Otimizador de um Grafo de Execução

ORIENTADO: PATRICIA RAMOS DE OLIVEIRA

ORIENTADOR: PROF. DR. ALEARDO MANACERO JR.

São José do Rio Preto

2003

Otimizador de um Grafo de Execução

ORIENTADO: PATRICIA RAMOS DE OLIVEIRA

ORIENTADOR: PROF.: DR. ALEARDO MANACERO JR

Projeto Final de Curso submetido ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas (Ibilce) da Universidade Estadual Paulista Júlio de Mesquita Filho, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação Departamento de Ciências de Computação e Estatística

_____ _____

Assinatura do Aluno Assinatura do Orientador

São José do Rio Preto

2003

Sumário

Lista de Figuras

Resumo

A área de computação de alto desempenho necessita de ferramentas precisas e eficientes para a análise de desempenho dos programas a serem executados nesse tipo de sistema. Manacero [7] propôs uma metodologia para essa tarefa baseado na simulação de um grafo de execução (grafo que representa os caminhos possíveis de execução de um programa), no qual uma ferramenta faria a geração desse grafo a partir do código executável. Nesse contexto passa a ser necessária a otimização desse grafo, reduzindo o total de vértices e arestas a serem simuladas. O objetivo desse trabalho foi a implementação desse módulo de otimização, produzindo resultados interessantes na operação de otimização.

Abstract

The High Performance Computing demands efficient and accurate tools to perform performance analysis of the software that will run on such systems. Manacero [7] propposed and approach for this job based on the simulation of execution graphs (graphs mapping the execution paths of a program). A tool built from this approach initially generates a graph from the executable code. Therefore, it beceomes necessary its optimization in order to reduce the number of edges and vertexes that have to be simulated. The goal of this work was the implementation of this optimization module. This work produced interesting results in the optimization of execution graphs.

Agradecimentos

Durante o desenvolvimento desse projeto obtive a ajuda de muitas pessoas, algumas diretamente do meu trabalho e outras indiretamente. Em seguida deixo meus maiores agradecimentos, pedindo desculpas caso não tenha citado nominalmente alguém em especial.

Ao meu orientador Aleardo Manacero Jr. pela confiança, paciência e suporte durante toda esse tempo.

A todos meus colegas de trabalho da CGB/Reitoria/UNESP, que estavam sempre me apoiando e incentivando para que eu concluísse esse trabalho.

A várias pessoas do IBILCE/UNESP, como o pessoal do laboratório de computação, funcionários da Biblioteca e professores do DCCE, que em vários momentos me deram suporte visto minha distância física.

Aos todos meus amigos e minha família que dividiram comigo toda essa fase da minha vida., que me apoiaram e me deram forças para que eu fosse em frente. Muito Obrigada!

Dedico este trabalho a minha mãe, Cleonice.

Capítulo 1

Introdução

Considerações Iniciais

Uma das mais críticas questões em sistemas de computação de alto desempenho é fazer um uso eficiente do sistema. Nesses casos, o alto custo dos equipamentos não permite o uso ineficiente dos mesmos e as necessidades de esforço computacional são elevadas para desprezar possíveis ganhos de eficiência. Quando se trata de computação paralela, pequenas diferenças de eficiência podem significar grandes variações nos custos de processamento.

Assim sendo, foi proposta por Manacero [7] uma metodologia para se fazer a predição de programas paralelos baseada na geração de grafos de execução a partir do código executável do programa e posteriormente sua simulação, juntamente com informações de configuração da máquina. Dentre as fases de geração do grafo e sua simulação, foi proposto um módulo de otimização do grafo, com o objetivo de reduzir o número de vértices a serem manipulados pelo simulador, uma vez que programas da área de alto desempenho costumam ser muito grandes, o que geraria grafos muito grandes. Uma melhoria no tempo de simulação é conseguida através da redução do número de vértices do grafo de execução, já que estes são os pontos de controle para simulação.

1.1. Objetivo do trabalho

Este trabalho tem com objetivo inicial examinar as possíveis técnicas de otimização para viabilidade de aplicação. As técnicas a serem analisadas fazem parte de otimização de código em compiladores, desenvolvidas em contexto de programas seqüenciais.

O objetivo final deste trabalho é a implementação do módulo de otimização do grafo de execução. As técnicas propostas nesse módulo baseiam-se em técnicas de otimização em compiladores, o que justifica a fase de estudo citada acima.

1.2. Relevância

Visto a importância do uso eficiente do sistema, principalmente em ambientes de alto desempenho, se faz necessário que esse desempenho seja definido e medido de alguma forma. Assim, os envolvidos no projeto do sistema e/ou programa têm condições de melhorar o mesmo, visando um melhor desempenho. Estas melhorias são possíveis através da análise dos dados fornecidos por ferramenta de análise de desempenho.

Considerando que programas da área de alto desempenho em geral são muito grandes, a aplicação da ferramenta de simulação do grafo de execução nesses programas, resultaria na simulação de grafos grandes.

1.3. Descrição do texto

Este trabalho está organizado de modo a fornecer primeiramente a fundamentação teórica do projeto, apresentando conceitos fundamentais sobre desempenho de programas, técnicas de otimização de código. Assim no capítulo dois são apresentados os fundamentos básicos sobre análise de desempenho de software, sobre a metodologia especificada por Manacero [7] e, principalmente, sobre técnicas para otimização de código utilizadas em compiladores, uma vez que parte das mesmas podem ser utilizadas ao otimizador de código.

No capítulo três, apresentam-se uma abordagem detalhada do módulo de otimização, assim como algumas alterações realizadas anteriormente a fase de implementação deste módulo, fase esta que se apresenta ao final deste capítulo.

Neste capítulo são apresentados os testes realizados com a implementação do módulo de otimização. Nele aparecem exemplificadas as fases de aplicação de cada uma das técnicas de otimização propostas no capítulo anterior.

Este trabalho encerra-se com a apresentação do capítulo cinco, onde se têm as conclusões deste trabalho, assim como algumas perspectivas para trabalhos futuros .

Capítulo 2

Análise de Desempenho e Otimização de Código

Considerações Iniciais

Neste capítulo são apresentados os conceitos mais fundamentais sobre desempenho de programas e técnicas de otimização de código. A discussão sobre desempenho de programas é naturalmente necessária pois esse projeto implementa um módulo de uma ferramenta para análise de desempenho.

Desse modo, apresenta-se inicialmente conceitos fundamentais sobre desempenho, incluindo formas de medição e análise. Depois faz-se também uma descrição da ferramenta em que esse projeto se insere, mostrando porque os chamados grafos de execução devem ser otimizados.

Numa etapa final do capítulo são apresentadas algumas técnicas para a otimização de código, as quais servem de base para a otimização de grafos de execução. O exame dessas técnicas permitira um melhor entendimento das opções de implementação adotadas para o módulo de otimização do grafo, que serão descritas no próximo capítulo.

2.1. Desempenho de Programas

No desenvolvimento de um programa, mesmo em suas fases iniciais, é necessário obter medidas sobre como será o seu comportamento no sistema para o qual ele está sendo projetado. Deste modo, as ferramentas para análise ou predição de oferecem mecanismos de instrumentação que forneça tais medidas. Essas medidas podem ser divididas em dois grupos:

- ***Teóricas**:* são geradas por modelos analíticos e apresentam como principal vantagem a simplicidade de sua obtenção, uma vez que existem apenas estimativas dos tempos de execução nos vários trechos do programa
- ***Experimentais**:* são realizadas durante a execução do programa. Existem inúmeras técnicas disponíveis dentro desta categoria: monitoração por "hardware", monitoração pelo sistema operacional, modificação do código fonte, modificação do código objeto e modificação do código executável.

As técnicas de monitoração têm como característica a precisão. Porém o uso da técnica de monitoração por "hardware" fica comprometida pelo alto custo de sua implementação. Já a segunda técnica apresentada é considerada primitiva e muito lenta pelo elevado número de interrupções do sistema geradas durante a medição.

As técnicas baseadas em modificação de código, seja no código fonte, objeto ou executável, são as mais usadas nas ferramentas de análise ou predição de desempenho, ainda que a precisão das medidas obtidas seja afetada.

2.1.1. Predição e análise de desempenho

Apesar de serem usadas com objetivos semelhantes, predizer e analisar o desempenho de um programa são atividades distintas. A preocupação básica na predição de desempenho é fornecer ao usuário um valor que indique se o desempenho esperado para o programa em questão é adequado ou não. Enquanto que, a análise de desempenho busca dar informações que possibilitem ao usuário alterar seu programa visando o alcance de um melhor desempenho.

As medidas necessárias em cada caso são diferentes. As medidas para análise de desempenho devem ser mais detalhadas do que para a predição. As medidas que interessam à predição de desempenho são mais simples e mais fáceis de serem obtidas.

2.1.2. Medidas de desempenho em sistemas paralelos

A principal medida de desempenho para sistemas paralelos é o **ganho de velocidade** (*"speedup"*), que indica o quão mais rápido um programa é executado em paralelo em relação a um programa de referência. Partindo-se deste conceito, diferentes abordagens na definição desse número foram criadas, sendo abordadas abaixo as principais:

- ***"speedup"* de tamanho fixo**: conhecida também como a **Lei de Amdahl,** procura determinar qual o máximo ganho de velocidade possível considerando o potencial de paralelismo existente no programa. Esta abordagem procura demonstrar que o ganho de velocidade não seria infinito mesmo se existissem infinitos processadores em paralelo.

$$\text{speedup de tamanho fixo} = (s + p) / \left(s + \frac{p}{N} \right)$$

  onde N é o número de processadores em paralelo, s representa a porção serial do programa e p sua porção paralela, com $s + p = 1$.

