COPPE/UFRJ

ESCALONAMENTO DISTRIBUÍDO DE LINHAS DE EXECUÇÃO PARALELAS ATRAVÉS DE REVERSÃO DE ARESTAS COM HIBERNAÇÃO

Carlos Augusto de Castro

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Priscila Machado Vieira Lima

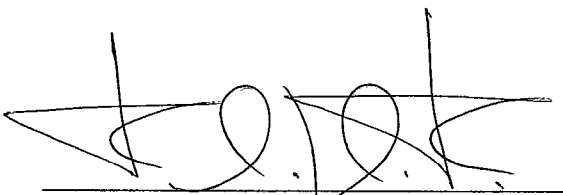
Rio de Janeiro
Março de 2009

ESCALONAMENTO DISTRIBUÍDO DE LINHAS DE EXECUÇÃO PARALELAS ATRAVÉS DE REVERSÃO DE ARESTAS COM HIBERNAÇÃO

Carlos Augusto de Castro

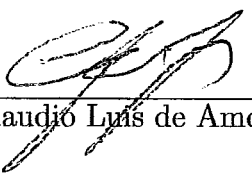
DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:



Prof. Felipe Maia Galvão França, Ph.D.

Prof. Priscila Machado Vieira Lima, Ph.D.



Prof. Claudio Luis de Amorim, Ph.D.

Dr. Geraldo Gil Veiga, Docteur

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2009

Castro, Carlos Augusto de

Escalonamento Distribuído de Linhas de Execução Paralelas Através de Reversão de Arestas com Hibernação/ Carlos Augusto de Castro. - Rio de Janeiro: UFRJ/COPPE, 2009.

XIV, 113 p.: il.; 29,7 cm

Orientadores: Felipe Maia Galvão França

Priscila Machado Vieira Lima

Dissertação (mestrado) UFRJ/ COPPE/ Programa de Engenharia de Sistemas e Computação, 2009

Referencias Bibliográficas: p. 84-90

1. Processadores múltiplos 2. Processamento especulativo 3. Paralelismo. I. França, Felipe Maia Galvão et al. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistema e Computação. III Título.

À minha esposa, pelas horas em que a deixei só.

Agradeço à UFRJ, ao IME, ao SERPRO, aos meus professores e à minha família

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

ESCALONAMENTO DISTRIBUÍDO DE LINHAS DE EXECUÇÃO PARALELAS ATRAVÉS DE REVERSÃO DE ARESTAS COM HIBERNAÇÃO

Carlos Augusto de Castro

Março/2009

Orientadores: Felipe Maia Galvão França

Priscila Machado Vieira Lima

Programa: Engenharia de Sistemas e Computação

Os recentes avanços da tecnologia, na direção de arquiteturas com múltiplos núcleos, força a mudança de paradigma de desenvolvimento de aplicações do modelo seqüencial para o de múltiplas linhas de execução (*threads*). O trabalho apresentado, faz uso do *SER - Scheduling by Edge Reversal*, algoritmo consistente e maduro, visando a migração de programas codificados para um modelo de execução sequencial, para uma eficiente execução no novo modelo. Isto é realizado através da substituição dos comandos seqüenciais *FOR* e *WHILE* por seus equivalentes paralelos, com mínimo esforço. Tais mecanismos foram modelados, a partir do *SER*, como fundamentação teórica, e dos conceitos de *POSIX Threads*, técnicas de escalonamento e de paralelização. O algoritmo obtido, *Gen_For* possui escalabilidade e facilidade na conversão de programas. Primeiramente o *Gen_For* foi utilizado em uma aplicação sintética, para aferimento de escalabilidade e aceleração, obtendo-se bons resultados. Em seguida, o *Gen_For* foi aplicado a produtos de matrizes esparsas. Foram antecipadas, como evolução futura, a paralelização de múltiplos laços, execução especulativa, balanceamento de carga em função da taxa de utilização dos núcleos e canalização por software.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DISTRIBUTED SCHEDULING OF PARALLEL THREADS BY EDGE REVERSAL WITH HIBERNATION

Carlos Augusto de Castro

March/2009

Advisors: Felipe Maia Galvão França

Priscila Machado Vieira Lima

Department: Computing and Systems Engineering

Recent technology trends towards multicore architectures forces the application development paradigm to change from the sequential model to a new, multithreaded one. This work is based in *SER - Scheduling by Edge Reversal*, a solid and stable algorithm, used as a substratum to convert programs coded in a sequential execution model to a safe execution in a multithreaded environment, by converting the sequential statements *FOR* and *WHILE* to their parallel equivalents *Gen_For* and *Gen_While*, with the slightest possible effort. *Gen_For* was shaped from *SER*, as a theoretical groundwork, with *POSIX* Threads concepts, scheduling and parallel execution techniques, as technological background. The derived algorithm *Gen_For* has a nice scalability and a friendly program conversion scheme. Good results were obtained as *Gen_For* was used in a generic application in order to measure its scalability and speedup. Besides, *Gen_For* was applied to sparse matrix product computations. While modeling *Gen_For*, parallel multi-loop, speculative execution, concurrency scheduling as a factor of processor usage and loop software pipelining were devised as future development.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

1.1 Motivação

Os recentes avanços da tecnologia, na direção das arquiteturas com múltiplos núcleos (*multicores*), força a mudança de paradigma de desenvolvimento de aplicações do modelo seqüencial para o com múltiplas linhas (*threads*) [41]. Considerando todo o acervo de software desenvolvido para o modelo atual (seqüencial) até hoje, o custo e a dificuldade para migrá-lo para o novo modelo (múltiplas linhas), é mais eficiente a substituição da infraestrutura de desenvolvimento e execução por uma nova, já adaptada para o novo modelo, onde o esfôrço do programador seja mínimo [11].

Uma outra visão é exposta em [8] onde os autores consideram que os desenvolvedores de sistemas operacionais, bancos de dados e máquinas virtuais teriam que se envolver com os detalhes de escrever programas paralelos Para todos os demais o desafio seria apenas de usar da melhor maneira possível os componentes para conseguir um sistema escalável.

Conforme Sangani, "Processadores com múltiplos núcleos são ótimos para fazer muitas coisas simultaneamente, mas que tal fazer uma única, mais depressa?" [41]

1.2 Objetivos e contribuições deste trabalho

A principal contribuição deste trabalho é oferecer um modelo de sincronização distribuído de linhas de execução, utilizando o algoritmo SERh - Escalonamento por Reversão de Arestas com hibernação (*Scheduling by Edge Reversal with*

hibernation) [9], contrastando com o *OpenMP - Open Multi-Processing* [34], cujo mecanismo de controle de sincronização é centralizado.

Através do padrão *POSIX*, o sistema operacional UNIX nos permite a execução de múltiplas linhas de execução dentro de um mesmo espaço de endereçamento. O padrão também nos oferece, dentre outros, comandos para iniciar, informar e aguardar o término de execução das linhas criadas, mas permanece a necessidade de análise e codificação da sincronização entre as múltiplas linhas criadas. A contribuição mais nuclear deste trabalho é a utilização do algoritmo SERh como modelo de sincronização distribuída de linhas de execução.

Está além do escopo deste trabalho a demonstração de que tal modelagem é sempre possível.

Como objetivo específico, este trabalho visa a migração de programas codificados para um modelo de execução seqüencial para uma eficiente e segura execução em um ambiente com múltiplas linhas. Através da substituição de comandos seqüenciais *FOR* e *WHILE* para seus equivalentes paralelos, transformando-os em linhas de execução, sincronizadas pelo SERh, desenvolvido ao longo do trabalho. Isto já foi estudado por [7], [36] e [33], dentre outros, mas normalmente sob aspectos específicos.

1.3 Estrutura do trabalho

O trabalho está baseado nas estruturas dos algorítimos distribuídos SER - Escalonamento por Reversão de Arestas (*Scheduling by Edge Reversal*) [3] e SERh - Escalonamento por Reversão de Arestas com Hibernação (*Scheduling by Edge Reversal with hibernation*) [9]. Na plataforma computacional *SMP -Symmetric multiprocessing* [19], utilizando a biblioteca padronizada *Pthreads* [23], comumente utilizada nos sistemas *UNIX*. Em esquemas de escalonamento usuais e

algumas ideias coletadas nas pesquisas sobre execução especulativa. Criaremos uma estrutura visando a conversão, com o menor esfôrço possível, de programas escritos na linguagem *C++* para tirar proveito de um processador com múltiplos núcleos.

O capítulo 2 apresenta a fundamentação teórica, sendo seguida, no capítulo 3, dos conceitos tecnológicos utilizados. No capítulo 4 é descrito o que foi desenvolvido e as limitações identificadas. Seus resultados são apresentados no capítulo 5 e sua conclusão no 6.

2 Fundamentação teórica

2.1 Escalonamento por reversão de arestas - *SER*

O escalonamento por reversão de arestas [3] é uma solução potencialmente ótima para o problema dos filósofos jantando de Dijkstra sob operação em alta carga, isto é, onde os processos continuamente requerem acesso à recursos compartilhados (existindo apenas dois estados possíveis, *READY* ou *WAIT*). O *SER* foi utilizado em aplicações distribuídas tais como circuitos digitais assíncronos [10] e simulações de redes neurais artificiais [4].

O *SER* trabalha sobre um grafo direcionado acíclico, no qual existe pelo menos um nó tendo todas as arestas incidentes, chamado *SUMIDOURO* (nó i em A na Figura 1). Existe também pelo menos um nó com todas as suas arestas direcionadas aos seus vizinhos, chamado *FONTE* (nós j e m em A na Figura 1). Sumidouros podem operar e reverter suas arestas para seus vizinhos, gerando uma nova orientação acíclica (em B na Figura 1), com novos nós *SUMIDOUROS* (nós l e k em B na Figura 1), que podem operar em seqüência. A aciclicidade do grafo é mantida porque as únicas arestas modificadas são as que pertencem à novas *FONTES* (que previamente eram *SUMIDOUROS*) e portanto, um ciclo não poderia ter sido criado através delas.

