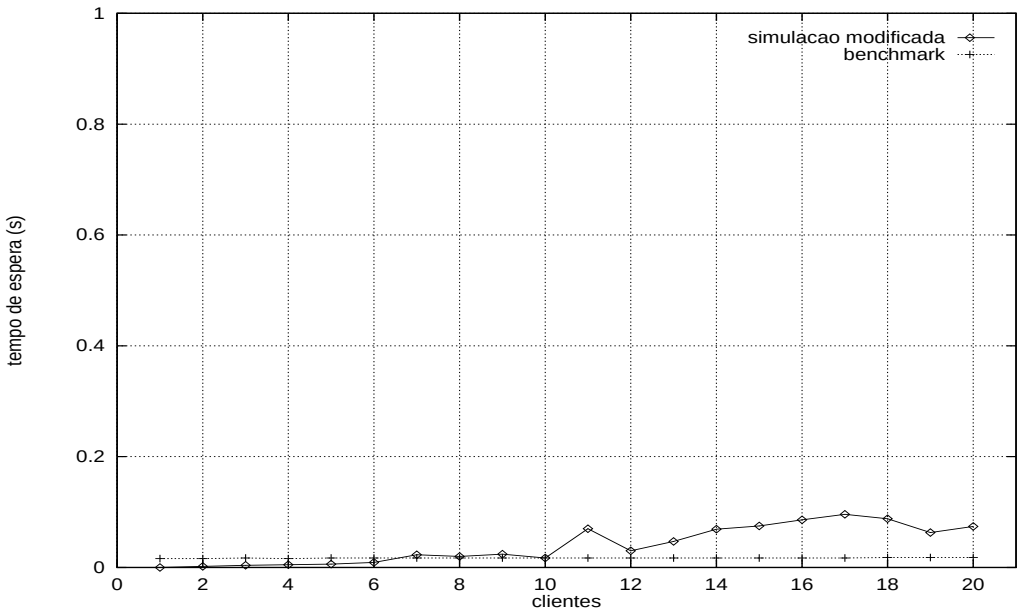



**Figura 4.17: Comparação entre os resultados da simulação (modelo modificado) e dos "benchmarks" - tempo de comunicação.**

# Capítulo 5

# Conclusões e perspectivas

Este último capítulo procura apresentar de forma sintética as conclusões que podem ser tiradas com uma leitura atenta dos demais capítulos dessa tese. Assim, primeiro serão relacionados os resultados obtidos no Capítulo Quatro que indiquem a viabilidade e as restrições do método aqui proposto. Após isso serão enunciados os pontos relevantes deste trabalho, e suas principais contribuições. Finalmente, serão indicadas direções para futuras pesquisas na área que possam usar as contribuições aqui fornecidas na melhoria de ferramentas e técnicas para análise de desempenho e, adicionalmente, ao desenvolvimento de técnicas para verificação de programas.

## 5.1 Viabilidade do método para predição de desempenho

Do que foi exposto ao longo da seção 4.4 podem ser tiradas as seguintes conclusões:

- A precisão do método é elevada quando se tem um número razoável de simulações, com taxa de erro média inferior a 3% para o caso estudado;
- O protótipo teve uma velocidade de execução excelente em média, embora precise refinamentos para que o tempo de execução de cada simulação seja mais uniforme;
- O método é bastante flexível, admitindo variações difíceis de serem implementadas em "benchmarks", tais como o teste sobre máquinas não disponíveis fisicamente ou variações de granulação sem a necessidade de alterações no programa;

Os resultados de precisão e velocidade do protótipo permitem concluir que o método para predição/análise de desempenho aqui proposto é viável pois conseguiu fornecer resultados precisos com um baixo custo de processamento e tempo. Além disso, sua flexibilidade para variação no tamanho de grão e do modelo estatístico para tomada de decisão em vértices com mais de uma aresta emergente faz com que, além de viável, a proposta tenha potencial de crescimento buscando-se o aperfeiçoamento do protótipo implementado e novas aplicações.

## 5.2 Relevância do método

Pelo exposto no Capítulo Dois e pelos resultados obtidos com o protótipo implementado tem-se que as principais contribuições desse método estão relacionadas com o método para a construção do modelo de representação do programa a ser analisado. O processo de predição de desempenho por simulação do grafo de execução do programa traz consigo diversas vantagens, várias delas enumeradas durante a descrição do método no Capítulo Três. Aqui serão resumidas tais vantagens e, a partir delas, apontadas as contribuições dessa metodologia.

As maiores vantagens da simulação do grafo de execução são:

- O modelo do programa é mais preciso que modelos analíticos;
- Seu custo operacional é inferior ao envolvido com "benchmarking";
- Permite a realização de medições para a obtenção de "profilings" ou traços de execução de modo não invasivo;

Como as vantagens listadas foram comprovadas através do protótipo implementado e do caso estudado, pode-se afirmar que a principal contribuição deste trabalho é a proposição de uma metodologia mais eficiente para a construção do modelo do programa e obtenção das medidas necessárias para a análise de desempenho.

Em relação a outros trabalhos na área este inova por não necessitar da máquina paralela para nenhum tipo de medição e por construir o modelo do programa diretamente a partir de seu executável. Quanto à máquina paralela, ele necessita apenas de informações que estejam disponíveis em catálogos e, com isso, permite que medição seja feita sem inserir código adicional no programa em análise. Quanto ao modelo do programa, ele é gerado já com informações detalhadas sobre otimizações introduzidas pelo compilador em tempo de execução.

Apesar de não ser uma solução definitiva para o problema tendo em vista o elevado grau de complexidade envolvido, é possível concluir que este trabalho contribui de modo significativo para o avanço de técnicas para a análise de desempenho de programas paralelos.

## 5.3 Perspectivas para trabalhos futuros

Durante a realização do trabalho aqui descrito foram observadas várias direções que merecem maiores investigações e/ou investimentos. O esforço consumido até o momento mostrou-se frutífero e pode levar a novos trabalhos na área. Deve-se salientar também que tais esforços podem se situar em duas frentes distintas: uma procurando aperfeiçoar o protótipo aqui desenvolvido e outra na busca por novas aplicações para a metodologia aqui utilizada. Segue-se uma lista de possibilidades para novos trabalhos com breves explicações do que pode ser feito em cada caso:

- Aperfeiçoamento do protótipo:
  - Melhoria do módulo de geração do grafo de execução, para que possam ser detectados problemas como os indicados em 4.4.1 para as cadeias longas de testes geradas por alguns compiladores;

    Já foi indicado que esse é um ponto crítico para a melhoria da velocidade de execução do simulador. Parte desse problema pode ser resolvido simplesmente através de otimizações mais grosseiras que aproximem tais cadeias por vértices tipo PASSAGEM. Porém, como tais cadeias são introduzidas pelo compilador, existe a possibilidade de corrigir o problema já na fase de interpretação das instruções caso se possa mapear as instruções de desvio condicional para instruções tipo "load/store" parametrizadas segundo uma taxa de acerto em memória "cache".
  - Desenvolvimento das bibliotecas de interpretação de instruções de máquina para diferentes processadores;

    Essa direção é óbvia após a validação do método e quando se tem como objetivo a construção de uma ferramenta para análise de desempenho mais versátil.
  - Implementação de mecanismos para obtenção de traços de eventos ou "profilings" no simulador;

    A Figura 3.15 indica a possibilidade de interação do usuário com o simulador através da definição das medidas de interesse. No protótipo apenas foram realizadas medições sobre tempo de execução, tempo gasto com comunicação (com

possibilidade de distinção entre envio e recebimento de informações) e tempo gasto com sincronismo. Uma medida não realizada que poderia ser facilmente obtida é o grafo de chamadas de funções, que poderia ser obtido usando vértices de chamada e retorno de funções como referência.

- Fazer a geração do grafo de execução a partir de alguma linguagem de especificação formal, tais como LOTOS ou SDL;

  Isso permitiria a predição de desempenho sem a existência do código executável, bastando que houvesse uma especificação suficientemente detalhada do programa. Essa perspectiva é muito interessante para a comparação entre algoritmos alternativos para a solução de um problema, a qual poderia ser feita sem a necessidade da implementação em linguagem de programação dessas alternativas.

- Novas aplicações:

  - Verificação de correção de programas;

    A simulação do grafo de execução do programa permite a localização de erros de fluxo no mesmo. Essa capacidade pode ser desenvolvida para que o método seja usado na área de testes de programas.

  - Ferramenta auxiliar ao trabalho de compiladores paralelos;

    Compiladores paralelos têm sua eficiência ligada ao problema de coloração de grafos cujos vértices são conjuntos de instruções do programa. A geração do grafo de execução pode ser modificada para que os vértices gerados passem a ser instruções não seqüenciais, que poderiam ser coloridos com mais facilidade.

  - Ferramenta auxiliar na avaliação de desempenho de máquinas ainda em desenvolvimento;

    Como o método aqui desenvolvido é preciso e necessita apenas de informações sobre a máquina e não de sua existência, é possível usá-lo para fazer análise de desempenho de uma máquina em desenvolvimento, desde que se conheça seu conjunto de instruções e que exista um compilador disponível para tal conjunto.

