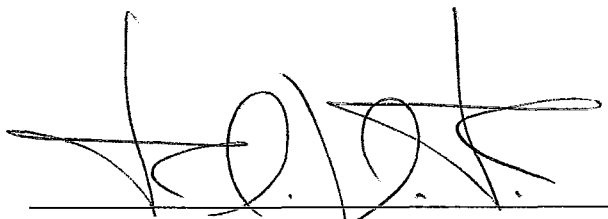
MAPEAMENTO DE EXTENSÕES DO PADRÃO OPENMP PARA CONSTRUÇÕES SINTÁTICAS DE UM SISTEMA MULTITHREADED EFICIENTE

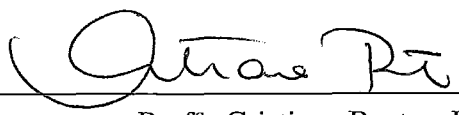
Patrícia Bittencourt Sampaio

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA 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.

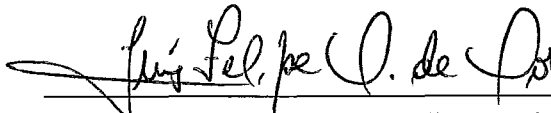
Aprovada por:



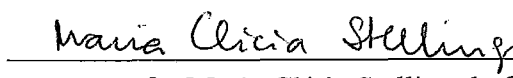
Prof. Felipe Maia Galvão França, Ph.D.



Prof[a]. Cristiana Bentes, D.Sc.



Prof. Luis Felipe Magalhães de Moraes, Ph.D.



Prof[a]. Maria Clícia Stelling de Castro, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

JUNHO DE 2007

SAMPAIO, PATRÍCIA BITTENCOURT

Mapeamento de Extensões do Padrão OpenMP para Construções Sintáticas de um Sistema Multithreaded Eficiente [Rio de Janeiro] 2007

XIII, 91 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2007)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1 - Modelos de Programação para Memória Compartilhada

2 - Tradução de Linguagens em Memória Compartilhada e Distribuída

3 - Sistemas Distribuídos

I. COPPE/UFRJ II. Título (série)

Aos meus pais...

Agradecimentos

Venho agradecer primeiramente à Deus por me conceder o privilégio de realizar este trabalho e por iluminar, a todo instante, o meu caminho, na trajetória da minha formação.

Agradeço em especial aos meus pais, Maria do Carmo e José Francisco, que mesmo em presença de momentos de dificuldade, me apoiaram durante toda a extensão deste trabalho. E aos meus queridos irmãos, Ricardo e Sabrina, pelo apoio prestado.

Ao meu orientador, prof. Claudio Luis de Amorim, que me recebeu no Laboratório de Computação Paralela (LCP), no qual o convívio com algumas pessoas permitiram o meu amadurecimento também no campo pessoal, sendo mais crítica e ponderada diante de algumas situações.

À dedicação daquelas que se tornaram duas amigas, prof^a. Maria Clícia de Castro e prof^a. Cristiana Bentes, pelo acompanhamento e orientação no trabalho. Agradeço imensamente por prestarem este apoio transmitido em reuniões freqüentes para as quais vocês tiveram de se deslocar da UERJ onde trabalham, até a Coppe.

À dois outros grandes amigos, Lauro Whately e Rafael Mendes, que de forma análoga se manifestaram, apresentando algumas sugestões, e sobretudo, pelo companheirismo nestes 3 anos de vivência no LCP. Aos demais colegas do LCP, João Maurício, Leonardo Pinho, Leonardo Bragatto, entre outros, pela solicitude com que me passaram algumas experiências e que serviram para o enriquecimento do meu conhecimento. E ainda, a outros colegas da comunidade de alunos e professores da Coppe com quem tive o prazer de conviver e de firmar amizade.

À colaboração da *Intel Corporation*, em particular ao prezado Lawrence Meadows, pelo atendimento e prontidão para eventuais solicitações. Em especial, pela concessão de licença, válida num período de 4 meses, do produto Cluster OpenMP, considerada esta uma ferramenta recente e não disponível para experimentação.

E, por fim, à todos aqueles que participaram da minha formação me incentivando, pelos quais tenho profunda admiração.

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

MAPEAMENTO DE EXTENSÕES DO PADRÃO OPENMP PARA CONSTRUÇÕES SINTÁTICAS DE UM SISTEMA MULTITHREADED EFICIENTE

Patrícia Bittencourt Sampaio

Junho/2007

Orientador: Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Esta dissertação apresenta uma proposta de tradução de extensões do padrão OpenMP para a linguagem Cilk pertencente a um sistema homônimo que permite execução *multithread* de forma eficiente. O padrão OpenMP institui a base deste trabalho por prover portabilidade em multiprocessadores e facilidade na programação de aplicações paralelas.

A tradução inclui o mapeamento de estruturas do OpenMP para Cilk, e também um processo de tradução para coordenar este mapeamento. Com a tradução foi possível avaliar se um modelo de execução de *threads* distinto do empregado pelo OpenMP provê vantagens para que aplicações escritas com o OpenMP sejam executadas em *clusters* de computadores através do *software* DSM *Clik*.

A correção e o potencial da tradução para multiprocessadores foram verificados com experimentos realizados neste ambiente. Outros experimentos permitiram a identificação de alguns de fatores que afetam o desempenho de programas traduzidos para *clusters* usando o *Clik*, de acordo com o mapeamento proposto. O trabalho revela o efeito destes fatores no *software* DSM, para que sejam reduzidos a fim de consolidar uma alternativa de extensão do padrão OpenMP a *clusters*.

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

MAPPING OF OPENMP EXTENSIONS TO SYNTATIC CONSTRUCTS OF AN EFFICIANTLY MULTITHREADED SYSTEM

Patrícia Bittencourt Sampaio

Junho/2007

Advisor: Claudio Luis de Amorim

Department: Systems Engineering and Computer Science

In this dissertation, we propose a translation from OpenMP extensions to the Cilk language that belongs to a homonym system wich efficiantly provides *multithread* execution. The OpenMP is the basis of this research as it provides multiprocessor portability and supports the parallel programming in an easy manner.

The translation includes the structure mapping from OpenMP to Cilk and a translation process so as to coordinate the mapping. The translation maked it possible to avaliate if a distict *thread* execution model from the one used by OpenMP would provide some facilities on the execution of OpenMP applications on *clusters* with the support of *Clik software* DSM.

We could verify the correction and the efficiancy of the translation for multiprocessor systems with experiments on this environment. Other experiments pointed out factors that compromises the translated programs performance to *clusters* using *Clik*, according to the proposal mapping. From another view, the research shows the effect of these factors on the *software* DSM used in order to be reduced and consolidate an alternative for extending the OpenMP to *clusters*.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A paralelização de aplicações, provenientes de diversas áreas, permite que respostas sejam fornecidas em um menor espaço de tempo. Embora benefícios sejam trazidos com a paralelização, alguns programadores, muitas vezes, relutam ao desenvolvimento de programas paralelos. Em particular, quando se emprega o modelo de comunicação de memória distribuída, baseado na troca de mensagens explícitas.

O modelo de comunicação baseado em troca de mensagens é comumente utilizado em ambientes onde a memória é disposta de forma distribuída. Isto é, ela é dispersa entre nós de processamento onde são criados processos para resolver tarefas. A descentralização da memória requer o envio e recebimento explícito de mensagens entre os processos. Dessa forma, é assegurado que cada processo possua as informações necessárias, não originadas localmente.

Tanto o envio como o recebimento de mensagens são introduzidos ao nível da aplicação. Sendo assim, o programador deve garantir que a informação correta chegue em tempo hábil ao destino determinado. Quando a natureza do problema exige um grande volume de mensagens, o programador passa a ter um esforço substancial para a utilização do modelo.

Uma outra abordagem que facilita o desenvolvimento de aplicações paralelas refere-se ao modelo de comunicação de memória compartilhada. A comunicação no modelo de memória compartilhada ocorre de maneira implícita. A leitura e a escrita de dados compartilhados na memória são realizadas de forma transparente ao usuário. Esse mecanismo fornece ao desenvolvedor a ilusão da inexistência de comunicação entre processos.

Um dos objetos de estudo deste trabalho consiste em uma API padrão e portável voltada para memória compartilhada. Esta API, provida pelo padrão OpenMP [1], surgiu como uma solução para limitações de portabilidade verificada pelo fato de bibliotecas

e compiladores usarem diferentes primitivas para plataforma de hardware. A especificação OpenMP foi concebida para multiprocessadores, incluindo sistemas baseados em barramento e a classe de sistemas de memória compartilhada distribuída (DSM), também conhecidos como *Non Uniform Memory Access Systems* (NUMA).

Embora o modelo de memória compartilhada com o uso do OpenMP em multiprocessadores seja capaz de facilitar a programação paralela, em termos de simplicidade de programação e portabilidade, o custo de uma plataforma como esta é alto.

Uma alternativa escalável de baixo custo, se comparada aos multiprocessadores, referem-se aos *clusters* de computadores. Como forma de permitir o pleno uso do modelo de memória compartilhada em sistemas paralelos, *softwares* DSM (SDSMs) oferecem suporte ao paradigma de memória compartilhada em *clusters*. O *software* DSM *Clik* [7], em particular, propõe um mecanismo eficiente de distribuição de tarefas entre os nós de um *cluster*.

Este trabalho tem como proposta a tradução de aplicações escritas com o padrão OpenMP para um modelo de execução distinto ao nível de tarefas de implementações do OpenMP. Este modelo de execução é baseado na criação dinâmica e assíncrona de *threads* e é utilizado pelo *software* DSM *Clik*. A motivação deste trabalho está em verificar se o modelo dinâmico e assíncrono provê vantagens e facilidades para a execução de aplicações escritas com o OpenMP em *clusters*, através do SDSM *Clik*.

A tradução é realizada sobretudo com o mapeamento de extensões de linguagem do OpenMP para estruturas equivalentes semânticamente na linguagem *Cilk*, utilizada pelo *Clik*. Inclui-se também na tradução, a definição de um processo de tradução para coordenar o mapeamento.

O SDSM *Clik* é derivado de um sistema voltado para multiprocessadores denominado *Cilk* [8]. Para tornar a tradução aplicável a multiprocessadores e também a *clusters*, um mapeamento complementar é proposto para *clusters*. Este mapeamento estende o adotado para multiprocessadores, de forma que para toda aplicação traduzida para *clusters*, é aplicado primeiro o mapeamento para multiprocessadores.

Alguns trabalhos [2, 3, 21, 6] encontrados na literatura, estendem o domínio do OpenMP, de modo a suportá-lo em sistemas distribuídos. O resultado consiste em uma implementação do padrão para *clusters* com o auxílio de SDSMs. O trabalho proposto, porém, não implementa uma versão do padrão neste ambiente. O propósito está em ampliar o domínio do OpenMP à *clusters* com a tradução de aplicações, sem que sejam necessárias modificações na especificação do OpenMP.

Esta proposta procedeu com a validação do mapeamento, seguida da avaliação de aplicações traduzidas de OpenMP para Cilk, através do mapeamento estabelecido. A validação e a análise são realizadas em um multiprocessador e também em um *cluster* de computadores, com o uso do *benchmark* NAS [10].

A avaliação realizada em multiprocessador demonstra que, para cada aplicação pertencente ao *benchmark*, obtêm-se desempenhos na mesma ordem de grandeza entre as versões em OpenMP e traduzida da aplicação. Testes em *cluster*, no entanto, permitiram verificar que, enquanto algumas destas aplicações adquiriram *speedup*, as mesmas aplicações traduzidas, e executadas com o sistema *Clik*, apresentaram *slowdown*. Os resultados provenientes do *cluster* contribuíram para que fossem identificadas algumas características do mapeamento que tornam-se críticas para o desempenho de aplicações no *Clik*.

Em síntese, as principais contribuições desta pesquisa são:

- Verificação, em um multiprocessador, da capacidade de aplicações escritas com um padrão que segue o paralelismo estático e síncrono obter um desempenho equivalente com uma linguagem voltada para o paralelismo dinâmico e assíncrono;
- Execução em *clusters* de computadores de aplicações escritas com um padrão para multiprocessadores, com o objetivo de estender o padrão para sistemas distribuídos;
- Identificação de algumas dificuldades em manter o modelo SPMD usado por um padrão de programação para multiprocessadores, no *software* DSM *Clik*;
- Verificação da capacidade em estender o padrão OpenMP para sistemas distribuídos sem propor alterações no padrão, através da tradução de aplicações.

Este trabalho está organizado da seguinte forma. O Capítulo 2 descreve o modelo de programação e de execução, tanto do padrão OpenMP como do sistema Cilk, seguido de uma comparação dos mecanismos de criação e sincronização de tarefas. O Capítulo 3 aborda o mapeamento de extensões de linguagem do OpenMP criteriosamente selecionadas, incluindo o processo de tradução para coordenar este mapeamento. O mapeamento é mensurado no Capítulo 4, onde são apresentados resultados de aplicações traduzidas para um multiprocessador, e para *clusters*. No Capítulo 5, são relatados alguns trabalhos relacionados. E, por fim, no capítulo 6 são apresentadas conclusões e trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Trabalhos anteriores buscam estender o padrão OpenMP a sistemas distribuídos, através de SDSMs. Nestes, no entanto, a extensão é realizada por meio da implementação do OpenMP em *clusters*. Isto significa que diretivas são incorporadas ao padrão e, em alguns casos, são propostas modificações na implementação de diretivas.

Em [2] é verificada a primeira iniciativa neste sentido. A proposta faz uso do SDSM Treadmarks [19] que provê execução *multithreaded* e coerência de memória. Uma interface para o sistema Treadmarks permite a compilação de um programa com diretivas do OpenMP para um programa que adota o estilo *fork-join* [23] de execução. O trabalho propõe modificações na especificação OpenMP para se adequar às transformações resultantes do processo de compilação.

Em [3] é proposta uma extensão do OpenMP a um *cluster* de SMPs (*Symmetric Multiproccessing Processors*). O paralelismo, neste tipo de sistema, necessita ser explorado tanto no nível de um único nó quanto no nível de um conjunto de nós, para se obter melhor desempenho. Assim, o trabalho apresenta o protótipo de um compilador que permite a comunicação entre *threads* dentro de um nó e entre nós de forma transparente. O compilador introduz funções adicionais à estrutura base do *runtime* de SDSMs e define diretivas de comunicação adicionais ao padrão OpenMP.

Kee et al [21] também propõem um ambiente de programação híbrido. A razão pela configuração híbrida é evitar o excesso de sincronização de operações com *locks* e barreiras implícitas do padrão OpenMP. O ambiente de programação denominado ParADE(*Parallel Application Development Environment*) consiste da tradução automática das aplicações OpenMP para um modelo híbrido de passagem de mensagens e memória compartilhada. A tradução é baseada no compilador descrito em [2] e não é voltada para a especificação mais atual do OpenMP (2.5).

Basulmallik et al [22] discutiram questões de desempenho do OpenMP em sistemas SDSMs. Eles apresentaram algumas técnicas de otimização para melhorar o desempenho nestes ambientes. Além disto, apontaram as sincronizações freqüentes, especificamente barreiras, como o principal obstáculo para um melhor desempenho. No entanto, não foi apresentado um sistema real para demonstrar as técnicas propostas.

Uma proposta mais recente partiu da própria Intel, pioneira há algum tempo no desenvolvimento de compiladores para multiprocessadores. Trata-se do sistema *Cluster OpenMP* [5], um software DSM, baseado em Treadmarks, que demonstra ser uma implementação do OpenMP eficiente [6] em um ambiente de *clusters*. Contudo, é uma solução paga, o que por sua vez inibe a realização de pesquisas com a ferramenta.

Estes trabalhos se revelam diferentes quanto à extensão do OpenMP aqui apresentada. A proposta deste trabalho não está na modificação ou no complemento de diretivas à API do OpenMP, de modo a prover uma implementação do OpenMP para *clusters*. O enfoque do trabalho está na utilização de uma linguagem baseada na execução dinâmica e assíncrona de *threads* para que seja gerada uma aplicação, onde a funcionalidade das estruturas do OpenMP sejam equivalentes. De modo que, com a tradução para esta linguagem seja possível avaliar se é possível adquirir ganhos com as estruturas baseadas em um modelo diferente de execução em um *cluster*.

Capítulo 3

Modelos de Programação para Memória Compartilhada

3.1 O Modelo de Programação OpenMP

Décadas atrás, o estado da arte de plataformas de hardware confrontava-se com as APIs padrões, para programação paralela, existentes. Nesse período, verificou-se a produção em larga escala de sistemas paralelos para memória compartilhada. Enquanto as APIs, com suporte ao paralelismo disponíveis, como MPI [11] e PVM [12], eram voltadas para sistemas de memória distribuída.

O padrão OpenMP surgiu, por uma iniciativa da comunidade de desenvolvedores com a colaboração de representantes da indústria de hardware e software, para suprir esta necessidade. O propósito era prover uma API padrão para multiprocessadores. E, ainda, que fosse portável, de forma que o seu uso não se limitasse a uma plataforma de hardware específica.

O padrão, não é, contudo, definido como uma linguagem. Ele apenas, através de sua API, confere o caráter paralelo a uma aplicação escrita puramente em C,C++ ou Fortran.

Muitas APIs provêem suporte ao paralelismo através de extensões de uma linguagem seqüencial. Essas extensões satisfazem três aspectos básicos da programação paralela: especificação de execução paralela, comunicação entre múltiplas *threads* e sincronização entre as *threads*. O padrão OpenMP atende igualmente a esses preceitos. Algumas abordagens, no entanto, empregam construções adicionais à linguagem base, ou utilizam chamadas de rotinas de biblioteca de tempo de execução, como feito pelo padrão MPI [11] e pelo pacote de *threads* Pthreads [13]. Já o OpenMP adota diretivas embutidas no programa seqüencial para exprimir o paralelismo.

A API da especificação OpenMP é, assim, constituída por um conjunto restrito de

rotinas de biblioteca de tempo de execução e de variáveis de ambiente, e, em grande parte, por extensões de linguagem baseadas em diretivas.

O objetivo desta seção não é discriminar cada componente presente na API do padrão, mas sim fornecer, em linhas gerais o entendimento desses componentes e como se classificam perante à especificação do OpenMP. No Capítulo 4, é descrita a sintática e semântica de um subconjunto, membro da API, utilizado para o propósito deste trabalho.

As rotinas de biblioteca e as variáveis de ambiente controlam parâmetros de execução das aplicações paralelas. O valor inicial de todas as variáveis de ambiente são dependentes de implementação, com exceção da variável *OMP_NESTED*, que é falso sempre no início. Os parâmetros de execução regidos por estas variáveis podem ser modificados após a execução do programa, mas somente se invocada a rotina de biblioteca correspondente.

A forma geral de uma diretiva em OpenMP é especificada como[1]:

```
#pragma omp nome-da-diretiva [cláusulas]
        bloco estruturado
```

A palavra chave *omp* distingue o *pragma* como sendo particular do padrão OpenMP. Desta forma, a diretiva é processada pelos compiladores que suportam OpenMP e ignorada por aqueles que não o suportam. O nome da diretiva comumente aparece acompanhado de uma lista de cláusulas que influenciam, em tempo de execução, o comportamento das *threads* na execução do bloco estruturado da diretiva. Estas cláusulas são comuns entre as diretivas. Entretanto, algumas delas não interagem com certas diretivas. De forma sucinta, as possíveis cláusulas providas pelo padrão são:

- *Cláusulas de Escopo ou de Estereótipo*: controlam o escopo das variáveis, se privativas ou compartilhadas, entre as *threads*, utilizadas no bloco estruturado indicado pela diretiva;

- *Cláusulas de Escalonamento*: afetam, de forma singular, os *loops* paralelos. Os tipos de escalonamento alteram a distribuição das iterações dos *loops* entre as *threads* criadas para o executarem;

- *Cláusulas Condicionais*: determinam se a diretiva é executada em paralelo ou seqüencialmente;

[1] A sintaxe geral, bem como as diretivas em particular, são apresentadas ao longo do texto em termos da linguagem C.

- *Cláusula de Ordenação*: impõe uma ordenação para garantir sincronização na execução das *threads* que participam de um *loop* paralelo;
- *Cláusula Copyin*: inicializa um tipo de variável privativa, chamada *threadprivate*;
- *Cláusula Nowait*: impede a execução de uma barreira implícita relacionada a diretiva que precede a cláusula.

Percebe-se que as cláusulas de escopo e a *copyin* influenciam as variáveis encontradas no respectivo bloco estruturado. Enquanto as demais, intervêem na execução de todo o bloco definido para a diretiva.

O bloco estruturado encontrado próximo à diretiva delimita o escopo da mesma. Quando este bloco está relacionado a uma diretiva que define uma região paralela, ele é dito como sendo a extensão léxica ou estática da região. Muitas vezes, porém, alguns subprogramas, ou seja, algumas subrotinas e funções, fazem parte desta estrutura de controle. Assim, a extensão dinâmica da região é definida de modo a agregar, não só a extensão estática correspondente, como também o código pertencente a estes subprogramas invocados no escopo da diretiva.

As extensões de linguagem do OpenMP se classificam em três categorias: **estruturas de controle paralelo, estruturas de escopo de dados e comunicação** e **estruturas de sincronização**. Cada uma delas, agrega determinadas diretivas, como parte da API da especificação. As próximas seções abordam estas categorias de extensões.

3.1.1 Estruturas de Controle Paralelo

Estas estruturas alteram o fluxo de controle de um bloco estruturado. Apenas um conjunto mínimo de diretivas pertencem a esta classe. Uma delas é aquela que delimita uma região paralela, sendo responsável por criar múltiplas *threads* para executarem concorrentemente a extensão dinâmica da região. A outra diretiva, proveniente deste conjunto, é capaz de dividir tarefas entre um grupo de *threads*. Com ela é possível explorar o paralelismo da aplicação no nível de *loop*.

3.1.2 Estruturas de Escopo de Dados e Comunicação

Esta classe de extensão de linguagem do OpenMP não está diretamente relacionada a diretivas que compõem a API do padrão. Trata-se na realidade da especificação do escopo de variáveis, mediante tipos definidos pelo padrão. Os tipos primitivos determinam se

certa variável possui uma única cópia compartilhada entre as *threads* criadas por uma diretiva de controle paralelo, ou se cada *thread* tem associada uma cópia privativa ao longo da estrutura de controle.

Em relação a esta especificação, todas as *threads* compartilham uma única *heap* global em um programa OpenMP. Portanto, a alocação na área de *heap* é uniformemente acessível por todas as *threads* pertencentes a um grupo. Em contrapartida, cada *thread* possui a sua pilha privativa, usada nas chamadas de subrotinas encontradas em uma estrutura de controle paralelo. Assim, todas as variáveis alocadas na pilha são tratadas como privativas de cada *thread.*

Como parte das normas que contemplam a especificação do escopo de dados, toda variável assume o escopo compartilhado no início do programa, ou seja, uma única cópia da variável é criada. Existem apenas três exceções a esta regra. Primeiramente, as variáveis utilizadas para indexação em *loops* paralelos são sempre consideradas como privativas. De forma similar, as variáveis locais de subrotinas chamadas em estruturas de controle paralelo, e também os parâmetros de valor de dentro de uma subrotina invocada, recebem o escopo privativo. A outra particularidade à regra, diz respeito às variáveis automáticas declaradas na extensão léxica de uma região paralela, que são designadas como privativas.

Como citado anteriormente, as cláusulas de escopo modificam o escopo das variáveis quando utilizadas em diretivas. Elas alteram igualmente o escopo das variáveis tratadas como compartilhadas, à priori.

Estas cláusulas, assim como a especificação atribuída pelo padrão quanto ao escopo das variáveis, configuram as estruturas de escopo de dados e comunicação.

3.1.3 Estruturas de Sincronização

A comunicação entre *threads*, segundo o modelo de memória compartilhada adotado pelo OpenMP, é realizada através de leituras e escritas em variáveis compartilhadas. Para coordenar o acesso a essas variáveis, evitando conflitos nas leituras e escritas, a API do padrão define esta classe de estruturas voltada para a sincronização entre as *threads*.

As duas formas mais elementares de sincronização provida pelo padrão são a exclusão mútua e o evento de sincronização. A exclusão mútua confere acesso exclusivo à determinada variável compartilhada por vez. Enquanto o evento de sincronização é tipicamente usado para marcar um ponto de sincronização entre múltiplas *threads* em execução. Ambos os mecanismos são apoiados por diretivas da API do OpenMP.

3.2 O Modelo de Execução do OpenMP

O padrão OpenMP segue o modelo *fork/join* [23] de execução. Por ele, um único processo, no início de um programa, executa a parte seqüencial. Ao alcançar um construtor paralelo, ele gera um conjunto de outros processos para executar a parte paralela simultâneamente. Ao final do bloco paralelo, o processo criado inicialmente retoma a execução sozinho.

Em OpenMP, os processos criados na execução de um programa consistem em uma abstração de um controle de fluxo independente. Sendo que cada implementação do padrão admite uma forma específica para prover esta abstração. Uma, pode, por exemplo, atribuir uma *thread* OpenMP como um processo do sistema operacional. Enquanto, outra implementação pressupõe que as *threads* equivalem a *Pthreads* [13]. De forma alheia à implementação escolhida, as *threads* OpenMP comportam-se da mesma maneira, de acordo com a especificação OpenMP.

A execução ocorre de forma análoga à descrita para o modelo *fork/join*. O processo que começa a execução do programa é denominado *master thread*. No momento em que a *master thread* encontra uma estrutura de controle paralelo, ela dá origem a algumas *threads* adicionais. Juntamente com elas, integra um grupo de *threads* responsável pela execução do bloco estruturado da diretiva de controle paralelo. Tal procedimento pode ser observado pela Figura 3.1.

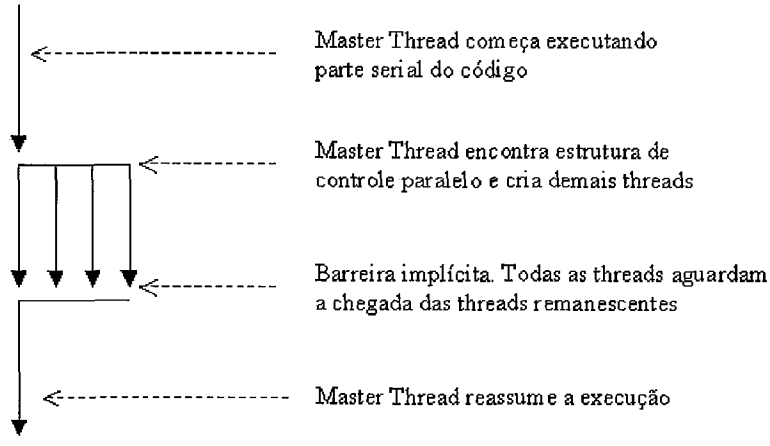


Figura 3.1: Modelo de Execução de um Programa em OpenMP

Após uma barreira implícita ser atingida por todas as *threads* do grupo, no final da estrutura de controle paralelo, apenas a *master thread* volta a executar.