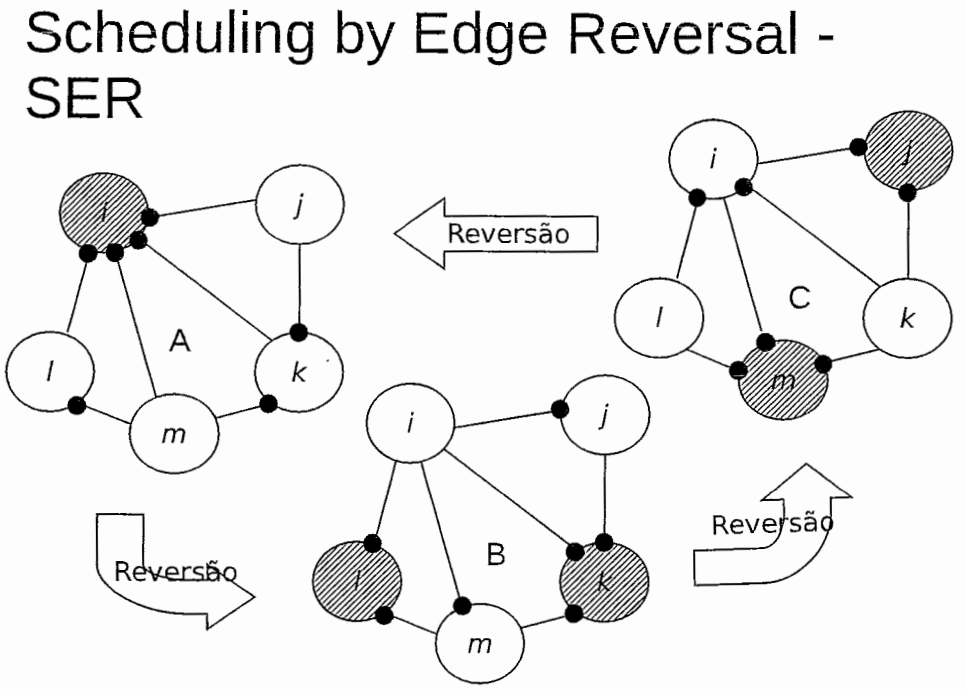


Figura 1: Dinâmica do algoritmo *SER*

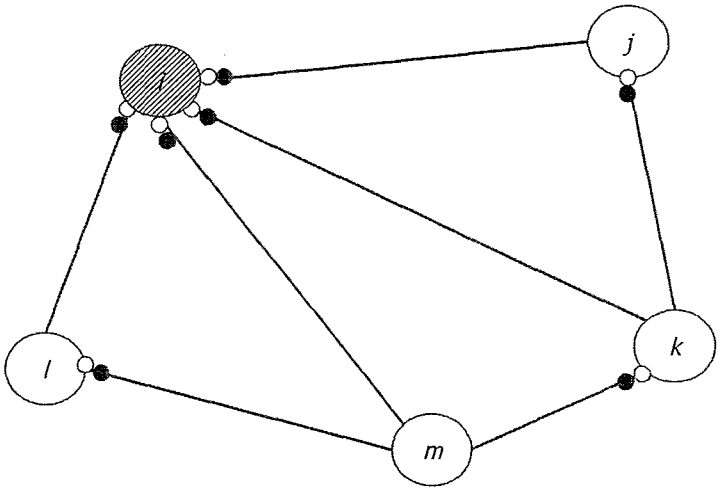


Figura 2: Estado de *READY* do nó i *SERh*

2.2 Escalonamento por reversão de arestas com hibernação - *SERh*

O Escalonamento por reversão de arestas com hibernação [9] é uma generalização do *SER*, descrito em 2.1, na qual os nós podem também estar em estado de hibernação - *HIBERNATION*, significando que um nó não requer acesso compartilhado aos recursos, além dos estados de *READY* para operar e *WAIT*. Estes três estados são modelados através da propriedade de dois diferentes tipos de *tokens*, *direito de reversão* (em preto nos grafos das Figuras 2 a 7) e *comunicado de não hibernação* (em branco nos grafos das Figuras 2 a 7).

Um nó é um *SUMIDOURO* (estado de *READY*) se ele possui o *token direito de reversão* em todas as suas arestas. Este é o caso do nó i da Figura 2.

Após a sua operação, ele entrará no estado de *WAIT* revertendo todos os *direitos de reversão* pelas as arestas nas quais ele possui também o *comunicado*

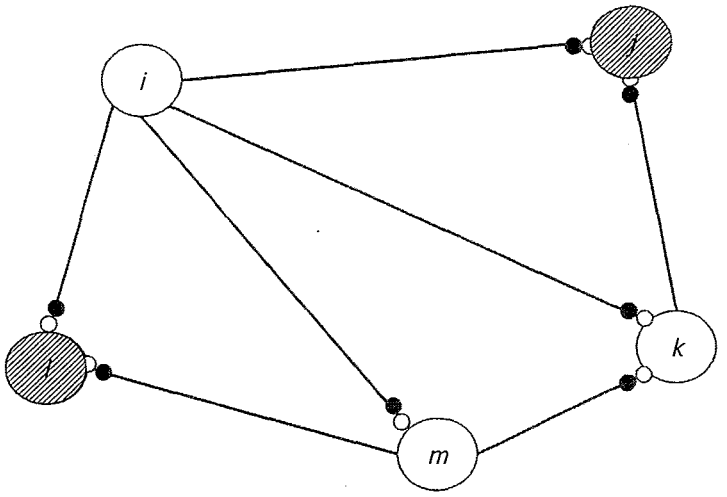


Figura 3: Estado de *WAIT* do nó i do *SERh*

de não hibernação (Figura 3).

Após a operação, para um nó entrar no estado *HIBERNATION*, ele enviará o token *direito de reversão* para todos os seus vizinhos, mas conservará o correspondente *comunicado de não hibernação* (Figuras 4 e 5).

Um nó só poderá deixar o estado de hibernação se receber de pelo menos um dos seus vizinhos um token direito de reversão e entrará no estado de *READY*, pelo envio de todos os tokens *comunicado de não hibernação* por ele mantidos, para todos os seus vizinhos (Figura 6 e 7).

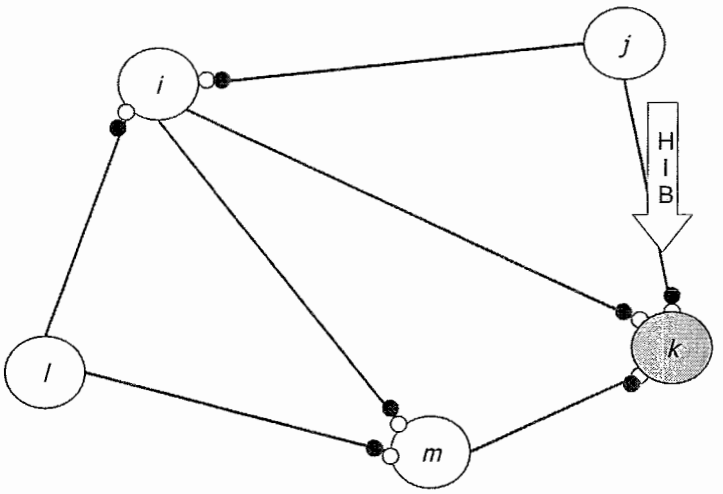


Figura 4: Estado de *HIBERNATION* do nó k (antes) *SERh*

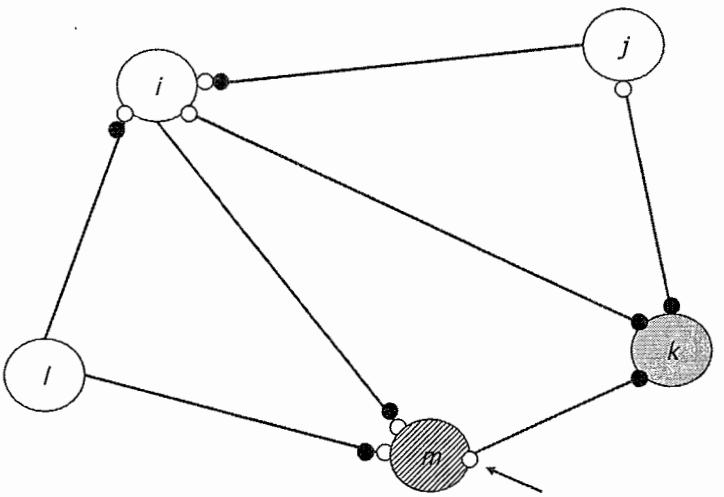


Figura 5: Estado de *HIBERNATION* do nó k (depois) *SERh*

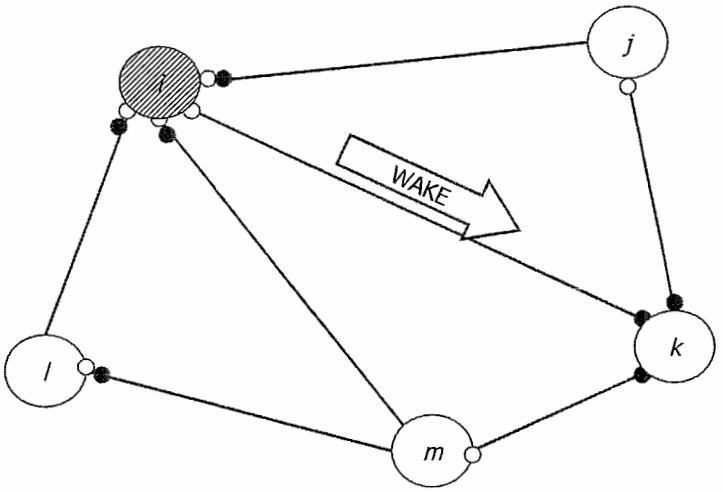


Figura 6: Nó k Retornando de *HIBERNATION* (antes) *SERh*

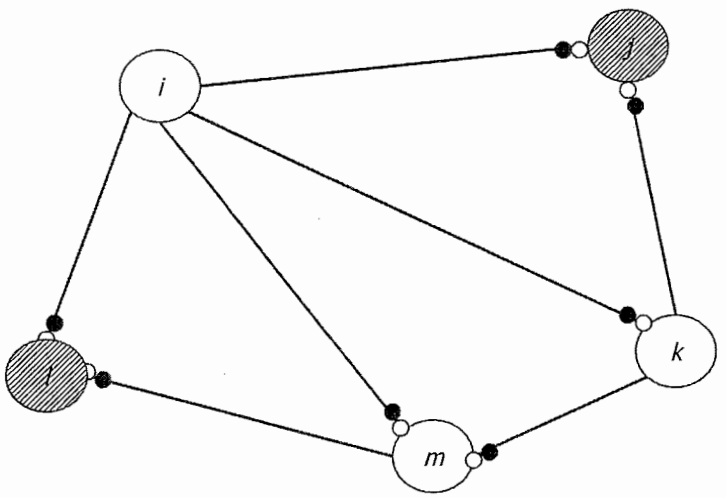


Figura 7: Nó k Retornando de *HIBERNATION* (depois) *SERh*

2.3 Multiprocessamento simétrico - *SMP*

A arquitetura SMP - Symetric (Shared Memory) Multiprocessors, conforme descrita por Patterson [19] permite que multiprocessadores com um número pequeno de processadores, compartilhem uma memória única e centralizada, interconectando os processadores à memória, por um barramento. Com grandes *caches*, o barramento e a memória, esta possivelmente com bancos múltiplos, podem satisfazer os requisitos de memória de um pequeno número de processadores. Substituindo o barramento único por múltiplos ou uma unidade de chaveamento, poderemos escalar esta arquitetura para algumas dúzias de processadores. A Figura 8 abaixo demonstra a arquitetura usada no trabalho.