# Apêndice A

# Estrutura geral do CPS

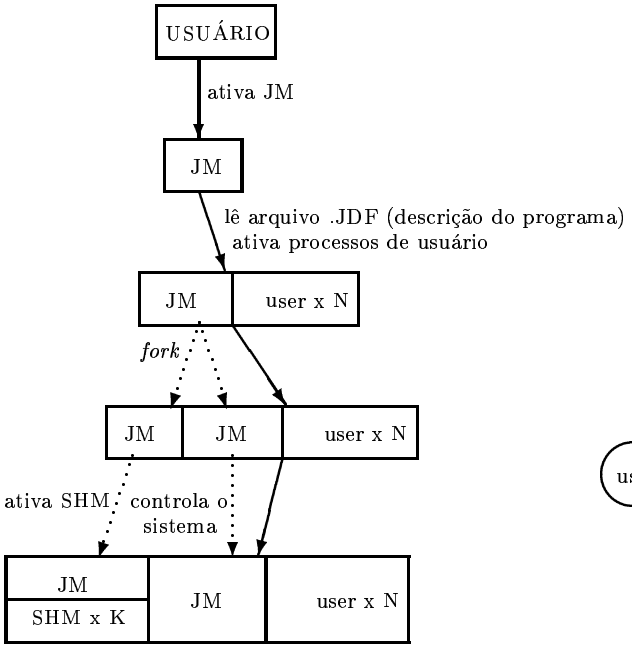
Neste apêndice descreve-se o **CPS** - **C**ooperative **P**rocesses **S**oftware [58, 21] - que é um pacote de ferramentas desenvolvido no Fermilab para a programação e execução de programas paralelos. Com seu uso é possível dividir um programa em processos que são executados de modo distribuído em uma rede de computadores.

Além de um conjunto de funções que podem ser chamadas pelo programa do usuário e implementam virtualmente o paralelismo no programa, existe um conjunto adicional de ferramentas que permite ao usuário do CPS o controle sobre a execução paralela de seu programa. Essas ferramentas são o "Job Manager" (JM) e o "Shared Memory Manager" (SHM), que devem interagir com o programa do usuário, mais precisamente com seus processos de usuário ("User Processes"). Em linhas gerais, é o JM quem controla toda a execução de um programa, enquanto o SHM é o responsável por toda comunicação entre o JM e os processos de usuário e entre processos de usuário residindo em máquinas distintas. A seguir descreve-se a forma como ocorre essa interação e algumas das facilidades oferecidas pelas funções do CPS [13].

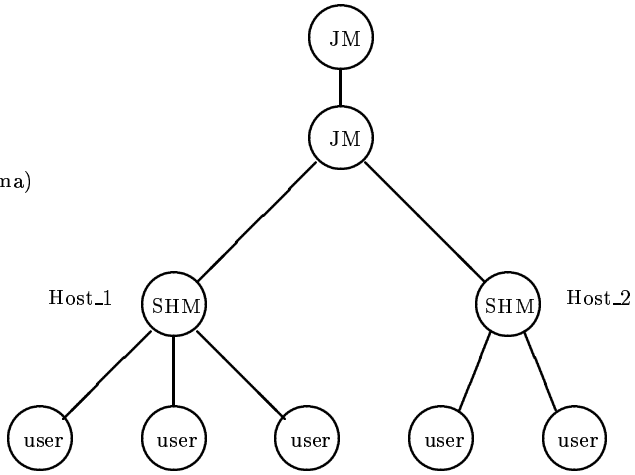
## A.1 Executando um programa com o CPS

A execução de um programa paralelo com o auxílio do CPS é simples. Na realidade, para o CPS, existem três tipos de processos comunicando-se entre si. Os processos são o SHM, os processos de usuário e o JM, que é quem inicia toda execução. Cabe ao usuário disparar o funcionamento do JM, o qual inicia cada um dos processos de usuário de acordo

[13] Uma excelente descrição do CPS e de sua aplicação para computação de alto desempenho no Fermilab foi feita por Rinaldo e Fausey em [51].




Seqüência de ativação de processos




Estrutura de comunicação entre processos

**Figura A.1: Ativação e comunicação de processos no CPS**

com informações a ele passadas e cria um SHM para cada processador em uso. A Figura A.1 apresenta essa relação seqüencial e a estrutura de comunicação entre processos no CPS.

Na figura faz-se uma referência ao arquivo de descrição do programa (.JDF), onde ficam armazenadas as informações sobre como o JM deve proceder na execução do programa. Essas informações se referem a quantos processos serão iniciados, em quais máquinas e com que códigos executáveis. A Figura A.2 apresenta um exemplo de um arquivo .jdf, no qual se destacam as seguintes linhas:

- Definição da classe (CLASS = 2), que define a classe do programa descrito nas linhas seguintes até a próxima ocorrência desse tipo de linha.
- Definição do programa (PROGRAM = REC_CLI.COM), que é quem define qual o programa a ser usado para os processos dessa classe.
- Especificação de processadores lógicos (LOGICAL CPU = 1,2,3,4,...,17), que permite a definição de quais processadores serão usados a partir do que estiver definido em outro arquivo de descrição do sistema (*.acp_sdf*), também apresentado na Figura A.2.

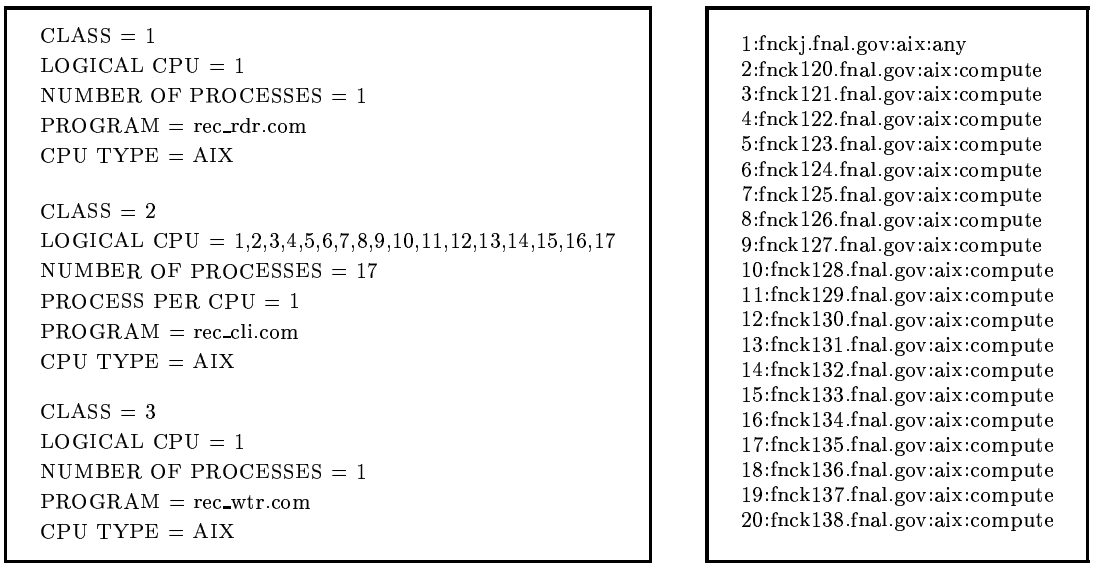



**Figura A.2: Exemplos de arquivos .JDF e .acp_sdf**

Esse arquivo (*.acp_sdf*) contém apenas uma lista numerada de nomes de máquinas, sistema operacional usado e função dentro do CPS. No exemplo da figura, estão listadas 20 máquinas, todas utilizando o ambiente AIX. O tipo de função de cada máquina pode ser *compute*, *I/O* ou *any*, sendo que máquinas do primeiro tipo apenas fazem processamento, as do segundo apenas entrada e saída de dados para o mundo externo (fitas por exemplo) e as do último tipo podem fazer tanto processamento quanto entrada e saída.

- Definição do número de processos (NUMBER OF PROCESSES = 17), que indica quantos processos serão executados para uma dada classe.
- Definição do número de processos por processador (PROCESSES PER CPU = 17), que diz quantos processos serão executados em cada processador definido para essa classe. O valor padrão é um processo por processador, que é adotado na ausência dessa linha.

Em resumo, para executar um programa que use o CPS é necessário que se criem dois arquivos, o de descrição do sistema e o de descrição da tarefa (ou "job"). O JM lê esses dois arquivos e inicia os processos necessários para que se execute o "job". A partir desse momento, o JM atua apenas para o atendimento de solicitações vindas dos processos ou de erros de execução. Todo o trabalho de dividir o programa em atividades que possam ser executadas em paralelo fica a cargo do programador, auxiliado pelo conjunto de funções de

sincronismo, comunicação e manejo de filas providas pela biblioteca do CPS.

## A.2 Programando com o CPS

A implementação de um problema usando o CPS é bastante simples, desde que o problema obedeça às seguintes restrições:

1. *Trabalhe com informações independentes*;

   O que significa permitir a cada processo operar sobre dados diferentes sem a necessidade de comunicação entre eles.

2. *Ofereça uma grande carga computacional por informação*;

   O que significa ter processos que gastem muito mais tempo processando cada informação dentro de um processador do que enviando ou recebendo essa informação.