O contexto de execução da *master thread* refere-se ao espaço de endereçamento con-

tendo todas as váriaveis especificadas no programa. Incluem-se nele as variáveis globais, aquelas alocadas na pilha de execução, e ainda as variáveis alocadas dinâmicamente (alocadas na *heap*). O seu contexto de execução persiste até o término da execução do programa. As outras *threads*, pertencentes a um grupo de execução, possuem em seu contexto a sua própria pilha, usada para inserção de *frames* na invocação de subrotinas. Conforme mencionado na Seção 3.1.2, esta pilha de execução é privativa de cada *thread*. Todas as outras variáveis presentes no contexto de execução das demais *threads*, que atuam em um grupo, possuem seu escopo definido através das estruturas de escopo de dados e comunicação.

3.3 O Modelo de Programação Cilk

Cilk é conceituamente considerada uma extensão *multithread* da linguagem C [9]. Neste trabalho, entretanto, o termo Cilk faz menção à implementação desta linguagem, que consiste em um sistema para multiprocessadores [8]. Casualmente o termo refere-se à linguagem deste sistema, no texto. Neste caso, ele é explicitamente expresso como sendo uma linguagem.

A extensão de linguagem básica de Cilk é composta por três palavras chave que provêem a abstração de *threads*, bem como mecanismos de sincronização:

- *cilk* - utilizada para definição de uma rotina *Cilk*;
- *spawn* - utilizada para criação dinâmica de *threads*;
- *sync* - permite a sincronização de *threads*.

Uma rotina *Cilk* é identificada pela palavra chave *cilk*. Assim como a definição do seu corpo e de sua lista de argumentos, as operações inerentes à rotina são executadas seqüencialmente de forma equivalente a uma simples rotina em C.

O paralelismo de *threads* é atribuído a uma rotina *Cilk*, a partir do momento que alguma rotina *Cilk* também é chamada dentro dela. A invocação desta rotina é feita com o uso da primitiva *spawn*, responsável pela criação dinâmica de uma *thread*. Analogamente à execução de uma rotina em C, quando uma *thread* é criada pela extensão provida por Cilk, a execução da rotina invocada é iniciada. Ao contrário de uma rotina em C, entretanto, na ausência de sincronização, a *thread* criada, denominada *filha*, pode executar em paralelo com aquela que a criou, a *thread-pai*. A *thread-filha*, por sua vez, pode criar outras *threads*, e assim sucessivamente, gerando um alto nível de paralelismo.

A sincronização ocorre na presença de uma primitiva *sync*, que funciona como uma barreira local. A primitiva impõe a condição de que a *thread-pai* reinicie sua execução somente após o término das suas *filhas*. Caso a primitiva não se encontre em uma rotina, ela é inserida implícitamente antes do seu retorno. Isto assegura que a rotina não termine enquanto existirem *threads-filhas* pendentes.

Na terminologia da linguagem Cilk, uma tarefa compreende uma seqüência de instruções delimitadas por um *spawn*, *sync* ou por um retorno de rotina. Cada tarefa associa-se a uma *thread*.

As primitivas *spawn* e *sync* têm funcionalidades semelhantes às primitivas *fork* e *join* do Unix [23]. Além de sua extensão básica, a linguagem Cilk suporta ainda acesso a dados com exclusão mútua, através das primitivas *lock* e *unlock*.

A Figura 3.2 mostra um exemplo típico de um algoritmo que usa a linguagem Cilk. O algoritmo calcula o *n*-ésimo número de Fibonacci.

```
cilk int fib (int n)
{
  int x, y;
  if (n < 2) return n;
  else {
     x = spawn fib(n-1);
     y = spawn fib(n-2);
     sync;
     return(x+y);
  }
}
cilk int main (int argc, int **argv)
{
  int n, result;
  n = (int)argv[1];
  result = spawn fib(n);
  sync;
  printf("Resultado: %d \n", result);
}
```

Figura 3.2: Geração do n-ésimo número de Fibonacci segundo a linguagem Cilk

Como é possível observar na Figura 3.2, uma *thread-pai* é criada no escopo do *main*, a qual cria duas *threads-filhas* responsáveis pelo cálculo de `fib(n-1)` e `fib(n-2)`.

Estas, por sua vez, criam suas *threads-filhas* formando uma árvore binária que descreve a computação do algoritmo. Pelo programa, nota-se que a linguagem Cilk permite expressar de forma natural o paralelismo alcançado com algoritmos recursivos, através da característica dinâmica e assíncrona da linguagem.

3.4 O Modelo de Execução do Cilk

O sistema de *runtime* do Cilk se encarrega de alguns aspectos na execução de um programa, como balanceamento de carga, paginação e implementação de protocolos de comunicação. Cabe a ele escalonar as *threads* entre os processadores de maneira eficiente.

Uma aplicação escrita com a linguagem Cilk, pode ser representada por um grafo acíclico direcionado que cresce dinamicamente. Cada *thread* corresponde a um nó do grafo. A Figura 3.3 ilustra um exemplo para uma aplicação. A dependência entre as *threads* é verificada pelas arestas que apontam para cima e para baixo. Aquelas que incidem para baixo indicam a criação dinâmica de *threads* contribuindo para a expansão do grafo. Já aquelas voltadas para cima sinalizam o retorno de uma *thread-filha* que faz contrair o grafo. Os retângulos, presentes no grafo, representam rotinas *Cilk* e esboçam o progresso da *thread* que executa a rotina.

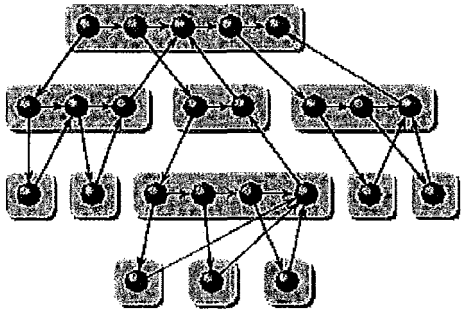


Figura 3.3: Grafo acíclico que exprime o comportamento de uma aplicação em Cilk

Uma das propriedades do escalonador do Cilk está em respeitar todas as dependências do grafo. Essas dependências formam uma ordem parcial, permitindo várias possibilidades de escalonamento das *threads*. As políticas de escalonamento obedecem a requisitos de espaço e tempo distintos. O requisito de espaço refere-se à área de armazenamento necessária para uma rotina *Cilk*, considerando apenas a alocação na pilha. O requisito de tempo está relacionado ao tempo gasto com comunicação, na existência de uma dependência entre *threads* hospedadas em processadores diferentes.

Embora não seja identificada uma política excelente para grafos *mutithread*, em termos gerais, todo programa baseado no paralelismo dinâmico e assíncrono gera um grafo bem estruturado [14]. E esse grafo pode ser eficientemente escalonado como descrito em [15].

A política de escalonamento adotada por Cilk descreve um mecanismo randômico denominado **roubo de trabalho**. Através dele, quando um processador não possui tarefas a executar, ele solicita uma tarefa a outro processador escolhido aleatoriamente. Quando ocorre a criação de uma *thread*, o processador corrente salva as variáveis locais da *thread-pai* na base de uma pilha de tarefas. Feito isto, ele começa a executar a tarefa correspondente à *thread-filha*. Ao finalizar a execução da *thread-filha*, a base da pilha é carregada e a *thread-pai* reassume. Ao solicitar uma tarefa, o processador requisitante rouba, se houver, uma tarefa do topo da pilha. Ou seja, da extremidade oposta daquela utilizada pelo processador que detêm a pilha. Caso ele não encontre uma tarefa a executar, outro processador vítima é escolhido randômicamente para o roubo.

O **roubo de trabalho** é implementado através de um protocolo de comunicação *request-reply* [24] entre o ladrão e a vítima.

3.5 Criação e Sincronização de tarefas nos modelos OpenMP e Cilk

Os mecanismos de criação e sincronização de tarefas caracterizam a natureza do paralelismo de um modelo de programação. Os modelos OpenMP e Cilk adotam naturezas distintas de paralelismo [16]. Como mencionado na Seção 3.4, o modelo do Cilk é mais voltado para o paralelismo dinâmico e assíncrono. Enquanto o modelo, provido pelo padrão OpenMP, explora de forma irrestrita o paralelismo estático e síncrono.

O paralelismo dinâmico se distingüe do paralelismo estático quanto ao particionamento de tarefas. No paralelismo dinâmico, enquanto existir tarefas a serem computadas, estas são gradativamente particionadas, e, desta forma, criadas. Isto acontece até não existirem mais tarefas a se executar. No paralelismo de dados, o particionamento é realizado uma única vez, antes do exercício da computação, sendo, portanto, dito como estático.

A sincronização de tarefas previamente criadas é expressa sob duas formas. Ora pelo paralelismo síncrono empregado pelo padrão OpenMP. Ora pelo paralelismo assíncrono predominante no modelo Cilk. No síncrono, os pontos de sincronização, nos quais são realizados alguns eventos para garantir a coerência da memória, são comuns entre as

tarefas criadas. Diferentemente do paralelismo assíncrono cujos pontos de sincronismo divergem de acordo com as tarefas.

O paralelismo estático e síncrono configura o modelo SPMD (*Single Program Multiple Data*) de programação que impera no modelo OpenMP, sendo, no entanto, não comumente empregado em Cilk.

Capítulo 4

Estudo e Tradução das Extensões do Padrão OpenMP para Cilk

Basicamente duas categorias formam as extensões de linguagem providas pelo OpenMP: as estruturas de controle paralelo e as de sincronização, usadas em um bloco de código estruturado, e ainda, as estruturas de controle de escopo de dados e comunicação, empregadas para variáveis em um programa. O subdomínio da API do OpenMP sobre o qual essas categorias de estruturas incidem são fortemente distintos. As estruturas de controle paralelo bem como as de sincronização são expressas através de diretivas definidas pela API. Já as estruturas de controle de escopo de dados e comunicação não são vinculadas diretamente a diretivas, mas à especificação e às cláusulas de escopo de dados, que interagem com algumas diretivas.

Outros aspectos tornam o subdomínio da API expresso por estas duas categorias de extensões do OpenMP diferentes. As diretivas associadas às estruturas de controle e de sincronização são mais facilmente identificadas do que a determinação do escopo de cada variável, em um programa com o padrão OpenMP. Isto ocorre em função das regras de escopo e particularidades definidas pelo padrão. Além disso, no caso das estruturas de escopo de dados, uma variável predominante em todo o programa pode ter seu escopo modificado inúmeras vezes, enquanto nas estruturas de controle paralelo e de sincronização um bloco estruturado é individual de cada diretiva, o que revela a ausência da propriedade de ubiqüidade verificada nas estruturas de escopo de dados.

Por estas razões, a tradução proposta das extensões do OpenMP para Cilk compreende o mapeamento das estruturas de controle paralelo e de sincronização e, de forma distinta, o mapeamento das estruturas de controle de escopo de dados e comunicação. Os mapeamentos reproduzem, com a extensão básica da linguagem Cilk, a semântica encontrada nas estruturas do padrão OpenMP.

Em vista da grande variedade de extensões de linguagem do padrão OpenMP, e das possíveis combinações entre elas, foi realizada uma análise de freqüência das extensões do OpenMP sobre o *benchmark* NAS [10]. Desta forma, a tradução incide sobre um conjunto mais restrito de extensões, extraídas pela análise de freqüência.

Este capítulo aborda, primeiramente, a análise de freqüência das extensões do OpenMP, e posteriormente, descreve o mapeamento estabelecido para as estruturas de controle paralelo e de sincronização, e para as estruturas de controle de escopo de dados e comunicação, que ocorrem com mais freqüência na análise realizada. Na Seção 4.4 é apresentado, como parte da tradução das extensões do OpenMP para *Cilk*, um processo de tradução que coordena o mapeamento proposto para as duas categorias de estruturas do OpenMP. O final do capítulo apresenta uma extensão do mapeamento das estruturas de controle de escopo de dados e comunicação para a tradução de aplicações em *clusters* de computadores.

4.1 Análise de Freqüência de Extensões do OpenMP

Para a análise de freqüência das extensões do OpenMP, foram utilizados os pacotes 2.3 e 3.0 do conjunto NAS [10]. Na análise, foram verificadas todas as extensões que ocorrem nas aplicações que compõem o *benchmark*, incluindo diretivas, cláusulas e rotinas de biblioteca. A partir desta verificação, as extensões que são utilizadas mais de uma vez em cada aplicação foram consideradas no mapeamento para Cilk. As extensões que não se aplicaram a esta condição receberam ainda assim um mapeamento para Cilk, na medida que o mapeamento demonstrava-se trivial.

Como mencionado na Seção 3.1, a API do OpenMP é formada por um conjunto de rotinas de biblioteca de tempo de execução, por variáveis de ambiente e por diretivas e cláusulas definidas pelo padrão. Em especial a API classifica as rotinas de biblioteca em função de três tipos:

- **Rotinas do Ambiente de Execução** - controlam o ambiente de execução paralelo;
- **Rotinas de Lock** - sincronizam o acesso a dados;
- **Rotinas de Temporização** - fornecem o tempo total decorrido com a aplicação.

As rotinas de *lock* foram desconsideradas no mapeamento pois possuem a mesma funcionalidade provida pela diretiva *critical*, identificada em algumas aplicações do *ben-*

chmark. Assim como essas rotinas, não foi definido um mapeamento para as de temporização já que foi utilizada a biblioteca do C padrão para a contabilização de tempo.

Dentre as rotinas do ambiente de execução, aquelas escolhidas pela análise podem ser observadas na Tabela 4.1. A definição de cada uma, segundo a API do OpenMP, pode ser identificada na mesma Tabela.

Rotinas de Biblioteca (C/C++)	**Descrição**
int omp_get_num_threads(void)	Retorna o número de *threads* que estão em execução
int omp_get_thread_num(void)	Retorna um identificador da *thread* dentro de um grupo, sendo sempre um valor no intervalo de 0 a *omp_get_num_threads()*-1

Tabela 4.1: Rotinas de Biblioteca de Execução do OpenMP

O ambiente de execução de uma aplicação escrita com o OpenMP é, em parte, especificado através de variáveis de controle internas da implementação do padrão. Essas variáveis internas podem ter seus valores inicializados pelas variáveis de ambiente da API. Através das rotinas do ambiente de execução é possível ler ou atribuir outro valor às variáveis de controle internas. Assim, de forma indireta, algumas destas rotinas possuem uma variável de ambiente associada. A rotina *omp_get_num_threads*, por exemplo, associa-se à variável *OMP_NUM_THREADS*, que define a quantidade de *threads* para uma aplicação. Em Cilk, a variável *OMP_NUM_THREADS* foi representada através de uma constante denominada *OMPCILK_TOT_THREADS*. O mapeamento, portanto, da rotina *omp_get_num_threads* consiste no retorno da contante *OMPCILK_TOT_THREADS*.

A rotina *omp_get_thread_num*, de forma exclusiva, não se relaciona a uma variável de ambiente. O seu mapeamento implica no retorno de uma variável chamada *ompcilk_tk*, utilizada para a criação de uma tarefa em Cilk, como apresentado na Seção 4.2.2.

O mapeamento das demais variáveis de ambiente ocorreu de forma implícita em função dos valores *default* atribuídos a elas. A Tabela 4.2 lista estas outras variáveis. Suas descrições e os valores *default* assumidos quando não se especifica um valor para cada uma, também encontram-se na tabela.

O mapeamento de algumas extensões de linguagem são, em geral, condicionados à implementação do OpenMP. Este é o caso, por exemplo, da variável *OMP_SCHEDULE*, já que o valor inicial assumido para a variável é dependente do compilador utilizado. Assim, foi considerado o escalonador **estático simples**, definido como o valor inicial da variável pelo compilador *Intel* [4], com o qual os mapeamentos propostos foram validados.

Variável de Ambiente	Descrição	Valor *Default*
OMP_SCHEDULE	Define o tipo de escalonador de *loop* e a quantidade de iterações a serem executadas por cada *thread*	Dependente da implementação (definido pelo compilador)
OMP_DYNAMIC	Permite ou não o ajuste dinâmico da quantidade de *threads* usadas em regiões paralelas	Dependente da implementação (definido pelo compilador)
OMP_NESTED	Permite ou não o alinhamento de regiões paralelas	*Falso*

Tabela 4.2: Variáveis de Ambiente do OpenMP

Entretanto, a variável *OMP_SCHEDULE* não possui uma correspondente no programa traduzido. Não é comum que o tipo de escalonador se altere no decorrer do programa escrito com o padrão OpenMP, como observado no conjunto de aplicações. Assim, para cumprir a função da variável, o comportamento do escalonador estático simples, descrito na Seção 4.2.2, é preservado em todos os *loops* que são paralelizados.

Na medida que não foi identificada nenhuma ocorrência das rotinas de ambiente que lêem e escrevem os valores correspondentes à variável *OMP_DYNAMIC*, nenhum mapeamento foi dedicado a ela. E, uma vez que o valor *default* atribuído pelo compilador *Intel* à variável é **Falso**, a mesma quantidade de *threads* é utilizada em cada região paralela, sendo esta quantidade também empregada na aplicação traduzida.

Nas aplicações do NAS não foram encontradas regiões paralelas capazes de se alinhar com uma região mais interna e, portanto, a variável *OMP_NESTED* tornou-se isenta de mapeamento. Apesar disto, a função exercida com o alinhamento de regiões paralelas é facilmente expressa em Cilk. Assim, na Seção 4.2.1, encontra-se o mapeamento proposto na existência de regiões alinhadas.

Em relação às diretivas presentes na API do OpenMP, 291 ocorrências foram verificadas com a versão 2.3 do conjunto NAS. Conforme mostra a Tabela (a) da Figura 4.1, desse total 85% correspondem às diretivas de região paralela e de divisão de tarefas (*omp parallel*, *omp parallel for*, *omp for* e *omp single*), enquanto as diretivas de sincronização contabilizam 15% do total (*omp master*, *omp barrier*, *omp critical* e *omp flush*).

Dentre as diretivas de sincronização, 2% delas referem-se à diretiva *omp flush*. O mapeamento desta diretiva tornou-se desprezível em função da complexidade quanto ao seu emprego e da introdução de *overheads* com o mapeamento da diretiva para Cilk, conforme esclarece a Seção 5.1.

Diretiva		Cláusulas	
omp parallel	7%	private	47%
		firstprivate	16%
		copyin	5%
		default	0%
		shared	0%
omp parallel for	2%	private	25%
		default	0%
		shared	0%
		reduction	12%
omp for	66%	private	3%
		nowait	19%
		schedule(static)	4%
		reduction	4%
omp master	8%		
omp barrier	3%		
omp critical	2%		
omp single	10%	nowait	10%
omp flush	2%		

Total de ocorrências das diretivas = 291

(a) Análise pelo Benchmark NAS NPB-2.3

Diretiva		Cláusulas	
omp parallel	14%	private	39%
		firstprivate	0%
		copyin	3%
		default	91%
		shared	36%
omp parallel for	18%	private	93%
		default	98%
		shared	19%
		reduction	12%
omp for	39%	private	0%
		nowait	69%
		schedule(static)	6%
		reduction	5%
omp master	18%		
omp barrier	2%		
omp critical	6%		
omp single	0%	nowait	0%
omp flush	3%		

Total de ocorrências das diretivas = 237

(b) Análise pelo Benchmark NAS NPB-3.0

Figura 4.1: Freqüência de Diretivas e Cláusulas do OpenMP

Na Figura 4.1, observa-se que algumas cláusulas são agrupadas por diretiva. A freqüência mostrada em função destas cláusulas refere-se ao percentual em que a diretiva em questão aparece acompanhada das cláusulas. Assim 47%, por exemplo, das diretivas *omp parallel* são sucedidas da cláusula *private*. Nota-se que nem todas as cláusulas estão presentes para cada diretiva. Apenas algumas cláusulas são consideradas representativas para uma diretiva específica, pela própria API do OpenMP.

A análise decorrente com o pacote NPB-3.0 do conjunto, resultou na identificação de 237 ocorrências de diretivas, como mostra a Tabela (b) da Figura 4.1, sendo 71% delas referentes às de região paralela e de divisão de tarefas e 29% às diretivas de sincronização.

As diferenças na freqüência das diretivas entre o pacote 2.3 e 3.0 do NAS é atribuída às melhorias realizadas com a atualização da versão do *benchmark*. Uma melhoria significativa está no aumento do emprego da cláusula *nowait* acompanhada da diretiva *omp for*, que despreza a execução de uma barreira implícita. E, ainda, na utilização da diretiva *omp master* em lugar da diretiva *omp single*, também como finalidade de eliminar a execução de uma barreira implícita após a *omp single*.

Todas as diretivas encontradas na Figura 4.1, com exceção da diretiva *omp flush*, foram consideradas no mapeamento para Cilk, em vista da ocorrência de cada uma nas duas

versões do NAS. Pela análise de freqüência, o conjunto de diretivas considerado constitui-se suficiente para expressar o paralelismo em uma aplicação com o OpenMP, já que a única diretiva não mapeada foi encontrada em apenas uma aplicação e contribui com somente 2% e 3% no total de ocorrência de diretivas nas versões 2.3 e 3.0 do NAS. Em relação às cláusulas, todas foram consideradas, excluindo-se a *schedule* e a *default*.

4.2 Mapeamento das Estruturas de Controle Paralelo e de Sincronização

No modelo de programação do Cilk é possível a escrita de um código usando o paralelismo estático e síncrono. Esta característica do Cilk facilitou o mapeamento das estruturas de controle paralelo e de sincronização com a representação, na liguagem Cilk, do modelo SPMD de programação, no qual o OpenMP é fortemente baseado.

Cabe ressaltar que, mesmo atribuindo a um programa traduzido o caráter estático e síncrono, o modelo de execução do Cilk é invariante. A política de escalonamento do modelo continua sendo sustentada pelo comportamento dinâmico e assíncrono das *threads*. Logo, o mapeamento das estruturas de controle paralelo e de sincronização foi estabelecido com a associação do OpenMP e do Cilk no nível de programação, mantendo-se inalterado o modelo de execução originário de ambos.

De acordo com a análise de freqüência realizada, as diretivas relacionadas às estruturas de controle paralelo que receberam um mapeamento são mostradas na Tabela 4.3. A Tabela 4.4 apresenta as diretivas mapeadas que se referem às estruturas de sincronização.

Diretivas de Região Paralela
`omp parallel`: provê paralelismo, com a réplica de um bloco de código entre *threads*
Diretivas de Divisão de Tarefas
`omp for`: provê paralelismo no nível de *loop*
`omp single`: impede a replica de código de uma região parelela, para ser executado por uma única *thread*

Tabela 4.3: Estruturas de Controle Paralelo Mapeadas

Diretivas de Sincronização
`omp master`: define um bloco de código para ser executado apenas pela *thread* responsável em criar outras
`omp barrier`: provê sincronização com barreira global entre um grupo de *threads*
`omp critical`: define um bloco de código para acesso exclusivo de *threads*

Tabela 4.4: Estruturas de Sincronização Mapeadas

De maneira geral, o mapeamento das estruturas de controle e de sincronização do OpenMP para Cilk se faz mediante a criação sucessiva de várias rotinas *Cilk*. Conforme exposto na Seção 3.1, toda diretiva do OpenMP é sucedida por um bloco estruturado que está condicionado as suas diretrizes. Para o mapeamento, as rotinas *Cilk* concentram blocos estruturados referentes a essas diretivas.

Em Cilk, existem dois tipos de sincronização. A sincronização do controle de fluxo, necessária para a coordenação das *threads* criadas antecipadamente, com a invocação de rotinas *Cilk*. E, a sincronização de dados para impor uma ordenação nas escritas efetuadas por aquelas *threads*. Ambos os tipos de sincronização ocorrem na invocação e retorno de uma rotina *Cilk*. Assim, um processo de **desmembramento de rotinas Cilk** foi utilizado para se manter intactos os pontos de sincronização do programa escrito com o OpenMP. E, ainda, para assegurar a correta execução do programa traduzido, no que diz respeito ao acesso coordenado às variáveis compartilhadas. Pelo processo, um ponto de sincronização, encontrado no código em OpenMP delimita o início ou término de uma rotina *Cilk*, no programa traduzido. Isto é, um ponto de sincronização e o outro que apareça imediatamente depois determinam o ínicio e fim da rotina *Cilk*, e o código presente entre ambos forma o escopo da rotina.

No **desmembramento de rotinas**, as rotinas *Cilk* criadas são designadas **rotinas desmembradas**. Para facilitar a identificação delas foi adotada uma nomenclatura. O nome destas rotinas é formado pelo prefixo *ompcilk*, seguido do nome da rotina original, onde o bloco estruturado pertencente a ela se encontra no código com o OpenMP, e sucedida por um número de identificação da rotina desmembrada. A rotina *"0"* é sempre aquela que contém inicialmente o bloco de código da estrutura de controle paralelo. As seguintes são nomeadas com o incremento de um. É importante notar que os **pontos de sincronização** (PSs) coordenam este processo de desmembramento e que entre as rotinas *Cilk* há sempre um PS.

O restante da seção aborda o mapeamento das diretivas listadas nas Tabelas 4.3 e 4.4. Para cada uma delas é apresentada uma breve descrição e o seu mapeamento, segundo o caso geral de uso da diretiva, ou de acordo com alguns casos particulares, onde incluem-se aqueles em que algumas cláusulas aparecem acompanhadas da diretiva.

4.2.1 Diretiva *omp parallel*

A diretiva *omp parallel* define que um bloco de código deve ser executado por múltiplas *threads*, determinando, assim, uma região paralela. Um identificador de início e de fim da diretiva é usado para delimitar o bloco.

No momento em que a *master thread* encontra uma diretiva *omp parallel*, ela cria um conjunto de *threads*. Este conjunto, que inclui a *master thread*, recebe uma cópia da região paralela que é executada concorrentemente.

O delimitador de fim da região possui uma barreira implícita que interrompe a execução de todas as *threads* criadas. As *threads* após sua execução atingem este ponto, onde aguardam o término de outras. Quando todas as *threads* alcançam a barreira são realizadas as operações relacionadas a coerência da memória compartilhada. Em seguida, apenas a *master thread* reassume a execução.

4.2.1.1 Caso Geral

Para o caso geral da diretiva, bem como para as demais mapeadas, é apresentada uma Tabela de mapeamento, onde na coluna da esquerda encontra-se a diretiva com a sintaxe do padrão OpenMP e na coluna direita a contrução equivalente em *Cilk*. Na Tabela, ainda é indicado, no canto superior esquerdo, se a diretiva está associada a um PS. Para a diretiva *omp parallel* é possível verificar o mapeamento realizado na Tabela 4.5.

Diretiva OpenMP [PS]	Construção Cilk

`#pragma omp parallel`
`{`
Bloco Estruturado
`}`
	`spawn ompcilk_rotina_0(...);`
	`sync;`
	`Cilk void ompcilk_rotina_0(...)`
	`{`
	Bloco Estruturado
	`}`

Tabela 4.5: Mapeamento Geral da Diretiva *omp parallel*

A região paralela é expressa pelo bloco estruturado da diretiva. Este bloco passa a ser compreendido pela rotina *Cilk*, denominada *ompcilk_rotina_0*[1]. A primitiva *sync* corresponde à barreira implícita no final da diretiva, garantindo a coerência de memória.