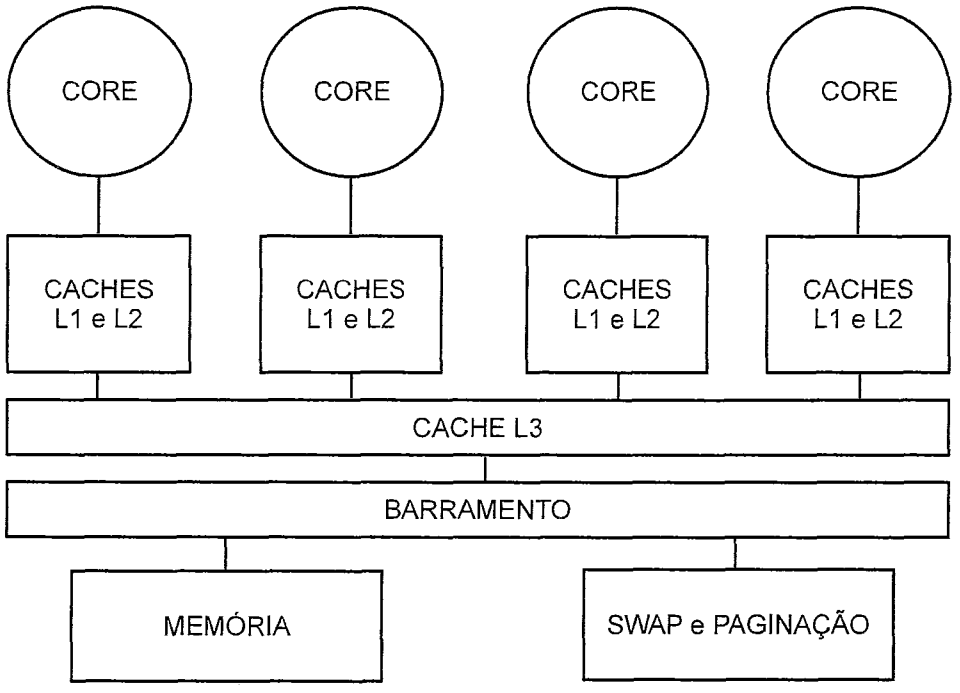


Figura 8: Arquitetura *SMP*

3 Conceitos tecnológicos

Os capítulos seguintes descrevem os conceitos que serão utilizados na modelagem dos algoritmos propostos para a solução do problema apresentado.

3.1 *POSIX THREADS*

POSIX (Portable Operating System Interface - IEEE 1003) THREADS [23], conforme conceituado em [20] é um padrão para linhas de execução, que define uma *API (Application Programming Interface)* padrão para criar e manipular estas linhas. As bibliotecas que implementam este padrão são chamadas *Pthreads*, sendo muito difundidas no universo *Unix* e outros sistemas operacionais semelhantes, como *Linux* e *Solaris*, daí o *X* ao final do acrônimo.

3.1.1 Conceito de linha de execução

Uma linha de execução (*thread*) é um fluxo de código executável dentro de um processo *UNIX*, que tem a possibilidade de ser escalonada para execução. Uma linha é mais simples para o sistema operacional criar, manter e gerenciar do que um processo por que muito pouca informação está associada a ela. Esta simplicidade sugere que uma linha tenha menor custo quando comparada com um processo. Todos os processos tem uma linha principal ou primária. A linha principal é o fluxo de controle do processo ou de execução de linhas. Um processo pode ter múltiplas linhas e, portanto, tantos fluxos de controle quantas são as linhas de execução. Cada linha executará independentemente e concorrentemente às demais, com a sua própria seqüência de instruções.

3.1.2 Requisitos de contexto de execução

Todas as linhas de execução dentro do mesmo processo existem no mesmo espaço de endereçamento. Todos os recursos pertencentes ao processo são compartilhados pelas linhas de execução. As linhas não tem a propriedade de nenhum recurso. Todos os recursos pertencentes ao processo são compartilhadas por todas as linhas do processo. Linhas compartilham descritores e ponteiros de arquivos, mas cada linha tem o seu ponteiro de programa, conjunto de registradores, estado e pilha. As pilhas das linhas estão dentro do segmento de pilha de seu processo.

O segmento de dados do processo é compartilhado pelas suas linhas de execução. Uma linha pode ler e escrever na memória do seu processo principal e este, por sua vez, também terá acesso aos dados. Quando o processo principal escreve na memória, qualquer de suas linhas filhas pode ter acesso aos dados. Linhas podem criar linhas dentro do mesmo processo. Linhas de execução também podem suspender, interromper e terminar outras linhas dentro do seu processo principal.

Linhas são entidades que executam e competem independentemente pelo uso do processador com outras linhas do mesmo ou de diferentes processos. Em um sistema com múltiplos processadores, linhas dentro de um mesmo processo podem executar simultaneamente em diferentes processadores. As linhas só podem executar nos processadores que forem designados para o processo principal. Se os processadores 1, 2 e 3 forem designados para o processo A, e o processo A tem três linhas, então cada linha será designada para um processador. Em um ambiente com um único processador, as diferentes linhas competem pelo uso deste processador.

A concorrência é obtida através da troca de contexto. Trocas de contexto ocorrem entre tarefas em um mono processador quando o sistema operacional é multitarefa. A multitarefa permite a mais de uma tarefa ser executada ao mesmo tempo em um único processador. Cada tarefa executa por um determinado intervalo de tempo. Quando este expira ou algum determinado evento ocorre, a tarefa é removida do processador e outra tarefa a ele é designada. Quando linhas estão sendo executadas concorrentemente dentro de um processo, então o processo é de múltiplas linhas. Cada linha executa uma sub-tarefa, permitindo a estas sub-tarefas do processo executar independentemente, sem relação com o fluxo principal do processo.

Com múltiplas linhas, as linhas podem competir pelo único processador ou serem designadas para processadores diferentes. De qualquer forma, uma troca de contexto ocorrendo entre linhas diferentes do mesmo processo requer menos recursos que uma troca de contexto ocorrendo entre linhas de diferentes processos. Um processo usa muitos recursos do sistema para manter o registro de suas informações e uma troca de contexto toma tempo para gerenciá-los. A maior parte da informação contida no contexto do processo descreve o espaço de endereçamento do processo e recursos a ele pertencentes.

Quando ocorre uma troca entre linhas de diferentes espaços de endereçamento, uma troca de contexto de processos deve ocorrer. Como linhas dentro do mesmo processo não possuem seu próprio espaço de endereçamento ou recursos, um menor registro de informações é requerido. O contexto de uma linha consiste apenas de uma identificação, uma pilha, um conjunto de registradores e uma prioridade. O conjunto de registradores contém o ponteiro de programa ou de instrução e

o ponteiro para a pilha. O segmento de texto de uma linha está contido no segmento de texto do seu processo. A troca de contexto de uma linha tomará menos tempo e usará menos recursos.

3.1.3 Comparação entre linhas e processos

Existem vários aspectos de uma linha de execução que são semelhantes a um processo. Ambos tem uma identificação, um conjunto de registradores, um estado, uma prioridade e estão submetidas à uma política de escalonamento. Como um processo, linhas tem atributos que a descrevem para o sistema operacional. Esta informação está contida em um bloco de informações, semelhante ao dos processos.

Os recursos abertos pelo processo (linha principal) são imediatamente acessíveis pelas linhas. Nenhuma preparação ou inicialização adicional é necessária. Linhas e processos filhos são entidades independentes do seu pai ou criador e competem pelo uso do processador. O criador do processo ou linha exerce algum controle sobre o processo criado. O criador pode cancelar, suspender, interromper ou trocar a prioridade. Uma linha ou processo pode alterar seus atributos e criar novos recursos, mas não pode acessar recursos pertencentes a outros processos.

A maior diferença é que cada processo tem o seu próprio espaço de endereçamento e as linhas não. Se um processo cria múltiplas linhas, todas elas estarão contidas no seu espaço de endereçamento. Processos filhos tem o seu próprio espaço de endereçamento e uma cópia do segmento de dados. Portanto, quando um processo filho altera suas variáveis ou dados, ele não afeta os dados do processo pai. Uma área de memória compartilhada tem que ser criada para

que processos pais e filhos possam trocar dados. Mecanismos de comunicação inter-processos, são usados para comunicar ou passar dados entre eles. Linhas do mesmo processo podem passar dados e se comunicar lendo e escrevendo diretamente em qualquer área que também é acessível pelo processo pai.

Algumas vantagens do uso de linhas de execução compreendem, menor uso de recursos do sistema para a troca de contexto, aumento no rendimento da execução da aplicação, nenhum mecanismo especial é requerido para comunicação entre linhas de execução e simplificação na estrutura do programa.

Podemos enumerar como desvantagens para esta utilização, o requerimento de sincronização para acesso concorrente à memória, poluição o espaço do processo e como existem dentro de um único processo, não podem ser reutilizadas por outros processos.

3.1.4 Funções da biblioteca *Pthreads*

Pthreads define um conjunto da linguagem C, implementado com um cabeçalho *pthread.h* e uma biblioteca *pthread.* O conjunto pode ser usado para criar, manipular e gerenciar linhas de execução assim como também para sincronização entre linhas através de *mutexes e signals.* Não relacionaremos as diferentes funções da biblioteca, apenas as mais comuns.

- Definição de uma nova linha de execução:

  pthread_t LINHA1;

Define uma linha de execução de nome LINHA1.

- Criação de linhas de execução

*int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);*

Se a execução for bem sucedida, uma nova linha de execução será criada e o código de retorno será zero, caso contrário, a linha não será criada e será retornado código de erro indicativo da falha. O parâmetro *thread* retorna a identificação da linha criada, *attr* controla os parâmetros de atributos da linha, caso seja nulo o padrão será utilizado, *start_routine* informa o endereço de início da linha de execução a ser criada e *arg* corresponde aos parâmetros que a ela serão passados.

- União de linhas de execução:

*int pthread_join(pthread_t thread, void **value_ptr);*

A função é usada para unir ou reunir fluxos de controle do processo, a função faz com que a linha de execução que chamou a função, suspenda sua execução até que a linha alvo termine. O parâmetro *thread* é a identificação da linha pela qual a linha que chamou a função está esperando. Se a função retorna corretamente, o estado de saída da função *pthread_exit()* executada pela linha de execução que está terminando é retornado no parâmetro *va-*

lue_ptr. A função retornará um código de erro caso falhe. Deve haver uma chamada à função *pthread_join()*, para cada linha de execução criada com o parâmetro *joinable*, pois no momento da união a memória por ela ocupada é liberada. De outro modo a memória será desperdiçada.

- Término de linhas de execução:

  *int pthread_exit(void *value_ptr);*

  A função é usada para terminar a linha de execução que a chamou. O parâmetro *value_ptr* é passado para a linha que chamou *pthread_join()* para esta linha de execução.

- Cancelamento de linhas de execução:

  int pthread_cancel(pthread_t thread);

  A chamada à função é uma solicitação para cancelar a linha de execução com a identificação *thread*.

- Definição de semáforos:

  pthread_mutex_t semaforo1;

Define um semáforo de nome *semaforo1*.

- Inicialização de semáforos:

*int pthread_mutex_init (pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);*

Inicializa um semáforo com o padrão.

- Bloqueio de semáforos:

*int pthread_mutex_lock(pthread_mutex_t *mutex);*

mutex é um semáforo definido anteriormente

- Liberação de semáforos:

*int pthread_mutex_unlock(pthread_mutex_t *mutex);*

mutex é o semáforo a ser liberado.

3.2 Técnicas de escalonamento

3.2.1 Escalonamento de um laço *DOALL*

Em um laço sem dependências de interação cruzada (*cross-reference*), todas as iterações são independentes entre si e podem ser executadas em qualquer ordem [26], [33]. O próprio nome do laço *DOALL*, sugere execução simultânea. Estratégias de escalonamento para determinar quais iterações devem ser executadas em quais processadores e em que momento, podem ser classificadas como estáticas ou dinâmicas, dependendo de quando a decisão é tomada.