O não atendimento à essas restrições não proíbe que o programa use o CPS, apenas dificulta sua programação e deteriora o seu desempenho. Além de atender às restrições é necessário que sejam identificadas as tarefas do programa que possam ser realizadas em paralelo. Em geral, exige-se que cada programa seja dividido em três classes, uma responsável por iniciar todos os processos e enviar dados para outras classes, outra responsável por fazer de fato o processamento das informações e, por fim, uma classe responsável por armazenar os resultados obtidos. Em cada classe podem existir vários processos, dependendo do grau de paralelismo existente na função da classe e na máquina em que o programa será executado. Para o teste apresentado no Capítulo Quatro existe um processo para a primeira e terceira classes, os quais executam sempre na máquina que disponha de unidades de fita 8mm. Já para a segunda classe são disparados tantos processos quantos forem as máquinas disponíveis para processamento, incluindo-se a máquina que abriga as outras duas classes.

Para cada classe existe um programa diferente, escrito em Fortran ou C, responsável pela execução das atividades reservadas para aquela classe. Para implementar cada programa é necessário que se estabeleça como as classes devem interagir. Dessa forma é possível determinar em que pontos do programa devem ser inseridas as chamadas para funções da biblioteca do CPS. Essas funções podem ser divididas em cinco grupos: sincronismo, troca de mensagens, manuseio de filas, serviços de chamada remota e utilidades. A seguir apresentam-se as principais funções dentro de cada grupo.

**A. Sincronismo**

- **cps_sync(lista, num)**, faz com que o processo que a chamou espere até que todos os processos contidos em *lista* chamem por essa função com o mesmo identificador de semáforo *num*, quando então são liberados.
- **cps_wait_queue(fila, estado)**, faz com que um processo espere até que *fila* esteja vazia ou cheia, dependendo do valor de *estado*.

**B. Troca de mensagens**

- **cps_declare_block**, define um bloco virtual de endereços onde serão feitas as transferências de dados para o processo que a chamar.
- **cps_get**, transfere as informações de um dado bloco para um vetor interno ao processo. Essa chamada é feita de modo síncrono, isso é, o processo fica parado esperando por sua finalização.
- **cps_send**, envia o conteúdo de um vetor para um bloco declarado em um ou mais processos remotos. Essa chamada é feita de modo síncrono, isso é, o processo fica parado esperando por sua finalização.
- **cps_start_send**, faz o envio de uma mensagem para o processo remoto retirado de uma fila de espera por comunicação. Essa chamada é assíncrona, ou seja, o processo continua a execução enquanto a transferência estiver sendo providenciada.
- **cps_wait_for_data**, coloca o processo em espera até que chegue informação em seu bloco declarado com *cps_declare_block*.
- **cps_start_gets**, recebe informações de modo assíncrono a partir de um processo remoto retirado de uma fila de espera.

**C. Manuseio de filas**

- **cps_declare_queue**, define que o processo que a chamou pode ser colocado na fila indicada em sua chamada.
- **cps_dequeue_process**, retira da fila indicada o processo que nela estivesse há mais tempo.
- **cps_queue_process**, coloca um ou mais processos na fila indicada em sua chamada.

**D. Serviços de chamada remota**

- **cps_declare_subroutine**, faz a declaração de uma função a ser chamada remotamente numa estrutura cliente-servidor. É o servidor quem deve ativar essa função para que os clientes possam fazer uso dela.
- **cps_call**, faz a requisição pelo serviço de uma função remota no processo especificado em sua chamada.

**E. Utilidades**

- **cps_init**, realiza toda a preparação para que o programa execute e informa ao JM que o processo está pronto. Essa função deve ser chamada antes de qualquer outra função do CPS.
- **cps_sleep**, faz com que o processo "durma" pela quantidade de segundos passada em sua chamada.
- **cps_sleep_process**, faz com que o processo "durma" até o final da execução do programa.
- **cps_stop_process**, termina a execução do processo que a chamar.
- **cps_stop_job**, termina a execução do programa como um todo. Pelo menos um dos processos deve fazer essa chamada ao final do programa, mas é necessário ter cuidado com processos que possam ainda estar executando, os quais serão encerrados com o risco de perder-se parte do seu processamento.

## A.3 CPS *versus* protótipo implementado

A estrutura de programas baseados no CPS foi aproveitada em dois pontos da implementação do protótipo usado nos testes desta proposta. Primeiro, foi necessário que o simulador fosse capaz de manipular grafos de classes diferentes que trabalhassem de modo cooperativo. Essa foi uma das razões para a introdução de vértices especiais para comunicação e sincronismo. Com eles ficou mais fácil fazer o trabalho de geração de eventos nos grafos dos processos.

Além disso, para facilitar a implementação do gerador do grafo de execução foram aproveitadas as chamadas para funções do CPS como delimitadores da execução do programa, em especial delimitadores de pontos de sincronismo e comunicação.

Apesar dessas restrições impostas ao protótipo a metodologia proposta não ficou descaracterizada, uma vez que todo o processo de geração do grafo de execução pode ocorrer sem o uso de funções como delimitadores e o simulador ganhou em funcionalidade, ao permitir também a cooperação entre processos diferentes e não simplesmente a execução de múltiplas instâncias de um mesmo processo.

# Apêndice B

# Física de altas energias e a E791

Neste apêndice serão apresentados os conceitos essenciais para a compreensão da magnitude dos programas examinados no Capítulo Quatro. Com isso, será feita uma breve descrição do que são experimentos com física de altas energias, passando-se então à descrição do experimento em que se usou tais programas.

## B.1 Física de altas energias

O estudo de partículas atômicas tem sido feito através de duas maneiras distintas: raios cósmicos e aceleradores de partículas. No primeiro caso os experimentos consistem em coletar grandes quantidades de partículas que incidem sobre a Terra, geralmente em laboratórios localizados em grandes altitudes para diminuir a interferência da atmosfera terrestre. No segundo caso, o que se faz é provocar a colisão de partículas de grande porte (prótons por exemplo) dentro de aceleradores e a partir disso coletar informações sobre as partículas resultantes do choque.

No Fermilab trabalha-se com a segunda forma de deteção de partículas usando-se um acelerador circular com diâmetro aproximado de dois quilometros, no qual podem ser feitos experimentos colidindo-se feixes de prótons e anti-prótons ou lançando-se feixes de partículas em alvos fixos.

A aceleração dos feixes de partículas é feita através de dois anéis de magnetos que alteram os campos elétrico e magnético sobre o feixe e fazem com que esse seja acelerado até velocidades próximas à velocidade da luz - reportam-se velocidades de até 99.98% da velocidade da luz. A Figura B.1 mostra um trecho do túnel onde está o acelerador, destacando-

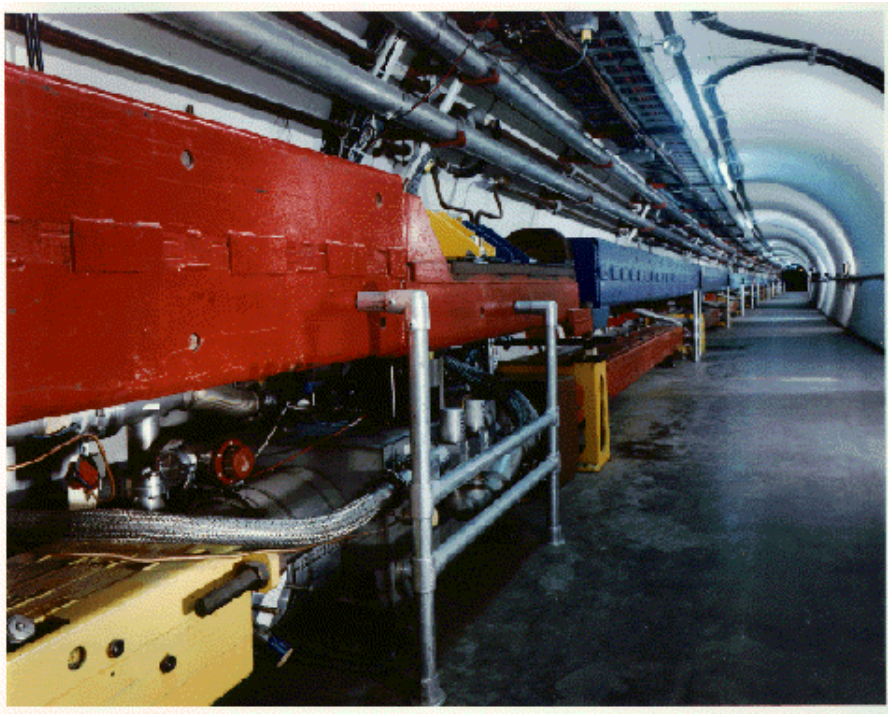


**Figura B.1: Túnel do acelerador de partículas do Fermilab**

se na parte esquerda os magnetos responsáveis pela aceleração das partículas. Já a Figura B.2 apresenta um esquema do acelerador, onde se destacam os detetores internos ao mesmo (*D0* e *Collider Detector*), o ponto de cruzamento de feixes nos anéis (próximo ao D0), a saída para experimentos de alvo fixo e os pré-aceleradores de prótons e anti-prótons.

