PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EXPLORANDO PROGRAMAÇÃO HÍBRIDA NO CONTEXTO DE *CLUSTERS* DE MÁQUINAS *NUMA*

NEUMAR SILVA RIBEIRO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes

Porto Alegre
2011

Dados Internacionais de Catalogação na Publicação (CIP)

R484e Ribeiro, Neumar Silva
Explorando programação híbrida no contexto de cluters de máquinas de clusters de máquinas NUMA / Neumar Silva Ribeiro. – Porto Alegre, 2011.
81 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes.

1. Informática. 2. Arquitetura de Computador. 3. Sistemas Híbridos. I. Fernandes, Luiz Gustavo Leão. II. Título.

CDD 004.22

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "**Explorando Programação Híbrida no Contexto de Clusters de Máquinas NUMA**", apresentada por Neumar Silva Ribeiro, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído , aprovada em 29/03/2011 pela Comissão Examinadora:

Prof. Dr. Luiz Gustavo Leão Fernandes– Orientador — PPGCC/PUCRS

Prof. Dr. César Augusto Fonticielha De Rose - — PPGCC/PUCRS

Prof. Dr. Rodrigo da Rosa Righ – — UNISINOS

Homologada em..24/05/11....., conforme Ata No. ...008... pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320–3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

Para minha querida Esposa e Meus Pais.

AGRADECIMENTOS

Primeiramente, quero agradecer a Deus por ter me abençoado, e ter feito eu perserverar até aqui.

A minha Esposa, que há nove anos é uma companheira que me dá o suporte necessário para todas as minhas conquistas.

A meus pais, João Batista e Marlene, e irmãos Edimar e Joselaine por tudo, principalmente pelo apoio durante todos estes anos de estudo.

Ao orientador Gustavo pela confiança e aposta, e sobretudo, pelo incentivo.

Ao pessoal da Coordenadoria de Informática da Procuradoria da República no RS, principalmente à seção de suporte técnico, onde atuo diretamente, pelo apoio e incentivo durante este projeto.

A todos os colegas pelas trocas de experiências, ajudas, mas, principalmente, pelo companheirismo. Em especial aos colegas do GMAP.

Ao LAD (Laboratório de Alto Desempenho) da PUCRS nas figuras do professor De Rose, por disponibilizar a estrutura para execução dos testes, e do colega Antonio de Lima, que colaborou muito na realização deste trabalho, por sua disposição.

A todos os amigos, que muitas vezes tive que deixar na mão, para poder terminar este projeto.

Meu muito obrigado a todos! Com certeza sem vocês a realização deste projeto não seria possível.

EXPLORANDO PROGRAMAÇÃO HÍBRIDA NO CONTEXTO DE *CLUSTERS* DE MÁQUINAS *NUMA*

RESUMO

Normalmente, utiliza-se o paradigma de troca de mensagens quando se está programando uma arquitetura do tipo *cluster*. Porém, quando se deseja programar uma máquina multiprocessada, é requirido o paradigma de memória compartilhada. Recentemente, o surgimento de novas tecnologias possibilitou a criação de *clusters* com nós multiprocessados. Nestas arquiteturas os nós são compostos por mais de um processador ou *core*, e compartilham a mesma memória. Este cenário, cria a possibilidade de usar novos modelos de programação híbrida. No amplo espectro de soluções possíveis para o desenvolvimento de código híbrido para *clusters* de máquinas multiprocessadas, a utilização da dupla MPI e OpenMP está emergindo como um padrão de fato. A maioria dos códigos híbridos MPI e OpenMP são baseados em um modelo de estrutura hierárquica, que torna possível a exploração de grãos grandes e médios de paralelismo no nível de MPI, e grão fino no paralelismo no nível do OpenMP. O objetivo é claramente tirar vantagens das melhores características de ambos os paradigmas de programação. Os nós desses *clusters* podem ainda ser máquinas *NUMA* (*Non-Uniform Memory Access*). Estas máquinas com acesso não uniforme à memória possibilitam que o desenvolvedor explore afinidade de memória, melhorando o desempenho da aplicação. O objetivo principal deste trabalho é investigar o uso de programação híbrida com MPI e OpenMP em *clusters* de máquinas NUMA, explorando afinidade de memória, visando identificar um conjunto de boas práticas de programação híbrida a serem utilizadas neste contexto.

Palavras-chave: MPI, OpenMP, Programação Híbrida, Máquinas NUMA, *Cluster*.