Escalonamento estático, ou pré-escalonamento, designa iterações para processadores específicos em tempo de compilação ou tempo de carga. Cada processador determina qual iteração ele executará baseado no seu número de processador. Por exemplo, as iterações 1 , $p+1$, $2p+1$,... serão executadas no processador 0, as iterações 2, $p+2$, $2p+2$,... no processador 1 e assim por diante, onde p é o número de processadores disponíveis. Esta estratégia de escalonamento, distribui igualmente as iterações, por todos os processadores numa tentativa de balancear a carga computacional. Como cada processador conhece seu número e como os identificadores de tarefas são locais, cada processador facilmente identificará suas tarefas e portanto teremos uma baixa carga de escalonamento.

Se um compilador pudesse predizer com precisão todos os tempos de execução, a carga computacional poderia ser perfeitamente balanceada para minimizar o tempo total de execução. Infelizmente, uma variedade de eventos imprevisíveis torna difícil estimar o tempo de execução de cada iteração. Por exemplo, saídas de desvios condicionais podem ser diferentes, falhas no cache, falhas de página, etc, estes eventos não podem ser previstos em tempo de compilação. É difícil

para a técnica de pré-escalonamento manter um bom balanceamento de cargas.

Escalonamento dinâmico de laço, também chamado auto-escalonamento, pode ser usado para transportar a decisão de escalonamento do tempo de compilação para o de execução, tornando cada processador responsável pela alocação do seu próprio trabalho. No auto-escalonamento processadores ociosos, determinam suas tarefas acessando uma variável compartilhada que indica a próxima iteração. O tempo para acessar esta variável comum e para removê-la da pilha de tarefas a serem executadas, introduz uma sobrecarga de trabalho, mas balanceia o tempo de execução total.

Como a carga geralmente pode não ser perfeitamente balanceada entre os processadores, cada processador pode terminar a execução de suas iterações em tempo diferente dos demais. Uma operação de sincronização é necessária ao final do laço, para prevenir processadores de executarem instruções posteriores ao laço antes que todas as iterações terminem. Uma barreira é normalmente utilizada para essa garantia.

3.2.2 Escalonamento em blocos

A discussão da Seção 3.2 assumia que um processador alocava apenas uma iteração por vez. Obviamente, esta estratégia exige N acessos para alocar N iterações, e produz custo maior no escalonamento. A quantidade de acessos pode ser reduzida, alocando-se blocos de iterações a cada acesso à fila de tarefas. Isto reduz o custo do escalonamento, mas aumenta o desbalanceamento devido à granularidade mais grossa das tarefas.

Estudos foram realizados e chegou-se à conclusão que para um grande número de iterações N, o melhor tamanho de bloco seria $c = N/p$, onde p é o número de processadores. Uma explicação para tal constatação com um grande número de tarefas, as variações no tempo de execução das tarefas. tendem a se cancelarem umas às outras. Portanto, a melhor performance é obtida minimizando o número de operações de escalonamento. Este custo mínimo de escalonamento ocorre quando um único grande bloco de tamanho N/p é designado no início do laço para cada processador.

Mesmo com as varições no tempo de execução tendendo a se cancelarem, esta prática é muito restrita. Uma implementação mais realista teria blocos de tamanho variável, alocando blocos maiores no início da computação para minimizar o custo do escalonamento, e menores no final para minimizar tempos de término desiguais.

3.2.3 Auto escalonamento guiado

Este algoritmo aloca um bloco de tamanho $c = R/p$, onde R é o número de iterações restantes. Portanto o termo R, é reduzido a cada passo do escalonamento, fazendo com que os blocos sejam maiores no início e menores no final do processo de escalonamento. Entretanto, é possível que vários blocos contendo uma única iteração sejam criados no final da execução. Como solução o algoritmo poderia especificar um bloco mínimo, para garantir que cada processador tenha uma carga mínima.

3.2.4 Escalonamento por fatoração

São escalonadas iterações em blocos de tamanhos iguais, sendo estes menores que no auto escalonamento guiado. O algoritmo distribui igualmente metade das iterações remanescentes entre os processadores a cada passo do escalonamento. Portanto todos os processadores executam aproximadamente o mesmo número de iterações no mesmo tempo. Como blocos menores são alocados perto do final da execução do laço, este algoritmo pode ainda compensar tempos de processamento desiguais.

3.2.5 Escalonamento trapezoidal

Este algoritmo utiliza uma função linear simples para determinar o tamanho do bloco a ser alocado no próximo passo. No início de um laço paralelo, o compilador ou o programador determina valores apropriados para os tamanhos inicial e final dos blocos a serem escalonados. O nome trapezoidal vem da observação que é uma função linearmente decrescente ao tamanho do bloco.

3.2.6 Escalonamento por afinidade

Combina estratégias estáticas e dinâmicas. Mantém um bom balanceamento de cargas provendo maior localidade de memória do que uma estratégia puramente dinâmica.

Em tempo de execução, cada processador mantém uma fila local dos trabalhos a ele designados. Depois de executar todas a iterações que à ele foram designadas, um processador verifica com os demais processadores se ainda existem iterações adicionais a serem executadas. Esta habilidade de "roubar"trabalhos [6], permite aos processadores balancear a carga dinamicamente quando a designação

estática inicial leva a um desbalanceamento.

A Tabela 1 exemplifica o número de iterações alocadas a um processador, em cada passo de escalonamento com N=1000 iterações e p=4 processadores, para as técnicas anteriormente descritas.

3.2.7 Escalonamento de linhas de execução

Quando um processo é escalonado para execução, é a linha de execução que utiliza o processador [20]. Se o processo só tem uma linha, é esta linha primária que é designada para para o processador. Se um processo tem múltiplas linhas e existem múltiplos processadores, estas múltiplas linhas são designadas para os processadores. Linhas de execução competem pelo uso do processador com todas as linhas dos processos ativos do sistema ou apenas com as linhas de um único processo.

Linhas de execução são colocadas nas listas de processos prontos classificadas pela sua prioridade. As linhas com a mesma prioridade são escalonadas para processadores de acordo com a política de escalonamento do sistema operacional. Quando não há processadores em número suficiente para atender a todas as linhas, então uma linha com prioridade mais alta pode tomar o lugar de uma outra que esteja executando. Se a nova linha que se torna ativa é do mesmo processo da que foi interrompida, então a troca de contexto acontece entre linhas de execução. Se a nova linha que se tornou ativa é de um outro processo, então ocorre uma troca de contexto

Linhas de execução tem o mesmo estado e transições que os processos. A

Tabela 1: Exemplos de técnicas de escalonamento
Número de iterações alocadas a um processador em cada passo de escalonamento com N=1000 iterações e p=4

Passo	Guiado	Fatoração	Blocos	Trapezoidal
1	250	125	250	76
2	188	125	250	73
3	141	125	250	70
4	106	125	250	67
5	79	63	-	64
6	59	63	-	61
7	45	63	-	58
8	33	63	-	55
9	25	31	-	52
10	19	31	-	49
11	14	31	-	46
12	11	31	-	43
13	8	16	-	40
14	6	16	-	37
15	4	16	-	34
16	3	16	-	31
17	3	8	-	28
18	2	8	-	25
19	1	8	-	22
20	1	8	-	19
21	1	4	-	16
22	1	4	-	13
23	-	4	-	10
24	-	4	-	7
25	-	2	-	4
26	-	2	-	-
27	-	2	-	-
28	-	2	-	-
29	-	1	-	-
30	-	1	-	-
31	-	1	-	-
32	-	1	-	-

Figura 9 ilustra os diferentes estados. São quatro os estados usualmente implementados: pronto, executando, esperando e bloqueado *(ready, running, stopped e blocked)*. No estado de pronto a linha pode ser eleita para execução. No estado executando, a linha está usando o processador. No estado esperando a linha está aguardando um sinal para retomar sua execução. No estado bloqueado, a linha espera um evento externo, acesso a dispositivo ou uma chamada ao sistema, ser completada.

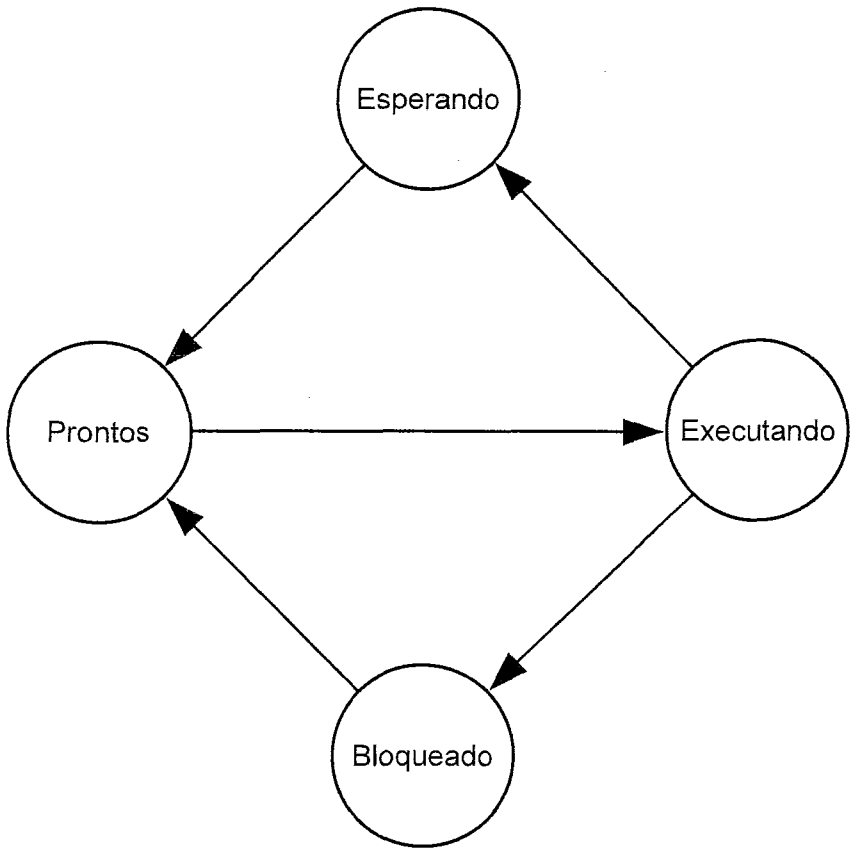


Figura 9: Fases da execução de uma linha

O escopo de contenção determina que conjunto de linhas de execução compete com uma determinada linha. Se uma linha de execução tem escopo de processo, ela competirá apenas com as demais linhas do mesmo processo. Entretanto

se ela tem escopo de de sistema, ela competirá com os seus pares e também com as outras linhas de mesmo escopo dos demais processos. Tomando-se como exemplo uma situação em que existem dois processos em um ambiente com três processadores. O processo A tem três linhas com escopo de processo e uma com escopo de sistema. Processo B tem duas linhas com escopo de processo e uma linha com escopo de sistema. As linhas do processo A com escopo de processo, competem pelo uso do processador A, designado ao processo A e as linhas do processo B com escopo de processo, competem pelo processador C, designado ao processo B. As linhas dos processos A e B, com escopo de sistema, competem pelo processador B, designado para o escopo de sistema.