- ***"speedup"* de tempo fixo:** objetivo principal 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 este princípio é aplicável, dizemos que o algoritmo ou máquina é **escalável**. A medida de escalabilidade é importante para que se saiba se é vantajoso aumentar o número de processadores de um sistema paralelo.

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

Do ponto de vista do usuário do programa paralelo, o mais importante é se o programa executa ou não mais rápido e não se o algoritmo é ou não escalável. Já do ponto de vista do administrador do sistema, nem sempre um melhor desempenho do sistema é alcançado com um tempo de execução menor. O que se busca agora é ter velocidade adequada com menor custo de processamento, considerando tanto necessidade quanto disponibilidade. A partir desta situação, e usando o *"speedup"* são formuladas outras medidas de desempenho, como eficiência, redundância, utilização e qualidade de paralelismo [6].

2.2. Métodos para predição e/ou análise de desempenho

Tendo examinado os conceitos básicos sobre medidas de desempenho, é necessário uma breve apresentação dos principais métodos utilizados para análise de desempenho. Estes se apresentam aqui divididos em três categorias, que são: métodos analíticos, métodos baseados em "benchmarking" e métodos baseados em simulação.

A. Métodos analíticos

Os métodos analíticos utilizam-se de técnicas de análise fundamentadas essencialmente na solução algébrica de modelos de redes de Petri e cadeias de Markov, nas quais o programa é definido como uma seqüência de estados.

Considerando que na maioria das vezes o programa é grande e complexo, a precisão dos resultados de desempenho diminui. Apesar do problema de precisão, esta umas das técnicas que permitem a predição de desempenho de um programa quando não se tem o programa e o sistema disponíveis Outro ponto favorável aos métodos analíticos é a sua velocidade de resposta.

B. Métodos baseados em *"benchmarking"*

Estes métodos para análise de desempenho têm como estratégia de medição executar o programa na máquina alvo e medir tudo que for desejável. Apresentam menor complexidade de implementação, pois suas medições são feitas diretamente sobre o conjunto cujo desempenho se deseja medir.

Apesar dos gastos diretos com a compra do equipamento e indiretos pela manutenção do mesmo, a medição de desempenho através de *"benchmarking"* tem sido bastante utilizada. Tendo como a favor sua relativa precisão e a disponibilidade da máquina.

C. Métodos baseados em simulação

Nessa categoria aparecem vários simuladores, grande parte deles baseados na simulação de eventos em redes de Petri, o que se assemelha ao descrito para os métodos analíticos. A principal diferença entre os métodos de simulação e os analíticos está em como os resultados e o modelo são obtidos. No caso dos métodos analíticos os modelos são compostos por sistemas de equações que representam o sistema em análise, enquanto que nos baseados em simulação, as equações são substituídas por regras de comportamento que ditam como os eventos ocorrem e modificam o estado do sistema.

2.3. Predição de Desempenho por Simulação do código Executável

Simulação é uma estratégia atrativa em termos de precisão de resultados, facilidade de modelagem e custos computacionais. Outros argumentos favoráveis ao seu uso são a flexibilidade no modelo de simulação e a velocidade em que se pode fazer as medições para

predição e desempenho. O problema encontrado no uso de simuladores está na obtenção de um modelo fidedigno ao que acontece no mundo real.

Nesta seção descreve-se resumidamente a técnica de análise de desempenho pela simulação do código executável de programas paralelos, proposta por Manacero, em sua tese de doutorado [7]. Esta técnica baseia-se na reescrita do código executável, transformando-o em um grafo de execução e, obtendo as medidas de desempenho através da simulação desse grafo. Com isso tem-se um modelo flexível e eficiente e, além disso, que utiliza medidas não invasivas caracterizando modelos confiáveis.

2.3.1. Descrição da metodologia

A metodologia proposta procura separar o modelo para o sistema em três diferentes modelos: um deles detalhando o programa que será analisado, outro com detalhes da máquina em que se fará o processamento futuro do programa e finalmente um terceiro modelo versando sobre a interação entre os dois primeiros no momento da execução. O princípio desta metodologia é o da metodologia dos "três passos" de Herzog .

Na metodologia, 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. 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, etc. Por fim, o modelo de 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 2.1. ilustra como são obtidos cada um desses modelos.

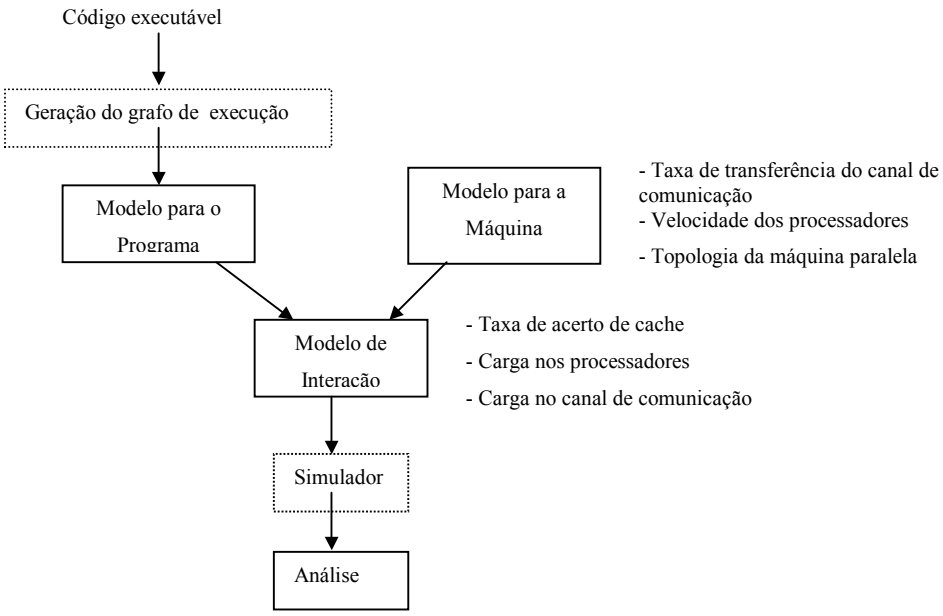


Figura 2.1: Obtenção dos modelos da metodologia de Herzog

Resumidamente, o método exige um programa executável, o qual é reescrito na forma de grafo de execução, reduzido para um grafo mínimo que ainda descreva o modelo do programa.

Posteriormente, este grafo mínimo é simulado, gerando dados sobre o desempenho do conjunto programa-máquina.

Vale ressaltar que o foco desse trabalho é no módulo de otimização do grafo de execução. Descrições detalhadas dos módulos de geração do grafo e de simulação do grafo podem ser encontradas em Moraes [8] e Tatsumi [10], respectivamente. A figura 2.2. mostra como esses módulos se relacionam e também como a metodologia de Herzog é mapeada neles.

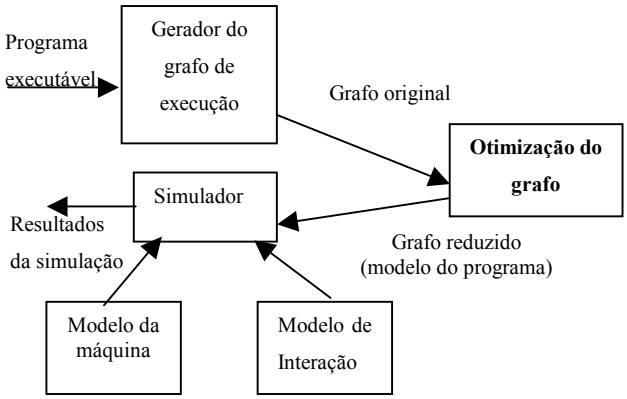


Figura 2.2.: Visão global do funcionamento do método

2.4. Otimização de código

A importância da otimização de código em compiladores tem crescido dramaticamente com o desenvolvimento dos computadores multiprocessados. Estas otimizações tradicionais foram desenvolvidas no contexto de programas seqüenciais e tornou-se essencial estender o escopo para programas explicitamente paralelos [9].

A otimização do grafo deve seguir os mesmos parâmetros definidos para a otimização de código realizada em compiladores. Essa restrição vem do fato de o grafo de execução ser, na pratica, uma representação simbólica do código do programa, incluindo os tempos de execução consumidos em cada trecho de código. Assim, qualquer redução no numero de vértices e arestas do grafo deve manter a correspondência entre os tempos modelados no grafo original e no grafo reduzido. Desse modo, as otimizações a serem implementadas seguirão, em grande parte, as técnicas usadas em compiladores, o que leva ao estudo preliminar de tais técnicas e, em especial, de técnicas voltadas para programas paralelos.

A maneira de tratar a otimização pode ser extremamente pragmática. É comum dizer que em um programa grande, a execução gasta 90% do tempo em 10% do código, e 10% do tempo nos 90% do código restante [1]. Isto acontece porque em geral um programa é constituído de inicializações e finalizações, entre as quais se encontra um conjunto de vários ciclos aninhados. O tempo maior de execução (“os 90%”) é gasto no ciclo mais interno (“os 10%”). A identificação dos trechos mais executados pode ser realizada por *profilers,* e a otimização pode se concentrar nestes trechos.

Considerando-se a quantidade de informação que se deseja manipular, o escopo da aplicação, as otimizações distinguem-se em: **otimizações locais**, que levam em consideração somente trechos pequenos de programas (por exemplo, trechos sem desvios, ou seja, trechos em linha reta), e **otimizações globais** que, consideram as inter-relações de todas as partes de um programa.