O bloco estruturado inserido na rotina *Cilk* referente a região paralela pode ser composto por várias diretivas OpenMP. Caso, sejam identificados PSs entre essas diretivas,

[1] *rotina* representa o mesmo nome da rotina do programa escrito com o OpenMP

ocorrem mais **desmembramentos de rotina**. Este fato ocasiona a separação do bloco estruturado com a criação de outras rotinas *Cilk*.

As reticências utilizadas na chamada e no cabeçalho da rotina desmembrada exprimem os parâmetros transmitidos, e referem-se às variáveis usadas na região que são declaradas externamente. A Seção 4.3 aborda o mapeamento do escopo de dados e descreve como mapear essas variáveis.

Um único desmembramento de rotina, denominado **desmembramento trivial**, é aquele no qual o escopo de uma rotina *Cilk* possui um fragmento de código do programa em OpenMP que não possui pontos de sincronização. Suponha que o bloco estruturado da tabela de mapeamento contenha diretivas sem pontos de sincronização. A região paralela deve ser mapeada com um único **desmembramento trivial**, onde todo o bloco é transportado para uma única rotina *Cilk*, como pode ser observado na Figura 4.6.

```
spawn ompcilk_rotina_0(...);
sync;

Cilk void ompcilk_rotina_0(...)  {
 int ompcilk_tk;
 for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)
 spawn ompcilk_rotina_1(ompcilk_tk, ...);
 sync;
}

Cilk void ompcilk_rotina_1(int ompcilk_tk, ...)  {
 Bloco Estruturado
}
```

Tabela 4.6: Desmembramento Trivial

De acordo com a Tabela 4.6, o *loop* na rotina *ompcilk_rotina_0* corresponde à criação de tarefas simultâneas para a execução, em paralelo, do bloco estruturado. Da mesma forma, o *loop* é executado concorrentemente por *threads* criadas no programa com o OpenMP. É possível notar que o identificador da rotina desmembrada referente à região paralela é acrescido de um. Como esta foi a primeira rotina desmembrada, ela é identificada como sendo a primeira. A variável *ompcilk_tk* corresponde ao número de identificação da tarefa.

A variável de ambiente *OMP_NESTED* define o comportamento de *threads* na ocorrência de alinhamento de diretivas *omp paralell*. Quando **verdadeiro**, as mesmas *threads* que executam uma região, também executam outra dentro dela. Quando **falso**, cada *th-*

read cria um subconjunto de *threads* para executar a região mais interna. Para suportar a o alinhamento de diretivas *omp paralell*, a criação de tarefas simultâneas é aplicada no *spawn*, observado na tabela, 4.6 que antecede a chamada da rotina *ompcilk_rotina_0*, por exemplo.

4.2.1.2 Caso Particular: combinada à diretiva *omp for*

Comumente, após uma diretiva *omp parallel* encontra-se uma diretiva de divisão de tarefas chamada *omp for*, cujo mapeamento é apresentado na Seção 4.2.2. Esta diretiva faz com que o grupo de *threads*, criado para a região paralela, execute o *loop* presente no bloco estruturado. As iterações do *loop* são divididas entre o grupo de *threads*. Cada *thread*, pode executar mais de uma iteração do *loop*. É importante salientar que o bloco da diretiva é estritamente composto pelo *loop* e que ele obedece à seguinte forma canônica:

```
for (index=start; index < end; expressão de incremento)
```

A variável *index* deve ser sempre do tipo inteiro. Outros operadores de comparação podem ser utilizados como condição de parada do *loop*. A restrição para as variáveis *start* e *end* é a de que sejam valores numéricos e que não se modifiquem a cada iteração do *loop*. E, finalmente, a expressão de incremento altera o valor do índice de acordo com um montante equivalente a cada iteração, através de um dos possíveis operadores de incremento da linguagem.

Diretiva OpenMP [PS]	Construção Cilk

`#pragma omp parallel for`
`for (i=...; i<...; i++)`
`{`
`...`
`}`
	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
	`spawn ompcilk_rotina_0(ompcilk_tk, ...);`
	`sync;`
	
	`Cilk void ompcilk_rotina_0(int ompcilk_tk, ...)`
	`{`
	divisão de tarefas estendida
	`for (i=...; i<...; i++)`
	`{`
	`...`
	`}`
	`}`

Tabela 4.7: Mapeamento da Diretiva *omp parallel* combinada à Diretiva *omp for*

A Tabela 4.7 mostra o mapeamento deste caso particular. Nela, o bloco estruturado definido pelo *loop* imediatamente depois da diretiva agrega um conjunto de operações executadas em paralelo. Estas operações são representadas na tabela por reticências.

De forma análoga, este mapeamento também faz uso do mecanismo de criação de tarefas simultâneas em Cilk, como se observa na coluna da direita.

No escopo da rotina *Cilk*, é realizada uma **divisão de tarefas estendida**. Esta divisão se refere à parte do código traduzido responsável por dividir as iterações do *loop* entre as *threads* de forma similar ao realizado com a diretiva *omp for* do OpenMP. Esta **divisão de tarefas estendida** é parte integrante do mapeamento da diretiva *omp for*, e é descrita na Seção 4.2.2.

4.2.1.3 Caso Particular: escopo de *loop* composto por Ponto de Sincronização

É possível encontrar dentro de uma diretiva *omp parallel* um *loop* que não é paralelo. Isto é, que não se inclui no escopo de uma diretiva *omp for*. Se este *loop* apresentar PSs no seu escopo, é aplicado o mapeamento mostrado na Tabela 4.8 a ele.

Diretiva OpenMP [PS]	Construção Cilk

`#pragma omp parallel`
`{`
`for (i=...; i<...; i++)`
`{`
Bloco Estruturado A
PS
Bloco Estruturado B
PS
`}`
`}`
	`spawn ompcilk_rotina_0(...);`
	`sync;`
	`Cilk void ompcilk_rotina_0(...)`
	`{`
	`for (i=...; i<...; i++)`
	`{`
	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
	`spawn ompcilk_rotina_1(omcilk_tk, ...);`
	`sync;`
	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
	`spawn ompcilk_rotina_2(ompcilk_tk, ...);`
	`sync;`
	`}`
	`}`

Tabela 4.8: Mapeamento da Diretiva *omp parallel* com Escopo de *Loop* composto por PSs

Para a demonstração deste caso foram utilizados dois PSs dentro do *loop* da região paralela. O mesmo mapeamento é respeitado mesmo que apenas um PS seja identificado. Os PSs ilustrados podem ser encontrados pela presença de diretivas de sincronização, ou de outras diretivas cujo término é definido por um PS. Em ambas as situações, o mapeamento se faz com a permanência do *loop* que agrega os blocos, A e B, na rotina desmembrada associada à região paralela. E, também, com a criação de tarefas simultâneas para as demais rotinas desmembradas, 1 e 2. Para o mapeamento, cada bloco estruturado intercalado por PSs deve compor o escopo de uma rotina desmembrada.

4.2.2 Diretiva *omp for*

Dentro de uma região paralela é possível explorar o paralelismo de maneira que *threads* recebam diferentes porções de uma estrutura de dados compartilhada ou que sejam atribuídas tarefas distintas a elas. A diretiva *omp for* atende ao caso em que uma estrutura compartilhada é particionada entre *threads*.

Como citado anteriormente, trata-se de uma diretiva complementar à diretiva *omp parallel* que explora o estilo de paralelismo SPMD. Pela diretiva, as iterações de um *loop* são divididas entre as *threads* criadas antecipadamente ao se executar a diretiva *omp parallel*.

No padrão OpenMP existem diferentes tipos de escalonadores de *loop* responsáveis pela divisão de tarefas realizada com a diretiva *omp for*. Aquele que opera como *default* é dependente da implementação do padrão OpenMP considerada. O compilador da *Intel*, por exemplo, utiliza o método estático simples. Nele cada *thread* recebe a mesma quantidade de iterações. Esta quantidade é calculada simplesmente pela divisão do total de iterações pela quantidade de *threads* existentes.

4.2.2.1 Caso Geral

O mapeamento da diretiva *omp for* assume a divisão de tarefas empregada pelo escalonador estático simples. Isto porque para a validação da tradução proposta foi realizada com o compilador da *Intel*, como especificado no capítulo 5. O cálculo de divisão de tarefas usada por este escalonador é expresso, em Cilk, com a programação explícita do cálculo no nível de aplicação. O cálculo efetuado, bem como, as variáveis utilizadas, por convenção, na programação do escalonador, podem ser verificadas na Tabela 4.9.

A Tabela 4.9 mostra na parte superior a forma canônica do *loop* encontrado no código com o OpenMP. Enquanto, na parte inferior da tabela, localiza-se o trecho de código usado para a programação do escalonador de *loop*. Note que o operador de comparação do *loop*, localizado na forma canônica, é preservado na tradução com a divisão de tarefas estendida.

Verifica-se que são criadas algumas variáveis. Dentre elas, *ompcilk_desloc* que recebe sempre o valor de início do índice do *loop* (*start*). À variável *ompcilk_limmax* foi atribuído o valor "0", já que o operador de comparação mostrado na forma canônica foi o sinal de menor (<). Esta variável recebe o valor "1", toda vez que o operador de comparação é expresso pelo sinal de menor ou igual (<=), ou, maior ou igual (>=). Pela variável *ompcilk_pendingtks* verifica-se se o total de iterações é divisível pela quantidade

Forma Canônica do Loop
`for (index=start; index < end; expressão de incremento)`
Divisão de Tarefas Estendida

`ompcilk_desloc =start`
`ompcilk_limmax = 0`
`ompcilk_totiters = (end) - ompcilk_desloc + ompcilk_limmax`
`ompcilk_btask = ( ompcilk_tk * (ompcilk_totiters/OMPCILK_TOT_TKS) ) + ompcilk_desloc`
`ompcilk_etask = ompcilk_btask + (ompcilk_totiters/OMPCILK_TOT_TKS) - ompcilk_limmax;`
`ompcilk_pendingtks = ompcilk_totiters -`
`- (ompcilk_totiters/OMPCILK_TOT_TKS) * OMPCILK_TOT_TKS;`
`if (ompcilk_tk == OMPCILK_TOT_TKS - 1)`
`{`
`if (ompcilk_pendingtks != 0)`
`ompcilk_etask = ompcilk_etask + ompcilk_pendingtks;`
`}`
`for (index=ompcilk_btask; index < ompcilk_etask; expressão de incremento)`
`...`

Tabela 4.9: Mapeamento da Diretiva *omp parallel* com Escopo de Loop composto por PS's

de *threads* utilizadas.

Esta divisão programada de forma explícita, passa a ser realizada por cada tarefa Cilk encarregada de executar o *loop*. Para distingüir a quantidade de iterações de cada tarefa é usado seu próprio identificador (*ompcilk_tk*), atribuído na criação de tarefas simultâneas. O mapeamento pode ser visto na Tabela 4.10.

Diretiva OpenMP [PS]	Construção Cilk

`#pragma omp parallel`
`{`
`#pragma omp for`
`for (i=...; i<...; i++)`
`{`
`...`
`}`
`}`
	`spawn ompcilk_rotina_0(...);`
	`sync;`
	`Cilk void ompcilk_rotina_0(...)`
	`{`
	`int ompcilk_tk;`
	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
	`spawn ompcilk_rotina_1(ompcilk_tk, ...);`
	`sync;`
	`}`
	`Cilk void ompcilk_rotina_1(int ompcilk_tk, ...)`
	`{`
	`for (i=...; i<...; i++)`
	`{`
	`...`
	`}`
	`}`

Tabela 4.10: Mapeamento Geral da Diretiva *omp for*

4.2.2.2 Caso Particular: Acompanhada da Cláusula *nowait*

Quando a cláusula *nowait* aparece imediatamente após à diretiva *omp for* a barreira implícita, embutida no final da diretiva, não é executada pelas *threads*. Desta forma, ao finalizar suas iterações, a *thread* prossegue sua execução, sem ter de esperar que as outras *threads* terminem as iterações que lhe foram destinadas.

Considerando o caso de uma diretiva *omp for*, sucedida da cláusula *nowait*, e ainda de outra diretiva provida por um PS, o mapeamento é realizado como mostra a Tabela 4.11.

Diretiva OpenMP []	Construção Cilk

`#pragma omp parallel`
`{`
PS
`#pragma omp for nowait`
`for (i=...; i<...; i++)`
`{`
`...`
`}`
 	
Bloco Estruturado
PS
`}`
	`spawn ompcilk_rotina_0(...);`
	`sync;`
	
	`Cilk void ompcilk_rotina_0(...)`
	`{`
	`int ompcilk_tk;`
	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
	`spawn ompcilk_rotina_1(ompcilk_tk, ...);`
	`sync;`
	`}`
	`Cilk void ompcilk_rotina_1(int ompcilk_tk, ...)`
	`{`
	divisão de tarefas extendida
	`for (i=ompcilk_btask; i<ompcilk_etask; i++)`
	`{`
	`...`
	`}`
	Bloco Estruturado
	`}`

Tabela 4.11: Mapeamento da Diretiva *omp for* acompanhada da cláusula *nowait*

Ao invés de se criarem duas rotinas *Cilk* para concentrar, respectivamente, o bloco estruturado da primeira e da segunda diretiva, apenas uma rotina *Cilk* é criada. Não é preciso gerar outro desmembramento de rotina, pois como a barreira implícita é ignorada, as mesmas *threads* que executam a primeira diretiva podem executar a segunda.

4.2.2.3 Caso Particular: Acompanhada da Cláusula *nowait* e presente em Bloco Estruturado Adjunto

Algumas vezes é possível identificar um alinhamento de diretivas em OpenMP, onde um bloco estruturado aparece imediatamente antes, e outro bloco imediatamente depois da diretiva *omp for*. Desta forma, observa-se uma sequência de três blocos consecutivos. O alinhamento caracteriza-se ainda pela inexistência de PSs entre os blocos estruturados, e pela presença da cláusula *nowait*, que acompanha a diretiva *omp for*.

No contexto deste trabalho, este alinhamento forma um **bloco estruturado adjunto**, como pode ser visto na Tabela 4.12.

Diretiva OpenMP []	Construção Cilk

`#pragma omp parallel`
`{`
PS
Bloco Estruturado A
`#pragma omp for nowait`
`for (i=...; i<...; i++)`
`{`
`...`
`}`
Bloco Estruturado B
PS
`}`
	`spawn ompcilk_rotina_0(...);`
	`sync;`
	`Cilk void ompcilk_rotina_0(...)`
	`{`
	`int ompcilk_tk;`
	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
	`spawn ompcilk_rotina_1(ompcilk_tk, ...);`
	`sync;`
	`}`
	`Cilk void ompcilk_rotina_1(int ompcilk_tk, ...)`
	`{`
	Bloco Estruturado A
	divisão de tarefas extendida
	`for (i=ompcilk_btask; i<ompcilk_etask; i++)`
	`{`
	`...`
	`}`
	Bloco Estruturado B
	`}`

Tabela 4.12: Mapeamento da Diretiva *omp for* acompanhada da cláusula *nowait* e presente em **bloco estruturado adjunto**

De acordo com a Tabela 4.12, ambos os blocos estruturados A e B são introduzidos na rotina que concentra o *loop* pertencente à diretiva *omp for*.

Ocasionalmente, mais diretivas *omp for* acompanhadas da cláusula *nowait*, ou genéricamente, outros trechos de código não seguidos por PSs, interceptam os blocos A e B ilustrados. Esses trechos de código devem ser integrados ao bloco adjunto. Para isto, a ordem em que aparecem no programa em OpenMP deve ser preservada. Esta compactação torna-se interessante, pois reduz a quantidade de desmembramentos de rotina.

4.2.3 Diretiva *omp single*

Um bloco estruturado contido em uma região paralela, em certos casos, não deve ser replicado ou compartilhado entre as *threads*. É desejável que este bloco seja executado por uma única *thread*. Para suprir esta necessidade o padrão OpenMP possui a diretiva *single*.

A diretiva é considerada um caso derivado de divisão de tarefas onde todo o processamento é realizado por uma única *thread*. Apesar das demais *threads* não participarem da execução, a diretiva pertence à classe de diretivas de **divisão de tarefas** por dois motivos. Primeiro, por estar condicionada ao fato de que a diretiva deve ser alcançada por todas as *threads* de um grupo. Segundo, porque, no final da diretiva, a execução de todas as *threads* é interrompida por uma barreira implícita.

A *thread* responsável em executar o bloco estruturado que acompanha a diretiva é dependente de implementação. Cada implementação possui a sua heurística na escolha da *thread*. As demais *threads* do grupo são bloqueadas na barreira implícita, no final da diretiva, até o término do bloco estruturado correspondente ser executado.

4.2.3.1 Caso Geral

O mapeamento geral da diretiva *omp single* é apresentado na Tabela 4.13.

Diretiva OpenMP [PS]	Construção Cilk
`#pragma omp parallel`	`spawn ompcilk_rotina_0(...);`
`{`	`sync;`
`#pragma omp single`	`Cilk void ompcilk_rotina_0(...)`
Bloco Estruturado	`{`
`}`	*Bloco Estruturado*
	`}`

Tabela 4.13: Mapeamento Geral da Diretiva *omp single*

O bloco da diretiva *single* é inserido na própria rotina *Cilk* criada para a região paralela. No modelo do Cilk, as *threads* relacionam-se entre si. As *threads-filhas* devem enviar as modificações realizadas em variáveis para a sua *thread-pai*. Para o mapeamento, foi selecionada a própria *thread-pai* para executar o bloco agregado à diretiva. Considerando a existência, em Cilk, do *overhead* com a criação e sincronização de *threads-filhas*, a opção pela *thread-pai* foi motivada por não contribuir para o aumento do *overhead*.

4.2.3.2 Caso Particular: Acompanhada da Cláusula *nowait* e pertencente a um Bloco Estruturado Adjunto

Neste caso a diretiva possui a cláusula *nowait*, e compõe um **bloco estruturado adjunto.**

A Tabela 4.14 mostra o mapeamento para este caso. Ao contrário do caso geral, o bloco estruturado associado a diretiva é executado pela *thread-filha* de identificação "0", e não a *thread-pai.* Isto porque a execução pela *thread-pai* ocasionaria mais um desmembramento de rotina e no *Cilk* o desmembramento gera pontos de sincronização.

Diretiva OpenMP []	Construção Cilk
`#pragma omp parallel`	`spawn ompcilk_rotina_0(...);`
`{`	`sync;`
PS	`Cilk void ompcilk_rotina_0(...){`
Bloco Estruturado A	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
`#pragma omp single nowait`	`spawn ompcilk_rotina_1(ompcilk_tk, ...);`
Bloco Estruturado B	`sync;`
	`}`
Bloco estruturado C	`Cilk void ompcilk_rotina_1(int ompcilk_tk, ...)`
PS	`{`
`}`	*Bloco Estruturado A*
	`if (id_task == 0)`
	Bloco Estruturado B
	Bloco estruturado C
	`}`

Tabela 4.14: Mapeamento da Diretiva *omp single* acompanhada da cláusula *nowait* e pertencente a um Bloco Estruturado Adjunto

4.2.4 Diretiva *omp master*

Na especificação 2.5 do OpenMP [1], a diretiva *omp master* não pertence à classe de sincronização. A diretiva *omp master* é usada para identificar um bloco estruturado incluído na região paralela que deve ser executado apenas pela *master thread.* Ela se distingüe das diretivas de divisão de tarefas por não possuir, por definição, a barreira implícita no final da diretiva. Por esse motivo, não é cabível o emprego da cláusula *nowait* acompanhada da diretiva.

4.2.4.1 Caso Geral

A implementação da diretiva *omp master* consiste apenas na utilização de uma estrutura condicional. Ela verifica se a *thread* corrente é a de identificação "0". Este número de identificação é sempre atrbuído à *master thread.*

A *master* associa-se à *thread-pai*, em Cilk, uma vez que a *thread-pai* é responsável por criar as outras *threads* que executarão uma região paralela. Assim, é apropriado manter o bloco estruturado relativo à diretiva *master* no escopo da rotina referente à região paralela, antes mesmo da criação de tarefas simultâneas, como mostra a Tabela 4.15.

Diretiva OpenMP []	Construção Cilk
`#pragma omp parallel`	`spawn ompcilk_rotina_0(...);`
`{`	`sync;`
`#pragma omp master`	`Cilk void ompcilk_rotina_0(...)`
Bloco Estruturado	`{`
`}`	*Bloco Estruturado*
	`}`

Tabela 4.15: Mapeamento Geral da Diretiva *omp master*

Observe que não há distinção entre o mapeamento da diretiva *master* e o mapeamento da diretiva *single*. Em ambos os casos, a alternativa mais apropriada foi atribuir à *thread-pai* a execução do bloco estruturado relacionado às diretivas.

4.2.4.2 Caso Particular: Pertencente a um Bloco Estruturado Adjunto

A diretiva pode, juntamente com outros trechos de código, vir a formar um **bloco estruturado adjunto**. Por definição, o término da diretiva não possui uma barreira implícita. Desta maneira, a diretiva não depende da cláusula *nowait* para compor o **bloco estruturado adjunto**. Isto faz com que a chance de que a diretiva faça parte do bloco aumente, em relação a outras diretivas que necessitam da cláusula *nowait* para estarem inseridas em um bloco adjunto.

Neste caso particular, é proposto o mapeamento da Tabela 4.16.

De forma equivalente ao caso particular da diretiva *omp single*, a verificação da identificação da tarefa é realizada dentro da rotina desmembrada.

Diretiva OpenMP []	Construção Cilk
`#pragma omp parallel`	`spawn ompcilk_rotina_0(...);`
`{`	`sync;`
PS	
Bloco Estruturado A	`Cilk void ompcilk_rotina_0(...)`
`#pragma omp master`	`{`
Bloco Estruturado B	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
	`spawn ompcilk_rotina_1(ompcilk_tk, ...);`
Bloco estruturado C	`sync;`
PS	`}`
`}`	
	`Cilk void ompcilk_rotina_1(int ompcilk_tk, ...)`
	`{`
	Bloco Estruturado A
	`if (id_task == 0)`
	Bloco Estruturado B
	Bloco estruturado C
	`}`

Tabela 4.16: Mapeamento da Diretiva *omp single* acompanhada da cláusula *nowait* e pertencente a um Bloco Estruturado Adjunto

4.2.5 Diretiva *omp barrier*

Por ser uma diretiva de sincronização, a diretiva *omp barrier* permite a coordenação das *threads* que executam uma região paralela. Quando cada *thread*, criada no início da região, atinge a barreira, ela espera a chegada de todas as outras. Somente após o término da barreira cada *thread* retoma a sua execução.

4.2.5.1 Caso Geral

O mapeamento da diretiva *omp barrier* é apresentado na Tabela 4.17.

Diretiva OpenMP [PS]	Construção Cilk

`#pragma omp parallel`
`{`
Bloco Estruturado A
`#pragma omp barrier`
Bloco Estruturado B
`{`
	`spawn ompcilk_rotina_0(...);`
	`sync;`
	`Cilk void ompcilk_rotina_0(...)`
	`{`
	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
	`spawn ompcilk_rotina_1(ompcilk_tk, ...);`
	`sync;`
	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
	`spawn ompcilk_rotina_2(ompcilk_tk, ...);`
	`sync;`
	`{`

Tabela 4.17: Mapeamento Geral da Diretiva *omp barrier*

A Tabela 4.17 mostra dois blocos estruturados separados por uma diretiva de barreira *omp barrier*. Ambos os blocos podem estar condicionados ou não a diretivas do OpenMP. O escopo da rotina *ompcilk_rotina_1* agrega o *Bloco Estruturado A*.

Com a presença da barreira ocorre outro desmembramento de rotina. Para isto, são geradas mais tarefas simultâneas, após o término das primeiras criadas. O *Bloco Estruturado B* pertence, assim, ao escopo da segunda rotina. Apesar de não estarem apresentadas na tabela, as rotinas de identificação 1 e 2, fazem parte do mapeamento, na conversão para Cilk.

Não foi encontrado nenhum caso particular no emprego desta diretiva considerando as aplicações do *benchmark* utilizadas para o mapeamento da diretiva.

4.2.6 Diretiva *omp critical*

A diretiva *omp critical* provê exclusão mútua na execução de um bloco estruturado encapsulado no escopo da diretiva.

Ao encontrar esta diretiva, determinada *thread* aguarda até que nenhuma outra *thread* esteja executando a seção crítica, para que ela tenha acesso exclusivo à seção. Ou, caso não existam *threads* na seção, ela poderá entrar imediatamente. Após a completa execução do bloco, a *thread* libera o acesso e sinaliza a liberação para eventuais *threads* em espera.

4.2.6.1 Caso Geral

O mecanismo de exclusão mútua, proveniente da linguagem Cilk, é implementado através de instruções em *assembly* alinhadas na rotina em C onde é exercida a chamada. O uso destas instruções seguem o formato GNU. No entanto, este formato não é suportado pelo compilador da *Intel* para a plataforma *Itanium* [4], onde o mapeamento proposto foi validado. Por esta razão, a estratégia de exclusão mútua inerente ao Cilk não pôde ser aplicada, o que inviabilizou, por um lado, o mapeamento da diretiva *omp critical* para Cilk.

Apesar da restrição das instruções em *assembly*, foi observado que, nas aplicações do *benchmarck* NAS onde a diretiva *omp critical* é usada, a seção crítica é empregada apenas para a redução de variáveis. Quando isto ocorre a diretiva *omp critical* pode ser traduzida a partir da mesma solução encontrada para a cláusula *reduction*. Assim, o mapeamento desta diretiva é apresentado na Seção 4.3.2, referente à cláusula *reduction*.

4.2.7 Subprogramas

Os subprogramas, neste contexto, se referem a rotinas invocadas no escopo de outras. Apesar de não se caracterizarem como diretivas, estes subprogramas devem ser cuidadosamente analisados na tradução de uma aplicação para Cilk. Sobretudo no que diz respeito a sua invocação. Esta seção dedica-se ao mapeamento aplicado em relação às chamadas destes subprogramas.

4.2.7.1 Caso Geral

Como pode ser visto na Tabela 4.18, existem duas alternativas para o mapeamento do caso geral. A escolha por uma delas está relacionada à existência ou não de, ao menos, uma

diretiva de região paralela na cadeia de chamadas iniciada pelo primeiro subprograma.

Construção OpenMP []	Construção Cilk Alternativa A	Alternativa B

`void main(...)`
`{`
`rotina(...);`
`}`
	`Cilk void main(...)`
	`{`
	`rotina(...);`
	`}`
		`Cilk void main(...)`
		`{`
		`spawn rotina(...);`
		`sync;`
		`}`

Tabela 4.18: Mapeamento Geral da Construção com Subprogramas

Quando um subprograma é encontrado, o escopo que o define deve ser verificado para saber se ocorrem diretivas de regiões paralelas. Se alguma diretiva de paralelização for encontrada a alternativa de mapeamento indicada corresponde à B. Caso contrário, a pilha de chamadas desencadeada, a partir do subprograma em questão, deve ser percorrida. É necessário identificar a última chamada de rotina e verificar o seu escopo, para a identificação de alguma diretiva de região paralela. Se, durante a busca, alguma diretiva de região paralela for encontrada, a alternativa para o mapeamento recomendada continua sendo a B. Porém, se a pilha tiver sido percorrida até o seu final e nenhuma diretiva do tipo foi identificada, é aconselhado o mapeamento descrito pela alternativa A.