A política de escalonamento e prioridade do processo se aplica à linha de execução primária. Cada linha de execução pode ter sua política separadamente da linha primária. Linhas de execução tem como prioridade um número inteiro o qual tem um valor mínimo e máximo. Um esquema de prioridades é usado para determinar que linha será designada para o processador em função do valor de sua prioridade, sendo a de mais alta executada antes da de menor prioridade, em um esquema similar ao de processos.

3.3 Técnicas de execução especulativa

Um compilador *TLS - Thread Level Speculation* - especulação a nível de linha de execução, quebra o código seqüencial, difícil de analisar em tarefas (*tasks*). Especulativamente as executa em paralelo, esperando não violar a semântica seqüencial [27].

O fluxo de controle do código seqüencial impõe uma relação de dependência de controle entre as tarefas, como predecessoras e sucessoras. O código seqüencial

também embute uma relação de dependência dos dados, nos acessos à memória realizados pelas diferentes tarefas, que a execução paralela não pode violar.

Uma tarefa é especulativa quando ela pode realizar ou ter realizado operações que violam dependências de controle ou de dados com as tarefas predecessoras. Quando uma tarefa não especulativa termina sua execução, ela está pronta para sua confirmação (*commit*). O papel da confirmação é informar ao restante do sistema que os dados por ela gerados são parte do estado não especulativo do programa. Entre outras operações, esta confirmação envolve passar o estado não especulativo para uma tarefa sucessora. As tarefas devem ser confirmadas dentro da ordem de predecessoras e sucessoras, descrita anteriormente. Enquanto as tarefas executam em paralelo, o sistema deve identificar violações de dependência entre elas. Quando uma violação de dependência é detectada, a tarefa é abandonada e todo estado por ela produzido é descartado. Tarefas sucessoras também são descartadas e serão reexecutadas.

Vários estudos foram realizados sobre arquiteturas de especulação a nível de linha de execução (*TLS*). Bhowmik *et al.* [5], apresenta um ambiente para fracionar programas seqüenciais em múltiplas tarefas. Quiñones *et al.* [38], com o compilador *MITOSIS*. Steffan *et al.* [42], considera outros aspectos além de violações de dependências, Johnson *et al.* [24], propõe uma execução para determinar o perfil da aplicação e selecionar as melhores linhas de execução. Marcuello *et al.* [30], analisa vantagens e desvantagens das diferentes técnicas de especulação, em [28], apresenta uma nova arquitetura para microprocessadores *TLS* e em [29], estuda a performance de vários preditores de valor. Zilles *et al.* [45], através da execução de uma versão aproximada, calcula valores que

serão usados na execução definitiva. Prabhu *et al.* [37] exemplifica como a *TLS* simplifica a paralelização manual e aumenta seu desempenho. Douillet *et al.* [13], gera automaticamente múltiplas linhas de execução usando canalização por *software* em arquiteturas de múltiplos núcleos.

3.4 Programação genérica - Metaprogramação

Dissecando a palavra metaprogramação, literalmente, significa "um programa sobre um programa", ou seja, um metaprograma é um programa que manipula código fonte [1]. Pode ser um conceito velho, mas é familiar. O compilador *C++* é um exemplo que manipula código *C++* e produz código assembler ou código de máquina. Analisadores, como *YACC*, são um outro tipo de programa que manipula programas. A entrada do *YACC* é uma descrição em alto nível do analisador escrito em termos de uma gramática e ligada a ações. O pré-processador da linguagem *C* incluído na suite de compiladores *GCC*, também é um programa em que os dados de entrada são um outro programa.

3.4.1 Metaprogramação em *C++*

Em *C++* o mecanismo de *TEMPLATE* é uma facilidade para metaprogramação nativa na linguagem [14]. Mais importante do que sua capacidade de fazer computações numéricas em tempo de compilação, é a capacidade do *C++* de computação com tipos. Podemos definir um *template* que será substituído pelo tipo de dado no momento da execução, permitindo assim codificação de funções independentes do tipo da variável.

3.4.2 Porque e quando metaprogramar

Uma razão óbvia para usar metaprogramação, é que, fazendo a maior quantidade de trabalho antes do programa resultante iniciar, teremos programas mais rápidos [39]. Quando uma gramática é compilada, *YACC* realiza uma grande análise, passos e otimizações que, se realizados em tempo de execução, poderiam degradar a performance do programa. Uma razão mais sutil, mas um argumento importante para usar um metaprograma é que o resultado da computação pode interagir mais profundamente com a linguagem alvo. Tal como quando você quiser que o código seja expresso em termos de abstrações no domínio do problema. Por exemplo, você pode querer que a matemática de vetores seja escrita usando notação de operadores de matrizes ao invés de laços sobre uma seqüência de elementos.

3.5 Paralelização de programas

Programação seqüencial já é considerada uma tarefa complicada. Mover para um modelo de programação de múltiplas linhas aumenta a complexidade e os custos envolvidos no desenvolvimento de software. Complexidade e custos, aumentados pela tarefa de reescrever o código legado, treinamento de programadores, aumento da complexidade de testes e esforços para evitar condições de corrida, *deadlocks* e outros problemas associados com programação paralela [36] [17]. Entretanto, a quantidade de paralelismo extraído geralmente é insuficiente para ocupar muitos núcleos.

O insucesso ocorreu por duas razões [7]. Primeiramente, nenhum sistema existente trouxe todo o variado grupo de técnicas de análise e otimização de compiladores e suporte de hardware para permitir extração de paralelismo. Mui-

tas aplicações poderiam ser paralelizadas através da integração destas técnicas em um moderno compilador, com uma visão de todo o programa. Segundo, para muitas aplicações que não são paralelizáveis por tais técnicas, o problema com a extração automática de linhas de execução, vem de restrições artificiais impostas pelos modelos de execução seqüenciais, tais como a necessidade do formato de uma informação a ser impressa ou armazenada ser rigidamente explicitado, não deixando nenhuma margem de variação possível para ser usada pelo compilador.

Em particular, somos frequentemente incapazes de especificar que múltiplas saídas possíveis de um programa existem. Por causa disto, o compilador é forçado a manter a única saída correta que um programa seqüencial especifica, mesmo quando outras desejáveis são possíveis. Expressando múltiplas ordens de execução, a paralelização pode ser alcançada automaticamente sem o custo da transformação para um modelo de múltiplas linhas.

4 Uso do SERh para controlar linhas de execução

Uma série de parâmetros devem ser levados em consideração ao usarmos o SERh para controlar linhas de execução.

4.1 Conversão de programas seqüenciais em paralelos

Por anos, contou-se com um constante crescimento na velocidade do relógio para consistentemente fornecer acréscimo de performance para um grande grupo de aplicações. Recentemente, entretanto, esta tendência mudou, a indústria de microprocessadores não pode mais aumentar a velocidade do relógio, devido a problemas de consumo de energia, dissipação de calor e outros fatores. Entretanto, o crescimento exponencial na quantidade de transistores continua forte, fazendo com que as empresas acrescentem valor aos seus produtos , produzindo *chips* que incorporam múltiplos processadores [36].

Enquanto *chip* com multiprocessadores (*CMP*) aumentam o rendimento de códigos multiprogramados ou de múltiplas linhas, eles não beneficiam diretamente muitas importantes aplicações seqüenciais existentes. Compiladores tem tido pouco sucesso ao extrair paralelismo ao nível de linhas de programas seqüenciais. Bons resultados foram obtidos em alguns poucos domínios restritos, mais notavelmente paralelizando aplicações científicas e/ou numéricas. Tais técnicas funcionam bem em poucos laços, manipulando estruturas muito regulares e passíveis de análise, consistindo de acessos previsíveis a vetores.

Em muitos casos, conjuntos de iterações de laços completamente independentes (*DOALL*), ocorrem naturalmente ou são facilmente expostos por trans-

formações. Estas técnicas, como muitos programas tem um complexo controle de fluxo, estrutura de dados recursivas ou acessos através de ponteiros genéricos, tornam-se inadequadas em geral. Como extração automática de linhas de execução se tornou difícil para compiladores alcançarem, arquitetos de computadores se voltaram para técnicas especulativas e de passos múltiplos, técnicas para fazer uso de contextos adicionais de hardware. Estas técnicas são promissoras, mas geralmente necessitam de suporte de hardware para lidar com recuperações em caso de falha de especulação ou lidar com o aquecimento de estruturas da micro-arquitetura. Estas abordagens também são limitadas pela crescente taxas de falhas de especulação, penalidades e poluição encontradas, quando se tornam mais agressivas. Até mesmo a melhor de todas estas técnicas não substitui a necessidade de extração de linhas de execução automáticas e não especulativas. Elas tem um importante papel ortogonal.

4.1.1 Obstáculos à extração de linhas de execução

Bastante paralelismo encoberto já existe em aplicações seqüenciais, este, porém, é devido ao controle de fluxo complexo e acessos irregulares a memória baseados em ponteiros. Esta paralelização não é do tipo *DOALL*, na qual técnicas de paralelização voltadas para aplicações científicas são excelentes. Ao invés disto, laços em programas *C/C++* geralmente tem uma ou mais cadeias com iterações cruzadas. Por sorte, em tais casos, partes de cada iteração podem ser canalizadas e sobrepostas a outras seções em iterações diferentes.

Esse paralelismo de canalização é geralmente explorado por técnicas ao nível de instrução *(ILP)*, tais como desdobramento de laços e canalização por software

(mas geralmente com sucesso variável devido a latências variáveis). Entretanto, como estas técnicas não extraem linhas de execução, elas não podem ser diretamente aplicadas a *chips* multiprocessadores.

Extrair paralelismo de canalização envolve particionar os laços de alguma forma. Naturalmente, para ser apropriado, os laços particionados devem representar uma parcela significativa do tempo total de execução e deve ser de longa duração (várias iterações por chamada) para se sobrepor a custos criados. Laços deste tipo geralmente existem em programas mas não são sempre visíveis para o compilador.

4.1.2 Análise da memória

Com objetivo de extrair paralelismo a nível de instrução (*ILP*), técnicas de análise de dependência de memória eficientes permitem extrações bem sucedidas de linhas de execução.

4.2 Fontes de paralelização - laços

Utilizar as iterações repetidas na execução de laços, foi bastante estudada por [32], [44], [26] e [43], dentre outros. Duas técnicas básicas para melhorar a performance de computadores - usando componentes mais rápidos para reduzir os tempos do ciclo do processador e explorando paralelismo para execução concorrente - não são mutuamente exclusivas. Entretanto, com a redução da taxa de aprimoramentos na tecnologia de semicondutores, arquitetos de sistemas estão se voltando cada vez mais para o processamento paralelo para aumentar o rendimento.

Limitando a extração do paralelismo a um bloco básico (uma seqüência de instruções sem desvios para dentro ou para fora do bloco) limita a aceleração máxima a apenas duas ou quatro vezes em programas. Entretanto, se a fronteiras dos blocos básicos puderem ser ignoradas, o paralelismo de todo o programa estaria disponível para ser explorado. Experimentos simulando este caso ideal, mostram que programas de aplicações científicas e de engenharia tem um alto grau de paralelismo inerente, embora programas fora destas áreas o tenham em menor grau.

Uma maneira para extrair este paralelismo potencial, é concentrar esforços no paralelismo disponível em laços. Como o corpo de um laço, pode ser executado muitas vezes, laços geralmente contribuem com uma grande porção do paralelismo de um programa.