As transformações de melhoria de código ainda podem ser classificadas entre **independentes e dependentes da máquina**. As otimizações independentes são aquelas que não levam em consideração a arquitetura da máquina alvo, e as independentes são as transformações feitas em nível de instruções e de modos de endereçamento.

2.4.1. Critérios na aplicação das transformações de otimização

Para que uma otimização seja aplicada a um programa, deve-se criar o seguinte fluxo [4]:

1. Decidir qual parte do programa otimizar e qual transformação particular será nela aplicada;
2. Verificar que a transformação não muda o significado do programa ou o modifique de um modo que seja aceitável para o usuário e;
3. Transformar o programa.

Nesta seção, serão tratados alguns pontos importantes para os dois primeiros tópicos, apresentada a análise de dependências que é usada para decidir e verificar muitas transformações. As possíveis transformações a serem aplicadas serão apresentadas mais a frente.

A. Corretude

Quando um programa é transformado, seu significado deve permanecer o mesmo. O modo mais fácil de realizar isto é exigir que o programa transformado realize exatamente as mesmas operações que o original, na exata ordem imposta pela semântica da linguagem. Uma definição mais prática é:

Uma transformação é legal se o programa original e o transformado produzem exatamente a mesma saída para todas execuções idênticas. Sendo que, duas execuções de um programa são idênticas se elas são abastecidas com a mesma entrada de dados e se todo par correspondente de operações não determinísticas em duas execuções produzem o mesmo resultado.

B. Escopo

As otimizações podem ser aplicadas a um programa em diferentes níveis de granulação. Quando o escopo da transformação é ampliado, o custo da análise geralmente aumenta. Alguns estágios de complexidade são:

- ***Instrução*** - as expressões aritméticas são a fonte principal do potencial de otimização.
- ***Bloco Básico -*** este é o foco das antigas técnicas de otimização, tendo como vantagem para a análise, a existência de somente um ponto de entrada. Um bloco básico é uma seqüência de comandos consecutivos no qual o fluxo de controle entra no início e chega ao final sem parada ou possibilidade de salto exceto ao final.
- ***Laço mais interno -*** importante quando a arquitetura alvo é a de alto desempenho. Várias técnicas são aplicadas somente no contexto de laços internos.

- ***Ninho de laço perfeito -*** um ninho de laço é um conjunto de laços, um dentro do outro. Um ninho é chamado de ninho perfeito se o corpo de todo laço, que não o mais interno, consiste somente do próximo laço no ninho. Como um ninho perfeito é sumarizado e organizado mais facilmente, várias transformações aplicam-se somente a ninhos perfeitos.
- ***Ninhos de laços comuns -*** qualquer ninho de laço, seja perfeito ou não.
- ***Procedimento -*** algumas otimizações, transformações de acesso à memória em particular, produzem melhor aproveitamento se forem aplicadas a um procedimento inteiro de uma só vez. Neste caso o compilador deve ser capaz de lidar com as interações de todos os blocos básicos e as transferências de controle dentro do procedimento. O termo padrão para otimizações em nível de procedimento na literatura é **otimização global**.
- ***Inter-procedural -*** vários procedimentos juntos freqüentemente expõe mais oportunidades para otimizações; em particular, a sobrecarga de chamadas de procedimento é muitas vezes significante, e pode ocasionalmente ser reduzida ou eliminada com a análise inter-procedural.

C. Análise de Dependência

Existem várias técnicas de análise usadas na otimização. Será feita aqui uma introdução da análise de dependência, sua terminologia e a teoria básica.

Uma dependência é um relacionamento entre dois cálculos que estabelece restrições em suas ordens de execução. A análise de dependência identifica estas restrições, que são então usadas para determinar se uma específica transformação pode ser aplicada sem que se mude a semântica do cálculo.

Existem dois tipos de dependências possíveis: *dependência de controle* e *dependência de dados*. Existe uma dependência de controle entre a instrução 1 e a instrução 2, denotado como $S_1 \xrightarrow{c} S_2$, quando a instrução S_1 determina se S_2 será executada . Por exemplo:

```
S1   if (a = 3)  then
S2   b = 10
end if
```

Duas instruções têm uma *dependência de dados* se elas não podem ser executadas simultaneamente devido ao uso conflitivo da mesma variável. Existem quatro tipos de dependência de dados: *dependência de fluxo* (também chamada de dependência verdadeira), *antidependência*, *dependência de saída* e *dependência de entrada.*

S_4 tem um *dependência de fluxo* em S_3 (denotado por $S_3 \rightarrow S_4$) quando S_3 deve ser executado primeiro, pois este escreve um valor que é lido por S_4. Por exemplo:

```
S3   a = c * 10
S4   d = 2 * a + c
```

S_6 tem uma *antidependência* em S_5 (denotado por $S_5 \xrightarrow{/} S_6$) quando S_6 escreve uma variável que é lida por S_5 :

```
S5 e = f * 4 + g
S6 g = 2 * h
```

Uma *antidependência* não restringe a execução tão firmemente como uma *dependência de fluxo.* Como anteriormente, o código executará corretamente se S_6 é retardado até mesmo depois que S_5 complete. Uma solução alternativa é usar duas posições de memória g_5 *e* g_6 para

guardar os valores lidos em S_5 e escrito em S_6, respectivamente. Se a escrita em S_6 se completa primeiro, o valor antigo ainda estará disponível em g_5.

Uma *dependência de saída*, (denotado por $S_7 \xrightarrow{o} S_8$), existe quando ambas instruções escrevem na mesma variável:

S_7a = b * c
S_8a = d + e

Novamente, como na *antidependência,* a replicação do carregamento permite as instruções executarem concorrentemente. Neste pequeno exemplo não há uso interposto da variável *a* e nenhuma transferência de controle entre as duas instruções, então o cálculo em S_7 é redundante e pode efetivamente ser eliminado.

O quarto possível relacionamento, chamado de *dependência de entrada*, acontece quando dois acessos à mesma posição de memória são ambos de escrita.

Um tipo não específico de dependência é denotado por $S_1 \Rightarrow S_2$. E no caso de dependência de dados, estamos sendo de certo modo imprecisos. As referências às variáveis individuais na instrução produz as dependências e, não as instruções como um todo. No exemplo de *dependência de saída,* as variáveis *b, c, d* e *e* podem todas serem lidas da memória em qualquer ordem, e os resultados de *b * c* e *d + e* podem ser executados assim que seus operandos forem lidos da memória. A expressão $S_7 \xrightarrow{o} S_8$ de fato significa que o carregamento do valor *b * c* em *a* deve preceder o carregamento do valor *d + e* em *a*. Quando existe uma possível ambigüidade, distingui-se usando diferentes referências à variável na instrução.

2.5. Transformações de otimização

Esta seção cataloga as transformações de otimização de programa mais comuns. Ao decorrer desta seção poderá verificar que a ênfase principal das otimizações em compiladores é nas transformações realizadas em laços, já que em geral é onde a maior parte do tempo de execução é gasto.

Uma referência padrão em compiladores, em geral, é o livro do “Dragão”, o qual este trabalho teve como referência para a maioria dos exemplos [1], assim como alguns antigos ensaios [3].

Como o número de otimizações de código aplicadas em compiladores é bem extenso, será tratado aqui somente algumas. Abaixo segue uma possível divisão:

a) Transformações de laço baseadas no fluxo de dados;
b) Transformações de reordenação de laço;
c) Transformações de reestruturação de laço;
d) Transformações de substituição de laço;
e) Transformações de acesso à memória;
f) Avaliação parcial;
g) Eliminação de redundância.

(a) Transformações de laço baseadas no fluxo de dados

Um grande número de otimizações clássicas de laços é baseado em análise de fluxo de dados, que é feito através das variáveis do programa.

- **Redução do complexidade baseado no laço**

A redução no tamanho substitui uma expressão no laço com uma outra que seja equivalente mas que use um operador menos caro. A figura 2.3.(a) mostra um laço que contém uma multiplicação. A figura 2.3.(b) é a versão transformada do laço onde a multiplicação foi substituída por uma adição.

Figura 2.3.: Redução de complexidade

- **Movimentação de código invariante no laço**

Quando um cálculo aparece dentro de um laço, mas seu resultado não muda entre as iterações, este pode ser movido para fora do laço.

A movimentação de código pode ser aplicada em expressões de alto nível no código fonte ou, a um baixo nível para cálculos de endereçamento.

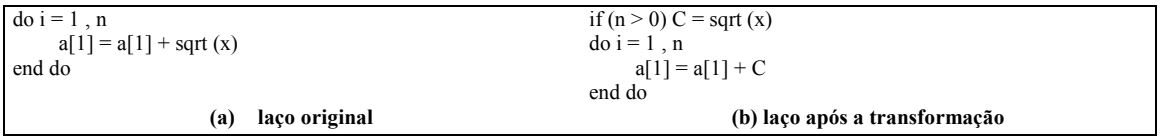


Figura 2.4.: Movimentação de código invariante no laço

A figura 2.4.(a) apresenta um exemplo no qual uma função cara é movida para fora do interior do laço. O teste no código transformado na figura 2.4.(b) garante que se o laço nunca for executado, o código movido não será também executado, a fim de que não se crie uma exceção.

- **"Desdobramento" do Laço**

O desdobramento do laço é aplicado quando um laço contém uma condicional, sendo uma condição do teste invariante no laço. O laço é então replicado dentro de cada um dos possíveis ramos da condicional, reduzindo o tamanho do código do corpo do laço e, possibilitando a paralelização de uma ramo da condicional.