4.2.7.2 Caso Particular: Localizados em Extensão Dinâmica em presença de Blocos Estruturados condicionados a Diretivas

A região paralela associada a uma estrutura de controle paralelo é formada pela sua extensão dinâmica (Seção 3.1). Esta extensão é composta pela parte estática, presente diretamente no escopo da região, e por subrotinas, ou subprogramas, chamados na região.

Quando um subprograma está presente na extensão dinâmica de uma região paralela, e no escopo dele verifica-se a existência de diretivas de divisão de tarefas ou de sincronização, o mapeamento proposto é o apresentado na Tabela 4.19.

Neste caso, a *rotina_A* é aquela que concentra a estrutura de controle paralelo. A *rotina_B* refere-se ao subprograma localizado na extensão dinâmica da região, cujo escopo é formado por alguma outra diretiva do OpenMP. Na coluna da direita, onde é apresentado o mapeamento, observa-se a presença de uma reticências no escopo da *rotina_B*. Ela representa o mapeamento adotado para a respectiva diretiva. Na realidade, neste caso particular, o escopo do subprograma deve ser cuidadosamente analisado para que sejam identificadas diretivas de forma a se utilizar o mapeamento correto para cada uma delas.

Diretiva OpenMP []	Construção Cilk

`#pragma omp parallel`
`{`
`rotina_B(...);`
`}`
 	
`void rotina_B(...)`
`{`
`#pragma omp ...`
Bloco Estruturado
`}`
	`spawn ompcilk_rotina_A_0(...);`
	`sync;`
	`Cilk void ompcilk_rotina_A_0(...)`
	`{`
	`spawn rotina_B(...);`
	`sync;`
	`}`
	`Cilk void rotina_B(...)`
	`{`
	`...`
	`}`

Tabela 4.19: Mapeamento Geral da Construção com Subprogramas localizados em Extensão Dinâmica em presença de Blocos Estruturados condicionados a Diretivas

4.2.7.3 Caso Particular: Localizados em Extensão Dinâmica em presença de Blocos Estruturados não condicionados a Diretivas

O mapeamento apresentado, à seguir, corresponde ao emprego particular de subprogramas, onde, ao contrário do caso expresso na Seção 4.2.7.2, os blocos estruturados do escopo do subprograma não são regidos por diretivas do OpenMP.

Diretiva OpenMP []	Construção Cilk

`#pragma omp parallel`
`{`
`rotina_B(...);`
`}`
 	
`void rotina_B(...)`
`{`
Bloco Estruturado
`}`
	`spawn ompcilk_rotina_A_0(...);`
	`sync;`
	`Cilk void ompcilk_rotina_A_0(...)`
	`{`
	`for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)`
	`spawn rotina_B(...);`
	`sync;`
	`}`
	`Cilk void rotina_B(...)`
	`{`
	Bloco Estruturado
	`}`

Tabela 4.20: Mapeamento Geral da Construção com Subprogramas localizados em Extensão Dinâmica em presença de Blocos Estruturados não condicionados a Diretivas

Segundo a Tabela 4.20, a *rotina_B* é chamada através da criação de tarefas simultâneas, uma vez que ela se encontra no escopo de uma região paralela.

4.3 Mapeamento das Estruturas de Controle de Escopo de Dados e Comunicação

Diferente de um programa escrito em uma linguagem de programação puramente seqüencial, nas aplicações escritas com o padrão OpenMP, o termo **escopo de uma variável** é usado para designar se uma variável é privativa ou compartilhada. Desta forma, se uma variável é privativa, uma cópia distinta dela é atribuída para cada *thread*. Se o escopo de uma variável é compartilhado, significa que um conjunto de *threads* compartilham uma única cópia da variável. Assim, dois tipos de escopo de dados básicos são definidos na especificação: **escopo privativo** e **escopo compartilhado**.

Estabeleceu-se um mapeamento para ambos os escopos em Cilk, considerando o mecanismo de suporte à memória compartilhada e privativa do Cilk.

A especificação do escopo de dados, definida pelo OpenMP, provê uma coletânea de regras para a determinação de atributos de compartilhamento para variáveis referenciadas em regiões paralelas. Estes atributos de compartilhamento as identificam como compartilhadas ou privativas. De acordo com a especificação, estas regras variam se a variável é referenciada em diretivas complementares às de região paralela, ou seja, naquelas de divisão de tarefas ou de sincronização. Ou, se é referenciada em uma região, mas fora de uma diretiva complementar. Estas regras demonstraram-se úteis na tradução de um código escrito com o OpenMP, na medida que facilitaram a atribuição do escopo destas variáveis em Cilk.

A seguir é apresentado inicialmente o mapeamento proposto do escopo privativo e compartilhado em Cilk. Uma abordagem utilizada para evitar a modificação intensa no acesso de algumas variáveis com o mapeamento, é apresentada em seguida. Posteriormente, é descrito o mapeamento que se aplica a variáveis referenciadas dentro e fora de uma diretiva complementar às de região paralela.

4.3.1 Escopo Compartilhado e Privativo

Em Cilk, uma variável torna-se compartilhada quando uma variável global é acessada por rotinas que operam em paralelo. Ou, ainda, através da passagem de ponteiros para rotinas criadas com a primitiva *spawn*, o que permite que mais de uma rotina acesse o objeto endereçado pelo ponteiro.

Assim, em relação ao escopo compartilhado, existem duas possibilidades de mapeamento. Caso a variável seja identificada como global, ela é mantida como global, no

programa em Cilk. Caso a variável tenha seu escopo modificado para compartilhado, ela recebe outro mapeamento.

As variáveis identificadas como globais recebem como mapeamento uma **instância global do mapeamento compartilhado**. Analogamente ao Cilk, toda variável global, em OpenMP, possui escopo compartilhado, salvo as variáveis encontradas em uma diretiva *threadprivate*. Deste modo, nada é alterado no acesso à variável.

Uma variável mapeada como uma **instância local do mapeamento compartilhado** possui seu escopo modificado no programa OpenMP. Ela é inicialmente identificada como privativa, por estar declarada dentro do escopo de uma rotina. Mas, devido ao fato de ser utilizada dentro de uma região paralela, ou ter seu escopo explícitamente modificado para compartilhado, a variável torna-se compartilhada. Neste caso em particular, o tipo de acesso à variável é alterado. Isto porque a variável mapeada é incluída em um registro que concentra todas as variáveis compartilhadas de uma rotina *Cilk*. Para que a área de memória ocupada pelo registro seja compartilhada, um ponteiro é alocado para ele e passado como argumento a todas as subrotinas que utilizem uma das variáveis que o registro concentra.

No Cilk, um ponteiro alocado na pilha pode ser passado de forma segura apenas para as rotinas, cujas *threads* que as executam tornam-se filhas. Somente a alocação na *heap* é capaz de assegurar a correta transmissão do ponteiro também no sentido inverso, ou seja, de uma rotina *filha* para a sua rotina *pai*. Desta forma, o ponteiro do registro utilizado para uma **instância local do mapeamento compartilhado** é alocado na área de *heap*.

Por convenção, o nome do registro usado por uma **instância local do mapeamento compartilhado** é formado pelo prefixo *OMPCILK*, seguindo do termo *SHD* (de *shared*), e concatenado ao nome da rotina em que é declarado. O ponteiro associado ao registro recebe o mesmo nome, com todos os termos em letra minúscula. Isenta-se da inclusão no registro, as variáveis já declaradas como ponteiros no programa com o OpenMP. Em relação a estas variáveis, basta que sejam alocadas na área de *heap* e utilizadas na passagem por parâmetro, como tradicionalmente feito em C.

Para o escopo privativo dos dados em OpenMP, o mapeamento consiste apenas na criação de uma instância local da variável em cada rotina onde a mesma é usada como privativa. Este mapeamento recebe o nome de **instância local do mapeamento privativo**. A solução, entretanto, é deficiente para alguns casos, onde a mesma instância da variável, apesar de privativa, deve existir em várias rotinas *Cilk*. Devido ao escalonamento do *Cilk*, uma variável privativa escrita por uma *thread* em uma rotina pode ser lida por outra

thread, possivelmente em um processador distinto, em outra rotina *Cilk*. Assim, para que a variável seja visível, é necessária a sua alocação na área de memória compartilhada. As variáveis que podem existir em várias rotinas *Cilk* e cujo escopo deve ser mantido como privativo são:

- aquelas que sofrem redução;
- aquelas subordinadas à diretiva threadprivate;
- aquelas que, por intermédio do mecanismo de mapeameamento das estruturas de controle paralelo e de sincronização, e não por particularidades do OpenMP, são afetadas pelo processo de **desmembramento de rotinas**.

As variáveis listadas anteriormente devem portanto persistir em várias rotinas com a definição do escopo compartilhado, e ao mesmo tempo serem utilizadas como privativas. Para contemplar essas variáveis, um mapeamento distinto foi introduzido com a terminologia de **instância global do mapeamento privativo**. O mapeamento é definido pela criação de um vetor compartilhado cujo tamanho é equivalente ao número total de *threads* que executam simultâneamente uma região paralela. No programa em Cilk o número total de *threads* é simbolizado pela constante *OMPCILK_TOT_TKS*. A i-ésima posição do vetor é escrita/lida pela tarefa, em Cilk, de identificação i, que possui acesso restrito às demais posições do vetor. Isto garante, por si só, o acesso privativo à variável. No entanto, para torná-la persistente nas rotinas *Cilk*, o vetor é adicionado ao registro compartilhado da rotina onde é declarado, juntamente com as variáveis mapeadas como uma **instância local do mapeamento compartilhado**.

4.3.1.1 A Abordagem Conservativa

Toda variável mapeada como uma **instância local do mapeamento compartilhado** é referenciada de forma simples, em um código com o OpenMP. Com a tradução cada ocorrência da variável necessita de modificação para que seja referenciada a partir do registro a que faz parte. Uma adaptação semelhante é exigida para uma **instância global do mapeamento privativo** devido à criação do vetor compartilhado associado ao mapeamento.

A abordagem conservativa é uma noção introduzida para o mapeamento das estruturas de dados que surge para facilitar a tradução para Cilk, sem ter que intervir concomitantemente no código. Pela abordagem, são mantidas exatamente as mesmas instâncias escalares das variáveis, como declaradas no programa com o OpenMP.

No caso das **instâncias locais do mapeamento compartilhado**, identifica-se a primeira rotina em Cilk invocada, para a qual o ponteiro do registro compartilhado é passado como argumento. Todas as escritas e leituras da variável que antecedem a chamada desta rotina, são consolidadas por meio da instância escalar conservada pela abordagem. Antes ainda que a rotina seja invocada, a instância da variável presente no registro é inicializada com o valor da instância escalar.

Quando uma **instância global do mapeamento privativo** é atribuída como mapeamento de uma variável encontrada na diretiva *threadprivate*, a abordagem conservativa é aplicada também. Todas a escritas e leituras, que ocorrem antes de uma região parela onde a variável é usada, são consolidadas com a instância escalar da variável.

4.3.2 Variáveis referenciadas dentro de Diretivas Complementares

Para as variáveis referenciadas nas diretivas de divisão de tarefas ou de sincronização, a especificação OpenMP 2.5 [1] aponta um dos três critérios de determinação de atributos de compartilhamento: **pré-determinado, determinado explícitamente** e **determinado implícitamente**. Esta seção indica o mapeamento do escopo de dados provido na Seção 4.3.1 perante os três critérios de determinação do atributo.

A Tabela 4.21 resume o mapeamento das variáveis cujo atributo de compartilhamento é pré-determinado.

Variável com Atributo Pré-Determinado	Mapeamento em Cilk
Presentes em diretiva *threadprivate*	*Instância Global do Mapeamento Privativo*
Declaradas no escopo da diretiva complementar com duração de armazenamento automático	*Instância Local do Mapeamento Privativo*
Alocadas na *heap*	*Instância Local do Mapeamento Compartilhado*
Membro de dados estático	*Instância Local do Mapeamento Compartilhado*
Indexação de loop regulado por diretiva de divisão de tarefas	*Instância Local do Mapeamento Privativo*

Tabela 4.21: Mapeamento de variáveis com atributo de compartilhamento pré-determinado

Em particular, é atribuída às variáveis identificadas em uma diretiva *threadprivate* uma **instância global do mapeamento privativo**, como mencionado.

De forma geral, uma **instância global do mapeamento privativo** é aplicada sempre que uma variável é escrita em uma rotina *Cilk* e imediatamente lida em outra rotina *Cilk*. Isto deve ser observado com cautela em relação às variáveis declaradas em uma rotina

OpenMP, que sofre desmembramento com a tradução para Cilk. Se a varíavel é sempre escrita no primeiro acesso a ela, em cada rotina desmembrada, uma **instância local do mapeamento privativo** é suficiente. O mesmo mapeamento é atribuído à variável que é escrita apenas na rotina original em OpenMP que desencadeou os desmembramentos de rotina, e somente lida em cada uma das rotinas desmembradas.

Como as variáveis vinculadas a uma diretiva *threadprivate* são comumente escritas ao longo do programa com o OpenMP é atribuída uma **instância global** a elas. O mapeamento destas variáveis segue a **abordagem conservativa**. Uma instância escalar conservada pela abordagem associa-se à cópia da variável da *master thread.* Assim, imediatamente antes da chamada da rotina correspondente à região paralela, no código em Cilk, o valor desta instância é conferido à posição *"0"* do vetor compartilhado originado pelo mapeamento.

Uma cláusula não diretamente relacionada ao escopo de dados, mas que é utilizada com a diretiva *threadprivate*, é a cláusula *copyin.* Esta cláusula implica na inicialização das cópias das *threads* de um grupo com o valor atribuído à cópia da *master thread.*

Para mapear a cláusula *copyin*, a inicialização do vetor compartilhado do mapeamento segundo uma **instância global do mapeamento privativo** é realizada com a atribuição do valor da posição *"0"* do vetor às demais posições. Desta maneira, é reproduzida a inicialização das cópias das demais *threads* de um grupo, no programa em OpenMP.

As variáveis referenciadas em diretivas complementares possuem seu atributo de compartilhamento determinado explícitamente quando elas são listadas em cláusulas de escopo que acompanham a diretiva.

Através das cláusulas de escopo, uma variável pode ter seu escopo alterado inúmeras vezes, de acordo com o tipo que recebe em cada região paralela. Assim, uma mesma variável pode receber o escopo privativo em uma região. Se presente na lista de outra cláusula definida para uma região subsequente, pode tornar-se compartilhada.

A Tabela 4.22 apresenta o mapeamento das variáveis com atributo determinado explícitamente segundo as cláusulas de escopo de dados em que se encontram.

O mapeamento mostrado na tabela para uma variável presente em uma cláusula *shared* só tem efeito na ocorrência de modificação do escopo da variável. Assim, se a variável sendo mapeada foi identificada como compartilhada previamente, o mapeamento atribuído antes, através de uma **instância global** ou **local do mapeamento compartilhado**, permanece inalterado. Porém, se a variável tiver recebido um mapeamento privativo anteriormente, com a conversão do escopo é criada uma outra instância para ela. Esta

Variável com Atributo Determinado Explícitamente (em cláusula)	Mapeamento em Cilk
shared	*Instância Local do Mapeamento Compartilhado*
private	*Instância Local do Mapeamento Privativo ou Instância Global do Mapeamento Privativo*
firstprivate	*Instância Local do Mapeamento Privativo ou Instância Global do Mapeamento Privativo*
lastprivate	*Instância Local do Mapeamento Privativo ou Instância Global do Mapeamento Privativo*
reduction	*Instância Global do Mapeamento Privativo e Instância Local do Mapeamento Compartilhado ou Instância Global do Mapeamento Compartilhado*

Tabela 4.22: Mapeamento de variáveis com atributo de compartilhamento determinado explícitamente

instância, é introduzida no registro compartilhado da rotina em Cilk relacionada à região paralela. Desta forma, todos os acessos à variável que ocorrem na região fazem uso desta instância.

As variáveis condicionadas à cláusula *private* responsável pela alteração para o tipo básico de escopo privativo obedecem a um dos mapeamentos: **instância local do mapeameanto privativo** ou **instância global do mapeamento privativo.** O primeiro caso é aplicado quando a variável não requer que a mesma instância persista entre uma rotina desmembrada e outra. Verifica-se essa condição quando a variável é usada em apenas uma rotina desmembrada, por exemplo. O mapeamento alternativo é incorporado na ausência desta condição.

De maneira similar à cláusula *shared*, o mapeamento apontado para uma variável em uma cláusula *private* surtem efeito somente perante a alteração do atributo de compartilhamento da variável. Caso este em que o contexto compartilhado da variável continua existindo. Uma situação particular ocorre quando todos os acessos de escrita e leitura da variável condicionada à cláusula *private* estiverem confinados em estruturas de controle paralelo e de sincronização. Se em todas estas estruturas verifica-se que a cláusula *private* é aplicada à variável, a instância compartilhada da variável torna-se desprezível. Para diminuir o uso da memória compartilhada, basta que seja concedida uma **instância local do mapeamento privativo** à variável.

Na Tabela 4.22, é possível reparar que as variáveis encontradas nas cláusulas seguintes, *firstprivate* e *lastprivate* podem receber as mesmas intâncias de mapeamento das va-

riáveis verificadas em cláusulas *private*. Isto porque as cláusulas *firstprivate* e *lastprivate* são derivadas da anterior.

Por definição, os valores das variáveis pertencentes à lista de uma cláusula *private* são indeterminados tanto na entrada como na saída de uma região paralela. Com a cláusula *firstprivate*, cada cópia privativa de uma variável pode ter seu valor definido na entrada de uma região, sendo inicializada pelo valor da cópia da *master thread*. Já a cláusula *lastprivate*, determina que no final da região, a cópia da *master thread* seja atribuída com o valor referente à cópia privativa da última iteração do *loop*.

Embora as duas cláusulas não tenham sido verificadas na análise de freqüência de diretivas e cláusulas realizada, foi concedido mapeamento para ambas, supondo que a diretiva de região paralela que antecede as cláusulas é aninhada a uma diretiva *omp for*.

Algumas particularidades, no entanto, foram observadas. Para a cláusula *firstprivate*, caso a variável sendo mapeada receba uma **instância global de mapeamento privativo**, imediatamente antes da criação de tarefas simultâneas, as posições do vetor compartilhado são inicializadas, como mostra a Tabela 4.23.

```
Cilk void rotina(...)
{
 ...
 (i) ompcilk_shd_rotina->var1[0] = ompcilk_shd_rotina->var1;
 ou
 (ii) ompcilk_shd_rotina->var1[0] = var1;

 for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)
 ompcilk_shd_rotina->var1[ompcilk_tk] = ompcilk_shd_rotina->var1[0];

 for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)
 spawn ompcilk_rotina_0(ompcilk_tk, ompcilk_shd_rotina, ...);
 sync;
}
```

Tabela 4.23: Particularidades do Mapeamento da Cláusula *firstprivate*: Inicialização segundo a Instância Global de Mapeamento Privativo

Pela tabela, é possível notar que, primeiramente a posição *"0"* do vetor compartilhado pode receber o valor da **instância local do mapeamento compartilhado** da variável *var1* (situação *i*), ou o valor da **instância global do mapeamento compartilhado** da variável *var1* (situação *ii*). Isto depende do mapeamento compartilhado que tenha sido atribuído a ela. Posteriormente, as demais posições do vetor são inicializadas com o valor da posição *"0"*.

No entanto, se a variável receber uma **instância local do mapeamento privativo**, de acordo com a Tabela 4.24, ela é declarada na rotina associada à região paralela (*ompcilk_var1*). Por conseguinte, é inicializada com o valor da variável se mapeada através de uma **instância local do mapeamento compartilhado** (situação *i*), ou através de uma **instância global do mapeamento compartilhado** (situação *ii*).

```
Cilk void rotina(...)
{
 ...
 for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)
 spawn ompcilk_rotina_0(ompcilk_tk, ompcilk_shd_rotina, ...);
 sync;
}
Cilk void ompcilk_rotina_0(int ompcilk_tk, OMPCILK_SHD_ROTINA ompcilk_shd_rotina, ...)
{
 Tipo_da_variavel ompcilk_var1;
 (i) ompcilk_var1 = ompcilk_shd_rotina->var1;
 ou
 (ii) ompcilk_var1 = var1;
 ...
}
```

Tabela 4.24: Particularidades do Mapeamento da Cláusula *firstprivate*: Inicialização segundo a Instância Local de Mapeamento Privativo

Para a cláusula *lastprivate*, se o mapeamento assumido para a variável corresponde a uma **instância global de mapeamento privativo**, o valor da instância compartilhada da variável é atualizada fora da rotina associada à região, como demonstra a Tabela 4.25.

```
Cilk void rotina(...)
{
 ...
 for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)
 spawn ompcilk_rotina_0(ompcilk_tk, ompcilk_shd_phrotina, ...);
 sync;

 (i) ompcilk_shd_rotina->var1 = ompcilk_shd_rotina->var1[OMPCILK_TOT_TKS-1];
 ou
 (ii) var1 = ompcilk_shd_rotina->var1[OMPCILK_TOT_TKS-1];
 ...
}
```

Tabela 4.25: Particularidades do Mapeamento da Cláusula *lastprivate*: Atualização segundo a Instância Global de Mapeamento Privativo

Como dito anteriormente, o valor atribuído à variável referente à cláusula *lastprivate* corresponde sempre ao da última iteração do *loop*. Como, no mapeamento da diretiva *omp for*, esta iteração é executada sempre pela tarefa de identificação *OMPCILK_TOT_TKS - 1*, o valor da instância compartilhada é igual ao da posição *OMPCILK_TOT_TKS - 1* do vetor.

Caso seja adotada uma **instância local de mapeamento privativo** à variável, o valor da sua instância compartilhada é atualizado, ainda, no escopo da rotina da região paralela. Este valor somente é modificado pela instância privativa da última tarefa, como pode ser visto na Tabela 4.26.

```
Cilk void rotina(...)
{
 ...
 for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)
 spawn ompcilk_rotina_0(ompcilk_tk, ompcilk_shd_rotina, ...);
 sync;
}
Cilk void ompcilk_rotina_0(int ompcilk_tk, OMPCILK_SHD_ROTINA ompcilk_shd_ro tina, ...)
{
 Tipo_da_variavel ompcilk_var1;

 for (i=...; i<...; i++)
 {
 ...
 }

 if (ompcilk_tk == OMPCILK_TOT_TKS-1)
 (i) ompcilk_shd_rotina->var1 = ompcilk_var1;
 ou
 (ii) var1 = ompcilk_var1;
}
```

Tabela 4.26: Particularidades do Mapeamento da Cláusula *l astprivate*: Atualização segundo a Instância Local de Mapeamento Privativo

Independente da instância conferida ao mapeamento privativo, se local ou global, o contexto compartilhado pode ter sido adotado pelo mapeamento **local** ou **global**, identificados nas tabelas, respectivamente, pelos casos *i* e *ii*.

Diferentemente da regra de outras cláusulas, uma mesma variável é passível de estar presente na lista das cláusulas *firstprivate* e *lastprivate*, simultâneamente, em determinada região. Para este caso, em particular, o mapeamento do contexto compartilhado da variável deve ser comum em relação às duas cláusulas.

Por fim, para a cláusula de escopo identificada como *reduction*, o mapeamento das variáveis encontradas nela consiste em conceber uma **instância global do mapeamento privativo** e uma **instância do mapeamento compartilhado** à variável pertencente a sua lista.

Caso a variável sendo reduzida seja declarada globalmente, uma **instância global do mapeamento compartilhado** é adotada. Caso contrário, é criada uma **instância local do compartilhado** para ela.

A cláusula *reduction* é diferente das demais cláusulas por estar associada ao armazenamento privativo e compartilhado de uma única variável. Ela permite uma operação de redução sobre variáveis escalares. Todas as variáveis condicionadas à cláusula são submetidas à redução, das suas instâncias privativas para a sua instância compartilhada. Isto é feito de acordo com um operador, definido juntamente com a lista de variáveis, o qual segue à propriedade associativa e comutativa.

Este mesmo mapeamento é atribuído à diretiva *critical*, caso seu emprego seja justificado por uma redução de variável. O mapeamento de uma operação de redução, em linhas gerais, é ilustrado na Tabela 4.27.

```
for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)
(i) ompcilk_shd_rotina->var1[ompcilk_tk] = var1;
ou
(ii) ompcilk_shd_rotina->var1[ompcilk_tk] = ompcilk_shd_rotina->var1;

for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)
spawn ompcilk_rotina_0(ompcilk_tk, ompcilk_shd_rotina, ...);
sync;

(i) var1 = 0;
ou
(ii) ompcilk_shd_rotina->var1 = 0;
for (ompcilk_tk=0; ompcilk_tk < OMPCILK_TOT_TKS; ompcilk_tk++)
(i) var1 += ompcilk_shd_rotina->var1[ompcilk_tk];
ou
(ii) ompcilk_shd_rotina->var1 += ompcilk_shd_rotina->var1[ompcilk_tk];
```

Tabela 4.27: Mapeamento de uma Operação de Redução

O operador de redução é denotado, neste caso, pelo sinal de soma, conforme exposto na última linha que realiza a redução da variável *var1*. Percebe-se que antes mesmo da operação de redução, a instância compartilhada da variável é zerada, uma vez que as instâncias privativas já tenham sido inicializadas.

O padrão OpenMP permite a alteração do escopo *default* de variáveis, através da cláusula de escopo *default*. No entanto, caso a cláusula não seja aplicada, o atributo de compartilhamento destas variáveis é compartilhado. Baseado na análise de freqüência de cláusulas consolidada, a cláusula não foi empregada em nenhuma região paralela, não sendo destinado nenhum tratamento para este caso.

As variáveis referenciadas em diretivas cujos atributos de compartilhamento não são pré-determinados nem determinados explícitamente possuem seus atributos implícitamente determinados. As variáveis que se enquadram neste caso são mapeadas como uma **instância global** ou **local do mapeamento compartilhado** em função de estarem ou não declaradas globalmente.

4.3.3 Variáveis referenciadas fora de Diretivas Complementares

As variáveis referenciadas apenas nas diretivas de regiões paralelas recebem o mapeamento proposto na Tabela 4.28.

Variável com Atributo Pré-Determinado	Mapeamento em Cilk
Variáveis estáticas declaradas em rotinas invocadas na região	*Instância Local do Mapeamento Compartilhado*
Demais variáveis declaradas em rotinas chamadas na região	*Instância Local do Mapeamento Privativo; ou Instância Global do Mapeamento Privativo*
Alocadas na *heap*	*Instância Local do Mapeamento Compartilhado*

Tabela 4.28: Mapeamento de variáveis com atributo de compartilhamento pré-determinado

Note que as variáveis, que não são estáticas, declaradas em rotinas chamadas dentro da região paralela, assumem a **instância global de mapeamento** quando são afetadas por desmembramentos de rotina. Do contrário, lhes é atribuído **uma instância local do mapeamento privativo**.