Os experimentos que utilizam dados coletados nos aceleradores procuram determinar a natureza da matéria, a interação existente entre partículas subatomicas e, em especial, comprovar experimentalmente previsões teóricas como a da existência dos "quarks", o último dos quais - o "top" - foi detetado no Fermilab em 1993 e teve sua descoberta recentemente anunciada. Para atingir esses objetivos são levantadas grandes quantidades de dados para que se tenha medidas estatisticamente confiáveis, como aponta Appel [10], ao listar dados sobre dez experimentos de produção do *quark* "charm" realizados em detetores de alvo fixo. Aliás, o maior desses experimentos é a fonte dos programas analisados neste trabalho.

A aquisição de dados é feita através de sensores espalhados em espectrômetros construídos para esse fim. A Figura B.3 apresenta o espectrômetro usado no experimento E791 do Fermilab. Esse espectrômetro mede cerca de 15 metros e é composto por diversas camâras, onde são feitas medições de sinais elétricos das partículas resultantes do choque de

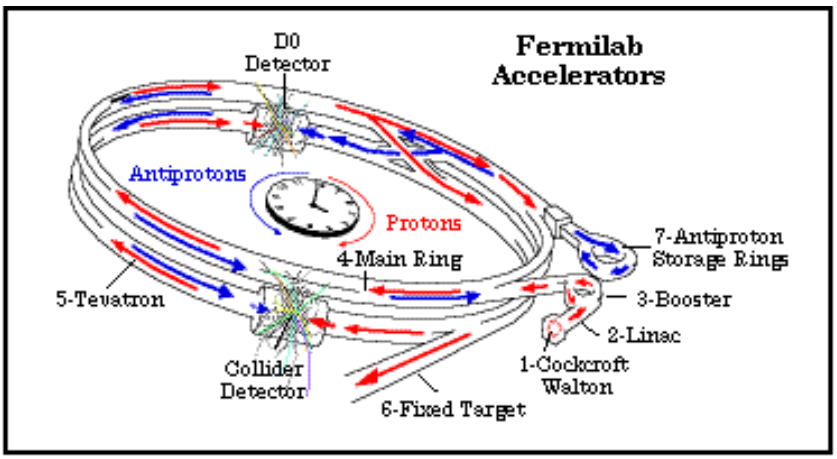



**Figura B.2: Diagrama esquemático do acelerador de partículas**

um feixe de pions contra um alvo de folhas de platina e diamante.

Um espectrômetro é composto por alvo, camâras de direção, contadores de Cerenkov, magnetos, calorímetros e planos de microfitas de silício (SMD's). Não cabe aqui examinar qual a função de cada um desses componentes na aquisição de dados, uma vez que isso reflete a física existente na determinação de trajetórias de partículas. Basta aqui saber que em todos existem diversos planos de sensores conectados a dispositivos especiais de aquisição de dados. Appel em [10] lista também a quantidade e o tipo de planos usados em alguns experimentos.

## B.2 O experimento E791

Como mencionado, o experimento E791 é o maior dos experimentos para a produção de *charms* realizados até hoje. Antes de examinar alguns dos detalhes técnicos do mesmo é interessante fazer um resumo de seu histórico. Daqui em diante esse experimento será chamado apenas de "E791" como um substantivo feminino.

A E791 é resultado de uma colaboração envolvendo mais de 70 pesquisadores de 14 instituições dos Estados Unidos, Brasil, México, e Israel [18, 55, 59]. Seu objetivo é a coleta da maior amostra possível de *charm* para a busca por decaimentos raros ou proibidos, o estudo de decaimentos ainda não observados e busca por estados de partículas com propriedades do *charm*. Ele é o quarto de uma série de experimentos usando o Tagged Photon Laboratory (TPL) do Fermilab, sendo o primeiro a coletar mais de um bilhão de eventos em seu espectrômetro.

A fase de coleta dos dados durou cerca de seis meses, terminando em janeiro de

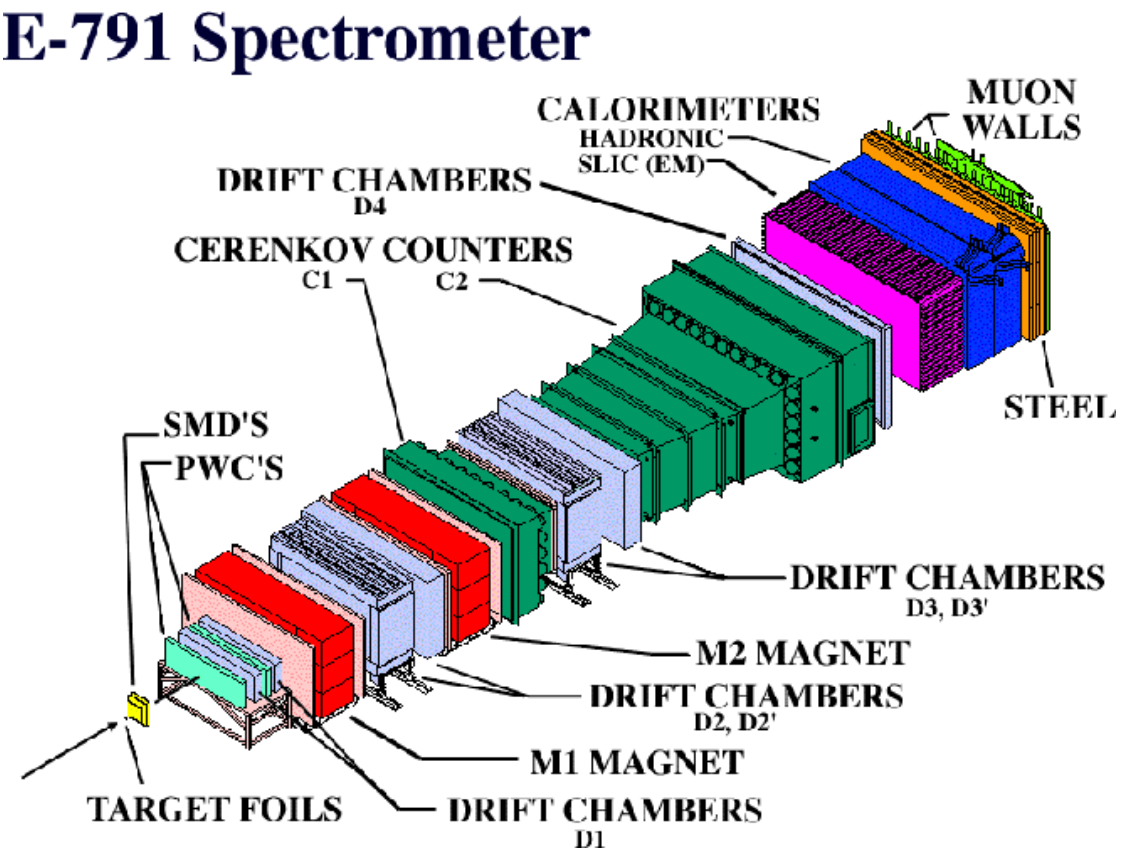



**Figura B.3: Diagrama esquemático do espectrômetro usado na experiência E791**

1992. Ao final desse período tinham sido armazenados cerca de 20 bilhões de eventos em 24.000 fitas de vídeo 8mm (2.3Gbytes cada), o que totalizou quase 50 Tbytes de informação. Com esse volume de dados espera-se que sejam reconstruídos 100.000 eventos contendo *charm*, o que significa mais de vinte vezes o melhor resultado obtido em experimentos anteriores. Atualmente os dados dessas fitas estão sendo analisados, após terem sido reconstruídos e filtrados entre 1992 e 1994.

Foi possível coletar essa quantidade de informações com a construção de um dispositivo especial de aquisição de dados ([8]). O sistema de aquisição de dados era composto por seis unidades de processamento compostas por 16 processadores 68020 cada (Figura B.4). Em cada uma dessas unidades eram conectadas sete unidades de fita 8mm. Dessa forma, a aquisição dos dados era feita de forma paralela em 42 fitas por sessão do experimento.

A aquisição em paralelo é possível pela característica inerentemente paralela dos eventos, isso é, um evento não depende do anterior nem influencia o seguinte. Assim, cada

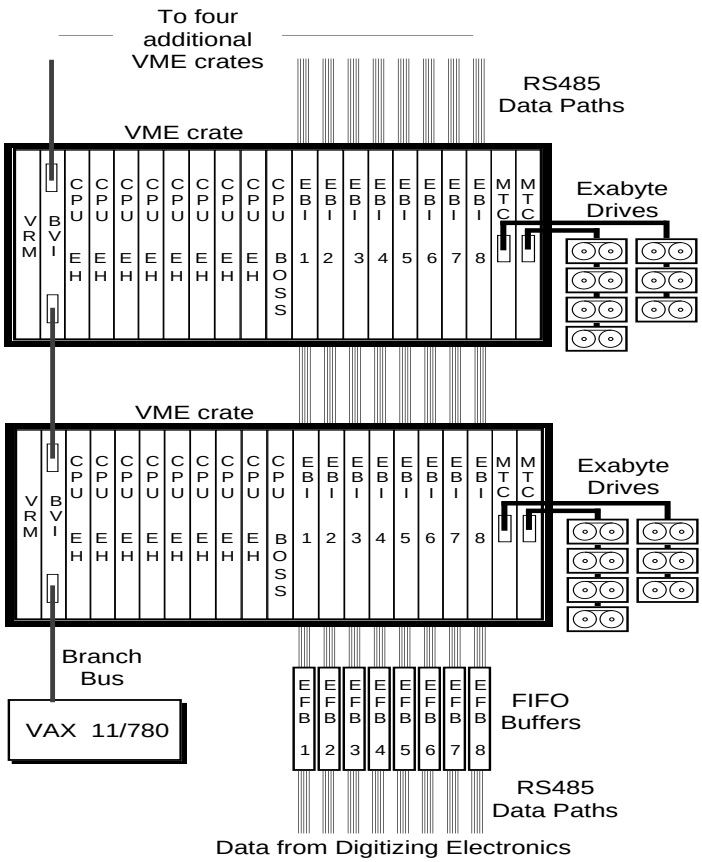



**Figura B.4: Unidades do sistema de aquisição de dados**

evento pode ser dirigido por software para uma fita diferente, permitindo que mais eventos possam ser coletados em um mesmo intervalo de tempo.