Processadores com canalização e de múltiplas instruções, como superescalares e máquinas com instruções de palavras muito longas (*VLIM - Very Long Instruction Word*), exploram o paralelismo de grão fino ao nível do conjunto de instruções. Estes processadores usam tanto técnicas em tempo de execução quanto em tempo de compilação, tais como busca avançada antecipada e canalização por software, para encontrar operações independentes para serem executados concorrentemente. Em contraste, multiprocessadores de memória compartilhada, exploram o paralelismo de grão grosso distribuindo completas iterações do laço para diferentes processadores.

Uma variedade de diferentes estratégias de escalonamento podem ser usadas

para determinar quais iterações devem executar por quais processadores. Como arquiteturas paralelas diferem em sobrecarga de sincronização, escalonamento de instruções, latência de memória e detalhes de implementações, determinar qual a melhor abordagem para explorar o paralelismo pode ser difícil.

É útil determinar o paralelismo máximo disponível em um laço, independentemente de restrições de recursos disponíveis, e usá-lo como um limite superior para determinar o quão perto, a execução produzida por uma técnica, está perto da máxima possível. O menor tempo para executar o laço completo é o tempo requerido para executar a maior cadeia estendida de dependências para cada ciclo no grafo de dependências. O menor tempo de execução de um laço, sem dependências de iterações cruzadas é o tempo requerido para executar o caminho crítico de uma única iteração.

4.3 O *Gen_For*

O comando *forall*, é uma importante construção de linguagem em muitas linguagens paralelas, ele especifica que computações podem ser realizadas independentemente. Embora sua necessidade seja largamente aceita, a definição do *forall* difere em cada linguagem, ele foi desenhado com um critério específico de implementação em cada uma [12].

A modelagem a seguir desenhada, fará uma implementação deste comando.

4.3.1 Modelagem com o *SER*

O comando *FOR* da linguagem *C++*, é definido na ISO/IEC 9899 [22], como:

for (cláusula-1 ; expressão-2 ; expressão-3) comandos

Definições:

1. A *expressão-2* é o controle, que é calculado antes da execução do corpo do laço.
2. A *expressão-3* é avaliada como uma expressão vazia depois da execução do corpo do laço.
3. Se *cláusula-1* é uma declaração, o escopo de qualquer variável por ela declarada é o restante da declaração e o laço completo.
4. Se *cláusula-1* for uma expressão, ela será avaliada como uma expressão vazia antes da primeira avaliação da *expressão-2*.
5. Ambas *cláusula-1* e *expressão-3* podem ser omitidas.
6. Se *expressão-2* for omitida ela será substituída por uma constante não zero.

A modelagem está sendo feita, baseada no algoritmo *SERh* escalonamento por reversão de arestas com hibernação, descrito na Seção 2.2. Podemos distinguir três fases distintas na execução do comando, a inicialização *FOR*, a execução dos procedimentos internos ao laço *PROC* e uma fase de finalização *END*. Na Figura 10 letra A representamos o grafo que modela o comando antes de ser ativado pelo comando imediatamente anterior (lembrando que o desenho espelha uma parte um programa maior).

A próxima etapa, espelhada na Figura 10, letra B, constitui a passagem para o estado de *READY* do nó *FOR* provocada pelo nó imediatamente anterior no

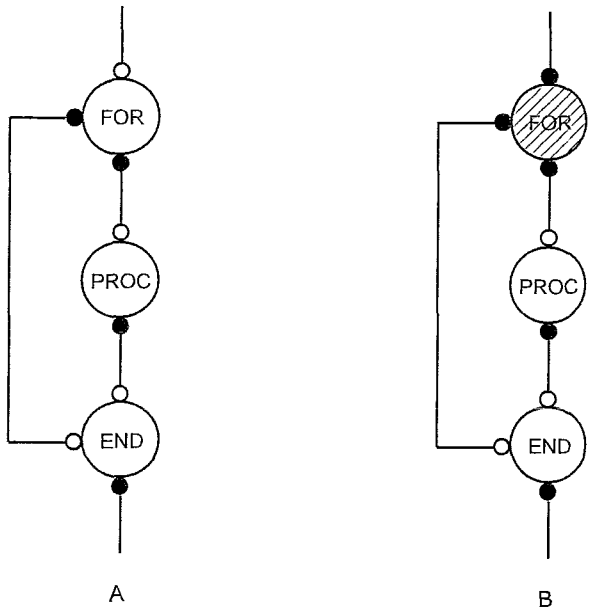


Figura 10: Inicialização do *Gen_For*

programa maior. O nó *FOR* executa o seu procedimento e entra em estado de hibernação, enviando os *tokens* de direito de reversão aos nós vizinhos.

A Figura 11, letra C, demonstra a etapa de execução dos comandos controlados pelo laço, que em seguida entra em hibernação. A Figura 11, letra D, demonstra a etapa de decisão de qual a próxima etapa a ser executada. Retorno ao início do laço, Figura 12, letra E ou saída do laço, executando a próxima instrução do programa, Figura 12, letra F. Observe-se que o grafo, retorna à sua configuração original, preparado para a próxima iteração do programa.

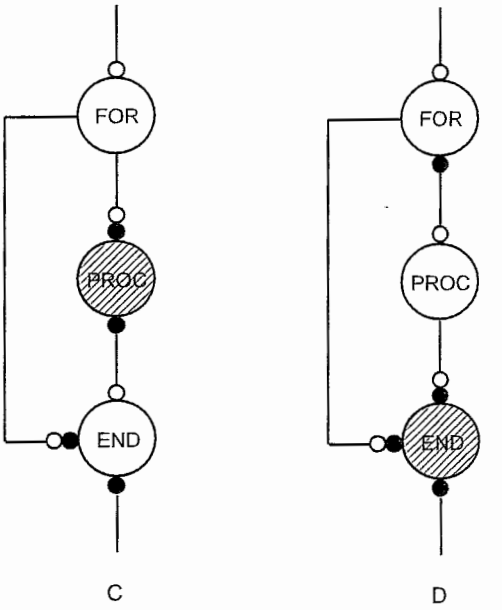


Figura 11: Execução do *Gen_For*

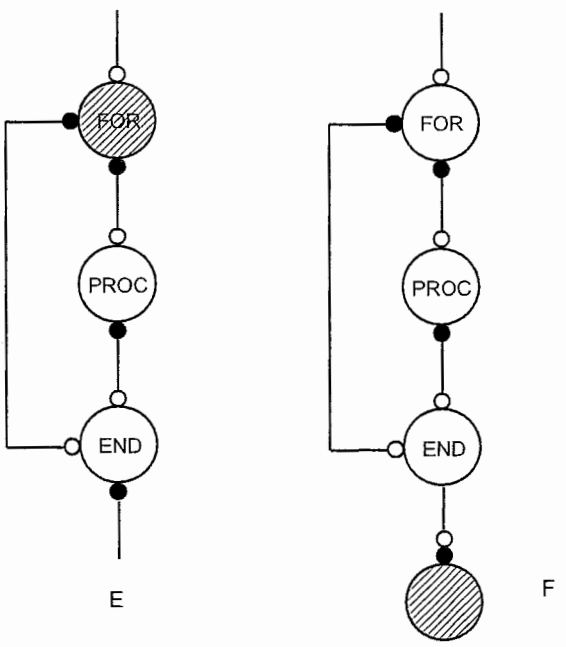


Figura 12: Finalização do *Gen_For*

Paralelização do *Gen_For*.

A partir o elemento atômico, modelando o comando *FOR*, da linguagem, poderemos paralelizar a execução deste comando, previamente parte de um programa. Sujeito às restrições, que serão descritas em descritas em 4.6. A paralelização seria feita, conforme ilustração da Figura 13, onde um grafo é substituído por N outros. N é função da quantidade de núcleos do equipamento. A quantidade de laços destinada a cada subdivisão do comando *FOR* original é determinada pelo tipo de escalonamento mais adequado à carga a ser suportada.

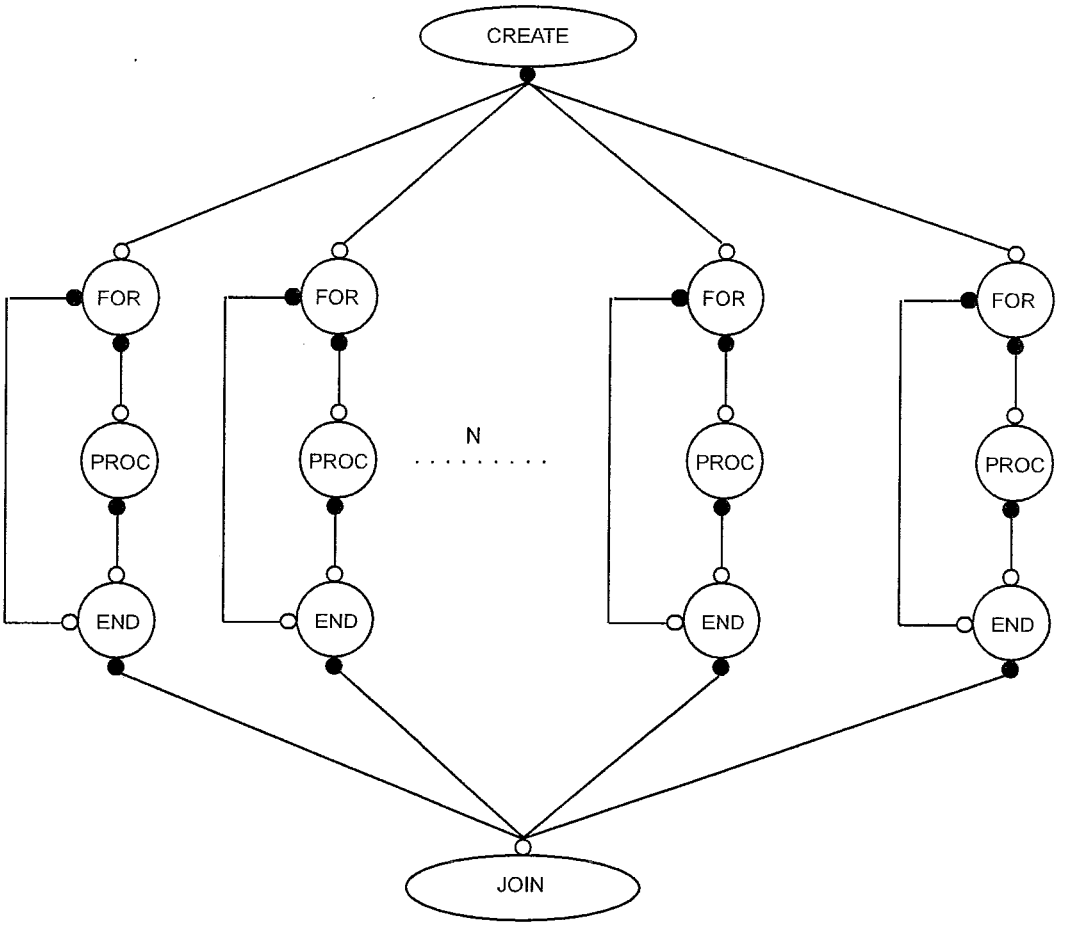


Figura 13: Paralelização do *Gen_For*

4.4 Modelagem com dependência de dados

Nesta seção é modelada a situação em que há dependência de dados de uma iteração com a imediatamente anterior.