Condicionais candidatas a esta técnica, podem ser detectadas durante a análise de movimentação de código, que identifica valores invariantes no laço.

Na figura 2.5.(a) a variável *x* é invariante no laço, permitindo que o laço seja desdobrado e que o ramo **true** seja executado em paralelo, como mostrado na figura 2.5.(b). Nota-se que,

como na movimentação de código invariante, se houver qualquer chance que a avaliação de condições causará uma exceção, deve-se garantir por um teste que o laço não será executado.

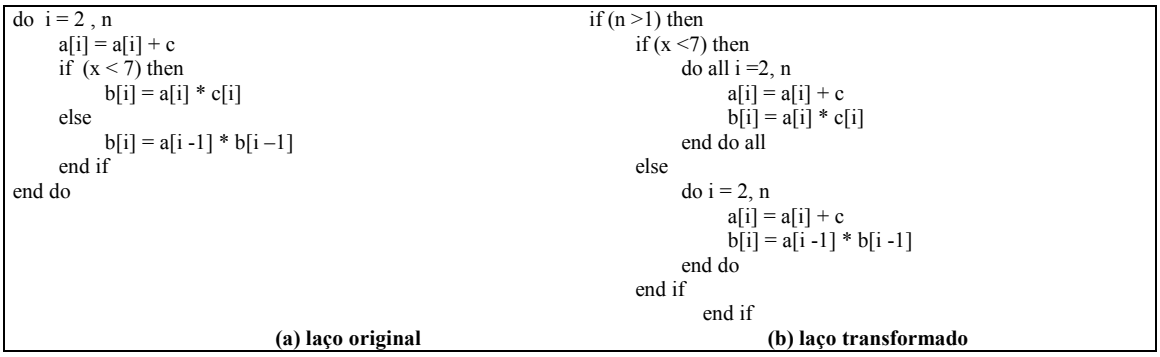


Figura 2.5.: Desdobramento de laço

(b) Transformações de reordenação de laço

As transformações desta classe, modificam a relativa ordem de execução das iterações de um ou mais ninhos de laços. Estas transformações são usadas principalmente para expor paralelismo e melhorar a localidade da memória.

Alguns compiladores só aplicam transformações de reordenação em ninhos de laços perfeitos. Para aumentar as possibilidades de otimização, pode-se às vezes aplicar *distribuição de laço* para se extrair ninhos de laços perfeitos de um ninho imperfeito.

- **Permutação de laço**

Esta técnica troca a posição de dois laços em um ninho perfeito, geralmente movendo um dos laços externos para a posição mais interna. A permutação é uma das transformações mais poderosas e pode aumentar o desempenho de vários modos. A permutação do laço pode ser realizada para:

- habilitar a vetorização permutando-se um laço interno dependente, com um externo e independente;
- melhorar o desempenho paralelo pela movimentação de um laço independente de um ninho de laços, para aumentar a granulação de cada interação e reduzir o número de barreiras de sincronismo;
- reduzir o número de iterações de um laço;
- aumentar o número de expressões invariantes do laço em um laço interno.

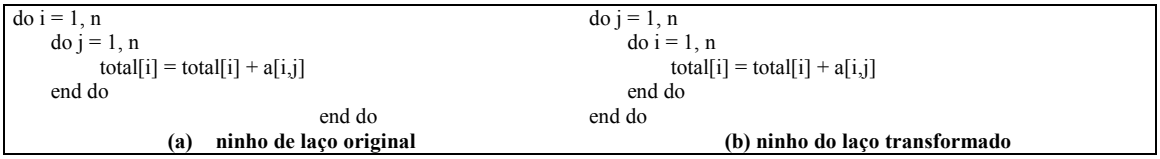


Figura 2.6.: Permutação de laços

Na figura 2.6.(a) acima, o laço mais interno acessa a estrutura de dados *a* com passo *n*. Pela permutação dos laços, converteu-se o acesso para um passo - 1 como mostrado na figura 2.6.(b).

- **Distribuição de Laço**

A distribuição (também chamada de rachamento ou divisão de laço) quebra um único laço em muitos. Cada um dos novos laços tem o mesmo espaço de iteração que o original, mas contém um subconjunto das instruções do laço original. Esta técnica é usada para:

- criar ninhos de laço perfeitos;
- criar sub-laços com menos dependências; e
- reduzir as solicitações de memória, pela iteração de menos vetores.

A Figura 2.7. é um exemplo na qual a distribuição remove dependências e permite que parte do laço seja executada em paralelo.

```
do i = 1, n
      a[i] = a[i] + c
      x[i+1] = x[i] * 7 + x[i+1] + a[i]
end do
```

(a) laço original

```
do all i = 1, n
      a[i] = a[i] + c
end do all
do i = 1, n
      x[i+1] = x[i] * 7 + x[i+1] + a[i]
end do
```

(b) laço modificado

Figura 2.7.: distribuição de laço

(c) Transformações de restruturação de laço

As técnicas desta categoria têm como característica a mudança da estrutura do laço, mas deixa os cálculos realizados por uma iteração do corpo de um laço e suas relativas ordens inalteradas.

- **Desenrolamento de laço**

O desenrolamento replica o corpo do laço um número de vezes chamado de fator de desenrolamento (*u)* e itera com passo *u* ao invés de 1 [3]. Os benefícios desta técnica têm sido estudado em diferentes arquiteturas .

Esta técnica pode melhorar o desempenho reduzindo o *overhead* no laço, aumentando o paralelismo de instrução.

Outras técnicas aparecem nesta categoria: *Software Pipelining*, Aglutinação de laço e sua variante Desmoronamento de laço, Normalização de Laço e outras. Todas tendo como objetivo, reduzir o *overhead* no laço e aumentar o paralelismo de instrução.

(d) Transformações de substituição de laço

As transformações de substituição de laço operam nos laços como um todo e alteram completamente suas estruturas.

- **Reconhecimento de redução**

Uma redução é uma operação que computa um valor escalar de um vetor. Reduções comuns incluem o cálculo da soma ou do valor máximo dos elementos em um vetor. Exemplo: a soma dos elementos de um vetor *a* é acumulado no escalar *s*. O vetor de dependência para o laço é (1) ou (<). Enquanto que um laço como vetor de direção (<) deve geralmente ser executado serialmente, as reduções podem ser paralelizadas se a operação realizada é associativa. A comutatividade fornece oportunidades adicionais para reordenação.

Outras possíveis transformações neste contexto são: Reconhecimento de idioma em laços e Escalarização de instruções vetoriais.

(e) Transformações de acesso à memória

As aplicações de alto desempenho apresentam o problema de limitação da memória, limitando assim o potencial de cálculo. Como resultado, otimizações do uso da memória tornou-se constantemente mais importante.

Fatores como reutilização, paralelismo e tamanho do conjunto de trabalho afetam o desempenho da memória de um sistema. O uso eficiente dos registradores é também crucial para um bom desempenho.

Algumas das possíveis otimizações de memória são: Enchimento de Vetores, Expansão escalar, Contração vetorial e Substituição escalar. Em [4] pode ser encontrado uma descrição de cada uma das transformações aqui citadas.

(f) Avaliação parcial

A avaliação parcial refere-se à técnica geral de executar parte dos cálculos em tempo de compilação. A maioria das otimizações clássicas baseadas na análise de fluxo de dados também são uma forma de avaliação parcial ou de eliminação de redundância.

- **Propagação de constante**

A propagação de constante é uma das mais importantes otimizações que um compilador pode realizar. Os programas tipicamente contêm muitas constantes; pela propagação delas através do programa, o compilador pode realizar uma quantia significante de pré cálculos. Mais importante ainda, é que a propagação revela muitas oportunidades para outras otimizações. Além das possibilidades óbvias como *eliminação de código morto*, as otimizações em laços são afetadas já que constantes aparecem freqüentemente em suas escalas de indução.

- **Dobramento de constante**

Esta técnica acompanha a propagação de constantes. Quando uma expressão contém uma operação com valores constantes como operandos, pode-se substituir a expressão com o resultado. Por exemplo, *x = 3 ** 2* torna-se *x = 9*. Tipicamente, constantes são propagadas e dobradas simultaneamente [1].

- **Propagação de cópia**

Otimizações como *eliminação da variável de indução* e *eliminação de subexpressão* pode ocasionar a criação de várias cópias do mesmo valor. O que se faz então, é a propagação da aparição original do valor eliminando-se as cópias redundantes.

A propagação de cópia reduz a sobrecarga de registradores e elimina a redundância nas instruções de movimentação de registrador para registrador.

- **Reassociação**

É uma técnica para aumentar o número de subexpressões comuns em um programa[3]. Em geral, este tipo de transformação é aplicada no cálculo de endereçamento dentro do laço quando realiza-se redução de complexidade em expressões de variáveis de indução. Os cálculos de endereçamento gerados consistem de várias multiplicações e adições. A reassociação aplica leis de associação, comutação e distribuição para reescrever estas expressões na forma soma de produtos.

- **Simplificação Algébrica**

Expressões aritméticas são simplificadas pela aplicação de regras algébricas. Um exemplo particular útil é o conjunto de identidades algébricas. Por exemplo, a instrução $x = (y * 1 + 0) /1$ pode ser transformada em $x = y$ se x e y são inteiros.

Aritmética de ponto flutuante pode ser problemática para simplificar; por exemplo, se x é um número de ponto flutuante com valor *Nan* (não um número), então $x * 0 = x$, ao invés de 0.

- **Redução da Complexidade**

Identidades como $x * 2 = x + x$ e $i * 2^c = i << c$, podem ser chamadas de *redução de complexidade* porque substituem uma operador caro por um operador mais barato e equivalente. Anteriormente, foi discutido a aplicação da redução de complexidade em operações que aparecem dentro de um laço.