Após o período de coleta de dados iniciou-se a preparação dos dados para análise. Essa preparação envolve duas etapas distintas: a reconstrução e filtragem dos eventos a partir dos dados digitalizados nas fitas e a seleção e classificação dos eventos reconstruídos. A primeira das etapas é caracterizada por uma elevada carga computacional, mas a segunda apresenta uma demanda muito maior por leitura e escrita de dados em fitas 8mm, como indicado em [40].

Para a segunda fase foi desenvolvido um programa que lê eventos já reconstruídos e os seleciona de acordo com parâmetros definidos segundo constantes físicas esperadas para cada tipo de decaimento. Esse processo é bastante simples, consistindo em uma seqüência de testes de decisão para cada tipo de partícula e a escrita dos eventos selecionados em duas fitas de saída, segundo o tipo de evento. Essa etapa foi processada em várias estações de trabalho, cada uma equipada com um conjunto de três unidades de fita, sendo uma dessas unidades

munida de um braço mecânico para a alimentação automática de até dez fitas de entrada. Isso possibilitou que a cada execução do programa fossem lidas de oito a dez fitas durante 20 a 25 horas.

Já o processo de reconstrução de eventos demanda muito mais tempo de processamento. Nas estações de trabalho usadas na época essa demanda representava cerca de uma semana de processamento por fita de dados digitalizados. Como foram gravadas cerca de 24.000 fitas, a única maneira de processar todos os dados em tempo hábil é com a utilização de máquinas paralelas ou distribuídas, como as "computer farms" utilizadas no Fermilab e nos outros três centros de processamento da E791. Com isso, o trabalho de reconstrução iniciado em novembro de 1992 pode ser concluído em agosto de 1994. O programa testado neste trabalho é o utilizado nessa fase.

Durante 1993, com o começo da segunda fase de seleção de eventos, iniciou-se também a análise física dos dados. Nessa fase cada programa é construído com um objetivo específico. Sendo assim, todo o trabalho de análise tem sido realizado em estações de trabalho individuais e mantidas pelos interessados em determinado tipo de evento.

Em paralelo a isso, para que os programas de análise sejam considerados corretos é necessário confrontá-los com resultados gerados por simuladores. Esses simuladores usam o método de Monte-Carlo para gerar vários milhões de eventos segundo as características dos detetores usados no experimento. Esses eventos são então analisados pelo programa em teste e comparam-se os resultados obtidos com os que foram gerados pelo simulador. Para a geração dos eventos por simulação são usadas as *farms* disponíveis para isso uma vez que a quantidade de eventos gerada é grande e o tempo de processamento para cada geração de evento também é grande.

## B.3 Reconstrução de eventos da E791

O sistema computacional para a reconstrução e filtragem de eventos usado na E791 é composto por três programas estruturados de acordo com a metodologia do CPS, ou seja, um programa para a leitura dos dados de entrada e passagem dos mesmos para um segundo programa que os manipula e passa os resultados para um terceiro programa, que grava os resultados em fita. Os programas de leitura e escrita ocupam aproximadamente 1 Mbyte cada, enquanto que o programa de processamento - cliente - ocupa mais de 4 Mbytes.

Cada um desses programas é composto por diversas rotinas em Fortran e algumas rotinas de manipulação de fitas em C. Para obter uma boa estruturação do programa e permitir

| | Programa | | |
|---|---|---|---|
| Etapa | reader | client | writer |
| Iniciação global | Inicia CPS, arquivos de histórico | Inicia constantes físicas | Inicia arquivos de histórico |
| Preparação de fitas de entrada | Monta fita de entrada e informa dados sobre ela | Coloca-se na fila e espera por dados | Monta fita de saída ou escreve novo cabeçalho |
| Processamento | Lê fita e envia dados para clientes | Reconstrução e filtragem de eventos | Recebe resultados e os escreve na fita de saída |
| Finalização de fita | Finaliza fita de entrada e prepara nova fita | Conclui eventos em análise | Encerra um arquivo na fita de saída |
| Finalização global | Completa estatísticas e finaliza a sessão | Espera pelo fim da sessão | Fecha fita de saída |

**Tabela B.1: Atividades em cada etapa dos programas**

uma reusabilidade do código paralelo, apenas uma função em cada programa é responsável pelo trabalho de controle do fluxo de dados. Essa função é dividida em cinco fases: iniciação global, preparação de fita de entrada, processamento específico, finalização de fita de entrada e finalização global. A Tabela B.1 apresenta o que cada programa executa em cada fase dessa função para o sistema de reconstrução de eventos.

Com a organização indicada pela Tabela B.1 fica fácil escrever novos programas paralelos, devendo ser alteradas apenas rotinas de leitura e escrita de dados e o programa que faz de fato o processamento. Esse último pode ser apenas uma rotina chamada durante a terceira fase do programa cliente.

O processamento das fitas de dados digitalizados foi realizado em cinco locais: Fermilab, University of Mississipi (UMiss), Centro Brasileiro de Pesquisas Físicas (CBPF), Ohio State University (OSU) e Kansas State University (KSU). Na realidade, a produção em KSU substituiu OSU após a mudança de alguns pesquisadores da colaboração da última para a primeira universidade.

Em cada um desses locais foram utilizadas diversas *farms*, com valores máximos de equipamentos indicados na tabela B.2. Observe-se que no Fermilab o número de *farms* variou de uma a seis, com configurações diferentes em vários momentos [41], sendo que ao final de 1993 a E791 era o maior usuário isolado do Fermilab, com capacidade de processamento equivalente a 1232 anos-Vax. Além disso era também o usuário mais eficiente, pois no momento em que o Fermilab relatava uma taxa de utilização de CPU's da ordem de 61% a E791 mantinha uma taxa média de 85%.

| Local | "Farms" | Máquinas | Fabricante |
|---|---|---|---|
| Fermilab | 6 | 105 | IBM e SGI |
| CBPF | 3 | 60 | Fermilab ACP2 |
| Umiss | 3 | 60 | DEC |
| KSU/OSU | 3 | 60 | DEC |

**Tabela B.2: Parque computacional máximo da E791**

A eficiência do processamento da E791 se deve principalmente a uma boa paralelização dos programas, administração eficiente do trabalho, acompanhamento constante do processamento, uso eficiente dos períodos improdutivos para montagem de fitas e uso das máquinas onde se realizava a leitura e escrita de fitas também para processamento.

Apesar de toda a otimização realizada, as *farms* mais velozes ainda levavam mais de cinco horas para processar cada fita de entrada, com uma taxa média de 38 eventos reconstruídos por segundo nas SGI's do Fermilab. Mesmo assim, os prazos para a reconstrução foram considerados curtos pelo volume de dados armazenados e os pesquisadores da colaboração tem produzido resultados interessantes do ponto de vista da Física nos últimos dois anos [3, 4, 5, 6].

# Apêndice C

# O processador MIPS R3000

## C.1 Descrição geral do processador

Neste apêndice descrevem-se algumas das principais características do processador MIPS R3000[14], que foi o modelo para a implementação do protótipo testado no Capítulo Quatro. A descrição que se segue procura apenas indicar os pontos da concepção do R3000 que foram usados para a implementação do protótipo e que devem ser tomados como referência para a implementação do gerador de código para outros processadores.

## C.2 Arquitetura

O MIPS R3000 é um processador RISC, contando com operação completa em 32 bits, "pipeline" de cinco estágios, controle interno de "cache" e de memória, esse último através de um "buffer" de tradução de endereços para mapeamento virtual-físico de memória (" **T**ranslation **L**ookaside **B**uffer -**TLB**"). Internamente possui 32 registradores de 32 bits, um contador de programas e dois registradores para armazenamento de operações de divisão e multiplicação inteiras. A Figura C.1 apresenta essa estrutura.