4.4.1 Dependência de dados

"Determinar a forma como uma instrução depende de outra é fundamental para determinarmos quanto paralelismo existe em um programa e como este paralelismo pode ser explorado". Patterson [19]

Faremos uma breve introdução aos três tipos de dependências: dados, nomes e controle.

Dizemos que uma instrução é dependente de dados de uma outra, se esta gera um resultado que será usado pela primeira, conforme exemplificado na Figura 14. Esta situação também pode ocorrer de uma cadeia de dependências, ou seja a instrução C é dependente de dados de uma instrução B, que por sua vez é dependente de dados de uma instrução A, fazendo assim com que C seja dependente de dados de A. Nem sempre temos uma visualização simples da dependência de dados, pois uma posição de memória pode ser referenciada de várias formas diferentes (endereço de base + deslocamento)

Na Figura 14, apresentamos a solução de canalização por software para o problema de dependência de dados entre linhas de execução consecutivas e nela podemos observar que esta dependência implica em redução do nível de paralelismo obtido com o modelo do Gen_For da Seção 4.3.

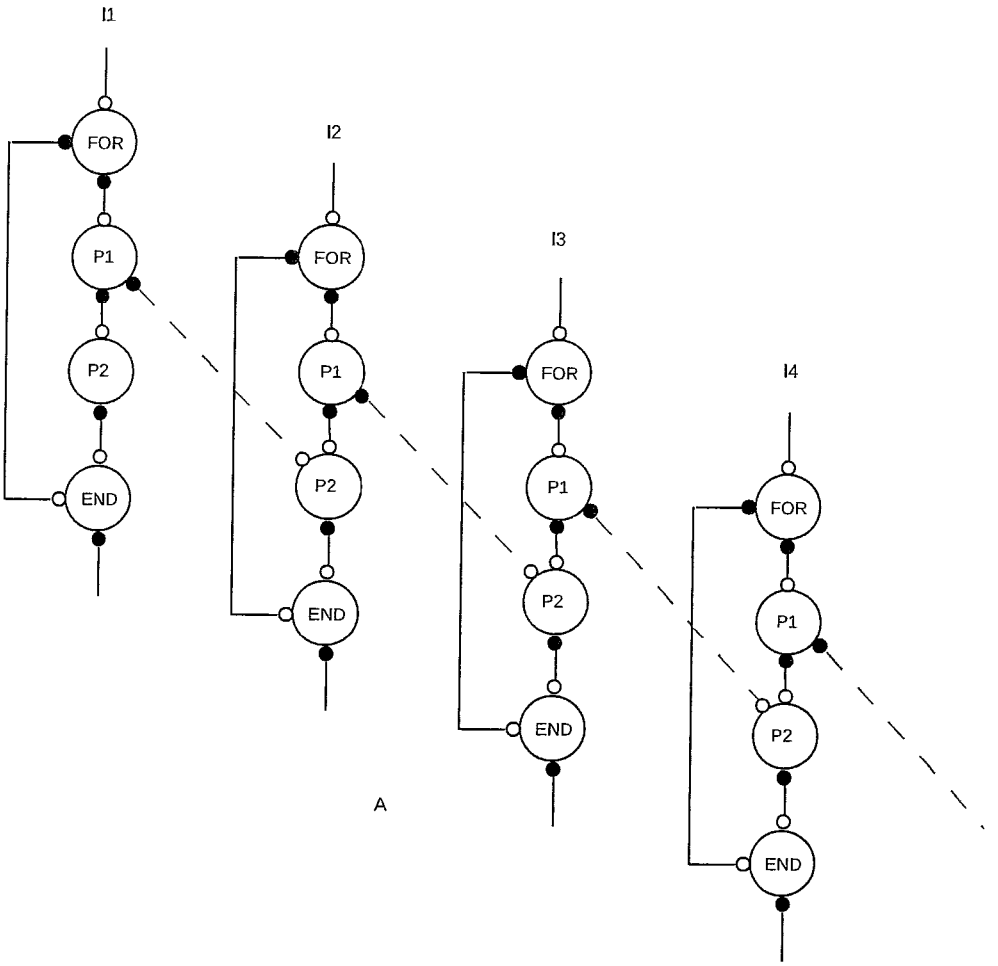


Figura 14: Inicialização do *Gen_For* com canalização

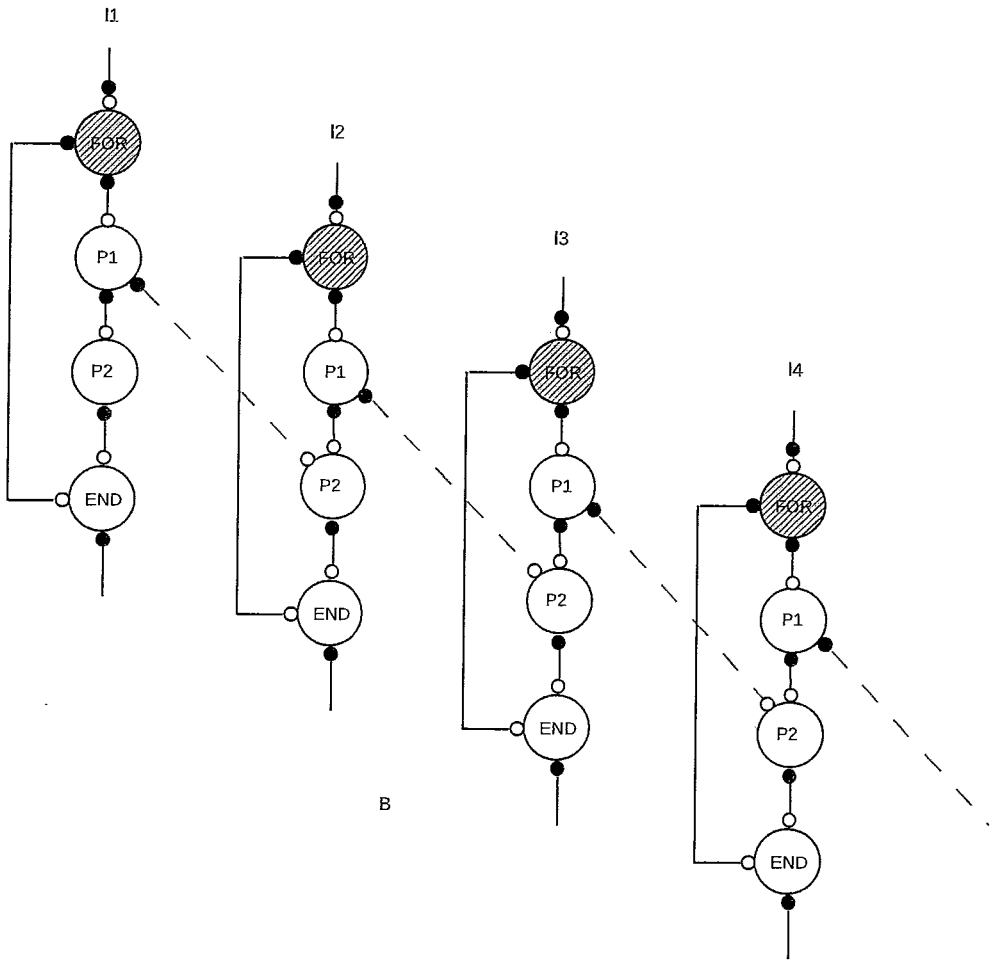


Figura 15: Execução do *Gen_For* com canalização parte I

4.4.2 Modelo

Na Figura 14, modelamos a solução que é obtida através de canalização por software, ou seja, a linha de execução foi dividida em duas partes *P1* e *P2*, de tal forma que a segunda parte *P2* de uma determinada iteração, depende de dados gerados na primeira parte *P1* da iteração imediatamente anterior. A dependência está simbolizada no modelo, através de arestas tracejadas entre *P2* de uma iteração e *P1* da iteração imediatamente anterior.

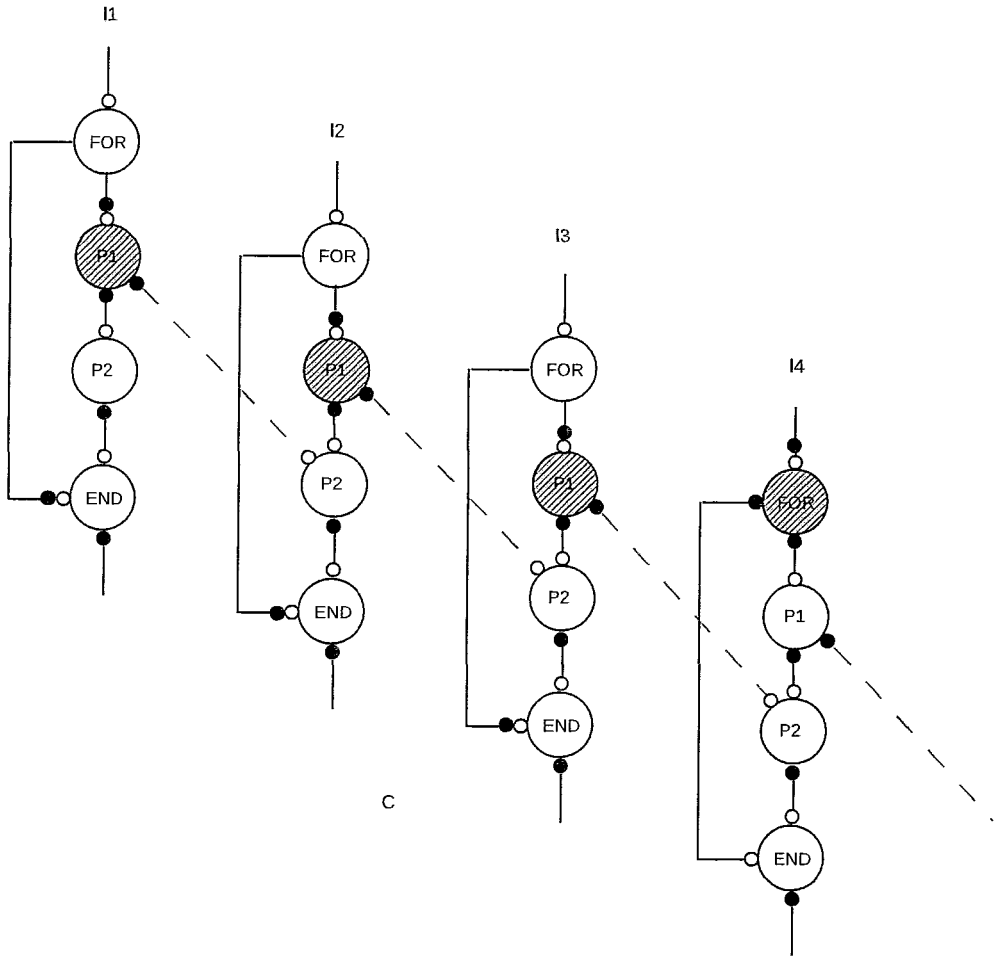


Figura 16: Execução do *Gen_For* com canalização parte II

Na Figura 15 é demonstrada a execução simultânea de quatro linhas de execução *I1, I2, I3* e *I4*. O procedimento *FOR* é iniciado simultaneamente em todas as quatro linhas. Na Figura 16 mostra as três linhas *I1, I2, I3* após o término do seu processamento inicial e executando seus procedimentos *P1*. Para efeito de demonstração do controle distribuído de sincronização obtida via SERh, a linha *I4* permanece ainda no seu procedimento inicial *FOR*, o que não impede a execução em paralelo de *P1* pelas linhas *I1, I2, I3*. Na Figura 17 temos nas iterações *I1* e *I2* os procedimentos *P1* concluídos e o procedimento *P2* já iniciado. Na iteração *I3*, mesmo com o procedimento *P1* da iteração *I2* já tendo sido