(g) Eliminação de redundância

Existem várias otimizações que aprimoram o desempenho pela identificação de cálculos redundantes e remoção dos mesmos[2]. Aqui neste texto já foi apresentada uma transformação desta, a *movimentação de código invariante em um laço*, onde um cálculo era realizado repetidamente, quando poderia ser realizado somente uma vez.

As transformações de eliminação de redundância removem dois outros tipos de cálculos: aqueles que são *inatingíveis* e aqueles que não são *imprestáveis*. Um cálculo é inatingível se nunca é executado; a remoção deste do programa não terá nenhum efeito semântico na execução do programa. O código inatingível é criado pelo programador (mais geralmente com códigos de *debug* condicionais), ou por transformações que deixaram códigos "órfãos" para trás.

Um cálculo é *imprestável* se nenhuma das saídas do programa são dependentes dele.

- **Eliminação de código inatingível**

A maioria dos compiladores realizam eliminação de código inatingível [2,3]. Em programas estruturados, existem dois modos básicos do código tornar-se inatingível. Se já é

sabido que uma condição é verdadeira ou falsa, um ramo da condicional nunca será adotado e seu código pode ser eliminado. Uma outra fonte comum para código inatingível é um laço que não realiza nenhuma iteração.

Em um programa não estruturado que usa instruções **goto** para transferência de controle, código inatingível não é óbvio pela estrutura do programa, mas pode ser encontrado pelo percorrimento do grafo do fluxo de controle do programa.

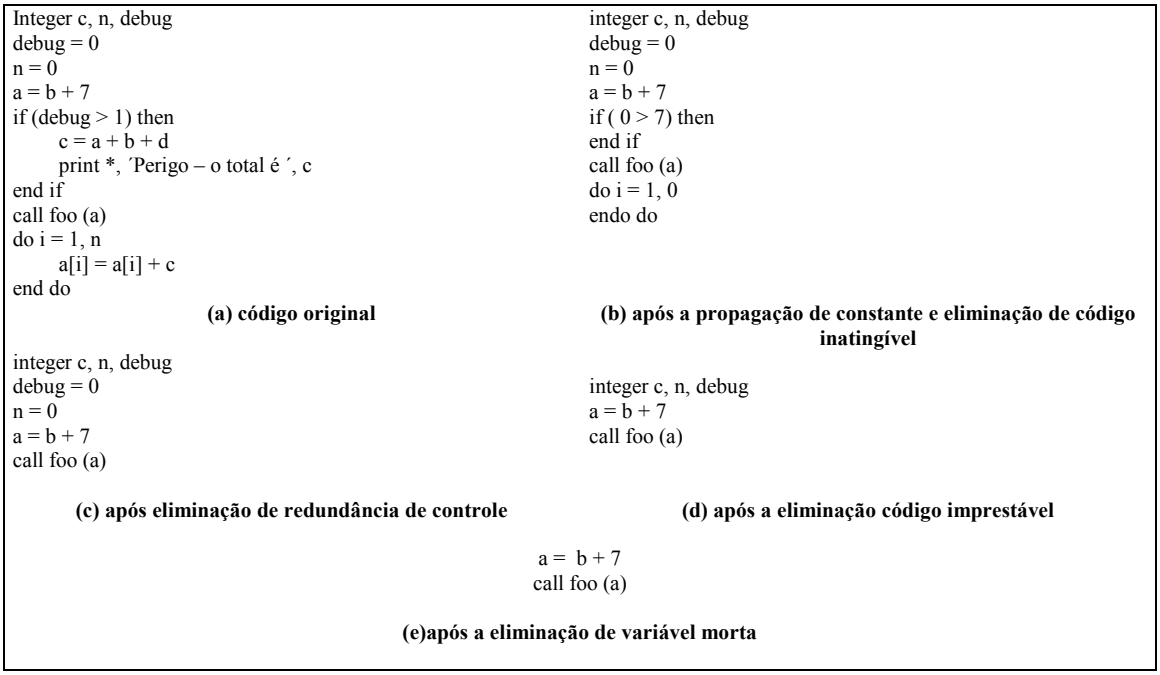


Figura 2.8.: Eliminação de Redundância

Códigos inatingíveis e imprestáveis são muitas vezes gerado pela *propagação de constante,* descrito anteriormente. Na figura 2.8.(a), a variável *debug* é uma constante. Quando seu valor é propagado, a expressão condicional torna-se *if (0>1)*. Esta expressão é sempre falsa, então o corpo desta condicional nunca será executado e pode ser eliminado, como mostra a figura 2.8.(b).

Similarmente, o corpo do laço **do** nunca é executado e então removido.

Código inatingível é também conhecido como *código morto*, mas esse termo também é aplicado à código imprestável, então escolheu-se aqui pelo termo mais específico.

Considerado as vezes como uma etapa separada, a *eliminação de redundância de controle*, que remove construções de controle como laços e condicionais quando estas se tornam redundantes (em geral como resultado da propagação de constante). Na figura 2.8.(b), o laço e as expressões de controle condicionais não são usadas e pode-se removê-los do programa como mostrado em 2.19(c).

- **Eliminação de código imprestável**

Código imprestável é criado geralmente por outras otimizações, como a eliminação de código inatingível. Quando o compilador descobre que um valor a ser computado por uma instrução não é necessário, ele pode removê-lo. Isto pode ser feito por qualquer variável não global e, que não esteja *viva* imediatamente após a definição da instrução. A análise de variável viva é um problema de fluxo de dados famoso [1]. Na figura 2.8.(c), os valores computados pelas instruções de recebimento não são mais usadas; sendo eliminadas em 2.8.(d).

- **Eliminação da variável morta**

Após uma série de transformações, particularmente otimizações de laço, existem variáveis cujos valores nunca serão utilizados. As variáveis desnecessárias são chamadas de *variáveis mortas*; a eliminação destas variáveis são otimizações simples [1,3].

Na figura 2.8.(d), as variáveis *c, n* e *debug* não são mais usadas e podem ser removidas; 2.8.(e) apresenta o código após a eliminação destas variáveis.

- **Eliminação de subexpressões comuns**

Em muitos casos, um conjunto de cálculos conterá subexpressões idênticas [1]. A redundância pode aparecer no código do usuário e nos cálculos de endereçamento gerados pelo compilador. Um compilador pode computar o valor de uma subexpressão, armazená-lo e, reutilizar o resultado armazenado [3]. A eliminação de subexpressões simples é uma transformação importante e, é quase que universalmente realizada. Embora seja boa a idéia de realizar a eliminação de subexpressões comuns sempre que possível, deve-se considerar a pressão de registradores correntes e o custo para refazer os cálculos.

2.5. Considerações Finais

O entendimento do projeto da ferramenta para analise de desempenho, como descrito nas seções iniciais desse capítulo e em [7], [8] e [10], facilita o trabalho de identificação de quais das técnicas apresentadas na seção anterior serão, de fato, úteis na redução do grafo de otimização.

No próximo capítulo se encontra a descrição da especificação e implementação do módulo de otimização do grafo de execução. Nele são apresentadas as razões pela escolha de cada uma das técnicas de redução efetivamente utilizadas no módulo, bem como a forma como sua inserção ocorreu.

Capítulo 3

Metodologia

Considerações Iniciais

Neste capítulo serão descritos primeiramente os passos envolvidos na elaboração do projeto. Em uma nova seção apresenta-se a especificação do módulo proposto por Manacero [7] juntamente com alterações realizadas, concluindo-se com uma descrição da implementação do mesmo.

3.1. Descrição da Metodologia

O projeto foi desenvolvido nas seguintes etapas:

- Análise das funções envolvidas na geração dos grafos de execução, para que fosse possível apresentar uma clara descrição da implementação do módulo de otimização;
- Alteração no arquivo de saída produzido pelo gerador de grafos de execução. Isto se fez necessário quando se observou a inexistência de informações essenciais para a aplicação das reduções que serão propostas;
- Codificação de funções responsáveis tanto pela alteração acima citada, quanto pelas reduções no grafo de execução;
- Execução de testes e avaliação dos resultados, cujas apresentações se fazem no próximo capítulo.

3.2. Módulo de otimização

Nesta seção será feita uma descrição detalhada do módulo de otimização. Porém, para que o mesmo possa ser melhor compreendido, é necessário que se apresente conceitos ligados ao grafo de execução, definindo o que esse grafo representa e como o mesmo é composto.

3.2.1. Representação e conceitos

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. Os vértices desse grafo definem pontos de processamento e as arestas indicam os caminhos que podem ser seguidos durante a execução do programa.

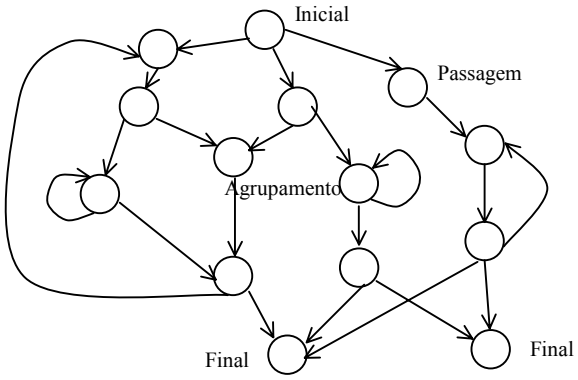


Figura 3.1: Grafo de Execução de um programa

Uma representação do grafo de execução é *G = (V,A)*, onde *G* é o grafo orientado, com um conjunto de vértices *V* e um conjunto de arestas *A*. Considerando a relação de interdependência, representada no grafo pelas arestas, entre os conjuntos de instruções, representados pelos vértices, na figura 3.2 abaixo, tem-se que as instruções representadas *v2* serão executadas somente após a conclusão das instruções em *v1*.