Nela pode ser destacada a separação dos registradores de controle dos registradores de programa. Assim, todo o controle de endereçamento na memória é feito em duas fases, uma com valores virtuais, contidos em registradores, e outra com valores físicos, calculados a partir da combinação entre endereços virtuais e mapas de endereçamento (TLB). Nesta estrutura existem dois pontos que foram usados na implementação e estão descritos a seguir:

[14] O leitor interessado na descrição detalhada desse processador pode conferir o livro de Gerry Kane [35], *MIPS RISC Architecture.*

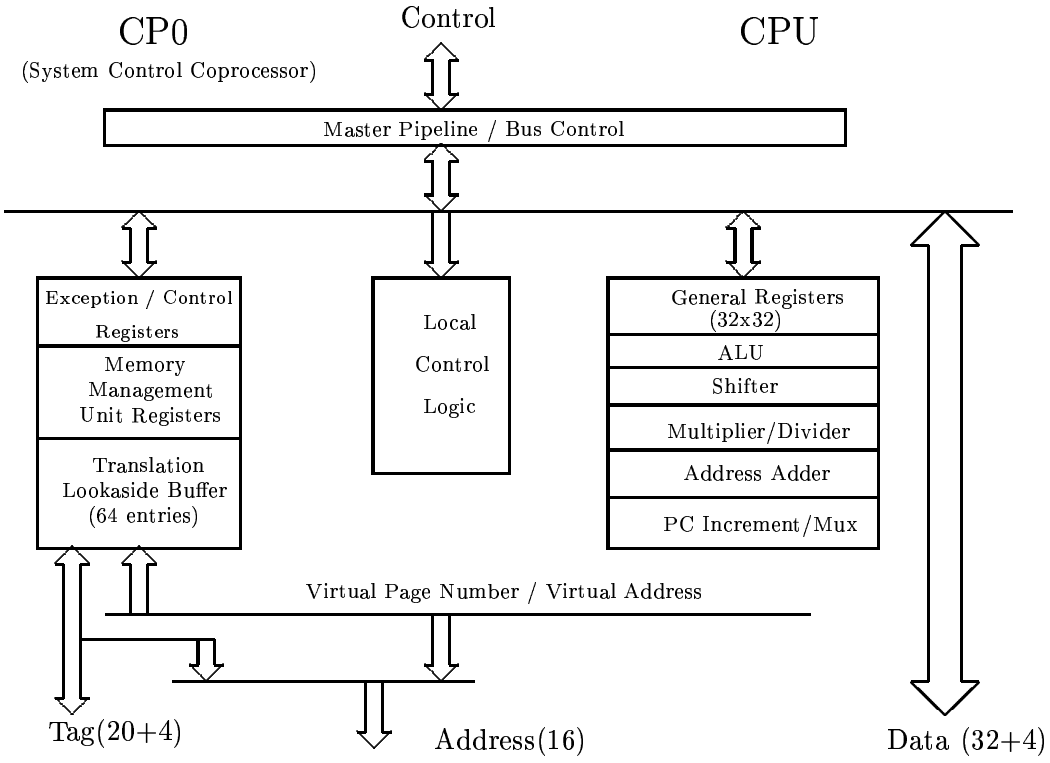



**Figura C.1: Blocos funcionais do MIPS R3000.**

o "pipeline" de instruções e o modo de mapeamento entre memória virtual (com endereços lógicos) e memória física.

**"Pipeline"**

O "pipeline" do R3000 possui cinco estágios, cuja seqüência de operação ao longo de uma instrução pode ser vista na figura C.2. Os estágios, e suas funções, são os seguintes:

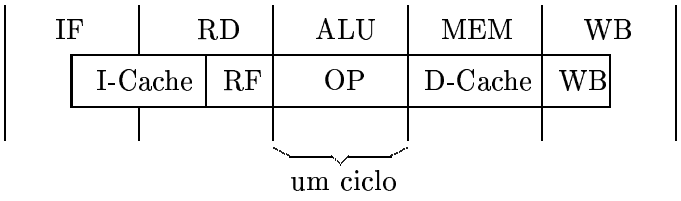



**Figura C.2: Seqüência de execução de instruções no "pipeline".**

1. **IF** - busca a instrução na memória;
2. **RD** - leitura dos operandos necessários nos registradores da CPU, feita durante a decodificação;
3. **ALU** - realiza efetivamente a operação sobre os operandos da instrução;
4. **MEM** - realiza o acesso à memória;
5. **WB** - escreve os resultados para o seu destino;

Essa divisão permite ao processador obter uma taxa aproximada de execução de uma instrução por ciclo de máquina. Isso apenas não é mantido para instruções usando um dos coprocessadores. Nesse caso a quantidade de ciclos por instrução é bem maior, como mostra a Tabela C.1. Deve-se salientar, no entanto, que o número de ciclos indicado nessa tabela inclui também ciclos onde podem ocorrer superposições de instruções de ponto flutuante. Foram essas quantidades de ciclos que o protótipo adotou para a geração do grafo de execução e no simulador como tempo de execução para cada instrução, tanto as de ponto flutuante como as demais.

| Instrução | Ciclos | Instrução | Ciclos |
|---|---|---|---|
| ADD.fmt | 2 | SUB.fmt | 2 |
| MUL.S | 4 | MUL.D | 5 |
| DIV.S | 12 | DIV.D | 19 |
| ABS.fmt | 1 | MOV.fmt | 1 |
| NEG.fmt | 1 | CVT.S.D | 2 |
| CVT.S.W | 3 | CVT.D.S | 1 |
| CVT.D.W | 3 | CVT.W | 2 |
| C.cond.fmt | 2 | BC1T/BC1F | 1 |
| LWC1 | 2 | SWC1 | 1 |
| MTC1 | 2 | MFC1 | 2 |
| CTC1 | 2 | CFC1 | 2 |

**Tabela C.1: Ciclos gastos em instruções do coprocessador**

### Espaço de endereços

O espaço de endereços virtual do processador R3000 é dividido em quatro segmentos, com um total de 4 Gbytes de memória, como mostra a Figura C.3. Metade desse espaço é reservado ao usuário numa área de endereços contíguos. Os outros 2 Gbytes são divididos entre três segmentos do núcleo do sistema, das quais duas não são mapeadas através da TLB

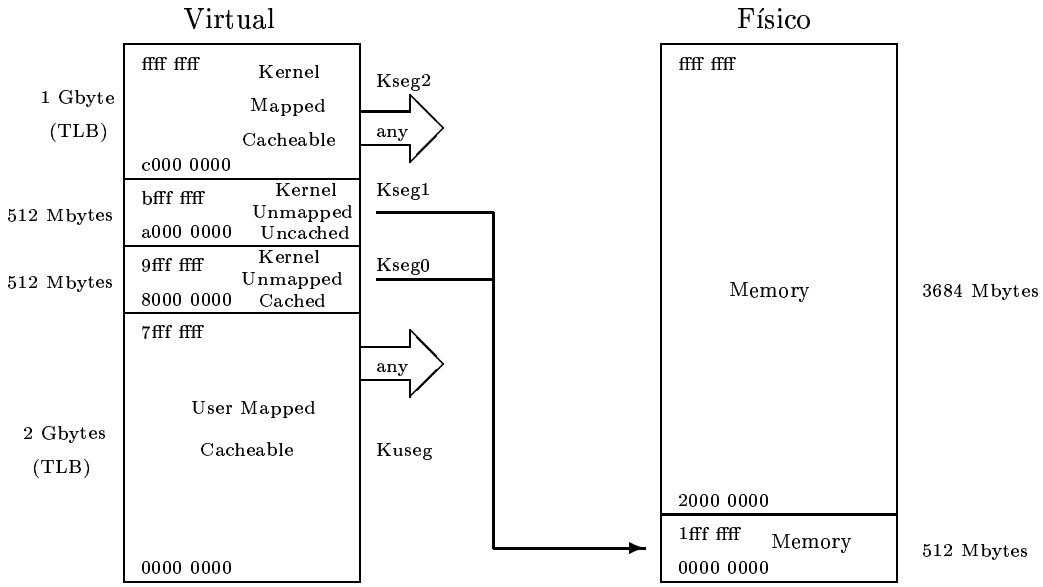



**Figura C.3: Mapeamento entre memória virtual e física.**

e ocupam 512 Mbytes cada. O mapeamento desses segmentos é feito diretamente nos primeiros 512 Mbytes da memória física. Além disso, o núcleo possui um segmento de 1 Gbyte mapeado através da TLB. Quando o processador opera em modo reservado ele tem acesso a esses segmentos, ficando o segmento de usuário mapeado byte a byte por um segmento virtual do núcleo (o *kuseg* da figura).

Esse formato de mapeamento foi usado para a verificação dos limites de endereçamento durante a geração do grafo de execução. Assim, para cada endereço de desvio verifica-se se ele ainda está dentro do espaço de endereços reservado para o usuário.

## C.3 Conjunto de instruções

Dentre todas as informações relativas ao R3000, aquelas relativas ao seu conjunto de instruções são as mais importantes para a implementação do protótipo. Uma característica dessas instruções é a sua divisão em três formatos básicos, segundo sua operação e modo de endereçamento. Isso simplifica sobremaneira a decodificação das instruções pelo processador e também pelo gerador do grafo. A identificação ocorre principalmente pelos bits 31-26, com algum apoio em outros conjuntos de bits para grupos especiais de instruções. A Figura C.4 apresenta o formato de cada uma dessas instruções.