4.4 Ordenação do Mapeamento das Extensões de Linguagem

A ausência de uma seqüência bem definida, que conduza o mapeamento para Cilk, pode afetar a capacidade da tradução de gerar um programa equivalente em Cilk, a cada tradução de uma mesma aplicação com o OpenMP. Isto é atribuído a duas razões. Primeiro porque o simples mapeamento de OpenMP para Cilk se aplica às extensões de linguagem de forma isolada, sem considerar nenhuma precedência no mapeamento delas. Segundo, como a tradução ainda é feita manualmente, cada tradutor pode adotar uma ordem distinta para conduzir o mapeamento. Assim, com a definição de seqüências de diretrizes para coordenar a tradução foi estabelecido um **processo de tradução**.

Embora não tenha sido automatizado, considera-se que o processo de tradução recebe como entrada um programa escrito com o OpenMP e gera, como saída, o programa traduzido. O processo é dotado de várias etapas com condições a serem verificadas. Quando fornecido o programa OpenMP, através destas condições, o processo identifica como o código foi escrito. E, assim, indica o mapeamento a ser adotado dentre aqueles discriminados nas seções 4.2 e 4.3.

Para contemplar as duas classes de estruturas, de controle e de dados, foram estabelecidas duas seqüências de etapas no processo de tradução. Ambas são representadas através de fluxogramas. Cada seqüência possui um conjunto de fluxogramas. Estes, por sua vez, sintetizam uma unidade da seqüência correspondente.

O **esquemático das estruturas de controle paralelo e de sincronição** configura a seqüência de fluxogramas que atende às extensões de linguagem de controle paralelo e sincronização. Três fluxogramas compreendem esta seqüência: o de **região seqüencial**, o de **desmembramento de rotina ocasionado por diretiva de região paralela**, e o de **região paralela**. Todos eles, integram uma unidade da seqüência a ser lida para que seja realizado o mapeamento de todas as estruturas de controle paralelo e sincronização presentes em um programa.

As unidades, expressas pelos fluxogramas, são encadeadas através de um determinado estado encontrado em uma delas onde a leitura da unidade corrente é paralizada para que uma outra unidade seja lida. A notação utilizada nos fluxogramas para denotar o estado, que alterna as unidades é identificado como um **processo**. A notação é expressa através de um retângulo que recebe uma tarja na sua extremidade inferior direita.

O ponto de partida para a leitura da seqüência referente às estruturas de controle para-

lelo, é feito por uma unidade denominada **geral**. O término da unidade **geral** assegura a conclusão do mapeamento de todas as estruturas de controle paralelo e de sincronização do programa.

As Figuras 4.2 e 4.3 mostram as unidades **geral** e de **região seqüencial**. Na unidade **geral**, é possível notar que, no momento, em que o estado denominado **Região Seqüencial** é atingido, a unidade de **Região Seqüencial** passa a ser lida. Somente após o término desta última, o estado seguinte da unidade **geral** é lido.

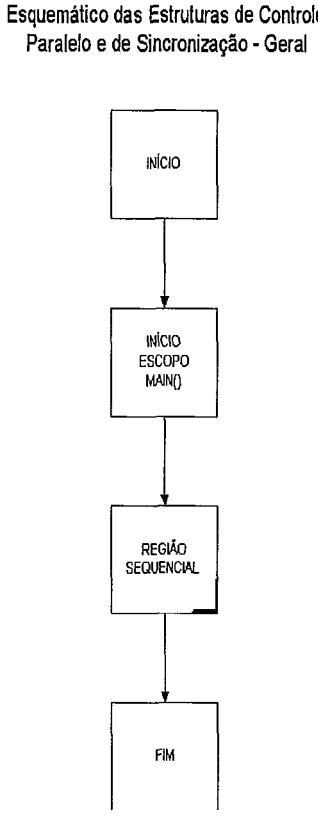


Figura 4.2: Unidade Geral do Esquemático das Estruturas de Controle Paralelo e de Sincronização

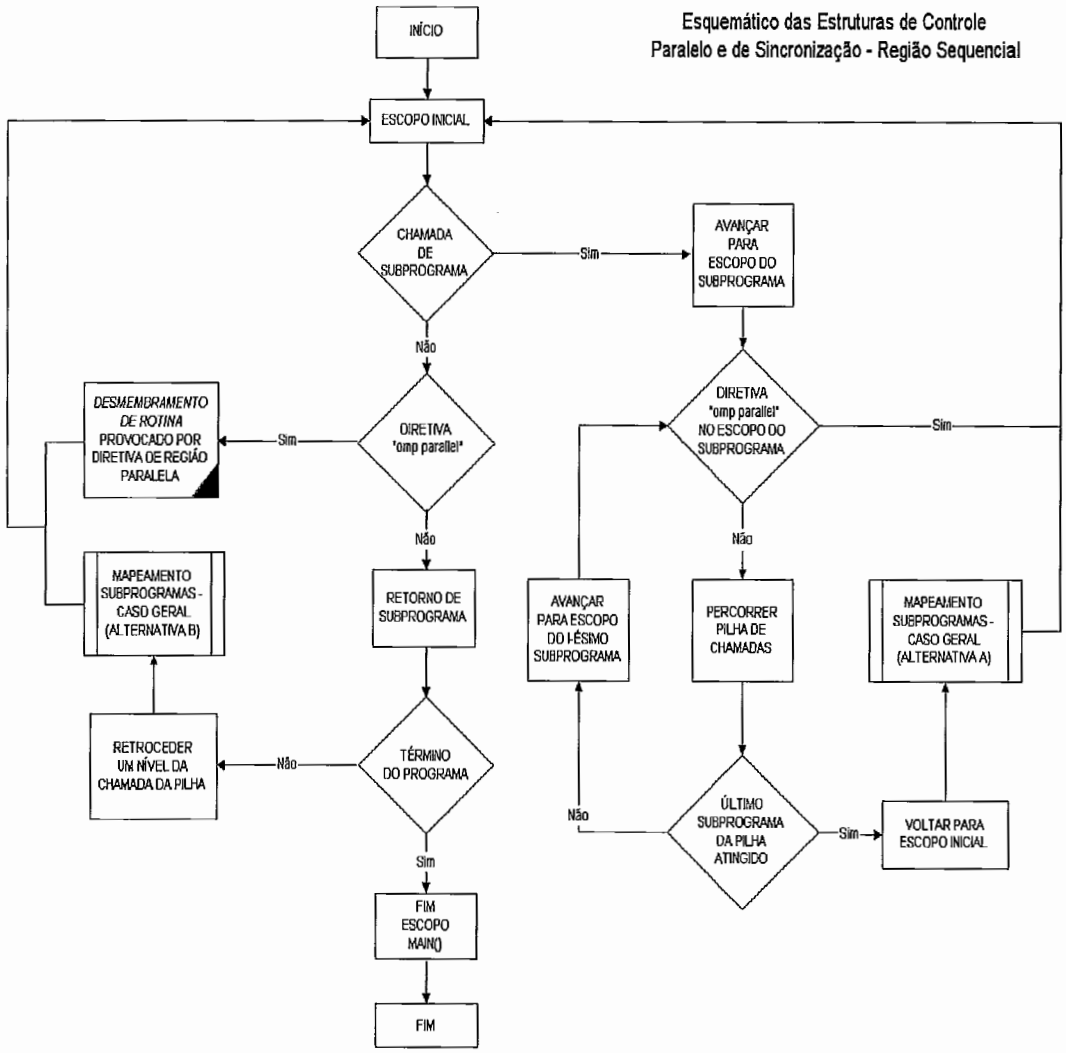


Figura 4.3: Unidade de Região Sequencial do Esquemático das Estruturas de Controle Paralelo e de Sincronização

A unidade de **região paralela** é lida a partir da unidade de **desmembramento de rotina**, pela transição para o estado de **Região Paralela**, conforme mostra a Figura 4.4. Quando o estado de **Região Paralela** é alcançado na unidade de **desmembramento de rotina**, é criada a rotina desmembrada inicial, identificada como *"0"*. Esta rotina inicial agrega, eventualmente, vários pontos de sincronização, que são inspecionados na unidade de **região paralela** a fim de se gerar novos desmembramentos de rotina.

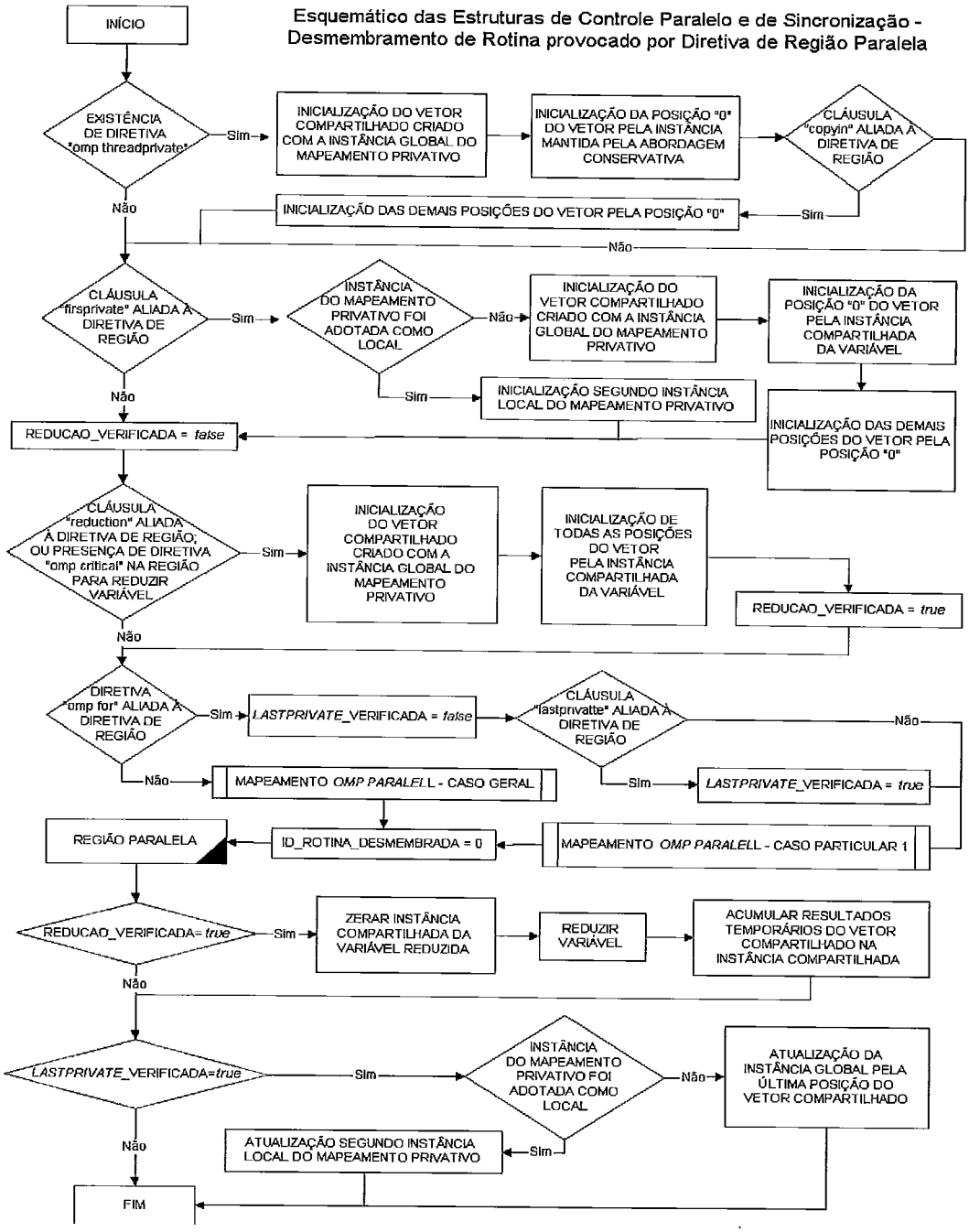


Figura 4.4: Unidade de Desmembramento de Rotina do Esquemático das Estruturas de Controle Paralelo e de Sincronização

No fluxograma da Figura 4.5, existe um estado denotado como **RP2**. Ele tem como função marcar a transição de um estado anterior para o próximo. A Figura 4.6 dá continuidade à unidade de **região paralela**. Nela verifica-se a transição de um estado para o **RP2**. Isso significa que quando esse estado é alcançado o estado seguinte será aquele indicado na Figura 4.5.

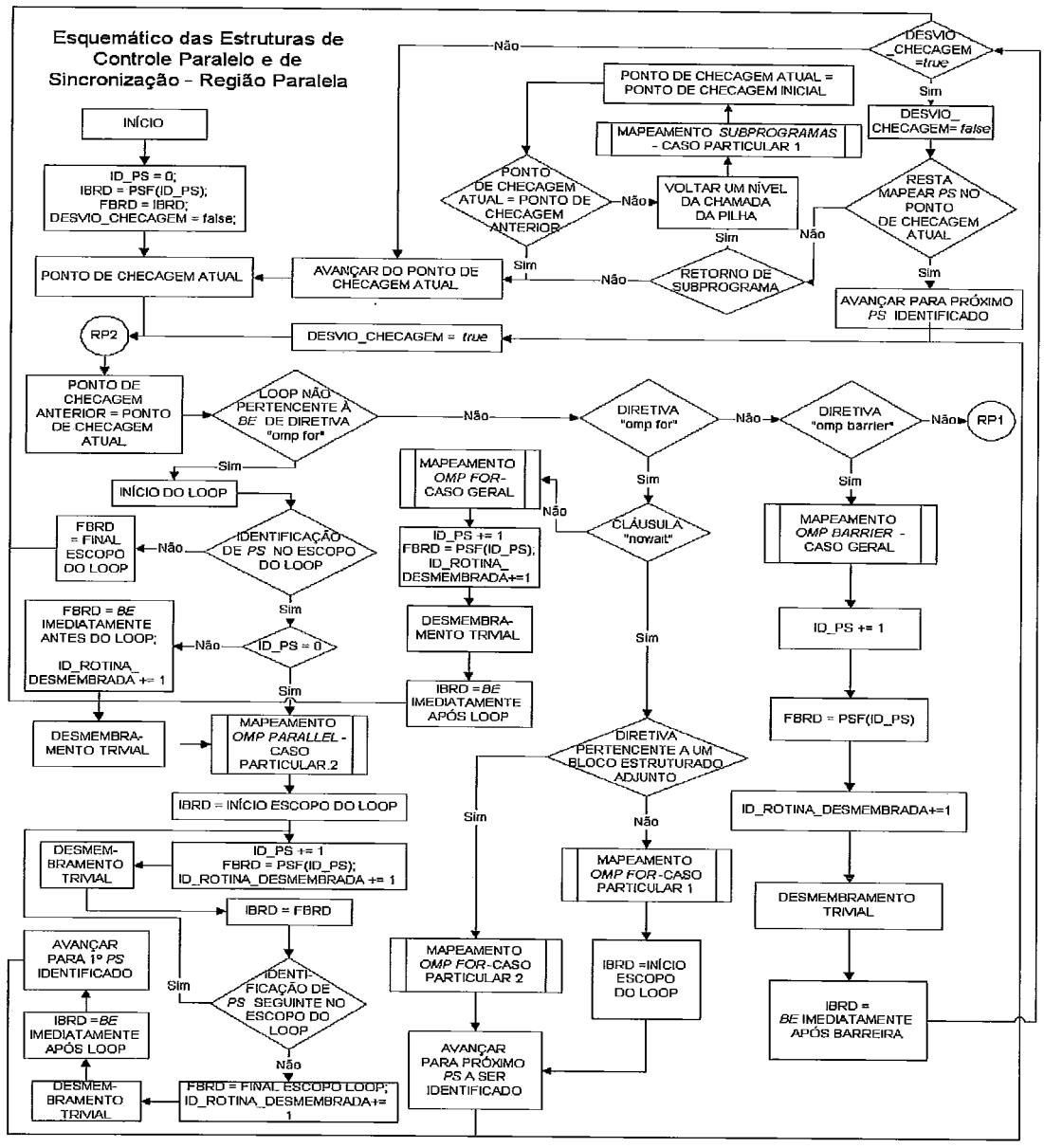


Figura 4.5: Unidade de Região Paralela do Esquemático das Estruturas de Controle Paralelo e de Sincronização

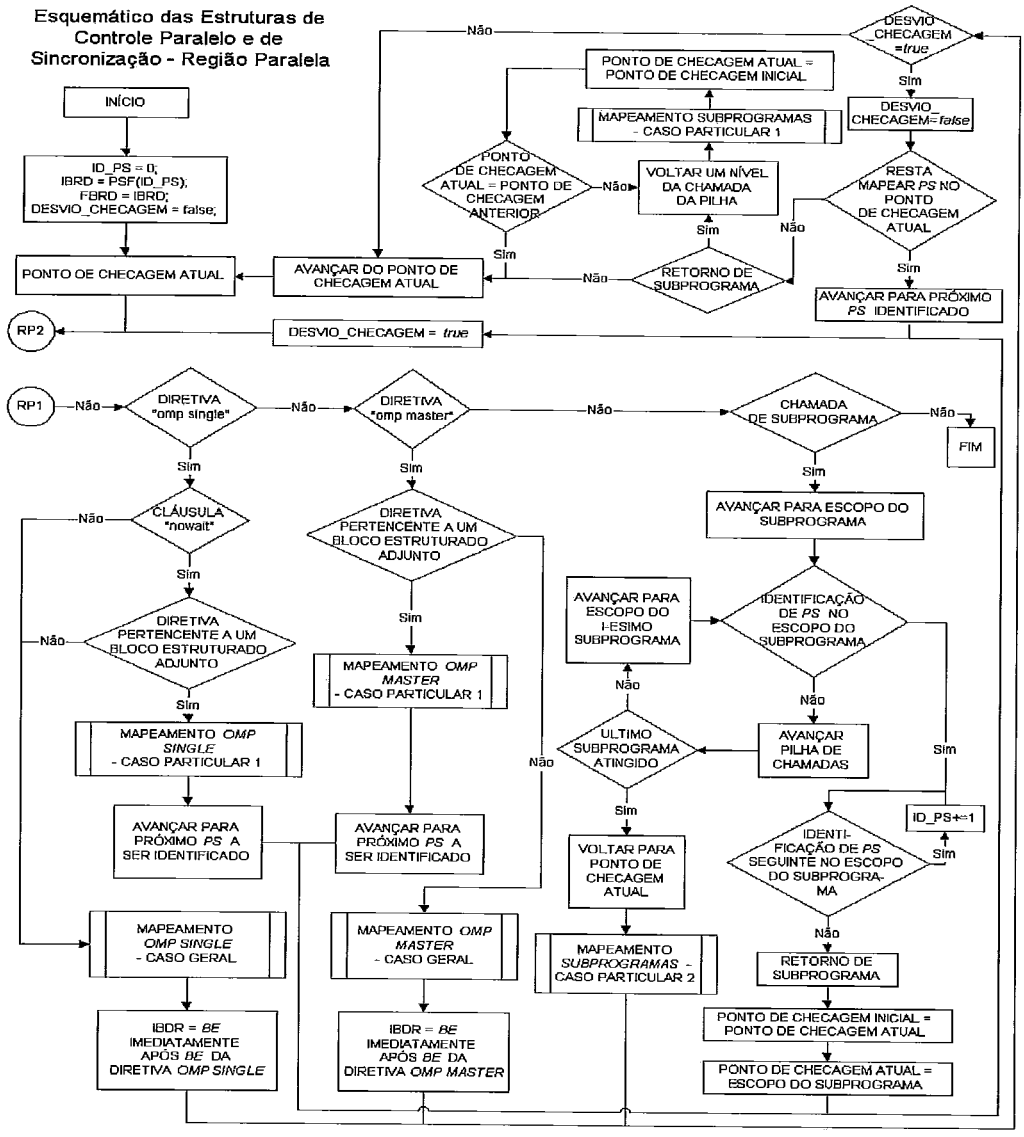


Figura 4.6: Unidade de Região Paralela do Esquemático das Estruturas de Controle Paralelo e de Sincronização - Continuação

Pela mesma razão atribuída ao estado **RP2**, o fluxograma da Figura 4.6, marca um estado denotado como **RP1**, que é alcançado como demonstra a Figura 4.5.

No fluxograma de **região paralela**, *ID_PS* indica o identificador de um ponto de sincronização, responsável por um novo desmembramento de rotina. A variável *IBRD* aponta o início do bloco da nova rotina desmembrada. Enquanto a variável *FBRD* indica o fim do bloco da rotina. Todo o trecho de código delimitado pelas variáveis *IBRD* e *FBRD* compõem o escopo da nova rotina desmembrada.

Uma função encontrada no fluxograma de **região paralela**, *PSF* (Função do Ponto de Sincronização), é utilizada para identificar os limites do bloco da rotina desmembrada. A

função deve retornar a última instrução do *BE* (Bloco Estruturado) anterior ao ponto de sincronização expresso pelo *ID_PS*.

A seqüência de fluxogramas voltada para as estruturas de controle de dados e comunicação é definida pelo **esquemático das estruturas de controle de dados e comunicação**. Nele, apenas mais uma unidade é verificada, além de uma unidade **geral** que marca o ponto de partida da leitura da seqüência. Tal unidade é identificada como **mapeamento influenciado por desmembramento de rotina**. O fluxograma presente da Figura 4.7 constitui a unidade **geral** do esquemático das estruturas de dados e comunicação.

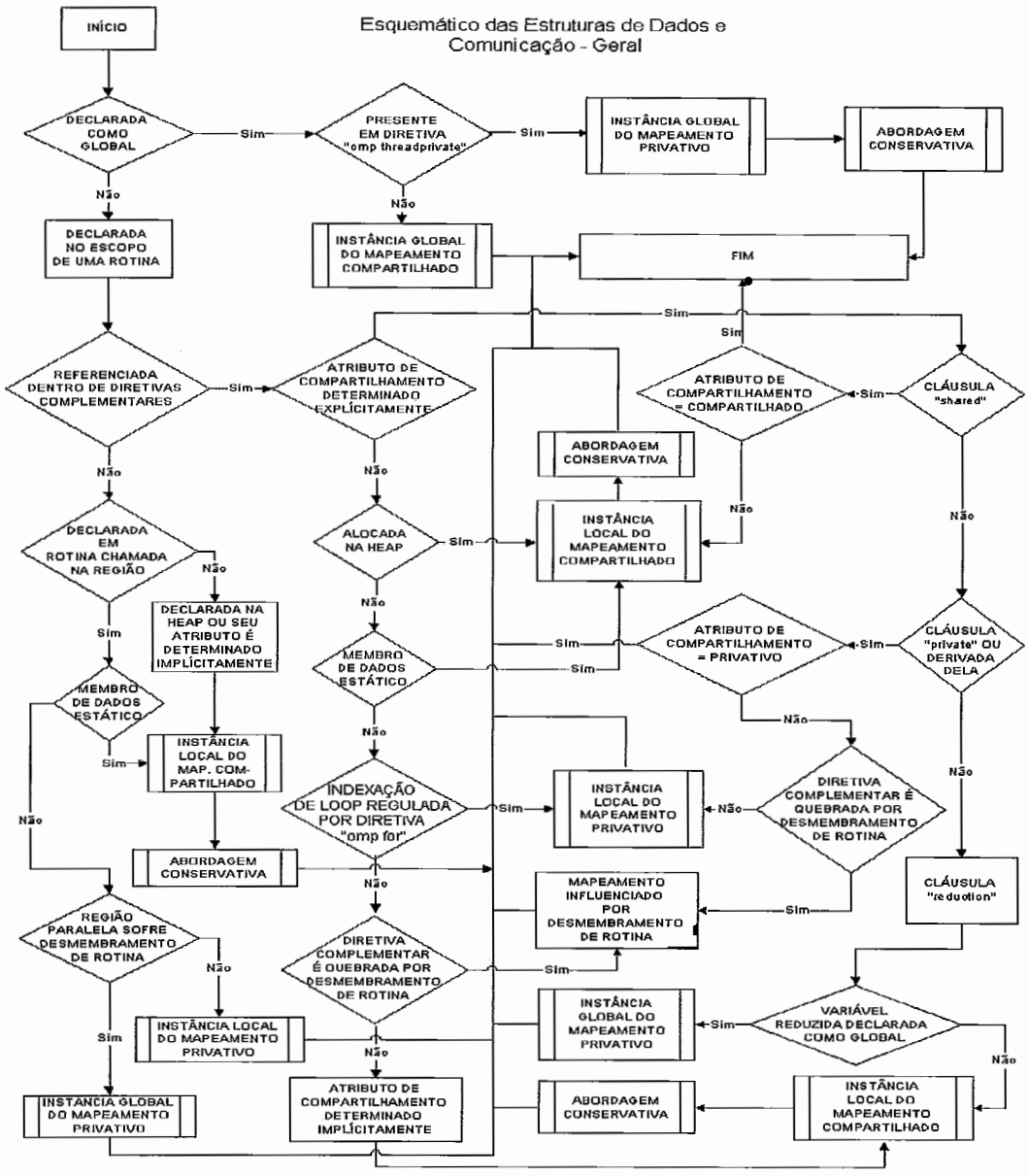


Figura 4.7: Unidade Geral do Esquemático das Estruturas de Dados e Comunicação

Na unidade **geral** observa-se a transição para o estado de **mapeamento influenciado por desmembramento de rotina**, que possui um fluxograma próprio, encontrado na Figura 4.8. No contexto das estruturas de dados e comunicação, a unidade **geral** deve ser consultada caso a caso, ou seja, para cada variável localizada na aplicação. De forma que o término da unidade caracteriza-se pelo mapeamento de uma única variável.

O fluxograma referente ao **mapeamento influenciado por desmembramento de rotina** trata o caso em que o mapeamento de uma variável é afetado pela própria tradução, de acordo com o escopo definido à variável, conforme descrito na Seção 4.3.1.

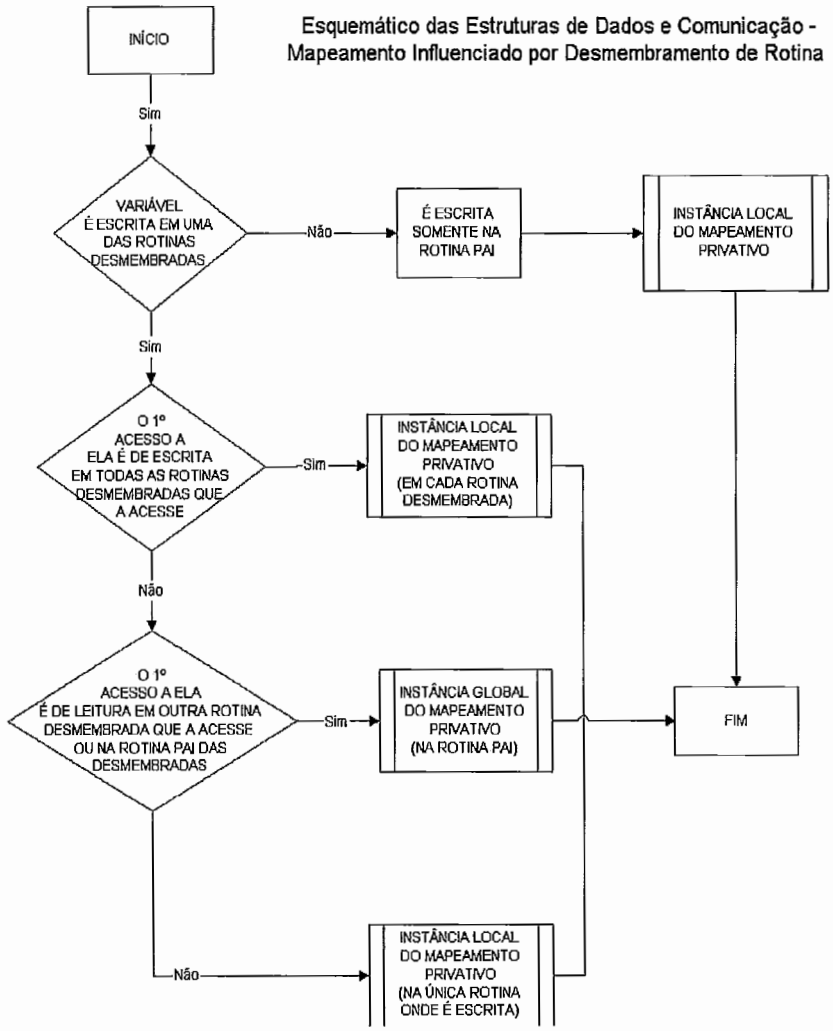


Figura 4.8: Unidade do Mapeamento Influenciado por Desmembramento de Rotina do Esquemático das Estruturas de Dados e Comunicação

Como o enfoque de ambos os esquemáticos está no mapeamento das estruturas, alguns estados presentes na seqüência de cada um correspondem ao próprio mapeamento atribuído a determinada estrutura. Estes estados são denotados por um retângulo cujos lados de menor aresta possui uma linha vertical.