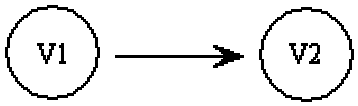


Figura 3.2: Interdependência entre dois vértices

Sendo assim, segundo os diferentes tipos de conexão com vértices vizinhos, tem-se uma classificação de vértices em cinco categorias: INICIAL, PASSAGEM, DECISÃO, AGRUPAMENTO e FINAL. A seguir tem-se a definição é das categorias citadas, seguindo de uma representação gráfica na figura 3.3:

- **Inicial**: um dado grafo de execução, possui somente um vértice dessa categoria, visto que este indica o ponto no qual se inicia a execução de um programa. Este

vértice é caracterizado por não possuir arestas incidentes, ou seja não possuir vértices ascendentes.

- **Passagem:** vértice caracterizado pela existência de apenas uma aresta chegando e uma aresta partindo, representando então trechos do programa em que não existem desvios condicionais.
- **Decisão:** são vértices que representam trechos do programa que contém o início de desvios condicionais. Estes vértices são caracterizados por possuírem um vértice ascendente e dois ou mais descendentes. Os vértices descendentes indicam os possíveis caminhos a serem seguidos durante a execução, dependendo do valor da condição testada no desvio.
- **Agrupamento:** vértices caracterizados pela existência de duas ou mais arestas incidentes, representando o agrupamento dos vários ramos criados por um desvio condicional.
- **Final:** estes vértices representam o final da execução do programa e, são caracterizados por não possuírem descendentes, ou seja, são terminais.

Um vértice pode pertencer a 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 3.1. A única restrição é que apenas exista um vértice do tipo inicial, quer este tenha um único ou vários descendentes (vértice de DECISÃO).

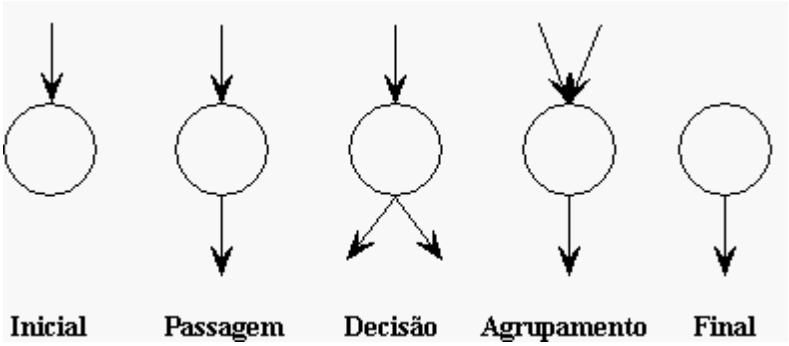


Figura 3.3:Tipos básicos de vértices

Levando-se em consideração as características dinâmicas da execução de um programa, tem-se uma outra classificação dos vértices:

- **Vértices de Execução**: indicam que o tempo de execução para o vértice não depende de fatores externos ao processador; podem ser de qualquer dos tipos listados anteriormente;
- **Vértices de Sincronismo**: indicam que a passagem para o vértice seguinte ocorrerá somente após o atendimento das restrições de sincronismo para o vértice em análise. O tempo consumido no vértice depende do instante em que essas restrições forem atendidas;
- **Vértices de Comunicação**: indicam que a passagem para o vértice seguinte depende do processo de comunicação definido no vértice. Desse modo, o tempo nele consumido dependerá da disponibilidade dos canais de comunicação e do tamanho do que estiver sendo transferido.

3.2.2. Reduções no grafo de execução

Como já mencionado, sendo os vértices do grafo de execução os pontos de controle para a simulação, a otimização é conseguida reduzindo-se o número de vértices do mesmo. No entanto, é preciso considerar que na simulação, menos vértices implicam resultados mais rápidos, porém menos precisos.

Neste caso, pode-se ter vários níveis de otimização, desde a máxima redução no gráfico, obtendo-se resultados menos precisos, passando por níveis intermediários de redução, até um nível mínimo de redução, tendo resultados mais precisos possíveis. O nível de otimização seria então definido segundo os interesses finais do usuário.

A seguir, serão apresentadas as técnicas de otimização, baseadas nas otimizações de código apresentadas no capítulo anterior.

Consta-se que a propagação de código, eliminação de código morto ou imprestável e eliminação de código redundante são as principais técnicas usadas. Entretanto, estas técnicas só podem ser aplicadas para um determinado conjunto de vértices como se observará adiante.

A. Aglutinação de vértices passagem

Um vértice passagem pode ser incorporado ao vértice destino de sua aresta emergente. Esta operação é válida desde o vértice não seja um vértice de sincronismo. A exceção ocorre

quando se desejar um nível de redução máxima.

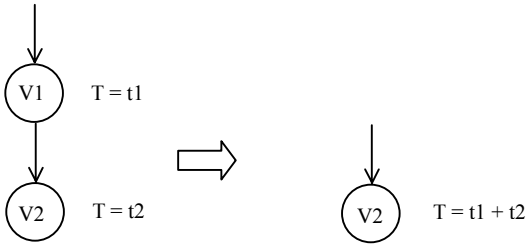
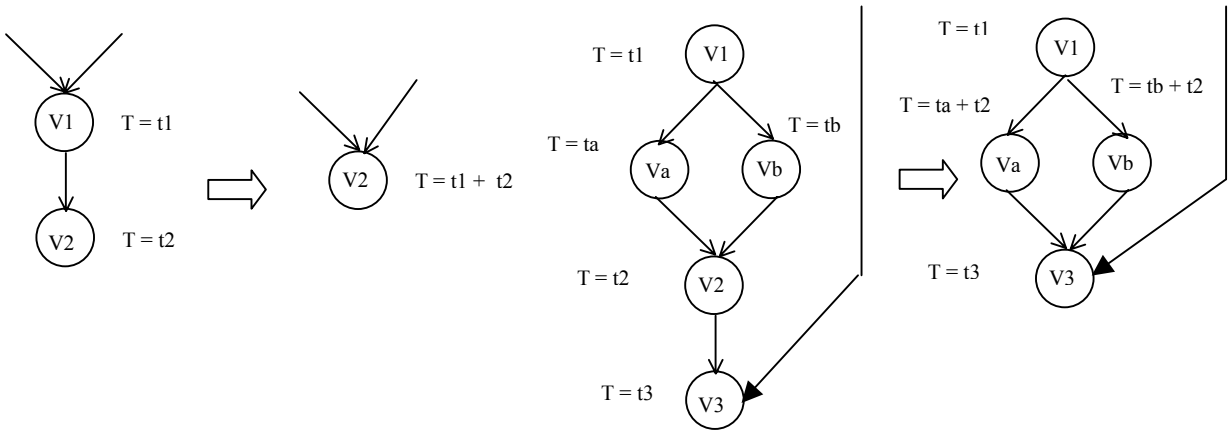


Figura 3.4: Aglutinação de vértices PASSAGEM

Tem-se a figura 3.4. para exemplificar a operação. Neste tipo de aglutinação de vértices, o tempo que seria consumido no vértice *V1* é acrescido ao tempo gasto no vértice *V2*. Podendo-se eliminar então o vértice *V1*, fazendo-se com que a aresta que nele incidia passe a incidir no vértice *V2*. No final, o grafo possui agora um ponto de controle a menos.

B. Aglutinação de vértices Agrupamento

Um vértice agrupamento pode ser incorporado ao vértice destino da aresta que parte dele, como mostra a figura 3.5.a., sendo que a operação de aglutinação é feita como na aglutinação de vértices passagem. A restrição a esta operação ocorre quando o vértice destino possui outras arestas incidentes.



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

Figura 3.5.: Aglutinação de vértices AGRUPAMENTO

Porém, mesmo no caso da existência de mais de uma aresta no vértice destino, ainda é possível realizar a redução do grafo. Na figura 3.5.b, temos o vértice agrupamento *V2* finalizando as arestas que partem do vértice de decisão *V1* e o vértice destino *V3* com arestas vindo de pontos distintos dos caminhos entre *V1* e *V2*.

A operação de aglutinação neste caso é feita removendo o vértice *V2*, passando-se as informações para seus antecessores, *Va* e *Vb*. Os antecessores apontam agora para o sucessor de *V2*, e têm seu tempo de execução acrescido do tempo que seria consumido por ele. Isto é, os tempos consumidos em cada um dos caminhos entre *V1* e *V2* continuam a ser os mesmos calculados antes de se realizar a redução.

Como dito anteriormente, isto apenas é possível no caso das arestas que incidem em *V3* venham de fora dos caminhos entre *V1* e *V2*.

C. Aglutinação de vértices comuns em ramos distintos

Nesta operação, o que se busca é a diminuição do número de vértices dentro dos vários ramos que partem de um vértice decisão. A restrição existe quando os vértices iniciais de cada ramo são pontos de sincronismo ou comunicação.