EXPLORING HYBRID PROGRAMMING IN THE CONTEXT OF CLUSTERS OF NUMA MACHINES

ABSTRACT

Typically, the message passing paradigm is the programming model used for cluster architectures. On the other hand, programming for multiprocessor architectures requires the shared memory paradigm. Recently, the emergence of new technologies enabled the creation of clusters with multiprocessor nodes. In these architectures, computing nodes are composed by more than one processor or core sharing the same memory. This scenario creates the possibility of using new hybrid programming models. In the spectrum of possible alternatives for hybrid programming for clusters of multiprocessors, the use of MPI along with OpenMP is emerging as a de-facto standard. Most hybrid MPI and OpenMP codes are based on hierarchical structure model, which allows exploring coarse and medium grain parallelism with MPI and fine grain parallelism with OpenMP. Clearly, the main idea is to take advantage of the best features of both programming paradigms. Additionally, the nodes of these clusters can be NUMA (Non-Uniform Memory Access) machines. These machines enable the developer to explore memory affinity, improving application performance. The main objective of this work is to investigate the use of hybrid programming with MPI and OpenMP in clusters of NUMA machines, exploiting memory affinity aiming at the identification of a set of good programming practices to be used in this context.

Keywords: MPI, OpenMP, Hybrid Programming, NUMA Machines, Cluster.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

NUMA	*Non-Uniform Memory Access*
cc-NUMA	*cache coherent Non-Uniform Memory Access*
SMP	*Symmetric Multiprocessor*
MPI	*Message Passing Interface*
API	*Application Programming Interface*
MAi	*Memory Affinity Interface*
CPU	*Central Processing Unit*
UMA	*Uniform Memory Access*
MPP	*Massively Parallel Processor*
ACPI	*Advanced Configuration and Power Interface*
SLIT	*System Locality Information Table*
SRAT	*System Resource Affinity Table*
INRIA	*Institut National de Recherche en Informatique et en Automatique*
RAM	*Random Access Memory*

SUMÁRIO

1. INTRODUÇÃO

Usualmente, utiliza-se o paradigma de troca de mensagens quando se está programando uma arquitetura do tipo *cluster* ou até mesmo uma máquina *MPP* (*Massive Paralell Processors*). Porém, quando se deseja programar uma máquina multiprocessada, como os computadores *SMPs* (*Symmetric Multiprocessor*), utiliza-se um paradigma de memória compartilhada. No entanto, o desenvolvimento de novas tecnologias possibilitou a criação de *clusters* com nós multiprocessados.

Com os nós de computação destes *clusters* compostos de mais de um processador ou *core*, e compartilhando a memória, é natural questionar a possibilidade de mudar os projetos de desenvolvimento de software que antes exploravam apenas o paradigma de troca de mensagens, para que agora também explorarem o paradigma de memória compartilhada. Códigos de troca de mensagens puros são, obviamente, compatíveis com *clusters* de nós multiprocessados. O ponto é que a comunicação por troca de mensagens dentro de um nó *SMP*, ou seja, na presença de uma memória compartilhada, não é necessariamente a solução mais eficiente. Existem algumas soluções possíveis para o desenvolvimento de código híbrido para memória compartilhada e memória distribuída, porém a utilização da dupla MPI [12] e OpenMP [20] está emergindo como um padrão de fato [24] [23]. A maioria dos códigos híbridos MPI e OpenMP são baseados em um modelo de estrutura hierárquica, que torna possível a exploração de grãos grandes e médios de paralelismo no nível de MPI, e grão fino no pa-ralelismo no nível do OpenMP. O objetivo é claramente tirar vantagens das melhores características de ambos os paradigmas de programação.

Os nós desses *clusters* podem ainda ser máquinas *NUMA* (*Non-Uniform Memory Access*). Estas máquinas com acesso não uniforme à memória possibilitam que o desenvolvedor especifique a localidade no momento de alocação da memória de modo a posicioná-la o mais próximo possível do *core* onde estará sendo executado o processo. Também é possível que o próprio sistema operacional defina a localidade utilizando um algoritmo de balanceamento de carga, por exemplo, baseado em estruturas chamadas *sched_domains* [10]. Porém esta ultima opção nem sempre garante um bom desempenho da aplicação.

1.1 Motivação e Objetivos

Com a maleabilidade do grão de paralelismo oferecido pela Programação Híbrida, ganhos de desempenho significativos podem ser obtidos em aplicações de alto desempenho. Este trabalho se propõe a desenvolver um estudo visando investigar maneiras de explorar os benefícios da programação híbrida no desenvolvimento de aplicações paralelas no contexto de *clusters* de *NUMAs*.

O objetivo principal deste trabalho é investigar o uso de programação híbrida com MPI e OpenMP em *clusters* de máquinas NUMA buscando o melhor dessa combinação. E como conclusão pretende-se obter as melhores práticas para ser utilizada na programação híbrida neste contexto.