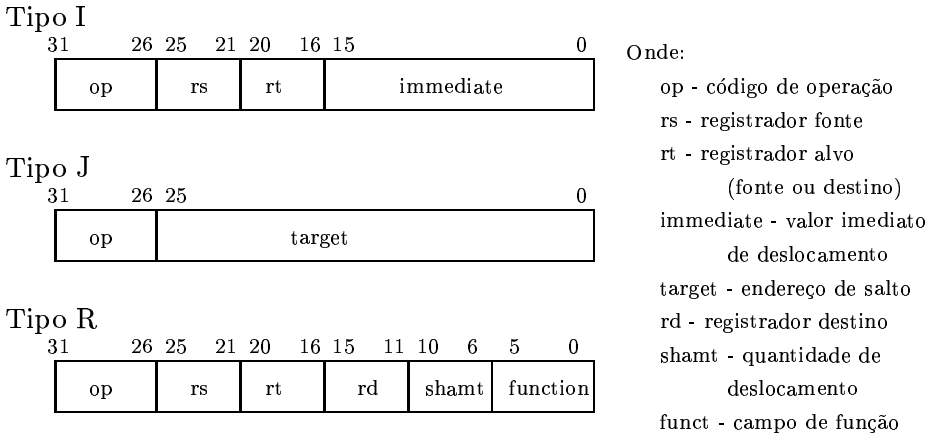



**Figura C.4: Formatos básicos de instruções no R3000.**

- Imediato (Tipo I)

  A instrução envolve o conteúdo de um registrador (indicado pelos bits 25-21 e mais um operando contido explicitamente nos 16 bits mais baixos da instrução. O resultado é armazenado num segundo registrador, indicado pelos bits 20-16.

- Salto (Tipo J)

  São instruções de desvio incondicional, logo contém apenas o endereço para o qual ocorrerá o desvio nos bits 25-0.

- Registrador (Tipo R)

  Essas instruções operam sempre com três registradores, dois para operandos e outro para o armazenamento do resultado. Além disso, a instrução precisa ser identificada através dos bits 5-0.

A divisão de instruções nesses formatos facilitou a interpretação do código executável, uma vez que cada grupo de instruções possui um formato padrão para a sua identificação e para a interpretação de seu significado semântico. O conjunto completo de instruções do MIPS R3000 pode ser visto na tabela C.2. Nela estão representadas todas as instruções, já divididas nos principais grupos semânticos disponíveis. A descrição precisa do que cada uma realiza e da forma como isso é feito pode ser encontrada em [35], embora seja necessário salientar que a implementação da biblioteca de manipulação do executável apenas foi possível com essas informações.

Dentre as instruções listadas na figura C.2 apenas JALR ("*Jump And Link Register*") apresentou uma maior dificuldade por utilizar o conteúdo de registradores adicionais, os

quais apenas são conhecidos através de uma manipulação conveniente da tabela de símbolos do programa. Todas as demais podem ser diretamente interpretadas, inclusive quanto ao conteúdo dos registradores, importantes apenas em tempo de execução e não para o estabelecimento dos caminhos possíveis dentro do grafo de execução.

28..26 Opcode

| 31..29 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | SPECIAL | BCOND | J | JAL | BEQ | BNE | BLEZ | BGTZ |
| 1 | ADDI | ADDIU | SLTI | SLTIU | ANDI | ORI | XORI | LUI |
| 2 | COP0 | COP1 | COP2 | COP3 | † | † | † | † |
| 3 | † | † | † | † | † | † | † | † |
| 4 | LB | LH | LWL | LW | LBU | LHU | LWR | † |
| 5 | SB | SH | SWL | SW | † | † | SWR | † |
| 6 | LWC0 | LWC1 | LWC2 | LWC3 | † | † | † | † |
| 7 | SWC0 | SWC1 | SWC2 | SWC3 | † | † | † | † |

2..0 SPECIAL

| 5..3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | SLL | † | SRL | SRA | SLLV | † | SRLV | SRAV |
| 1 | JR | JALR | † | † | SYSCALL | BREAK | † | † |
| 2 | MFHI | MTHI | MFLO | MTLO | † | † | † | † |
| 3 | MULT | MULTU | DIV | DIVU | † | † | † | † |
| 4 | ADD | ADDU | SUB | SUBU | AND | OR | XOR | NOR |
| 5 | † | † | SLT | SLTU | † | † | † | † |
| 6 | † | † | † | † | † | † | † | † |
| 7 | † | † | † | † | † | † | † | † |

18..16 BCOND

| 20..19 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | BLTZ | BGEZ | | | | | | |
| 1 | | | | | | | | |
| 2 | BLTZAL | BGEZAL | | | | | | |
| 3 | | | | | | | | |

25..23 COPz

| 22,21,16 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0,0,0 | MF | MT | BCF | | CO | | | |
| 0,0,1 | | | BCT | | | | | |
| 0,1,0 | | | | | | | | |
| 0,1,1 | | | | | | | | |
| 1,0,0 | CF | CT | | | | | | |
| 1,0,1 | | | | | | | | |
| 1,1,0 | | | | | | | | |
| 1,1,1 | | | | | | | | |

2..0 COP0

| 4..3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | | TLBR | TLBWI | | | | TLBWR | |
| 1 | TLBP | | | | | | | |
| 2 | RFE | | | | | | | |
| 3 | | | | | | | | |

†Reservado para futuras versões

**Tabela C.2: Códigos das instruções do MIPS R3000**

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] Adve, V.S.; "Analyzing the behavior and performance of parallel programs"; tese de doutorado apresentada ao Depto. de Ciência da Computação da Universidade de Wisconsin (EUA), disponível por ftp anônimo em ftp.cs.wisc.edu, no arquivo endereçado por pub/tech-reports/reports/93/tr1201.ps.Z , 1993.

[2] Aho, A.V. and Sethi, R. and Ulmann, J.D.; "Compilers: principles, techniques and tools"; Addison-Wesley Pub. Co., 1986.

[3] Aitala, E.M. et al.; "Search for $D^0\overline{D}^0$ mixing at the Fermilab E791 experiment"; *FERMI-PUB-96/109-E*, aceito para publicação em Phys. Review Letters.

[4] Aitala, E.M. et al.; "Mass splitting and production of $\Sigma_c^0$ and $\Sigma_c^{++}$ measured in 500 GeV/c $\pi^-$-N interactions"; *Phys. Letters*, vol B379, no. 1-4, p. 292-298, june 1996.

[5] Aitala, E.M. et al.; "Asymmetries between the production of $D^+$ and $D^-$ mesons from 500 GeV/c $\pi^-$-nucleous interactions as a function of $x_F$ and $p_t^2$", *Phys. Letters*, vol. B371, no. 1-2, p. 157-162, march 1996.

[6] Aitala, E.M. et al.; "Search for the flavor-changing neutral current decays $D^+ \rightarrow \pi^+\mu^+\mu^-$ and $D^+ \rightarrow \pi^+e^+e^-$"; *Phys. Review Letters*, vol. 76, no. 3, p. 364-367, jan. 1996.

[7] Alpern, B. and Carter, L.; "The myth of scalable high performance"; in *Proceedings of the 7th SIAM Parallel Computing Conference*, San Francisco, February 1995.

[8] Amato, S. et alii; "The E791 parallel architecture data acquisition system"; *Nuclear Instruments and Methods in Physics Research A*, n. 324, p. 535-542, 1993.

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

[10] Appel, J.A.; "Hadroproduction of charm particles"; *Annual Review of Nucelar and Particles Science*, Vol. 42, p. 367-399, 1992.

[11] Bowen, J.P.; "From programs to object code and back again using logic programming: compilation and decompilation"; *Journal of Software Maintenance: Research and Practice*, vol. 5, n. 4, p. 205-234, 1993.

[12] Breuer, P.T. and Bowen, J.P.; "decompilation: the enumeration of types and grammars"; *ACM Trans. on Programming Languages ans Systems (TOPLAS)*, vol. 16, n. 5, p. 1613-1647, 1994.

[13] Bradley, D.K. and 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.

[14] Browne, J.C. and Adiga, A.K.; "Graph structured performance models"; em *Performance Evaluation of Supercomputers*; J.L. Martin (editor), Elsevier Science Pub. B.V. (North-Holland), p. 239-281, 1988.

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

[16] Calzarossa,, M., Massari, L., Merlo, A. and Tessera, D.; "Parallel performance evaluation: the Medea tool"; em LNCS 1067, *Intl. COnf. and Exhibition on High-Performance Computing and Networking*, Brussels, Bélgica, p. 522-529, 1996.

[17] Couch, A.L.; "Locating performance problems in massively parallel execution"; *Proceedings of the IEEE*, vol. 81, n. 8, p. 1116-1125, 1993.

[18] Cremaldi, L.M. et alii; "Fermilab E791"; in *Proc. of the XXVI Int'l Conf. on High Energy Physics, Vol. 1*, p. 1058-1061, Dallas, Texas, August 1992.

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

[20] Estrin, G., Fenchel, R.S., Razouk, R.R. and Vernon, M.K.; "SARA (System ARchitect Apprentice): modeling, analysis, and simulation support for design of concurrent systems"; *IEEE Trans. on Software Engineering*, vol. 12, n. 2, p. 293-311, 1986.