No caso das estruturas de controle paralelo e de sincronização, o mapeamento refere-se aos descritos na Seção 4.2. O mapeamento é identificado pelo nome da respectiva diretiva, sendo especificado se trata-se do caso geral ou particular do mapeamento.

Já para as estruturas de controle de dados e sincronização o mapeamento é apontado como sendo uma **instância local/global do mapeamento compartilhado** ou uma **instância local/global do mapeamento privativo**. A implementação para os dois casos é descrita, na Seção 4.2.

4.5 Extensão do Mapeamento para o *SDSM Clik*

Para permitir que aplicações traduzidas sejam utilizadas em *clusters* de computadores é empregado o sistema *Clik* [7], que provê a abstração de uma memória única e compartilhada no *cluster*.

Se a aplicação for usada em *clusters*, entretanto, não basta apenas que seja traduzida, conforme sugerido neste capítulo. Algumas modificações se fazem necessárias para o mapeamento das estruturas de dados e comunicação das aplicações com o *Clik*. Sobretudo, porque a estrutura de processos e memória do sistema *Clik* é conceitualmente diferente da utilizada no sistema *Cilk*, para multiprocessadores.

Assim, um mapeamento extensivo é aplicado às estruturas de dados e comunicação das aplicações para *cluster*. O mapeamento das estruturas de controle paralelo e de sincronização não difere daquele voltado para um multiprocessador.

Esta seção tem como objetivo descrever, sucintamente, características do sistema *Clik*. E, posteriormente, introduzir a extensão do mapeamento para as estruturas de dados e comunicação de aplicações para o *Clik*.

4.5.1 Sistema Clik

A proposta deste software DSM é a de prover um mecanismo eficiente de distribuição de tarefas nos nós de um *cluster*. Para isto, o algoritmo de roubo de trabalho da versão 5.3.2 do sistema *Cilk* é adaptado pelo *SDSM*.

O sistema *Clik* inicia um conjunto de processos trabalhadores em cada nó. Cada trabalhador tem sua fila de tarefas independente. Todas as filas encontram-se vazias de início, exceto a do trabalhador *"0"* que é responsável pela execução da primeira *thread*, denominada *pai*. À medida que a *thread-pai* cria *threads-filhas* novas tarefas são inseridas na fila, as quais podem ser roubadas por outros trabalhadores.

O algoritmo de roubo de trabalho procede da forma descrita a seguir. Primeiramente, o trabalhador tenta obter uma tarefa de outro trabalhador localizado no mesmo nó. Caso não consiga obter uma tarefa, ele, então, inicia o roubo de trabalho remotamente, enviando uma mensagem requisitando trabalho a outro nó, escolhido aleatóriamente. Ao receber uma resposta de que o nó também não tem trabalho, ele inicia o algoritmo de roubo de trabalho remoto novamente. Caso contrário, ele insere a *thread* a ser executada na sua fila de tarefas local. O roubo de trabalho só é ativado novamente quando esta fila de tarefas esvaziar.

Uma página virtual equivale à unidade de coerência da memória compartilhada, empregada pelo *Clik*. A área de memória compartilhada necessária para a execução de aplicações *multithreaded* é provida de um protocolo de consistência relaxada baseado em residência (*home-based*), semelhante ao sistema HLRC [18]. O protocolo utiliza invalidações para garantir a coerência de uma página, baseando-se em intervalos que formam uma ordenação parcial das escritas feitas na memória. Um trabalhador é notificado de invalidações ao receber uma tarefa de outro nó (pelo roubo de trabalho) ou quando executa uma operação de *sync*.

O sistema *Clik* implementa um protocolo de múltiplos escritores, permitindo que escritas em áreas de dados diferentes de uma mesma página sejam realizadas concorrentemente. Estas escritas são enviadas para o nó *home* da página sob a forma de uma estrutura chamada *diff*. Isso ocorre em três momentos: quando uma nova tarefa é criada (com *spawn*), na execução de um *sync* e no retorno de uma tarefa. Quando um trabalhador sofre uma falha de página, ele requisita uma cópia atualizada dela ao nó *home* da página.

Segundo resultados obtidos [7], o sistema *Clik* revela-se competitivo em relação ao HLRC, sendo este último considerado estado-da-arte em termos de software DSM.

4.5.2 Mapeamento das Estruturas de Dados e Comunicação para Clusters

Considerando que o código que venha a ser executado no *Clik*, tenha sido previamente traduzido de OpenMP para Cilk, a transposição para o *Clik* exige algumas modificações.

A versão atual do sistema *Clik* não aceita o uso de variáveis globais. De maneira que, quando uma mesma variável é empregada entre várias rotinas, ela é sempre passada como argumento. Esta restrição requer que todas as variáveis declaradas globalmente recebam uma **instância local do mapeamento compartilhado**, no mapeamento extendido, ao invés de uma instância global.

Outra particularidade se refere à área de *DSM* no *Clik*. Para a alocação e liberação de espaço na área de *DSM* devem ser usadas, respectivamente, as seguintes rotinas: *cilk_dsm_malloc* e *cilk_dsm_free*. Todos os dados compartilhados e que são escritos mais de uma vez, não somente na inicialização deles, devem estar presentes nesta área. Em termos de mapeamento, as **instâncias de mapeamento** que são compartilhadas entre as *threads* necessitam estar alocadas na área de *DSM*. Ambas as funções de gerência de memória compartilhada distribuída devem ser utilizadas em rotinas *Cilk*.

Capítulo 5

Resultados Experimentais

Este capítulo aborda os resultados dos experimentos realizados para validar o mapeamento proposto de cada extensão de linguagem do OpenMP e o processo usado para coordenar este mapeamento. Além disto, o capítulo se concentra na verificação da qualidade da tradução, com a análise de desempenho de algumas aplicações. A validação do mapeamento consiste em verificar se dados de saída obtidos com uma aplicação, originalmente escrita com o OpenMP, e com a mesma aplicação traduzida, coincidem. O método usado para realizar esta verificação é descrito em cada experimento. A qualidade da tradução é verificada através da análise de desempenho entre as duas versões das aplicações consideradas, a original onde se emprega o OpenMP, e a traduzida.

Dois experimentos distintos foram realizados. No primeiro deles, utilizou-se um multiprocessador, uma vez que representa o ambiente para o qual tanto o padrão OpenMP como o sistema *Cilk* foram projetados. No segundo experimento, utilizou-se um *cluster* de computadores, onde algumas aplicações foram executadas, utilizando o *software DSM Clik*.

5.1 Aplicações Utilizadas

As aplicações utilizadas nos dois experimentos, descritos em seguida, são provenientes do conjunto NAS [10], onde o padrão OpenMP é empregado. Uma particularidade do conjunto NAS é a de que todas as aplicações, com exceção de um núcleo, são escritas em Fortran. Para o trabalho, entretanto, se fazia necessária a utilização de aplicações escritas em C, pelo fato da linguagem *Cilk* ser uma extensão do C. Assim, foi considerado o pacote 2.3 do conjunto NAS, escrito em C, pelos mesmos desenvolvedores do *OMNI OpenMP Compiler* [2].

Da mesma forma que na versão obtida diretamente do repositório do conjunto NAS,

o pacote em C é composto por cinco núcleos e três aplicações simuladas.

Os núcleos reproduzem a computação realizada por cinco métodos numéricos para aplicações de *Dinâmica de Fluidos Computacional* (CFD). As três aplicações simuladas incorporam grande parte da movimentação de dados e computação encontrados em códigos CFD. As aplicações pertencentes ao pacote, que foram utilizadas nos experimentos, são descritas a seguir.

- **IS** - Núcleo que provê uma ordenação de inteiros relevante para simulações de método de partículas. Ele testa o desempenho na velocidade da computação de inteiros e em comunicação.

- **CG** - Núcleo que usa o método de **Gradiente Conjugado** para testar computações e comunicações através de uma matriz com posições de entrada geradas randômicamente.

- **MG** - Núcleo que usa um método *MultiGrid* para resolver uma equação escalar de Poisson 3-D. O algoritmo testa distâncias curtas e longas de movimentação de dados.

- **EP** - Núcleo do *benchmark* embaraçosamente paralelo que gera curvas Gaussianas de forma randômica. Em contraste com as outras aplicações, o núcleo não requer comunicação entre processadores.

- **FT** - Núcleo que soluciona equações diferenciais parciais 3-D usando um método espectral baseado em FFT (*Fast Fourier Transform*), sendo efetuada uma FFT para cada dimensão. Consiste em um rigoroso teste de desempenho de comunicação de longa distância.

- **BT** - Uma aplicação CFD simulada que usa um algoritmo implícito para resolver equações de Navier-Stokes 3-D. A solução de diferença finita por uma fatorização é decomposta nas dimensões x, y e z, em função das quais um sistema resultante com 5x5 blocos tridiagonais é resolvido.

- **SP** - Aplicação CFD simulada cuja estrutura é similar ao da BT. Sendo que o sistema, resultante de uma fatorização *Beam-Warming* e composto por equações lineares pentadiagonais escalares, é resolvido para cada dimensão.

Vale ressaltar que a aplicação LU não foi considerada nos experimentos realizados por fazer uso da diretiva *omp flush* que não foi mapeada. A diretiva *omp flush* foi usada na aplicação para exercer a função de um *pipeline*, em que uma *thread* produz um valor para ser lido por outra. Pela diretiva, é definido um ponto de sincronização que provê uma visão consistente da memória, em relação às variáveis especificadas na diretiva. Este ponto de sincronização só poderia ser mapeado através de um *sync*, que implica na consistência de todos os dados compartilhados escritos pelas *threads-filhas*, e não somente em relação aos dados indicados pela diretiva, no modelo de execução do *Cilk*. Com o uso da primitiva *sync*, a sincronização gerada seria muito maior do que a necessária para a consistência dos dados no *pipeline*, portanto, optou-se por não estabelecer mapeamento para a diretiva *omp flush*.

Quando o *benchmark NAS* é utilizado na literatura [25, 26, 27, 28], comumente são encontrados os desempenhos das versões em Fortran das aplicações que o compõem. Assim, antes de serem realizados experimentos com a versão em C dessas aplicações, foi realizado um experimento preliminar, onde os tempos seqüenciais da versão em C foram comparados com os tempos seqüenciais da versão em Fortran do pacote 2.3. Neste experimento, o tamanho da entrada utilizado para cada aplicação foi o da classe C. Esta classe é adotada pelo *benchmark* NAS como sendo a mais representativa, em termos de massa de dados. A Tabela 5.1 mostra o tamanho da entrada, em função do número de inteiros da classe C, para cada uma das aplicações.

Aplicação	Tamanho (inteiros)
IS	2^{27}
CG	150000
MG	512^3
EP	2^{32}
FT	512^3
BT	162^3
SP	162^3

Tabela 5.1: Tamanho das Aplicações do NPB-2.3 referente à Classe C

Para a equivalência dos níveis de otimização e outras propriedades do compilador, foi utilizada a versão 9.1.036 do compilador Intel para Fortran e a versão 9.1.038 do compilador Intel para C/C++. O *flag* de otimização *-O2* foi utilizado, nas 10 execuções de cada aplicação. Para aquelas que fazem uso de função de geração de números randômicos foi empregada aritmética inteira de 8 *bits*.

A Tabela 5.2 apresenta os tempos seriais, em segundos, de ambas as versões das apli-

cações, sendo que apenas o núcleo IS não encontra-se na Tabela. O IS é a única aplicação em C, presente no repositório do conjunto NAS, e, portanto, a versão considerada é extraída diretamente do repositório.

Aplicação	T.S. da versão em Fortran	T.S. da versão em C
CG	435.51	473.04
MG	285.72	763.48
EP	672.41	687.50
FT	997.27	1152.42
BT	4103.17	4691.10
SP	2258.31	2619.43

Tabela 5.2: Tempo Seqüêncial (T.S.) em segundos das Aplicações Traduzidas

Pela Tabela 5.2, observa-se que o núcleo MG sofreu um aumento mais acentuado, no tempo seqüencial, com a versão em C. Na versão em C desta aplicação, em particular, a alocação dinâmica de alguns vetores é mais utilizada. Isto provocou perda de desempenho com o acesso extra à memória, e com a baixa localidade no acesso à memória, conforme descrito em [10].

Na versão em Fortran para OpenMP das aplicações BT e SP, em especial, a dimensão de dois vetores temporários, usados no cálculo para solução do sistema de equações, foi reduzida de 5 para 3, em relação à versão seqüencial destas aplicações. Isto fez com que a versão em Fortran, de ambas as aplicações, usasse apenas 1/6 do total de memória necessário na versão seqüencial delas. Esta otimização, porém, não foi incorporada na versão em C das duas aplicações, de modo que nesta versão a quantidade de escritas e leituras na memória é maior do que na versão em Fortran. Por isso, é observado, na Tabela 5.2, uma diferença mais significativa no tempo seqüencial entre as duas versões da BT e SP.

Apesar de terem sido encontradas uma diferença de tempo maior entre a versão em C e Fortran de algumas aplicações, este fator não impediu a verificação da funcionalidade da tradução, assim como, a análise de desempenho, através delas também, conforme mencionado na seção seguinte.

5.2 Experimento 1: Multiprocessador

Para demonstrar a efetividade da tradução proposta, usando o processo de tradução e os mapeamentos apresentados no Capítulo 4, foi comparada a execução das aplicações, originalmente escritas com o OpenMP, com a execução das versões traduzidas dessas aplicações, em um ambiente comum entre o OpenMP e o sistema *Cilk*, ou seja, em um multiprocessador.

Os experimentos realizados em um multiprocessador tiveram como propósito, em particular, encontrar construções em *Cilk* que apresentassem funcionalidade e desempenho equivalentes para as extensões do OpenMP consideradas, a fim de atestar a efetividade do mapeamento.

5.2.1 Descrição do Ambiente

O multiprocessador utilizado constitui uma arquitetura NUMA (*Non Uniform Memory Access*). Em uma arquitetura NUMA cada processador possui sua memória local. Outros processadores podem acessar esta memória local, no entanto, caracterizando uma arquitetura de memória compartilhada. A arquitetura é chamada **não uniforme** pois o tempo de acesso à memória torna-se maior quando o processador não acessa a sua própria memória. O ambiente empregado possui as seguintes características:

- Plataforma SGI Altix 350 ;
- 8 CPUs Intel Itanium2 64 bits ;
- 28 GB RAM Compartilhada - NUMA ;
- 4 Mb de cache L3 ;
- Kernel Linux versão 2.4.2.1 ;
- Sistema Cilk versão 5.4.2 ;
- Compilador Intel C/C++ versão 9.1.038 Itanium ;

5.2.2 Validação

A correção da tradução proposta, incluindo os mapeamentos, é verificada por meio da validação das aplicações traduzidas e pela validação das construções em *Cilk* usadas nos

mapeamentos. Assim, foi verificado se os mesmos dados de saída, das aplicações utilizadas no experimento, seriam obtidos, antes e depois da tradução delas. E, ainda, se as construções em *Cilk*, dos mapeamentos propostos, eram capazes de gerar os mesmos valores para as variáveis escritas na extensão do OpenMP correspondente.

Inicialmente, as aplicações descritas na Seção 5.1 foram traduzidas, segundo o processo de tradução e os mapeamentos apresentados no Capítulo 4. Com a tradução, cada aplicação do *benchmark*, identificada como a *versão em OpenMP*, recebeu uma outra versão, denominada *versão em Cilk*. A menor classe de dados de entrada foi empregada para ambas as versões.

Em particular no Cilk, os códigos escritos para este sistema, passam por um preprocessador que interpreta a linguagem Cilk, e em seguida são compilados. Para manter a equivalência entre os compiladores nas duas versões das aplicações, o código em Cilk pré-processado foi fornecido como entrada para o compilador da Intel, especificado na Seção 5.3.1, que suporta o OpenMP. Sendo, ainda, considerados exatamente os mesmos *flags* de compilação entre as duas versões.

Cada aplicação, originalmente pertencente ao NAS, possui um teste de verificação dos dados de saída. Estes mesmos testes foram usados nas aplicações traduzidas, a fim de validá-las. Para a validação das construções em *Cilk* dos mapeamentos, foram comparadas as variáveis escritas em cada construção, separadamente, em relação à respectiva extensão do OpenMP.

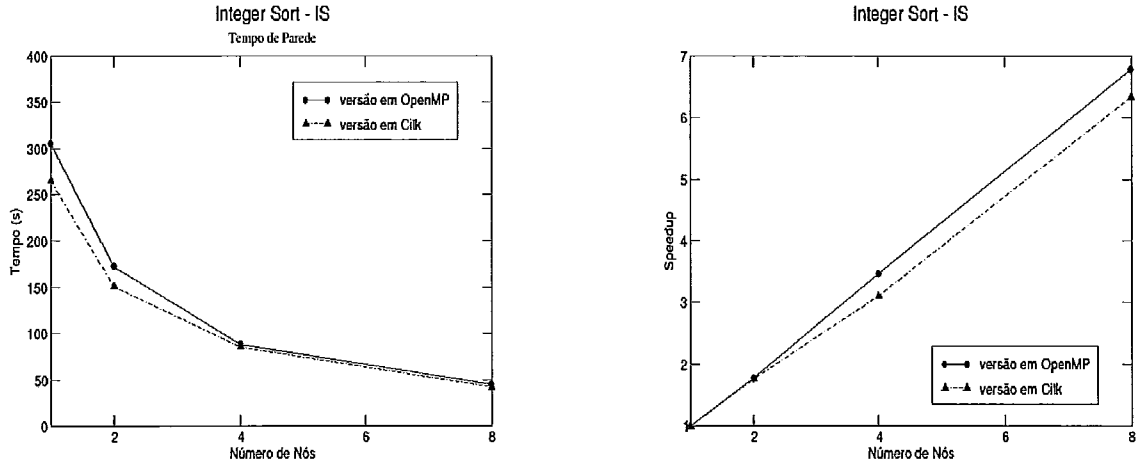
A comparação dos resultados de saída atestou a correção da tradução. Em seguida, foi analisado, no multiprocessador, o desempenho das aplicações traduzidas, quando comparadas as suas versões originais escritas com o OpenMP.

5.2.3 Análise de Resultados

Nesta seção são apresentados os desempenhos no multiprocessador adquiridos com as versões em OpenMP e em *Cilk*, das aplicações traduzidas, durante a validação. Os dados de entrada são expressos pela classe C do *benchmark*, cujo tamanho é apresentado na Tabela 5.1, para cada aplicação. Os desempenhos foram verificados em função do *speedup* de cada aplicação. O *speedup* exprime o quanto a versão paralela de uma aplicação é mais rápida do que a sua versão seqüiêncial.

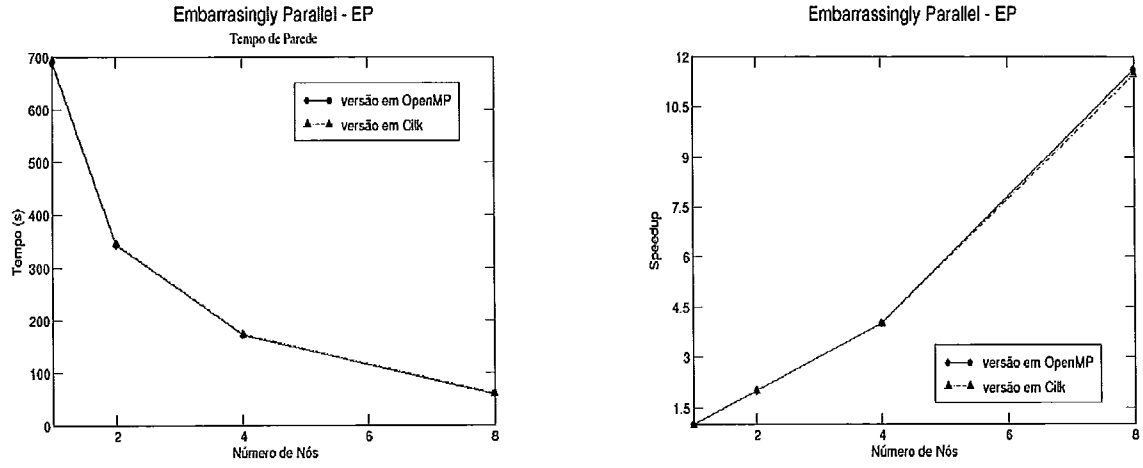
Os *speedups* foram calculados pela divisão do tempo total de execução da versão paralela, de acordo com o número de nós de processamento, pelo tempo total de execução seqüêncial. Para cada aplicação, é mostrado um gráfico com o *speedup* da sua *versão*

OpenMP e da sua *versão Cilk*. A quantidade de nós, para cada aplicação, foi variada em 2, 4 e 8, considerando que um nó equivale a um processador. Em cada processador foi criada uma única *thread*. No OpenMP, a quantidade de *threads* por processador foi definida através de variáveis de ambiente. Já no *Cilk*, por definição, é criada uma única *thread*, por processador. Assim, tornou-se suficiente especificar o número adequado de processadores para que a mesma quantidade de *threads* fosse criada, no *Cilk*. As Figuras 5.1, 5.4, 5.3, 5.2 e 5.5, mostram os *speedups* dos núcleos que compõem o *benchmark*.



(a) Tempo Total de Execução (b) *Speedup*

Figura 5.1: Desempenho do Núcleo IS

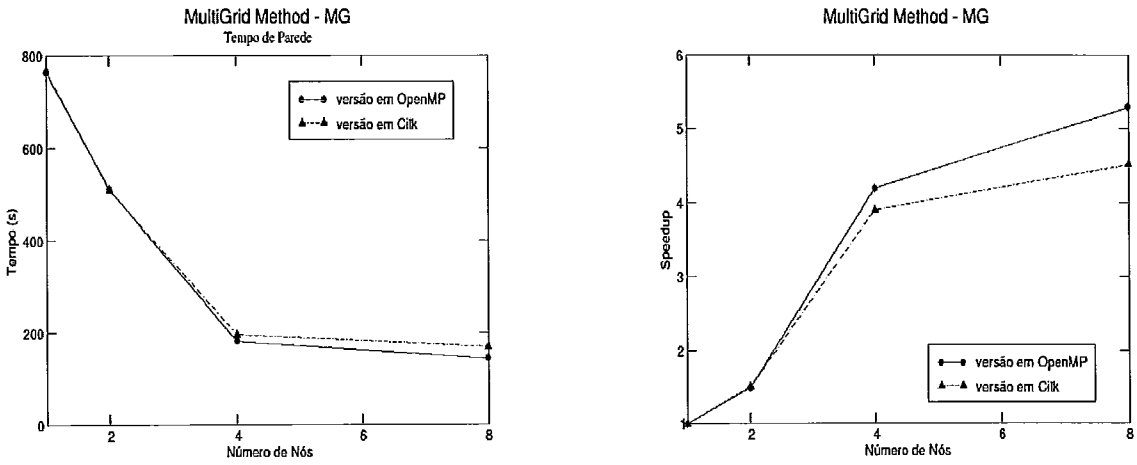


(a) Tempo Total de Execução (b) *Speedup*

Figura 5.2: Desempenho do Núcleo EP

Dentre os núcleos, observa-se que o desempenho da versão traduzida das aplicações é semelhante ao da versão *em OpenMP* delas. Em relação ao núcleo IS é possível verificar uma ligeira diferença no tempo seqüêncial da *versão em Cilk*, se comparada à *versão em OpenMP*. Foi constatado que o tempo de inicialização do programa com OpenMP é superior ao da *versão em Cilk*. É exatamente duas vezes maior: o tempo de inicialização na *versão em OpenMP* é equivalente à 80s, e na *versão em Cilk* de 40s.

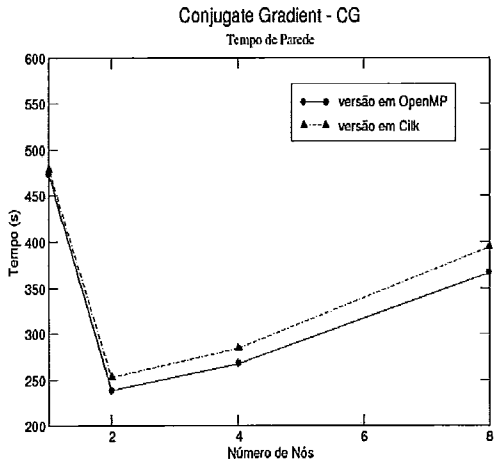
A diferença, observada com a execução serial da versão em OpenMP do IS, deve-se ao tempo gasto na inicialização de variáveis expressas na cláusula *copyin*, que acompanha a diretiva de região paralela, do início do programa. Pela cláusula, os valores, atribuídos à cópia da *master thread*, de cada variável presente na cláusula, são propagados à cópia de toda *thread* que executa a região paralela. Quando mais de um processador é utilizado, enquanto algumas *threads* tem suas cópias inicializadas, outras já passaram desta etapa e executam a computação encontrada na região, propriamente. Desta forma, é possível sobrepor o *overhead* da inicialização daquelas variáveis, com computação. Quando apenas um processador é usado, ainda assim, os valores dessas variáveis são propagados, antes da execução da região, por característica do compilador utilizado. Neste caso, no entanto, não é possível sobrepor o tempo gasto na inicialização com computação útil, tornando o *overhead* mais visível, na execução sequencial do IS. Cabe ressaltar que o *overhead* no uso da cláusula *copyin* foi identificado, especificamente, para a versão do compilador e na plataforma utilizada no experimento.