Como caso de estudo desse trabalho, uma aplicação previamente paralelizada no âmbito do

GMAP está sendo utilizada: NUMA-ICTM (uma aplicação para Categorização de Modelos de Tesselações paralelizadas para máquinas NUMA com OpenMP) [8] . Essa aplicação também foi objeto de estudo do trabalho: HPC-ICTM: um Modelo de Alto Desempenho para Categorização de Áreas Geográficas [26].

1.2 Trabalhos Relacionados

O Programa de Pósgraduação em Ciência da Computação da PUCRS já possui trabalhos abordando tecnologias que são padrão para o densenvolvimento em aplicações paralelas como MPI e OpenMP. Pretende-se então utilizar esses trabalhos como ponto inicial para desenvolver uma solução híbrida. Este trabalho foi ideializado para utilizar os recursos que os *clusters* estão apresentando atualmente. Ou seja, os nós normalmente são *multicore* e, em algumas situações, também apresentam recursos de máquinas NUMA.

1.2.1 Paralelização ICTM

O ICTM (*Interval Categorizer Tessellation Model*) [2] é um modelo multi-camada baseado no conceito de tesselações para categorização de áreas geográficas. O conceito de multi-camada está baseado no fato de que a categorização de uma mesma região poderá considerar diferentes características tais como: sua topografia, vegetação, clima, uso do terreno, entre outras. Cada uma destas características é representada no modelo como sendo uma camada. Através de um procedimento apropriado de projeção em uma camada base, é possível construir uma categorização final, a qual permite uma análise de como estas características são combinadas. Esta análise possibilita, para os pesquisadores da área, uma compreensão geral de dependências mútuas entre elas. Não é objetivo deste trabalho apresentar maiores detalhes sobre o modelo ICTM, visto que outros trabalhos já apresentaram essas informações com qualidade.

De fato, é pretendido apresentar os dois trabalhos que já foram desenvolvidos utilizando o modelo ICTM. O primeiro trabalho que será utilizado como referência é o trabalho *HPC-ICTM: um Modelo de Alto Desempenho para Categorização de Áreas Geográficas* [26] de Rafael Krolow Santos Silva, realizado no Programa de Pósgraduação em Ciência da Computação da PUCRS, que apresentou uma solução utilizando o MPI. Este trabalho detalha um estudo sobre diversas maneiras de implementar uma solução paralela utilizando *clusters* e o padrão MPI. O trabalho ainda apresenta soluções que utilizam Grades Computacionais, mas este assunto está fora do escopo desta análise. O trabalho [26] ainda discute alguns modelos de programação para enfrentar o desafio de paralelizar o ICTM. E neste ponto, o autor explica detalhadamente como cada modelo é mapeado. Posteriormente, apresenta os resultados das suas implementações.

O segundo trabalho que será utilizado como referência é o *NUMA-ICTM: Uma versão paralela do ICTM explorando estratégias de alocação de memória para máquinas NUMA* [7] de Márcio Bastos Castro, realizado no GMAP (Grupo de Modelagem de Aplicações Paralelas) no Programa de Pósgraduação em Ciência da Computação da PUCRS. Este último implementou uma versão paralela

do ICTM utilizando OpenMP. E em seguida também implementou uma segunda versão utilizando a *interface* MAi (*Memory Affinity Interface*) [21] para viabilizar a utilização da Libnuma. A MAi é utilizada para facilitar a alocação da memória levando em conta a localidade dos dados, o que é crucial para se obter um bom desempenho em máquinas NUMA [25].

Portanto, a partir destes dois trabalhos foi levantada a idéia de apresentar um estudo de uma versão híbrida para o ICTM, versão nova que poderá ser implementa inclusive em *clusters* de máquinas NUMA. Posto este objetivo seguiu-se então estudos sobre programação híbrida. Certamente, os trabalhos citados aqui não são os únicos referentes a programação híbrida. São apenas aqueles que foram analisados durante o desenvolvimento desta pesquisa.

1.2.2 Programação Híbrida

Vários trabalhos anteriores já foram feitos considerando-se um modelo híbrido de programação paralela utilizando a combinação de MPI e OpenMP . No trabalho [6] foi comparado uma versão híbrida MPI/OpenMP do *NAS benchmarks* com a versão em MPI puro, e observaram que o desempenho depende de vários parâmetros, tais como padrões de acesso de memória e desempenho de hardware.

O trabalho [13] utilizou o caso específico de um código de modelagem de elemento discreto, considerando que os resultados gerais em código MPI puro superaram o código híbrido, e que o paralelismo de grão fino exigido pelo modelo híbrido resulta em pior desempenho do que em um código OpenMP puro também. Já o trabalho [27] constatou que, em certas situações, o modelo híbrido pode oferecer melhor desempenho que o MPI puro, mas que não é ideal para todas as aplicações. O trabalho [19] examinou a interação entre processos MPI e *threads* OpenMP, e ilustrou uma ferramenta que pode ser usada para examinar o funcionamento de uma aplicação híbrida. O trabalho [16] também analisou *NAS benchmarks* e descobriu que o modelo híbrido tem vantagens sobre redes com conexões mais lentas. Bons desempenhos de código híbrido foram alcançados em *clusters* de SMPs como por exemplo em [3].