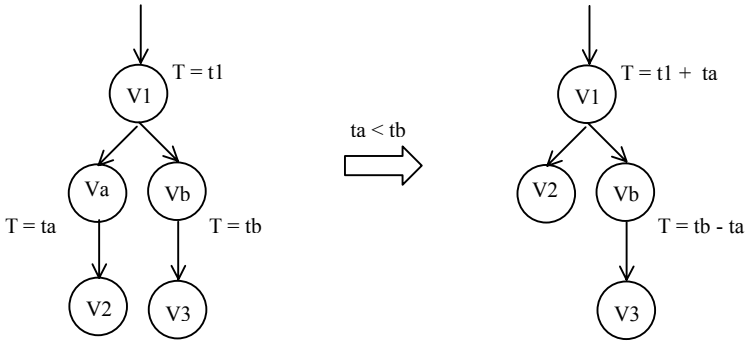


Figura 3.6: Aglutinação de vértices comuns

Com o auxílio da figura 3.6, verifica-se os procedimentos desta redução. Verificada a não existência de pontos de sincronismo entre os sucessores de *V1*, procura-se qual deles possui o menor tempo de execução. Este vértice é eliminado, na figura esse vértice é representado por *Va*, fazendo-se com que a aresta que nele incidia passe a incidir no vértice *V2*. No caso de existência de outros vértices com tempo de execução igual a ta, os mesmo podem ser eliminados. O tempo de execução *ta* é acrescido ao tempo de *V1* e decrescido dos tempos em cada um dos outros caminhos.

3.3. Preparações para a implementação

O ponto de partida para a aplicação das reduções no grafo é o arquivo de saída produzido pela função de geração do grafo de execução do programa. Este arquivo, em verdade uma tabela *hash*, que contém todas as funções do programa está estruturado da seguinte forma: para cada uma das funções do programa é criada um grafo, com informações sobre os vértices pertencentes a este grafo. A estrutura dos vértices dos grafos é composta por informações como o tipo de vértice, endereço inicial e final do vértice, número de ciclos de máquina consumido pelas instruções do vértice, dados sobre o ponto de sincronismo ou comunicação e apontadores para os vértices sucessores.

Como não se obteve a documentação formal sobre a representação dos tipos de vértices apresentados, e na estrutura utilizada até então não estava previsto a determinação de quantos arestas apontam para um determinado vértice, procederam-se as seguintes ações:

1. Especificação arbitrária da representação dos tipos de vértices, para que se possa manipulá-los de forma adequada na aplicação das técnicas de redução;
2. Determinação dos vértices agrupamento, atribuindo a informação de quantas arestas incidem neste vértice.

A classificação utilizada para os possíveis tipos de vértices de um grafo é dada abaixo:

0	→	passagem
1	→	decisão
2	→	agrupamento
3	→	chamada de função
4	→	comunicação
5	→	sincronismo
6	→	retorno de função

Figura 3.7.: Tipos de vértices do grafo

Inicialmente a função *GetTrees* gera um arquivo para cada grafo encontrado. A aplicação da função *GroupingVtx* determina os possíveis vértices agrupamento existentes e seta o campo de que quantifica o número de vértices que apontam para cada vértice do grafo. Esta

informação que pode é obtida verificando quantas vezes o endereço inicial de um dado vértice é citado nos endereços de destino do vértices restantes. Na figura 3.8. abaixo, tem-se a representação proposta de um dado vértice.

Endereço inicial do vértice	Tipo do vértice	Número de arestas que chegam ao vértice	Tempo de execução	Endereço para vértice sucessor 1	Endereço para vértice sucessor 2	Endereço para vértice sucessor n

Figura 3.8.:Representação de uma vértice do grafo

Tem-se então uma tabela *hash* indexada pelo endereço origem de cada vértice, de forma similar ao arquivo entrada inicial. Esta tabela será o ponto de partida para a aplicação das reduções propostas em 3.2.2.

3.4. Redução de vértices decisão

Tendo-se então as técnicas de redução do grafo e considerando as características dos vértices envolvidos em cada uma delas, conclui-se que a ordem de aplicação das técnicas é um fator importante. Determina-se que a ordem a ser aplicada é: decisão → agrupamento → passagem. Deste modo, não existe o risco de se invalidar a aplicação das demais reduções, já que vértices passagem são consumidos em todas as reduções.

Como citado anteriormente, o escopo de aplicação das reduções são os arquivos gerados pela função *GetTrees*.

A. Redução de vértices decisão

O que se faz na implementação da redução de vértices decisão é vasculhar a tabela *hash* de um grafo por vértices de decisão. Achado o vértice, recupera-se os endereços dos vértices por este apontado, se verifica o caso desses vértices não serem de sincronismo ou comunicação, determina-se o vértice que apresenta o menor tempo de execução. Este será excluído do grafo.

Para se deletar este vértice são realizadas algum passos: atualização do endereço de destino do vértice decisão, atualização dos tempos de execução do vértice destino e de todos os

outros imediatamente sucessores a este. Repete-se a análise para o mesmo vértice.

Analisa-se todos os vértices restantes do grafo buscando por novas possibilidades de aplicação desta redução, até que todos os vértices tenham sido examinados.

Neste trabalho observou-se que no caso do vértice escolhido para deleção for um vértice agrupamento, decide-se por não deletá-lo. Cria-se aqui então, uma nova restrição.

B. Redução de vértices agrupamento

Na redução de vértices agrupamento, o que se faz é semelhante ao da redução anterior. Busca-se por vértices agrupamento, examinando-se o vértice destino do mesmo. atentando para as restrições apresentadas em 3.2.2.B. No caso de não existir outros vértices apontando para este vértice destino, o vértice agrupamento é deletado e se faz a atualização do tempo de execução do vértice destino. Os vértices que incidiam sobre o vértice deletado passam a incidir sobre o vértice que era seu destino, isto é, atualiza-se os campos destino dos vértices envolvidos.

Como citado na especificação desta redução, a exceção ocorre das demais arestas que apontam para este vértice destino não venham do fluxo entre o vértice decisão (imediatamente anterior ao de agrupamento) e o vértice destino. Isto é feito, comparando todos os endereços envolvidos no fluxo. Verifica-se a existência de outros vértices agrupamento passíveis de redução.

C. Redução de vértices passagem

Esta técnica é a que apresenta aplicação mais simples. Busca-se o vértices de passagem que por ventura não tenham sido deletados nas reduções anteriores. Achado o vértice passagem, se verifica a condição deste ser de comunicação ou sincronismo e se o usuário deseja não considerar esta restrição (nível de otimização máxima). Em caso afirmativo de deleção, atualiza-se o tempo do vértice destino e o endereço destino do vértice anterior ao de passagem.

Semelhante as outras reduções, a busca no grafo continua até que todos os vértices tenham sido analisados. Todas os arquivos gerados pela função *GetTrees* após serem submetidos a etapa de otimização, no final serão concatenados, apresentando a mesma estrutura do arquivo de entrada inicial. Porém agora, com uma redução do número de vértices, o que implica resultados mais rápidos na etapa de simulação.

3.6. Considerações Finais

No decorrer deste capítulo foram apresentadas as etapas do desenvolvimento do projeto, incluindo a metodologia proposta assim como alterações necessárias para que fosse possível a aplicabilidade da metodologia. No capítulo seguinte, tem-se informações da execução de testes, avaliação dos resultado.

Capítulo 4

Testes e resultados

Considerações Iniciais

Neste capítulo serão descritos os testes realizados para o módulo implementado a partir das definições feitas no capítulo anterior. A máquina utilizada foi uma *Sun UltraSPARC II*, com sistema operacional *Solaris* versão 9, disponível na Coordenadoria Geral de Bibliotecas, Reitoria- UNESP.

4.1. Fase de testes

Como exemplo, será utilizado uma tabela *hash*, representado na figura 4.1, para ilustrar o funcionamento do módulo de otimização do grafo de execução. Como citado no capítulo anterior, esta tabela *hash* foi produzida pelo módulo de geração do grafo. Os vértices deste grafo apresentam as informações: tipo de vértice, endereço inicial, tempo de execução e endereços destino.

```
...
00017124 set_strings()
00017280 show_pages
...
Tree Number 10
vtx 0 00017124 5    00017138
vtx 0 00017138 106  00017154
vtx 1 00017154 125  000171a4 000171c8
vtx 0 000171a4 3    000171b4
vtx 0 000171b4 104  000171f4
vtx 0 000171c8 4    000171d8
vtx 0 000171d8 106  000171f4
vtx 0 000171f4 125  00017244
vtx 0 00017244 3    00017254
vtx 6 00017254 104  return
Tree Number 11
vtx 1 00017280 5 00017294 00017650
.
.
.
```

Figura 4.1: Arquivo contendo as árvores do programa

Na seção 3.3. foram apresentadas as etapas iniciais para a fase de testes: codificação dos tipos de vértices encontrados na tabela *hash* citada acima e, a criação de tabelas para cada uma das árvores de grafos encontradas na tabela inicial . Estes procedimentos são realizados pela função *GetTrees*.

Numa segunda fase, tem-se então várias tabelas, uma para cada árvore encontrada, que serão analisadas pela função *GroupingVtx*, para identificação dos vértices agrupamento.

O resultado da análise de um dos arquivos é apresentado na figura 4.2 , a tabela passa a ter uma nova coluna, que na figura aparece em negrito, quantificando o número de arestas que apontam para o dado vértice. Para melhor orientação nas etapas de testes, foi incluída uma primeira coluna com o número do vértice.

```
Tree Number 10
01 vtx 0 0 00017124 5    00017138
02 vtx 0 1 00017138 106  00017154
03 vtx 1 1 00017154 125  000171a4 000171c8
04 vtx 0 1 000171a4 3    000171b4
05 vtx 0 1 000171b4 104  000171f4
06 vtx 0 1 000171c8 4    000171d8
07 vtx 0 1 000171d8 106  000171f4
08 vtx 2 2 000171f4 125  00017244
09 vtx 0 1 00017244 3    00017254
10 vtx 6 1 00017254 104  return
```

Figura 4.2: Arquivo contendo os vértices de uma árvore

Tendo agora a árvore com os vértices na estrutura proposta em 3.2.3, pode-se dar início a aplicação das técnicas de redução. Como determinado em 3.4, seguiremos a ordem de aplicação: redução de vértices decisão → redução de vértices agrupamento → redução de vértices passagem.