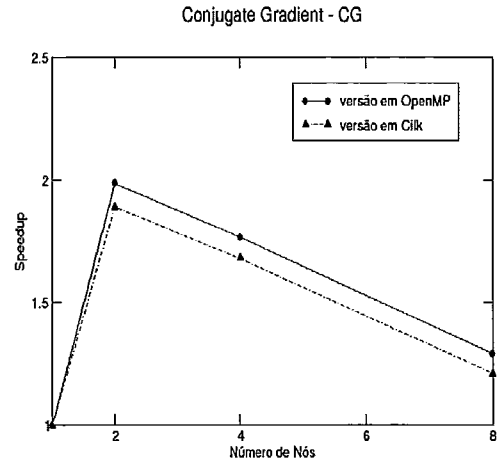
(a) Tempo Total de Execução

(b) *Speedup*

Figura 5.3: Desempenho do Núcleo MG



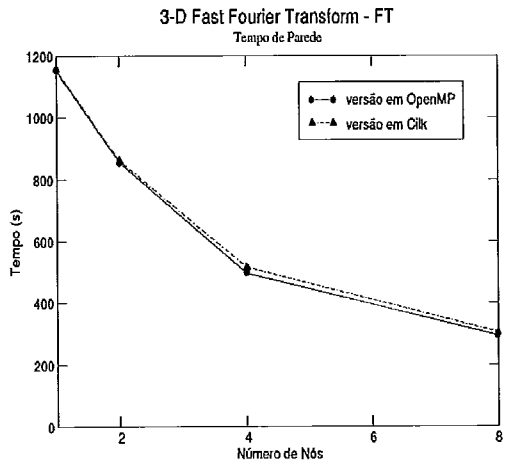
(a) Tempo Total de Execução



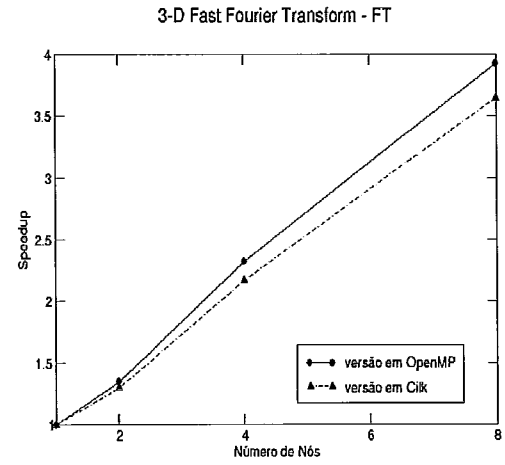
(b) *Speedup*

Figura 5.4: Desempenho do Núcleo CG

Para o núcleo CG, foi observado um *slowdown*, pelo fato da memória *cache* não ser aproveitada da melhor forma na aplicação, conforme descrito no artigo original da versão 2.3 do NAS [10]. A aplicação CG foi paralelizada com a adoção de diretivas em muitos *loops* de baixa granularidade, de maneira que a paralelização gera mais *overheads* do que benefícios na execução da aplicação com uma quantidade de nós superior a 2.



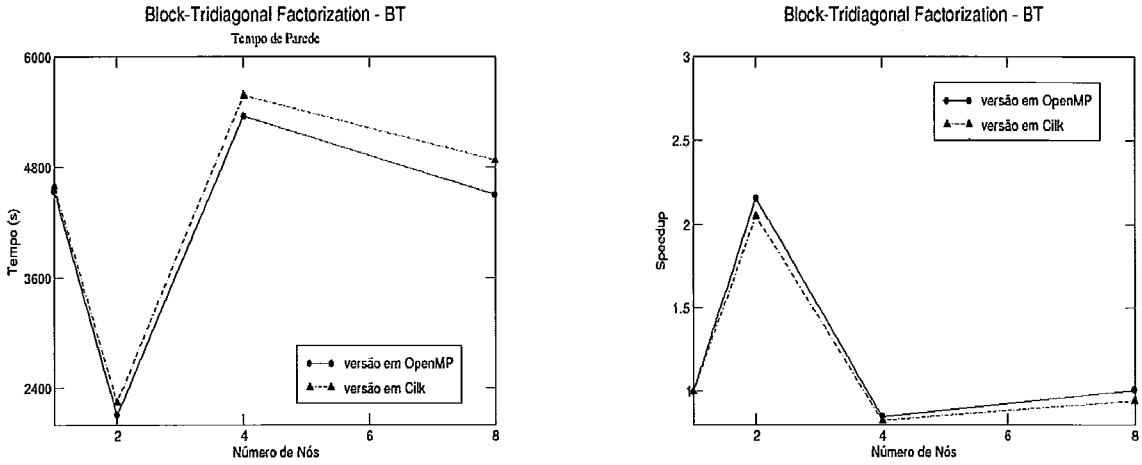
(a) Tempo Total de Execução



(b) *Speedup*

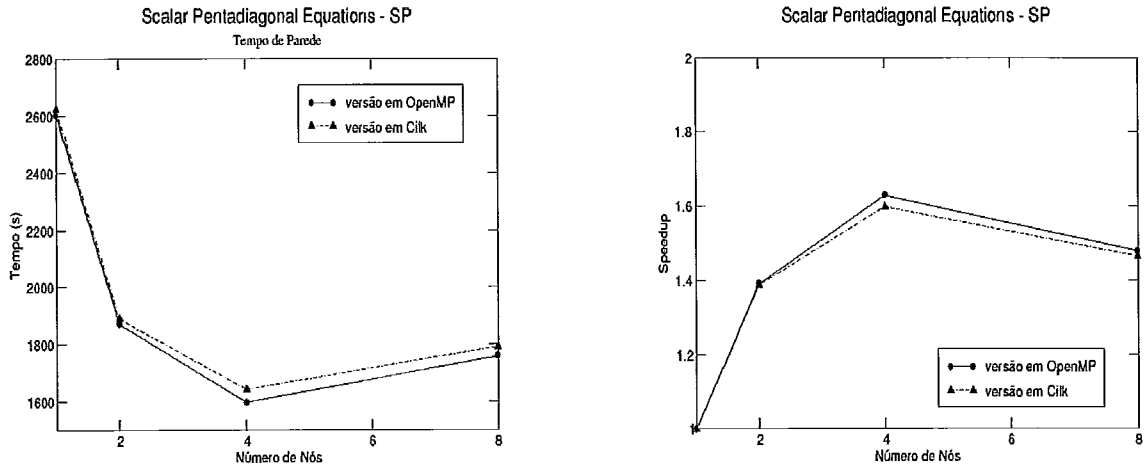
Figura 5.5: Desempenho do Núcleo FT

Nas Figuras 5.6 e 5.7, encontram-se, respectivamente, os desempenhos verificados para duas aplicações CFD simuladas do *benchmark*: BT e SP. Como mencionado anteriormente, não foram aplicadas otimizações no consumo de memória das versões em C da BT e SP. Em especial, alguns vetores temporários usados na solução de sistemas de ambas as aplicações, são mantidos com cinco dimensões, quando, na realidade, poderiam ser reduzidos para três. O grande consumo de memória demanda mais tempo na computação a ser realizada, além de aumentar o tempo gasto com sincronização e coerência de memória.



(a) Tempo Total de Execução (b) *Speedup*

Figura 5.6: Desempenho da Aplicação Simulada BT



(a) Tempo Total de Execução (b) *Speedup*

Figura 5.7: Desempenho da Aplicação Simulada SP

Embora, na análise de desempenho das aplicações traduzidas, as aplicações SP, BT e, ainda, o núcleo CG, tenham apresentado *slowndown* com o código em *Cilk*, o *slowndown* também é verificado para a *versão em OpenMP* das aplicações. O desempenho proporcional entre a *versão em OpenMP* e a *versão em Cilk*, encontrado para todas as aplicações consideradas, demonstram a efetividade da tradução proposta. A efetividade da tradução comprova a qualidade do mapeamento e do processo utilizado para coordenar o mapeamento, quando as aplicações a serem traduzidas são executadas em um multiprocessador.

5.3 Ambiente de Experimento 2: Cluster de Computadores

O emprego de aplicações traduzidas em um *cluster* de computadores tornou-se interessante, na medida que *overheads* existentes em função do ambiente, passam a influenciar o desempenho dessas aplicações.

Para apoiar a execução das aplicações traduzidas em um *cluster*, provendo a abstração de uma memória compartilhada no sistema distribuído, foi utilizado o *software* DSM *Clik*, cujas características são mencionadas na Seção 4.5.1. Os experimentos apresentados nesta seção tem como objetivo investigar se o modelo SPMD usado com a linguagem *Cilk* na tradução, provê vantagens e facilidades para aumentar o desempenho de aplicações OpenMP em um *cluster*.

5.3.1 Descrição do Ambiente

A configuração do ambiente, utilizado neste experimento, é mostrada a seguir:

- 8 CPUs Pentium IV 32 bits / 2.8 GHz ;
- 1 GB RAM ;
- 1 MB de cache ;
- Kernel Linux versão 2.6.7 ;
- Sistema Clik (baseado no Cilk versão 5.3.2) ;
- Compilador Intel 9.1 para *Cluster OpenMP*;

5.3.2 Validação

As mesmas aplicações usadas, no experimento anterior, foram executadas, no *cluster*, para a validação das aplicações e dos mapeamentos, neste ambiente. Para isto, foram geradas duas novas versões (*em OpenMP e em Cilk*) de cada aplicação, baseadas nas versões usadas no multiprocessador. A versão traduzida das aplicações para um multiprocessador, não poderia ser aplicada diretamente em um *cluster*. Como exposto na Seção 4.5, foi necessário adotar os mapeamentos e regras pertencentes à extensão do mapeamento para *clusters*, a partir do código traduzido para multiprocessadores. De forma análoga, diretivas especiais [6] foram adotadas na versão com o padrão OpenMP das aplicações, para uma implementação particular do padrão provida pelo compilador da *Intel* [4]. Esta implementação é capaz de manter uma memória compartilhada no *cluster*, com o auxílio do *software Treadmarks* [19].

As versões *em Cilk* das aplicações foram executadas no *cluster*, utilizando o SDSM *Clik*. Uma particularidade deste software DSM, em relação ao sistema *Cilk*, para multiprocessadores, é a de que no *Clik* pode ser definido uma quantidade superior a uma *thread*, em cada nó de processamento. Estas *threads* recebem o nome no *Clik* de *threads trabalhadoras*, ou simplesmente de **trabalhadores**. A fim de estabelecer a mesma quantidade total de *threads*, usadas na execução da versão *em OpenMP*, em cada nó foi criada somente uma *thread trabalhadora*, sendo escolhido um número de nós equivalente ao total de *threads* empregadas na versão *em OpenMP*. A quantidade de *threads* usadas para a versão com o OpenMP foi definida por variáveis de ambiente.

Para a execução de ambas as versões de cada aplicação, a menor classe de dados de entrada estabelecida pelo *benchmark* NAS, foi considerada. A correção da tradução, no *cluster* foi atestada a partir dos testes de verificação realizados pelo NAS. Assim como no experimento no multiprocessador, as construções em *Cilk* foram validadas no *cluster*, com a comparação das variáveis escritas nelas, em relação à diretiva do OpenMP correspondente.

5.3.3 Análise de Resultados

Esta seção apresenta os resultados de desempenho de algumas aplicações, dentre as utilizadas na validação, no *cluster*. Não foi possível a execução de todas as aplicações validadas, neste ambiente. Para a extensão do mapeamento no *Clik*, é necessário que variáveis, antes mapeadas como privativas, tornem-se compartilhadas e sejam alocadas

na área de DSM. Essa modificação gera um consumo extremamente alto de memória em muitas aplicações. Em particular, apenas três aplicações validadas puderam ser executas no ambiente, com uma quantidade suficiente de memória. Ainda assim, uma menor classe de dados de entrada foi empregada. A Tabela 5.3 apresenta as aplicações executadas para este experimento, com o tamanho da classe de dados utilizada e o tempo seqüêncial consumido por cada uma.

Aplicação	Classe	Tempo Sequencial (s)
EP	B	198.08
IS	B / 20 iterações	69.60
CG	NA=37520, NONZER=12, NITER=16, SHIFT=60.0	41.18

Tabela 5.3: Tempo Serial das Aplicações usadas no *Clik*

Para a aplicação CG, em especial, foi adotada uma classe de dados intermediária, não definida pelo NAS, cujo tamanho a qualifica como sendo a que sucede a menor de todas, definida pelo *benchmark*.

A figura 5.8 mostra os *speedups* obtidos para cada aplicação presente na Tabela 5.3. O número de tarefas foi alternado em 1, 2, 4 e 8, de modo que fosse igual à quantidade de nós no *cluster*. O número de trabalhadores por nó, no *Clik* foi fixado em 1, sendo, também, criada apenas uma *thread* por nó na implementação do OpenMP utilizada.

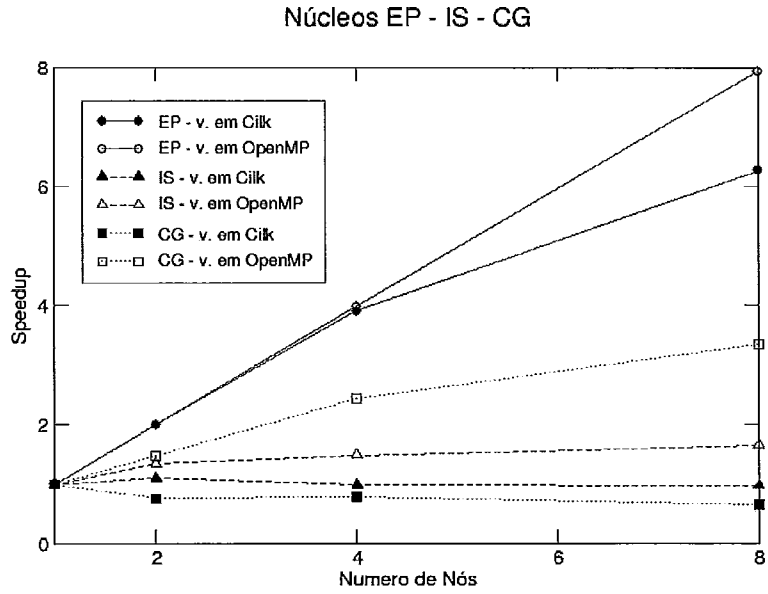


Figura 5.8: Desempenho dos Núcleos EP, IS e CG no Cluster

Embora a figura apresente os resultados de desempenho das duas versões das aplicação, não foi possível comparar os resultados entre as versões, no *cluster*, pelo fato do sistema *Clik* ser voltado para arquiteturas de 32 bits, enquanto a implementação do OpenMP, provida pelo compilador da Intel suporta apenas arquiteturas de 64 bits. Assim, os resultados de desempenho adquiridos com a versão *em OpenMP* foram utilizados apenas como referência para verificar se um ganho proporcional poderia ser obtido com as aplicações em *Cilk*.

Pela figura percebe-se que a *versão em Cilk* de todas as aplicações, com exceção do núcleo EP, adquiriram *slowdown*, enquanto a *versão em OpenMP* das mesmas aplicações obtiveram *speedup*.

Para identificar a causa do *slowdown* dos núcleos IS e CG, no *Clik*, foi realizada uma análise de *breakdown* dessas aplicações. O *breakdown* de cada uma é apresentado nas seções seguintes.

5.3.3.1 *Breakdown* do Núcleo *IS*

As Figuras 5.9, 5.10 e 5.9 mostram o *breakdown* do núcleo IS utilizando-se 2, 4 e 8 nós, respectivamente. Cada barra representa o *breakdown* de um trabalhador. A legenda apresentada na figura revela os seguintes componentes do tempo total da aplicação:

- *Computação* - tempo decorrido apenas com o algoritmo da aplicação;
- *Computação de Diffs* - cálculo das diferenças aplicadas a páginas;
- *Intervalo de Finalização* - tempo decorrido desde a última *thread* executada até o término da aplicação;
- *Atualização de Página* - tempo gasto com o envio de *diffs*;
- *Busca por Trabalho* - tempo gasto com o algoritmo de roubo de trabalho;
- *Handler* - tempo decorrido com falha de acesso a páginas;
- *Aplicação de WN* - tempo de notificação de escrita em uma página;
- *Finalização* - tempo decorrido para o término da aplicação;

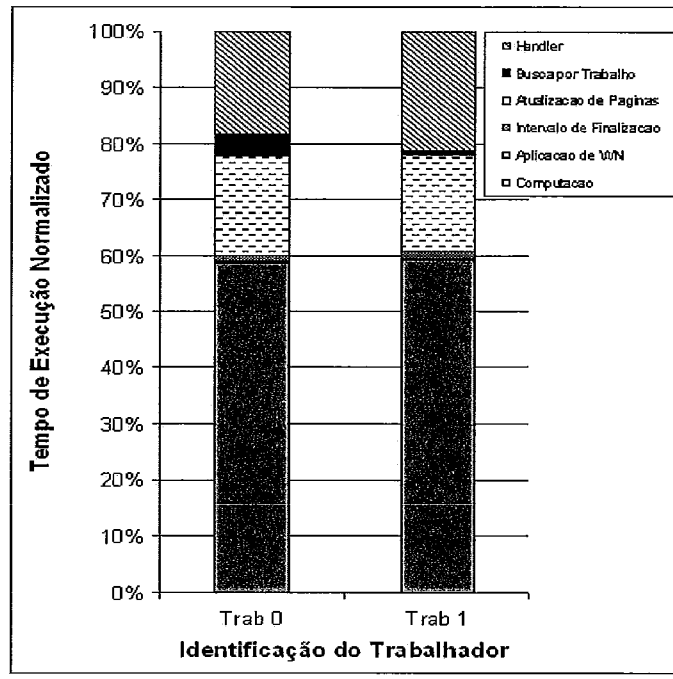


Figura 5.9: *Breakdown* do Núcleo IS com 2 nós

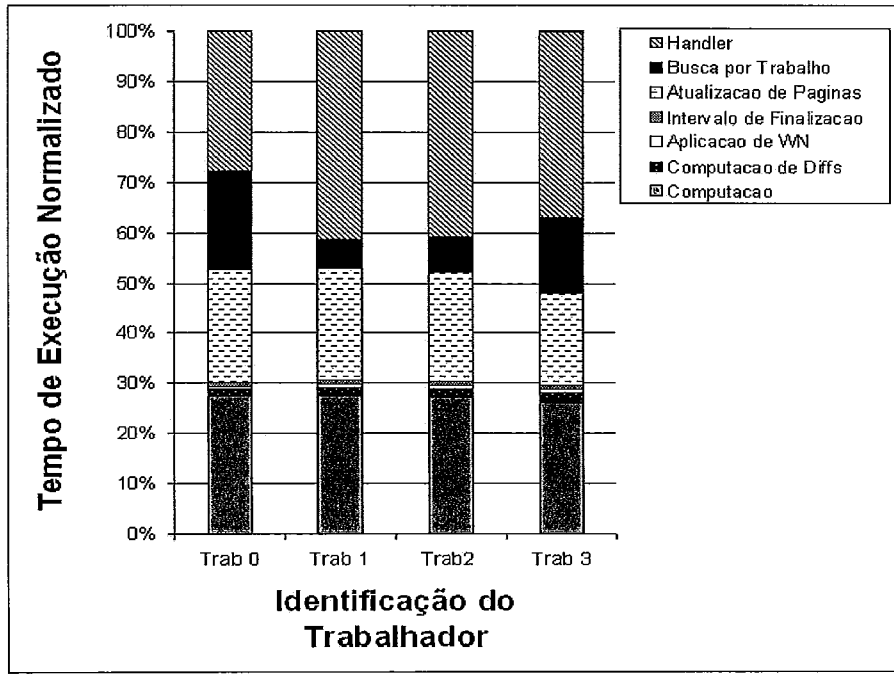


Figura 5.10: *Breakdown* do Núcleo IS com 4 nós

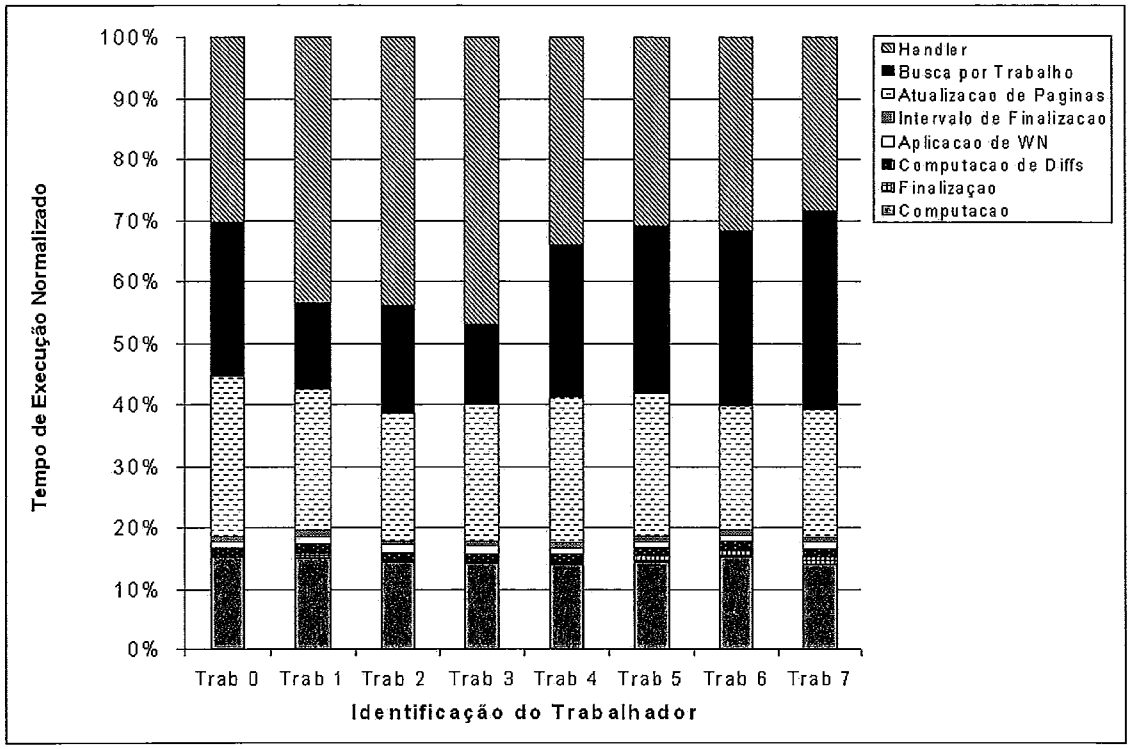


Figura 5.11: *Breakdown* do Núcleo IS com 8 nós

De acordo com estatísticas providas pelo *Clik*, o tempo de computação no IS reduz com o aumento do número de nós. Apesar desta redução, o tempo despendido com os *overheads* do sistema aumenta gradativamente, fazendo com que a aplicação adquira *slowdown*. Dentre estes *overheads*, aqueles que justificam a degradação do desempenho são o tempo gasto com *handler*, com *atualização de páginas* e com *busca por trabalho*, como se pode observar nos gráficos.

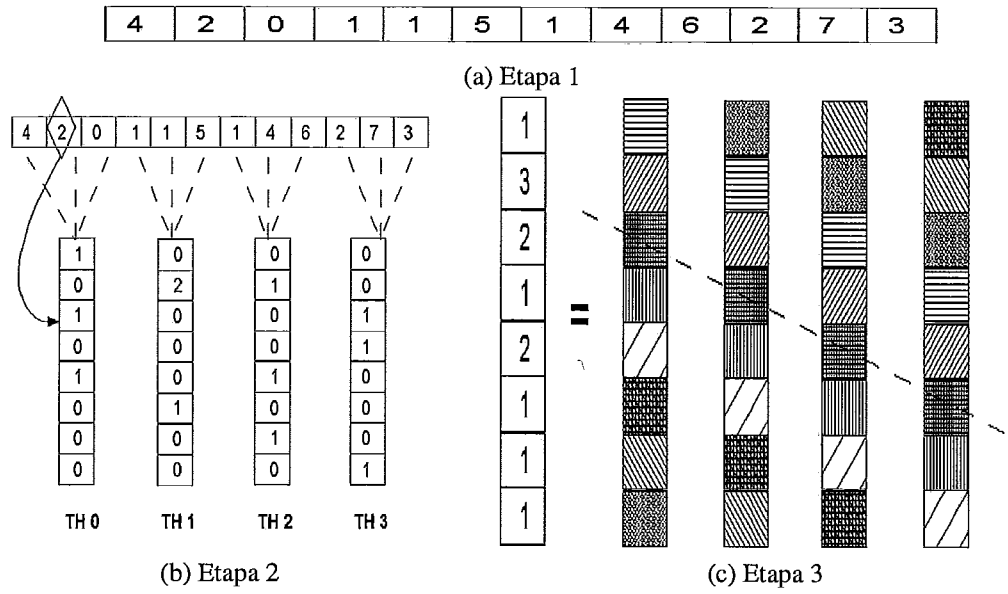


Figura 5.12: Etapas do Algoritmo do Núcleo IS

O algoritmo presente no núcleo IS funciona fundamentalmente com as etapas apresentadas na Figura 5.12. Primeiramente é gerado um vetor de inteiros desordenado (5.12a), por uma função de geração de números randômicos. O vetor desordenado é dividido em partes iguais, de acordo com o número de *threads* que executarão a aplicação, para a leitura de elementos (chaves) a serem usados nesta etapa inicial. No exemplo, o vetor de 12 elementos foi particionado, de forma que cada uma, das 4 *threads* criadas, leêm 3 elementos do vetor, distintamente. Na etapa seguinte do algoritmo, são contabilizadas as freqüências das chaves pertencentes ao vetor. De acordo com a seta pontilhada, mostrada na Figura 5.12b, cada *thread* usa as 3 chaves, que compõem a sua partição do vetor, como índice de um vetor temporário que cada *thread* possui para contabilizar a freqüência das suas chaves. Na etapa final, a freqüência das chaves são sobrepostas pelo número de ocorrências encontrado para cada chave pelas *threads*. É utilizado, para isto, um vetor compartilhado com as freqüências resultantes das chaves. Esta etapa foi paralelizada no algoritmo, de forma que enquanto uma *thread* copia, para uma faixa do vetor resultante, a freqüência identificada pela *thread* das chaves encontradas nesta faixa, outra *thread* copia a freqüência por ela contabilizada de uma faixa distinta. Na Figura 5.12c, as faixas de mesma textura, verificadas no vetor temporário de cada *thread*, expressam a coordenação das escritas no vetor compartilhado. Em um instante de tempo T, as *threads* estarão escrevendo freqüências de chaves distintas no vetor.

A escrita coordenada dos resultados na última etapa do algoritmo é realizada por meio de primitivas de sincronização. Na versão traduzida da aplicação, portanto, foram introduzidos *syncs* para prover a sincronização. No modelo de execução do *Clik*, no entanto, a única maneira de fazer com que o vetor temporário, de cada *thread*, com as freqüências das chaves persistisse entre a execução de um ponto de sincronização e outro, é através da criação de um vetor compartilhado, na memória compartilhada. Em função do tamanho da instância de mapeamento adotada a esta variável, o tempo relacionado com atualizações e falta de página, na versão *Cilk*, aumentou consideravelmente, sendo a principal razão pelo *slowdown*.

A queda de desempenho também é atribuída ao tempo despendido com busca por trabalho. Este tempo tornou-se alto em virtude de uma quantidade razoável de trechos seqüenciais do código originalmente presente no *benchmark*. Enquanto apenas uma *thread* tornou-se responsável por executar estes trechos de código, outras executavam o algoritmo de roubo de trabalho, cujo *overhead* de busca por trabalho passou a afetar a aplicação.