[21] Fausey, M.R. et alii; "CPS & CPS batch reference guide - version 2.8"; Fermi National Accelerator Laboratory, Batavia, Illinois, january of 1993.

[22] Gandra, M., Drake, J.M. and Gregorio, J.A.; "Performance evaluation of parallel systems by using unbounded generalized stochastic Petri nets"; *IEEE Trans. on Software Engineering*, vol.18, n. 1, p. 55-71, 1992.

[23] Gelenbe, E. and Liu, Z.; "Performance analysis approximations for parallel processing in multiprocessor systems"; em *Parallel Processing*, M. Cosnard, M.H. Barton e M. Vanneschi (editores), Elsevier Science Pub. B.V. (North-Holland), 1988.

[24] van Gemund, A.J.C.; "Compile-time performance prediction of parallel systems"; em LNCS 977, *8th Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, p. 299-313, 1995.

[25] Gill, D.S., Zhou, S. and Sandhu, H.S.; "A case study of file system workload in a large-scale distributed environment"; *ACM Sigmetrics*, vol. 22, n. 1, p. 276-277, may 1994. Versão integral disponível por ftp anonimo em ftp.csri.toronto.edu, como relatório técnico CCSRI-281.

[26] Graham, S.L. and Kessler, P.B. and McKusick, M.K.; "gprof: a call graph execution profiler"; *ACM Sigplan Notices*, vol. 17, n. 6, p. 120-126, 1982.

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

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

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

[30] Hollingsworth, J.K. and Miller, B.P.; "SLACK: a new performance metric for parallel programs"; relatório técnico tr1260 do Depto. de Ciência da Computação da Universidade de Wisconsin (EUA), disponível por ftp anônimo em ftp.cs.wisc.edu, no arquivo endereçado por tech-reports/reports/95/tr1260.ps.Z , 1995.

[31] Hondroudakis, A., Procter, R. and Shanmugam, K.; "Performance evaluation and visualization with VISPAT"; em LNCS 964, *Third Intl. Conf. on Parallel Computing Technologies*, St. Petersburg, Russia, p. 180-185, 1995.

[32] Hwang, K.; "Advanced computer architecture: parallelism, scalability, programmability"; McGraw-hill, Inc., 1993.

[33] Iazeolla, G. and Marinuzzi, F.; "LISPACK - A methodology and tool for the performance analysis of parallel systems and algorithms"; *IEEE Trans. on Software Engineering*; vol. 19, n. 5, p. 486-502, 1993.

[34] Kapelnikov, A., Muntz, R.R. and Ercegovac, M.D.; "A methodology for the performance evaluation of distributed computations"; em *Distributed Processing*; M.H. Barton, E.L. Dagless and G.L. Reijins (editores), Elsevier Science Pub. B.V. (North-Holland), p. 465-479, 1988.

[35] Kane, G.; "MIPS RISC architecture"; Prentice-Hall, Inc., 1988.

[36] Kim, J. and Shin, K.G.; "Execution time analysis of communicating tasks in distributed systems"; *IEEE Trans. on Computers*; vol. 45, n. 5, p. 572-579, 1996.

[37] Kitajima, J.P. and Plateau, B.; "Modelling parallel program behaviour in ALPES"; *Information and Software Technology*, vol. 36, n. 7, p. 457-464, 1994.

[38] Larus, J.R. and Ball, T.; "Rewriting executable files to measure program behavior"; relatório técnico tr1083 do Depto. de Ciência da Computação da Universidade de Wisconsin (EUA), disponível por ftp anônimo em ftp.cs.wisc.edu, no arquivo endereçado por pub/tech-reports/reports/92/tr1083.ps.Z , 1992.

[39] Malony, A.D. and Mertsiotakis, V. and Quick, A.; "Automatic scalability analysis of parallel programs"; em LNCS 794, *Proc. of the 7th Int'l Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, p. 139-158, Viena, 1994.

[40] Manacero Jr., A.; "Comparisons of CPS performance under different network loads"; Fermilab Technical Report FERMI-PUB/033-94, Batavia, Illinois, 1994. Também disponível por ftp anônimo em ftp.ibilce.unesp.br, no arquivo endereçado por pub/uploads/cps3.ps.gz .

[41] Manacero Jr., A; "FNAL farms: production report and improvements"; E791 Internal Memo in *E791 Collaboration Meeting*, January 1994. Também disponível por ftp anônimo em ftp.ibilce.unesp.br, no arquivo endereçado por pub/uploads/e791memo.ps.gz .

[42] Marsan, M.A., Balbo, G. and Conte, G.; "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems"; *ACM Trans. Comput. Syst.*, vol. 2, n. 2, p. 93-122, 1984.

[43] MIPS Computer Systems; "RISCompiler and C programmer's guide"; MIPS Computer Systems, Sunnyvale, CA, EUA, 1989.

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

[45] Peterson, J.L.; "Petri net theory and the modeling of systems"; Prentice-Hall, 1981.

[46] Pierce, J. and Mudge, T.; "IDTrace - A tracing tool for i486 simulation"; relatório técnico CSE-TR-203-94 do Depto. de Eng. Elétrica e Ciência da Computação da Universidade de Michigan (EUA), disponível por ftp anônimo em ftp.eecs.umich.edu, no arquivo endereçado por techreports/cse/1994/CSE-TR-203-94.ps.Z , 1994.

[47] Pittman, T. and Peters, J.; "The art of compiler design: theory and practice", Prentice-Hall, Inc., 1992.

[48] Ponder, C. and Fateman, R.J.; "Inaccuracies in program profilers"; *Software - Practice and Experience*, vol. 18, n. 5, p. 459-467, 1988.

[49] Reed, D.A.; "Experimental analysis of parallel systems: techniques and open problems"; em LNCS 794, *Proc. of the 7th Int'l Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, p. 25-51, Viena, 1994.

[50] Reiser, J.F. and Skudlarek, J.P.; "Program profiling problems and a solution via machine language rewriting"; *ACM Sigplan Notices*, vol. 29, n. 1, p. 37-45, 1994.

[51] Rinaldo, F.J. and Fausey, M.R.; "Event reconstruction in high-energy physics"; IEEE Computer, p. 68-77, june 1993.

[52] Sahni, S. and Thanvantri, V.; "Performance metrics: keeping the focus on runtime"; *IEEE Parallel and Distributed Technology*, vol. 4, n. 1, p. 43-56., 1996.

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

[54] Satterthwaite, E.; "Debugging tools for high level languages"; *Software - Practice and Experience*, vol. 2, n. 3, p. 197-217, 1972.

[55] Sidwell, R.A. et alii; "E791 status report"; in *7th Meeting of the American Physical Society, Division of Particles and Fields - DPF 92, vol. 1*, p. 672-674, Batavia, Illinois, November 1992.

[56] Sötz, F.; "A method for performance prediction of parallel programs"; em LNCS 457, *CONPAR 90-VAPP IV, Joint Intl. Conf. on Vector and Parallel Processing*, p. 98-107, 1990.

[57] Srivastava, A. and Eustace, A.; "ATOM: a system for building customized program analysis tools"; *ACM Sigplan Notices*, vol. 29, n.6, p. 196-205, june 1994.

[58] Sullivan, K.Q. et alii; "CPS users's guide - version 2.9"; Fermi National Accelerator Laboratory, Batavia, Illinois, june of 1993.

[59] Summers, D.J. et alii; "Charm physics at Fermilab E791", in *Proceedings of the XXVIIth Rencontre de Moriond*, p. 417-422, Les Arcs, Savoie, France, March 1992.

[60] Sun, X.H. and Li, L.M.; "Scalable problems and memory-bound speedup"; *Journal of Parallel and Distributed Computing*, 1993.

[61] Sun, X.H. and Zhu, J.; "Performance prediction: a case study using a scalable shared-virtual-memory machine"; *IEEE Parallel and Distributed Technology*, vol. 4, n. 4, p. 36-49, 1996.

[62] Thomasian, A. and Bay, P.F.; "Analytic queueing netowrk models for parallel processing of task systems"; *IEEE Trans. on Computers*; vol. 35, n. 12, p. 1045-1054, 1986.

[63] Trivedi, K.S., Haverkort, B.R., Rindos, A. and Mainkar, V.; "Techniques and tools for reliability and performance evaluation: problems and perspectives"; em LNCS 794, *Proc. of the 7th Int'l Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, p. 1-24, Viena, 1994.

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

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

[66] Wall, D.W.; " Predicting program behavior using real or estimated profiles"; Nota Técnica TN-18 do DEC Western Research laboratory, Palo Alto, EUA, disponível por ftp anônimo em gatekeeper.dec.com, no arquivo endereçado por pub/DEC/WRL/research-reports/WRL-TN-18.ps, dec. 1990.

[67] Wall, D.W.; "Systems for late code modification"; em *Code Generation - Concepts, Tools, Techniques*; R. Giegerich, S.L. Graham (editores), p. 275-293, Spring-Verlag, 1992.

[68] Zuberek, W.M.; "Performance evaluation using unbounded timed Petri nets"; em *Proc. of the Third Intl. Workshop on Petri nets and Performance models*, Kyoto, Japan, p. 180-186, 1989.