A função *ReduceVtx* abre o arquivo para a análise, analisando o campo tipo de vértice, até encontrar um vértice decisão. Nesse exemplo, é o vértice na linha 3 , verifica-se os campos de tipo do vértice dos vértices por este apontados, como nenhum é de sincronismo ou comunicação, compara-se os tempos de execução e tem-se que o vértice da linha 4 será excluído. Atualiza-se os tempos de execução do vértice decisão e do vértice da linha 6, assim como o vértice agrupamento passa a apontar para o vértice da linha 5. Abaixo, tem-se o resultado obtido dessa redução, representado na figura 4.3:

```
Tree Number 10
01 vtx 0 0 00017124 5    00017138
02 vtx 0 1 00017138 106  00017154
03 vtx 1 1 00017154 128  000171b4 000171c8
05 vtx 0 1 000171b4 104  000171f4
06 vtx 0 1 000171c8 1    000171d8
07 vtx 0 1 000171d8 106  000171f4
08 vtx 2 2 000171f4 125  00017244
09 vtx 0 1 00017244 3    00017254
10 vtx 0 1 00017254 104  return
```

Figura 4.3: Aplicação da redução de vértices decisão

Seria possível deletar ainda mais vértices pela aplicação dessa redução, porém para não se eliminar a possibilidade de exemplificação da redução de vértices agrupamento no mesmo arquivo, decidiu-se parar após a deleção de somente mais um vértice. Na figura abaixo, tem-se a saída:

```
Tree Number 10
01 vtx 0 0 00017124 5    00017138
02 vtx 0 1 00017138 106  00017154
03 vtx 1 1 00017154 129  000171b4 000171d8
05 vtx 0 1 000171b4 103  000171f4
07 vtx 0 1 000171d8 106  000171f4
08 vtx 2 2 000171f4 125  00017244
09 vtx 0 1 00017244 3    00017254
10 vtx 0 1 00017254 104  return
```

Figura 4.4: Nova aplicação da redução de vértices decisão

Passa-se então para a busca por vértices agrupamento, representado pelo vértice da linha 8. Obtêm-se o vértice por ele apontado, e verifica-se o campo arestas que chegam, como o valor é 1 (só é apontado pelo vértice da linha 8). Decide-se por eliminar o vértice agrupamento, atualizando-se endereços e os tempos de execução, como mostrado abaixo. E por sua vez, o vértice da linha 9 passa a ser um vértice de agrupamento, como pode-se ver na figura 4.5.

```
Tree Number 10
01 vtx 0 0 00017124 5    00017138
02 vtx 0 1 00017138 106  00017154
03 vtx 1 1 00017154 129  000171b4 000171d8
05 vtx 0 1 000171b4 103  00017244
07 vtx 0 1 000171d8 106  00017244
09 vtx 2 2 00017244 128  00017254
10 vtx 0 1 00017254 104  return
```

Figura 4.5: Aplicação da redução de vértice agrupamento

Como na redução anterior, se poderia aplicar novamente esta redução, concluindo com a eliminação também do vértice da linha 9:

```
Tree Number 1
01 vtx 0 0 00017124 5    00017138
02 vtx 0 1 00017138 106  00017154
03 vtx 1 1 00017154 129  000171b4 000171d8
05 vtx 0 1 000171b4 103  00017254
07 vtx 0 1 000171d8 106  00017254
10 vtx 2 2 00017254 104  return
```

Figura 4.6: Nova aplicação da redução de vértice agrupamento

```
Tree Number 10
03 vtx 1 1 00017154 249  000171b4 000171d8
05 vtx 0 1 000171b4 103  00017254
07 vtx 0 1 000171d8 106  00017254
10 vtx 2 2 00017254 104  return
```

Figura 4.7: Árvore final com os vértices restantes

Numa última fase tem-se a aplicação da redução dos vértices passagem que por ventura não tenham sido consumidos nas reduções anteriores. Com as sucessivas aplicações dessa redução tem-se a árvore no arquivo dado na figura 4.7.

Ao final tem-se uma árvore com somente quatro pontos de controle a serem passados ao simulador.

4.2. Tempo de processamento

Após as exemplificações dadas na seção anterior, foi submetida ao módulo implementado, uma tabela *hash*, contendo quarenta e quatro grafos, totalizando mil e quinhentos vértices. Com análise desse processamento, pode-se ter uma melhor dimensão do poder de otimização do módulo proposto.

O tempo consumido na execução nas fases de preparação do arquivo de entrada e na otimização propriamente dita dos grafos foi de aproximadamente cinco minutos. Provavelmente este tempo poderia baixar caso a fase da nova codificação dos vértices não tivesse que ser realizada, isto é, que o arquivo de entrada estivesse corretamente codificado e já com a informação de quantas arestas apontam para cada vértice.

O que se observa é que o tempo consumido na execução de uma árvore de grafos deste tamanho pode ser considerado pequeno frente ao possível ganho de velocidade que se poderá conseguir na fase simulação, afirmando-se então a viabilidade do módulo de otimização.

Capítulo 5

Conclusões e Perspectivas

Neste último capítulo procura-se apresentar de forma sintética as conclusões após a realização do projeto. Assim, têm-se primeiramente as impressões quanto a viabilidade da implementação proposta, seguidas pelas considerações sobre a aplicabilidade de otimizações severas. Finalmente serão indicadas as perspectivas para trabalhos futuros que possam utilizar as contribuições aqui fornecidas.

5.1. Conclusões

Do que foi exposto ao longo dos capítulos 3 e 4, podem ser retiradas as seguintes conclusões.

- Considerando as otimizações propostas e implementadas nos capítulos anteriores, vemos que as mesmas permitem uma boa redução no tamanho do grafo. Nos dados apresentados no capítulo anterior, verificou-se um caso onde um grafo que inicialmente apresentava dez vértices termina a fase de otimização somente com quatro vértices;
- Otimizações mais severas, como por exemplo otimizações que não respeitassem as restrições impostas nas reduções apresentadas, implicariam em uma maior redução do número de vértices do grafo resultando em uma simulação mais rápida. No entanto, é preciso que isso implicaria na perda de precisão, o que não era de interesse desse trabalho.
- Na última seção do capítulo anterior verificou-se que o tempo consumido na execução do módulo não é tão grande, frente ao provável ganho de velocidade na fase de simulação, com a redução obtida do grafo.

5.2. Perspectivas

Durante o trabalho desenvolvimento desse trabalho foram observadas considerações a serem feitas em trabalhos futuros.

- Investigação do impacto de otimizações severas, para que se possa obter dados reais sobre a perda de precisão. Considerando que esta perda não seja tão expressiva, se teria uma ganho maior de velocidade mantendo-se ainda uma boa precisão.
- Seria interessante também o desenvolvimento de interfaces para o controle das otimizações no grafo. Deste modo, o usuário poderia interagir, habilitando as diferentes reduções e suas restrições, podendo-se ter vários níveis de otimização, desde a máxima redução no grafo, obtendo-se resultados menos precisos, passando por níveis intermediários de redução, até uma redução mínima, obtendo resultados mais precisos possíveis.

Referências Bibliográficas

1. AHO, A. V.; SETHI, R.; ULLMAN, J.D., **Compilers, Principles, Tecniques, and Tools**, Addison-Wesley, 1988.

2. AHO, A. V.; ULLMAN J.D., **The Theory of Parsing, Translation and Compiling**, Vol. 02, Prentice Hall, 1973.

3. ALLEN, E. F ; COCKE F., **A Catalogue of optimizing Transformations**, anais do 5th Courant Computer Science Symposium, em Randall Rustin(editor),p. 1-30, 1972, New York, USA.

4. BACON, D. F.; GRAHAM, S. L.; SHARP, O. J., **Compiler Transformations for High-Performance Computing,** disponível em http://www.cs.berkeley.edu/~yelick/titanium/papers/bacon.ps Acesso em 01 ago 2003.

5. GOLDBERG, C.P., **A comparison of Certain Optimization Techniques**, anais do 5th Courant Computer Science Symposium, em Randall Rustin(editor), p. 31-50, 1972, New York, USA.

6.HWANG, Kai., **Advanced Computer Architecture: Parallelism, Scalability, Programmability**, McGraw-Hill, 1993.

7. MANACERO JR., A., **Predição do Desempenho de Programas Paralelos por Simulação do Grafo de Execução**, Tese de Doutorado submetida a Faculdade de Engenharia Elétrica e Computação, Universidade Estadual de Campinas, em, Campinas-SP, 1997.

8. MORAES, M. D., **Gerador do Grafo de Execução para Análise de Desempenho de Sistemas Paralelos**, Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Instituto de Biociências, Letras e Ciências Exatas, Universidade Estadual Paulista, São José do Rio Preto - SP, 1999.

9. SARKAR, V., **Parallel Programs and their Classification**, anais do 6th International Workshop, LCPC'93, p.633-655, em Agosto de 1993, Portland, Oregan, EUA.

10. TATSUMI, E. S., **Simulador de grafos de Execução para Análise de Desempenho de Programas Paralelos**, Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Instituto de Biociências, Letras e Ciências Exatas, Universidade Estadual Paulista, São José do Rio Preto, 2000.