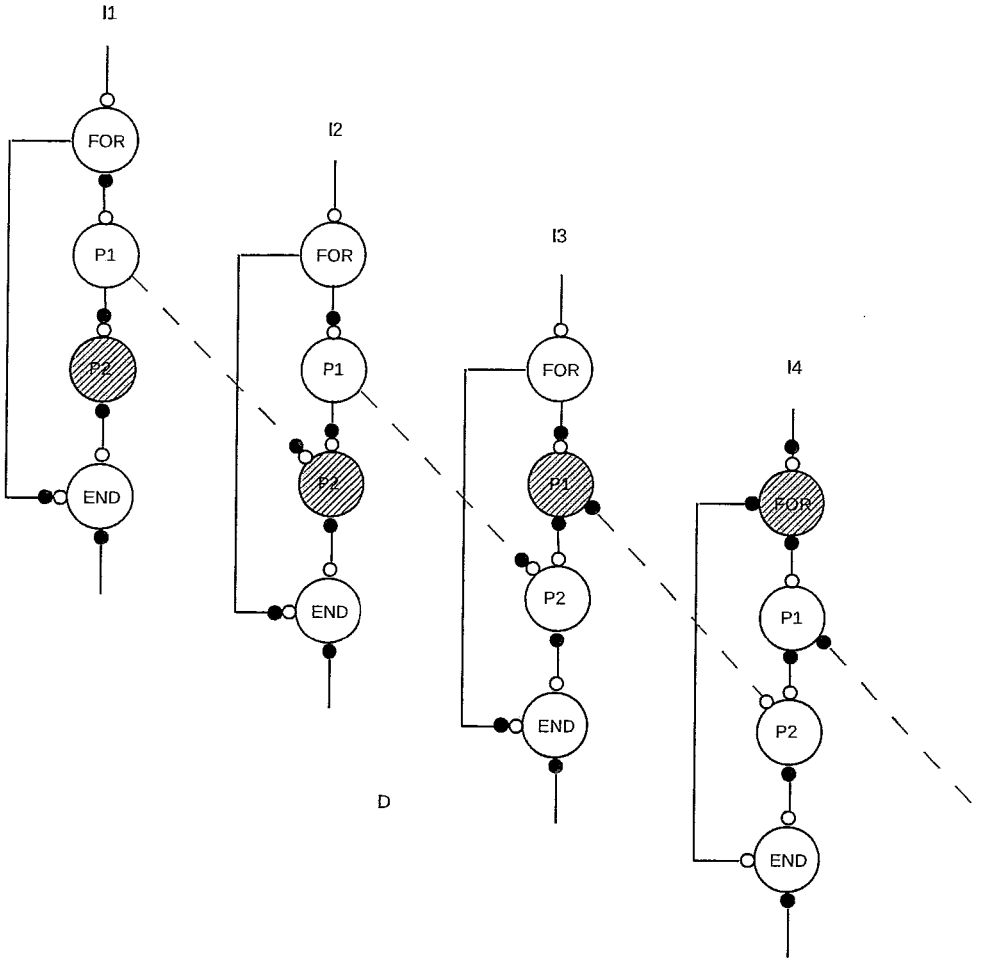


Figura 17: Execução do *Gen_For* com canalização parte III

concluído, seu procedimento *P2*, não poderá ser executado, pois o procedimento *P1* desta linha ainda não foi concluído.

Podemos observar na Figura 17 o efeito de canalização nas iterações *I2* e *I3* quando o procedimento *P1* da linha *I3* é executado em paralelo com o procedimento *P2* da linha *I2*.

4.5 O *Gen_While*

O comando WHILE, conforme descrito na norma ISO/IEC 9899 [22] é:

while (expressão) comandos

A avaliação da *expressão* de controle ocorre antes da execução do corpo do laço.

O comando *WHILE* pode ser modelado pelo mesmo algoritmo descrito para o comando *FOR*. Deveremos fazer restrições de liberdade no seu uso, tais como o controle do laço do *WHILE*, deve ser facilmente identificável. Algumas generalizações são possíveis, mas fora do escopo do trabalho, tais como considerar a execução paralela de múltiplos laços, ou a canalização de iterações que tenham interdependência e talvez execuções especulativas. Estes aspectos serão considerados na Seção 6.2, possibilidades futuras.

4.6 Limitações impostas

A execução das linhas geradas, pelo escalonador produzido no trabalho, será efetuada em um ambiente operacional pré-definido, e, conseqüentemente, apresentará suas restrições inerentes, a seguir comentadas.

4.6.1 Dependências entre as iterações do laço

Na construção do modelo, ficou implícito que não consideramos a dependência de dados ou de controle entre as iterações do laço, ou seja cada uma das diferentes iterações são consideradas como independentes entre si. Esta é uma restrição bastante forte, mas na Seção 5.3.3 - trabalhos relacionados, listamos duas aplicações comerciais, uma da *Oracle* e outra da *Microsoft* com a mesma restrição. Na Seção 6.2 - desenvolvimento futuro, está descrito, como, através de execução especulativa podemos contornar o problema, para considerar estas dependências.

4.6.2 Uso do escalonador do Sistema Operacional

As linhas geradas na conversão do programa escrito no modelo seqüencial convertido para o modelo paralelo, são executadas embutidas em um processo. O qual por sua vez está sob um sistema operacional, que determina o modelo de escalonamento. Este modelo de escalonamento, será aplicado aos processos concorrentes no mesmo ambiente. Em sendo selecionado o processo onde está sendo executado o programa convertido, de nosso interesse, a ele também será aplicado este modelo de escalonamento.

É possível interferir no processo de escalonamento das várias linhas, através da atribuição a elas, de diferentes níveis de prioridades que seriam levados em consideração pelo sistema operacional quando da seleção da próxima tarefa a obter o uso de recurso processador.

Nos testes realizados, as prioridades das linhas criadas não foram alteradas, ou seja, foi mantida em cada uma a prioridade da linha principal de execução,

fazendo com que todas as prioridades sejam idênticas. Na conversão efetuada, nosso interesse é de que todas as linhas terminem sua execução simultaneamente, sem que nenhum núcleo do processador fique ocioso. Trata-se da paralelização de um laço de execução e após seu término, a instrução seguinte do programa será executada. As computações que seguem o laço, ficarão esperando o seu término.

4.6.3 Coerência do cache

Na descrição do ambiente utilizado, verificamos que cada núcleo tem a sua própria *cache L2*. A comutação entre entre as diferentes, linhas provocada pelo escalonamento, descrito no item anterior, nos leva ao problema de coerência de *cache* conforme abordada em [19]. Uma linha tem os seus dados transferidos para a *cache* do núcleo para o qual foi escalonada, executa durante uma fatia de tempo, e é interrompida. Quando retomar a sua execução pode ser escalonada para um outro núcleo, diferente do anterior, forçando com que seus dados sejam transferidos para o *cache* do novo núcleo. Os novos processadores de múltiplos núcleos já dispõem de arquitetura que permite configuração de afinidade [21] em hardware, permitindo ao programador definir que, em uma determinada execução de linhas, não ocorra a migração destas entre os núcleos.

4.6.4 A biblioteca *Pthreads*

A codificação apoiada na biblioteca *Pthreads* impõe como restrição que a arquitetura do equipamento utilizado seja *SMP*. Para utilizarmos uma arquitetura, como por exemplo em *cluster*, seria necessário a codificação com o padrão *MPI - Message Passing Interface*. Mas o trabalho seria aproveitado, pois a modelagem da solução é a mesma, independentemente da plataforma de codificação.

4.7 Processo manual de paralelização

O programa a ser paralelizado é um programa que está operacional e em uso, ou seja foi analisado, codificado, testado, homologado e está sendo rotineiramente utilizado. Quaisquer alterações a serem feitas devem ser bastante seguras e exaustivamente testadas.

Como o modelo está em *C++*, nesta linguagem também deve estar o programa a ser paralelizado. A interferência devida ao processo de paralelização, deve ser a menor possível, limitando-se a cópias e deslocamentos de partes do programa. É necessária uma análise prévia do perfil de execução do programa a ser paralelizado, com o objetivo de determinação precisa das iterações dos *LOOPs* a serem paralelizados. Iterações estas, para esta fase do método de paralelização, devem estar dentro das limitações impostas na Seção 4.5. Não devem ocorrer interferências entre iterações e que estas possam ser executadas em qualquer ordem.

O processo de paralelização manual compõe-se de quatro etapas: primeiramente movemos os comandos que compõe a iteração do comando *FOR*, anotado como 1A, na Figura 18, para o interior da função *iterador*, anotado como 1B, na Figura 19. Em seguida movemos as duas funções *iterador* e *gera_threads_mestre*, anotados como 2B na Figura 19, para a posição anotada como 2A na Figura 18. No terceiro passo, movemos os comandos que substituirão o *FOR* original, anotados como 3B na Figura 19. O quarto passo é igualar as variáveis globais *tudo* e *i*, que controlarão as iterações, aos valores inicial e final do controle do comando *FOR*.

No primeiro passo, modificamos o escopo das variáveis que pertenciam a *main*

```
{
//      int* m;
//      int     n,iarow[m+1],jarow[iarow[m+1]-1],out;    //
//      double  arow[iarow[m+1]-1],diag[m],b[m],dtmp[n]; //  parameters
//      int     ltmp[n];                                 //  ex-boolean

//      char    fname[256];                              //
      int     j,k,irow,jrow,icol,index,x=1;                  // variable
      double  atmp, dif;                                   //
      int     nzflag;                                  //  ex-boolean
      time_t start, end;


//     ------------------------------------------------------------------
//     Open matrix file
//     ------------------------------------------------------------------
//     open(unit=out,file=trim(fname),status='REPLACE')
//     ------------------------------------------------------------------
//     Initialize index array
//     ------------------------------------------------------------------


      for (icol=0;icol<=(*n-1);icol++) {
         ltmp[icol] = 0;
         dtmp[icol] = 0;
      }
      *nz = 0;
      index=0;
//     ------------------------------------------------------------------
//     Compute number of nonzeros in product
//     ------------------------------------------------------------------
      for  (irow=0;irow<=(*m-1);irow++){
         *nz = *nz + 1;    // Count diagonal element

//       ------------------------------------------------------------------
//       Traverse row and scatter column indices
//       ------------------------------------------------------------------
       for (j=(iarow[irow]-1);j<=((iarow[irow+1]-1)-1);j++){
           ltmp[(jarow[j]-1)] = 1;
        }
//       ------------------------------------------------------------------
//       Compare with columns in transpose to build upper diagonal
//       ------------------------------------------------------------------
```