5.3.3.2 *Breakdown* do Núcleo *CG*

O *breakdown* da aplicação CG para 2, 4 e 8 nós é apresentado nas Figuras 5.13, 5.14 e 5.15.

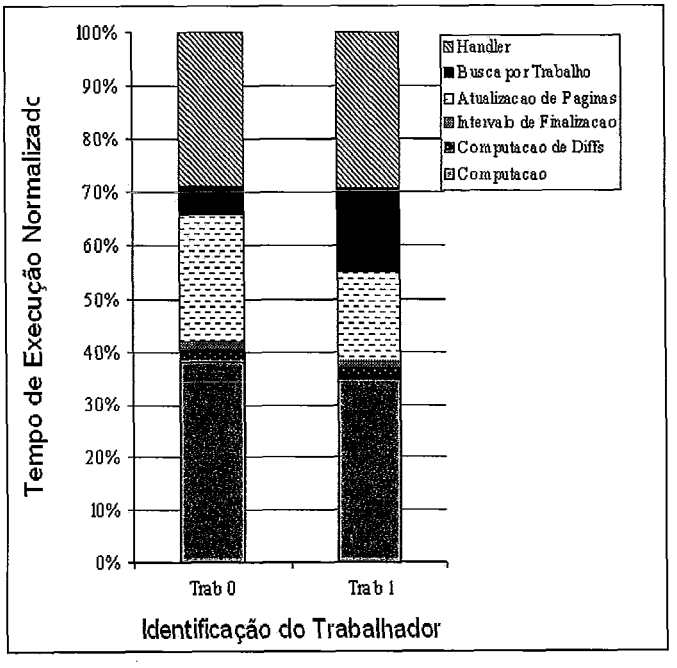


Figura 5.13: *Breakdown* do Núcleo CG com 2 nós

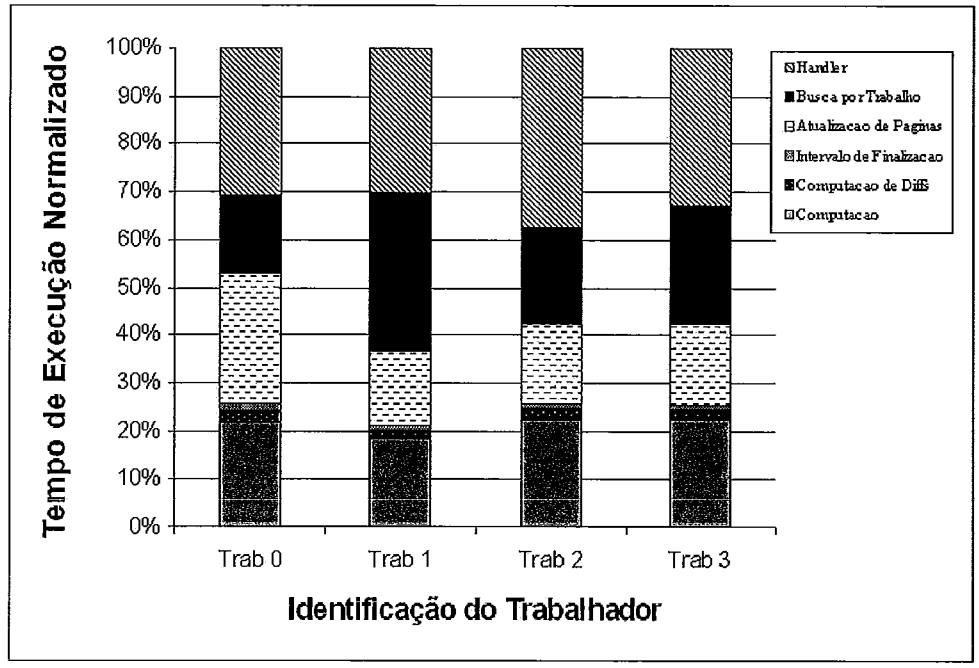


Figura 5.14: *Breakdown* do Núcleo CG com 4 nós

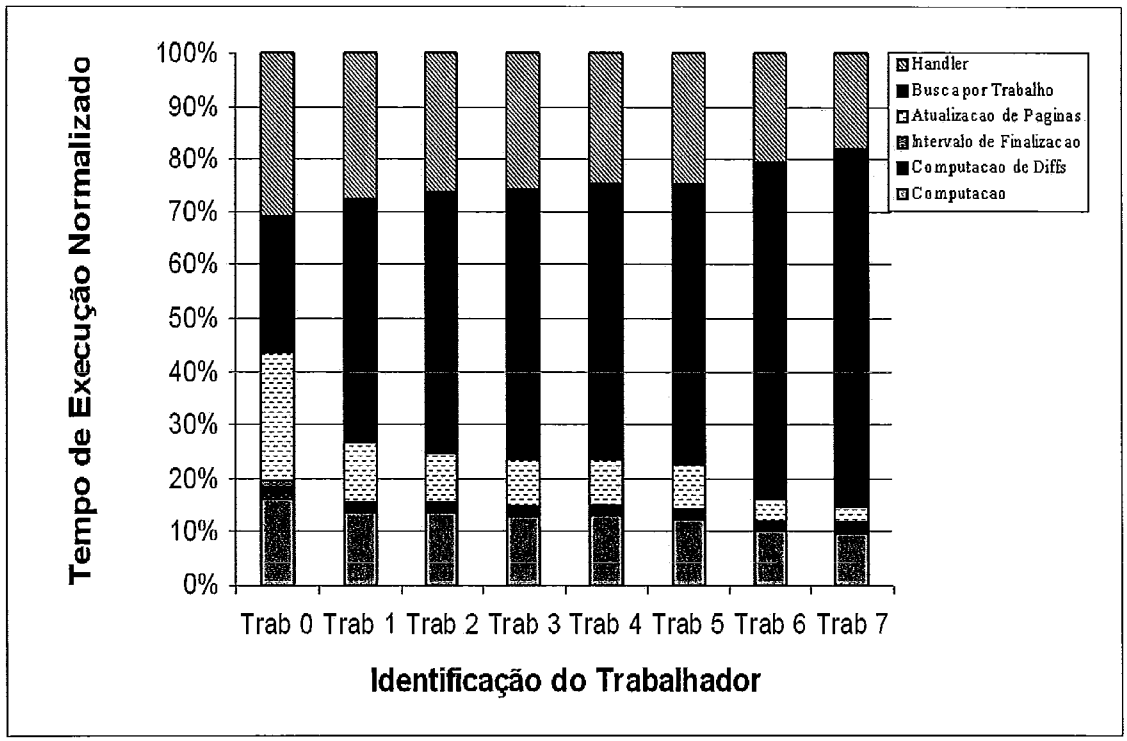


Figura 5.15: *Breakdown* do Núcleo CG com 8 nós

Da mesma forma que o núcleo IS, embora o tempo com *computação* regrida com o acréscimo de nós, os *overheads* tornam-se altos o suficiente para fazer com que a aplicação obtenha *slowdown*.

Pelos gráficos, nota-se um padrão que ocorre com o tempo consumido com busca por trabalho. Este tempo torna-se maior com o emprego de mais nós. Nesta aplicação, além da existência de longos trechos seriais, aqueles que são paralelizados contêm tarefas de pequena granularidade para serem executadas pelos trabalhadores criados no *Clik*.

A existência de tarefas de pequena granularidade fazem com que um trabalhador que obtenha uma dessas tarefas execute ela rapidamente, e, logo após adquira outra. Este rápido consumo de tarefas impede que outro trabalhador, que tenha enviado uma mensagem de roubo de trabalho, consiga obter uma tarefa, quando o envio da mensagem de roubo e a entrega da tarefa custam mais tempo do que a própria execução da tarefa. Isto implica no aumento do *overhead* com busca por trabalho, como aconteceu para o caso da CG.

Não se pode afirmar, no entanto, que o aumento da granularidade de tarefas é capaz de diminuir a busca por trabalho, com base apenas no desempenho analisado do núcleo CG neste experimento. Isto, na realidade, é averiguado com um outro experimento apresentado à seguir.

5.3.3.3 Aumento da Granularidade de Tarefas

Os experimentos apresentados nesta seção tem como finalidade verificar se os *overheads* que afetam o desempenho de uma aplicação, *em Cilk*, com pequena granularidade, diminuem com o aumento da granularidade, no sistema *Clik*.

Duas aplicações em potencial para serem usadas neste experimento, por apresentarem *slowdown*, seriam, primeiramente, a CG por, inclusive, possuir muitas tarefas de granularidade fina, e também o núcleo IS. A CG, porém, revela muitas dependências de dados, impossibilitando a agregação de tarefas para tornar a granularidade delas mais grossa. Já no IS foi verificado que quaisquer tentativas de aumentar a granularidade provocam a alteração do algoritmo do núcleo.

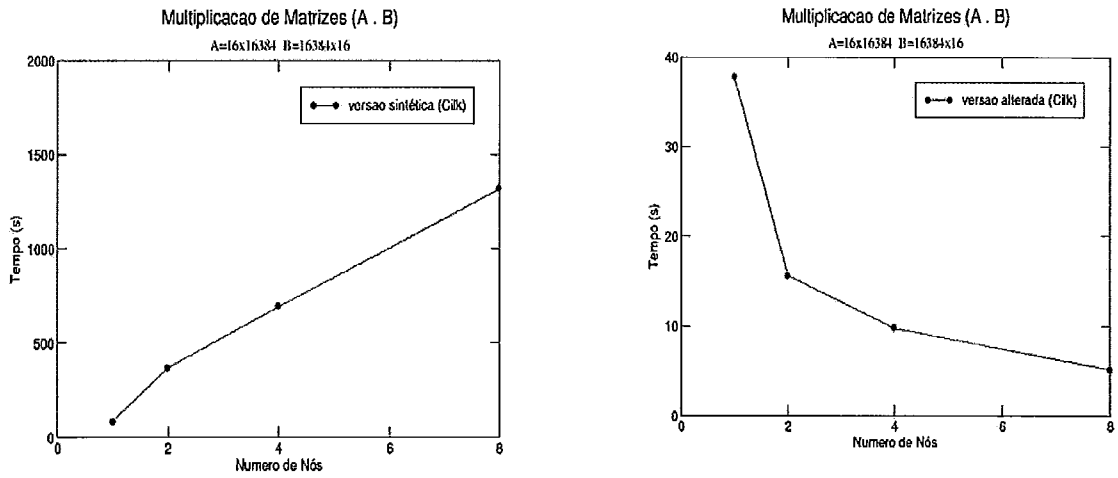
Assim, foi utilizada uma aplicação sintética, ou seja, escrita com o objetivo de se realizar o experimento. A aplicação efetua a multiplicação de duas matrizes não quadradas.

Foram escritas duas versões da aplicação em *Cilk* para serem executadas, com o sistema *Clik*, no mesmo ambiente descrito na Seção 4.5.1. Na primeira versão, a fase da multiplicação dos elementos, entre as duas matrizes, é separada da fase de soma dos elementos, que gera a matriz resultante. Na segunda versão, ambas as fases são agrupadas,

a fim de aumentar a granularidade da tarefa atribuída a cada *thread*. As duas versões da aplicação empregam, como dados de entrada, os seguintes números de linhas e colunas:

- Matriz A : Número de Linhas = 16 ; Número de Colunas = 16384
- Matriz B : Número de Linhas = 16384 ; Número de Colunas = 16

O desempenho das duas versões, em *Cilk*, são mostrados na Figura 5.16, onde o tempo seqüencial de ambas foi de aproximadamente 40 segundos. A Figura 5.16a apresenta o tempo total de execução da primeira versão da aplicação sintética, onde tarefas de menor granularidade são atribuídas às *threads*. O resultado obtido com a utilização de tarefas com granularidade maior, na segunda versão da aplicação, pode ser verificado na Figura 5.16b. O tempo total de execução é apresentado, em função do número de nós, sendo criada uma única *thread* trabalhadora em cada nó.



(a) Versão Sintética

(b) Versão Alterada

Figura 5.16: Tempo Total de Execução da Aplicação MM (Multiplicação de Matrizes)

Pelos gráficos, é possivel verificar que o tempo reduz com o aumento da granularidade das tarefas. Este aumento foi obtido, especificamente, com a junção das duas rotinas, criadas na primeira versão da aplicação, para realizar as duas fases descritas do algoritmo.

Em especial, os *overheads* de atualização e espera por páginas, e de busca por trabalho, existentes no *Clik*, agem de maneira dominante na primeira versão da aplicação. Os *overheads* consumiram, em média, 301.78s, 633.84s e 1258.16s do total de execução da aplicação, usando 2, 4 e 8 nós, respectivamente. Pela Figura 5.16a, percebe-se que o tempo gasto com estes *overheads* é bem próximo do tempo total de execução nesta

versão. O tempo com computação útil, somado com os *overheads* mencionados, basicamente resultam no tempo total de execução. Sendo assim, o tempo com computação útil demonstra-se bem inferior, em relação ao tempo consumido com os *overheads*.

O grande impacto que os *overheads* citados exercem nesta versão da aplicação deve-se à quebra das duas fases do algoritmo utilizado. Esta quebra gerou a criação de duas rotinas em *Cilk* e a inclusão de um ponto de sincronização ao final de cada uma. Desta forma, o tempo para que todas as páginas necessárias, na execução de cada rotina, fossem enviadas aos nós, e ainda, o tempo de atualização das páginas, entre uma rotina e outra, representaram um custo muito alto, se considerada a granularidade da computação atribuída às duas rotinas. A granularidade fina afetou o desempenho, na medida que provocou um desbalanceamento da computação útil entre as *threads* trabalhadoras, já que umas adquiriram mais tarefas do que outras.

Na segunda versão da aplicação, os *overheads* provenientes do *Clik* diminuem de maneira considerável, de forma que o tempo com computação útil passa a ser dominante. Com a junção das rotinas, eliminou-se um ponto de sincronização, reduzindo a quantidade de atualizações e espera por páginas. Os *overheads* relacionados às atualizações e espera por páginas não geram um custo muito alto, em função da maior granularidade atribuída à computação encontrada na rotina. Esta rotina agregada permitiu a criação de tarefas com granularidade mais grossa, o que por sua vez fez com que algumas *threads* executassem uma tarefa por um período maior de tempo, enquanto outras tornaram-se aptas a adquirir outra tarefa, para a execução de forma balanceada da aplicação.

Com este experimento, foi possível demonstrar que os *overheads* que afetam o desempenho de uma aplicação com baixa granularidade de tarefas, no *Clik*, podem ser reduzidos com o aumento da granularidade, de tal modo que uma aplicação com *slowdown*, pode adquirir *speedup*, como ocorreu para a aplicação de multiplicação de matrizes.

5.4 Discussão

Pelos experimentos realizados foi verificada a correção da tradução de aplicações do conjunto NAS originalmente escritas com o OpenMP em dois ambientes distintos, ou seja, em um multiprocessador e em um *cluster* de computadores.

A efetividade da tradução foi atestada através da comparação de desempenho das aplicações do NAS, escritas com o OpenMP, e da versão traduzida de cada uma no multiprocessador. Este ambiente demonstrou-se adequado neste experimento por não introduzir

overhead com comunicação tão significativo quanto em um *cluster* de computadores. Além disso, os multiprocessadores são largamente empregados pelo padrão OpenMP e pelo sistema *Cilk*. Todas as aplicações traduzidas apresentaram um desempenho semelhante ao observado com a versão original delas. Este resultado assegurou a efetividade da tradução para esta plataforma.

Através de um *cluster* de computadores alguns resultados de desempenho de aplicações previamente traduzidas para multiprocessador foram obtidos. As estruturas de escopo dados e comunicação destas aplicações receberam o mapeamento segundo a extensão do mapeamento proposta para *clusters*. Assim, as aplicações traduzidas puderam ser usadas neste ambiente com o auxílio do *software DSM Clik*, e a versão destas aplicações escritas com o OpenMP, através de uma implementação do OpenMP para *clusters*. Entretanto, a arquitetura suportada pelo *Clik* e por esta implementação do OpenMP são incompatíveis. Neste sentido, os resultados de desempenho são considerados apenas preliminares, na medida que tornou-se relevante adquirir uma noção quantitativa do desempenho das aplicações traduzidas para o *Clik*. Esses resultados permitiram verificar que o modelo SPMD de programação aplicado à linguagem Cilk pela tradução provoca algumas desvantagens diante dos *overheads* existentes no *software DSM Clik*.

No experimento em *clusters* observou-se que uma aplicação que possui variáveis mapeadas com uma **instância global do mapeamento privativo** são suscetíveis à perda de desempenho no *Clik*. Esta perda é atribuída ao *overhead* necessário para tornar a variável mapeada, de escopo privativo, persitente nas rotinas desmembradas onde é utilizada. Como o *Clik* não suporta variáveis globais, a única maneira de torná-la persistente é alocando-a na memória compartilhada, o que aumenta o *overhead* com atualizações de páginas.

Cabe ressaltar que as rotinas desmembradas na tradução são criadas devido à existência de pontos de sincronização pela utilização de diretivas de barreira, ou ainda, pela ausência de cláusulas *nowait*. Em um programa escrito com o OpenMP estes pontos sincronização são responsáveis pela sincronização de todas as *threads* que executam uma região paralela. O modelo de programação da linguagem *Cilk*, no entanto, não permite a sincronização de maneira global, apenas as *threads-filhas* da *thread* corrente são sincronizadas. Este fato requer o retorno de rotinas para manter a sincronização entre as *threads*. Este retorno está diretamente relacionado ao desmembramento de rotinais que resulta na criação de tarefas de menor granularidade do que com a utilização de uma região paralela com o OpenMP. Desta forma, o modelo SPMD de programação é melhor explorado no

OpenMP com a criação de tarefas de granularidade mais grossa, expressa pelo segmento presente em uma região paralela.

No experimento com o *Clik*, foi verificado que uma aplicação já paralelizada pelo OpenMP com muitas tarefas de granularidade fina pode ter seu desempenho afetado pelo *overhead* de busca por trabalho. Em uma aplicação como esta, uma única *thread* trabalhadora, no *Clik*, adquire tarefas e as executa com uma freqüência que impede que outras *threads* façam o mesmo. Isto gera um desbalanceamento de tarefas entre as *threads* criadas. Este desbalanceamento é provocado sobretudo porque no *Clik* torna-se difícil estabelecer uma quantidade fixa de tarefas entre *threads*, como proposto pelo modelo SPMD. Apesar da liguagem *Cilk* permitir que o modelo SPMD seja expresso com a definição inicial de uma mesma quantidade de tarefas entre *threads*, ainda assim essas tarefas são atribuídas dinamicamente às *threads*. Desta forma, umas recebem eventualmente mais tarefas do que outras. Este aspecto, gera um impacto grande em um *cluster* pois, ao contrário de um multiprocessador, o *overhead* associado ao algoritmo de roubo de trabalho aumenta devido ao maior tempo despendido com mensagens de roubo.

No experimento, o aumento da granularidade de tarefas, entretanto, demonstrou que a distribuição de tarefas pode se tornar mais equilibrada. Conseqüentemente, os *overheads* que afetam uma aplicação com baixa granularidade podem ser reduzidos, aumentando o desempenho da aplicação.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho foi apresentada uma proposta de tradução de aplicações paralelas escritas com o padrão OpenMP para a linguagem Cilk, que emprega um modelo de execução distinto do utilizado pelo OpenMP. A tradução inclui o mapeamento de estruturas do OpenMP para Cilk, e também um processo de tradução para coordenar este mapeamento.

Foi verificado o potencial da tradução quando esta é aplicada a multiprocessadores. Resultados preliminares obtidos com a execução de aplicações traduzidas para *cluster*, através do *SDSM Clik*, permitiram verificar que o modelo SPMD mantido com a tradução para Cilk pode aumentar os *overheads* existentes neste ambiente. O modelo SPMD não gera *overheads* significativos em um multiprocessador devido à baixa latência na comunicação entre os processadores.

Os resultados obtidos com a tradução para *cluster*, permitiram constatar que a existência de muitos pontos de sincronização em regiões paralelas do OpenMP geram tarefas de baixa granularidade no programa traduzido. Isto afeta as aplicações traduzidas para *cluster*, pois aumenta o *overhead* na execução do algoritmo de roubo de trabalho devido a latência mais alta na comunicação entre os processadores neste ambiente.

Também tornou-se difícil manter o modelo SPMD com o *Clik*, em virtude do sistema não distribuir estaticamente tarefas entre nós do *cluster*, de modo que as *threads* criadas para uma aplicação recebessem a mesma quantidade de tarefas para executar.

Deste modo, a extensão do OpenMP para *clusters* a partir da tradução para o *Clik*, não trouxe benefícios com a permanência do modelo SPMD na tradução proposta, de acordo com os resultados preliminares obtidos. Porém, existe a possibilidade de que a extensão usando o *Clik* traga vantagens com um estudo que se concentra na tradução voltada para o modelo de programação dinâmico e assíncrono da linguagem *Cilk*. Ou seja, sem o uso do modelo SPMD no programa traduzido.

Uma das aplicações traduzidas no experimento em *clusters* demonstrou que o mapeamento de variáveis privativas que persistem em todo o programa com o OpenMP, traz um efeito negativo quando aplicada a extensão do mapeamento neste ambiente com a alocação da variável na memória compartilhada. Primeiro, porque o mapeamento gera um consumo alto de memória para as aplicações que fazem muito uso das variáveis associadas ao mapeamento. Segundo, porque estas aplicações sofrem queda de desempenho com o aumento dos *overheads* de atualizações e espera por página existentes no *Clik*. Vislumbra-se que a integração do *Clik* com uma camada de software para gerenciamento de memória, denominada *Momemto* [29], será capaz de atender ao consumo de memória das aplicações que possuem variáveis do tipo mencionado. Os *overheads* para manutenção destas variáveis na memória compartilhada, podem, ainda assim, afetar o desempenho destas aplicações. Para reduzir estes *overheads* é passível de investigação o suporte a variáveis globais no *Clik*, provido por um mecanismo que gere menos custo do que a alocação na memória compartilhada.

Identificadas soluções capazes de diminuir o efeito dos fatores descritos que prejudicam a tradução proposta na execução de aplicações no *Clik*, torna-se interessante a automatização do processo de tradução. Desta forma, o padrão OpenMP pode ser estendido a *clusters* com o uso do *Clik*, sendo apenas necessária a tradução automática de estruturas do OpenMP para a linguagem *Cilk*.

Bibliografia

[1] OpenMP Architecture Review Board OpenMP: Simple, Portable, Scalable SMP Programming http://www.openmp.org. Acesso em: Junho/2007.

[2] Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano and Yoshio Tanaka. Design of OpenMP Compiler for an SMP Cluster In *Proceedings of the 1st European Workshop on OpenMP (EWOMP'99)*, pp. 32-39, Sep. 1999.

[3] H. Lu and Y. C. Hu and W. Zwaenepoel. OpenMP on Network of Workstations In *Proceedings of Supercomputing'98*, 1998.

[4] Intel Corporation. Intel C/C++ Compilers. http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm. Acesso em: Junho/2007.

[5] Intel Corporation. Cluster OpenMP for Intel Compilers. http://www.intel.com/cd/software/products/asmo-na/eng/329023.htm. Acesso em: Junho/2007.

[6] Jay P. Hoeflinger. Extending OpenMP to Clusters Copyright 2006, Intel Corporation.

[7] S. Rafael Mendes. Técnicas para Execução Eficiente de Aplicações Multithread em um Cluster de Computadores Tese de Mestrado, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, 2005.

[8] MIT Laboratory for Computer Science Cilk 5.3.2 Reference Manual, November, 2001. http://supertech.lcs.mit.edu/cilk. Acesso em: Junho/2007.

[9] Don Dailey and Charles E. Leiserson Using Cilk to Write Multiprocessor Chess Programs citeseer.ist.psu.edu/dailey01using.html. Acesso em: Junho/2007.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks *The International Journal of Supercomputer Applications*, 5(3):63-73, Feb. 1991.

[11] Peter Pacheco. Parallel Programming with MPI *Morgan Kaufmann Publishers Inc.*, October, 1996.

[12] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing *Concurrency: Practice and Experience*, vol. 2(4), pp. 315–339, Dec. 1990.

[13] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. Pthreads Programming: A POSIX Standard for Better Multiprocessing *O'Reilly Associates*, Sep. 1996.

[14] Robert D. Blumofe and Charles E. Leiserson. Space-efficient Scheduling of Multithreaded Computations In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 362-371, May 1993.

[15] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, Nov. 1994.

[16] Ioannis E. Venetis and Dimitrios S. Nikolopoulos and Theodore S. Papatheodorou. A Transparent Operating System Infrastructure for Embedding Adaptability to Thread-Based Programming Models *Lecture Notes in Computer Science*, vol(2150), pp. 514, Feb. 2001.

[17] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menon. Parallel Programming in OpenMP *Morgan Kaufmann Publishers Inc.*, 2001.

[18] M. Rangarajan and L. Iftode. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance In *Proceedings of the 3th Extreme Linux Workshop*, pp. 341-352, Oct. 2000.

[19] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared Memory Computing on Networks of Workstations IEEE Computer, 29(2):18-28, 1996.

[20] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. An Overview of the SUIF Compiler for Scalable Parallel Machines In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.

[21] Y. S. Kee, J. S. Kim, and S. Ha. ParADE: An OpenMP Programming Environment for SMP Cluster Systems In *Proceedings of SC'03*, pp. 15-21, Nov. 2003.

[22] A. Basumallik, S. -J. Min, and R. Eigenmann. Towards OpenMP Execution on Software Distributed Shared Memory Systems In *ISHPC - LNCS 2327*, pp. 457-468, 2002.

[23] Xavier Martorell et al. Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors In *Proceedings of 13th international conference on Supercomputing*, pp. 294-301, Jun. 1999.

[24] Jorge A. Cobb and Mohamed G. Gouda. The Request-Reply Family of Group Routing Protocols *IEEE Transactions on Computers*, v. 46, n. 6, pp. 659-672, Jun. 1997.

[25] George Cybenko and Lyle Kipp and Lynn Pointer and David Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks In *Proceedings of 1990 International Conference on Supercomputing*, pp. 254-266, v.18, n3, 1990.

[26] C. Amza and A. L. Cox and S. Dwarkadas and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer In *Proceedings of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, pp. 261-271, 1997.

[27] C. Amza and A. L. Cox and S. Dwarkadas and L-J. Jin and K. Rajamani and W. Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory In *Proceedings of the IEEE, Special Issue on Distributed Shared Memory*, pp. 467-475, v. 87, n. 3, 1999.

[28] Ramesh C. Agarwal and Bowen Alpern and Larry Carter and Fred G. Gustavson and David J. Klepacki and Rick Lawrence and Mohammad Zubair. High-Performance Parallel Implementations of the NAS Kernel Benchmarks on the IBM SP2 *IBM Systems Journal*, pp. 263-272, v. 34, n. 2, 1995.

[29] T. Trevisan, C. Amorim, L. Whately and V. Costa Distributed Shared Memory in Kernel Mode In *Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing (SCAB-PAD'02)*, pp. 159-168, Oct. 2002.