O trabalho [9] investiga o desempenho de uma aplicação híbrida sobre dinâmica molecular, e compara o desempenho do código híbrido com o mesmo código paralelo usando MPI puro. Neste trabalho, foi analisado desempenho de ambas as versões em dois sistemas *multicore* de alto desempenho. As conclusões apontaram que em muitas partes da aplicação o código em MPI puro, nos *clusters multicore*, apresentaram melhor desempenho.

Os trabalhos [24] e [23] apresentam muitas informações relevantes sobre programação híbrida, como conceitos e alguns procedimentos mostrando a melhor forma de implementar soluções utilizando programação híbrida. Estes trabalhos foram utilizados para a compreensão deste modelo de desenvolvimento. Na SuperComputing 2010 que ocorreu no mês de novembro de 2010, os autores apresentaram um fórum acerca de programação híbrida, onde citaram inclusive programação híbrida em *clusters* de máquinas NUMA [22]. De um modo geral, os autores apresentam bons resultados em *clusters* de máquinas NUMA utilizando a ferramenta de linha de comando *numactl*, e não a biblioteca *libnuma*, como está sendo utilizada nesta pesquisa.

1.3 Estrutura do Volume

Esta Dissertação é composta de 8 capítulos, sendo o primeiro esta introdução. O restante está organizado da seguinte forma:

- **Capítulo 2**: Apresenta as principais características de uma arquitetura NUMA, além de alguns conceitos relevantes para o bom entendimento desta arquitetura e a correta utilização dos seus recursos;
- **Capítulo 3**: Relaciona as principais ferramentas necessárias para a utilização dos recursos das máquinas NUMA, e ainda apresenta uma *interface* para facilitar a utilização destes recursos;
- **Capítulo 4**: Faz uma introdução à programação híbrida, explica os principais modelos de programação híbrida, e também suas vantagens e desvantagens;
- **Capítulo 5**: Descreve o caso de estudo utilizado neste trabalho e discute a abordagem proposta para a solução híbrida;
- **Capítulo 6**: Apresenta os resultados obtidos com as duas implementações híbridas descritas no trabalho;
- **Capítulo 7**: Conclui a análise dos resultados obtidos, além de elencar algumas boas práticas.

2. ARQUITETURAS NUMA

A seguir serão apresentadas algumas das principais características de máquinas NUMA e ainda alguns conceitos relevantes para um bom entendimento destes detalhes. Os sistemas NUMA são compostos por nós, que possuem um processador, *multicore* ou não, e uma memória local, todos esses nós estão interconectados por uma rede especializada, o que oferece um **único espaço de endereçamento da memória para todo o sistema**, proporcionando inclusive a coerência da cache dos diversos processadores [18].

Em sistemas SMP o acesso à memória é uniforme, o que significa que o custo de acesso a qualquer local da memória é o mesmo. Já nos sistemas NUMA, o tempo de acesso à memória é não uniforme, visto que quando o processador acessa os dados que estão na memória local experimenta uma latência bastante inferior do que quando acessa os dados que estão na memória de outro nó [5].

Nos sistemas NUMA, esse custo para acesso à memória varia de acordo com a **Distância NUMA** ou **Fator NUMA**. Essa distância é basicamente a razão entre o tempo de acesso á memória de um nó local e o tempo de acesso à memória de um nó remoto. As distâncias NUMA são reconhecidas pelos sistemas operacionais em unidades definidas pela ACPI (*Advanced Configuration and Power Interface*) [1] SLIT (*System Locality Information Table*). De acordo com o padrão ACPI, à distância NUMA é armazenada na tabela SLIT e é um valor normalizado por 10. Também é definido pelo padrão a tabela SRAT (*System Resource Affinity Table*). Esta tabela é utilizada pelo sistema operacional de modo a auxiliar nas tarefas de alocação, fazendo com que os dados sejam alocados de preferência na memória do nó local do processo.

Na Figura 2.1 tem-se um modelo de um esquema de um sistema NUMA. O sistema mostrado é composto de quatro nós. Cada nó possui dois processadores, uma memória local e sua cache. Imaginando que existe um processo executando no processador 4, no nó 2, então a distância NUMA para o acesso à memória local é 10. No entanto, se esse processo precisar acessar dados na memória localizados no nó 1, então esse acesso terá uma latência maior, e esse valor pode ser expresso pela distância NUMA, que nesse exemplo pode ser colocada como 20, que é um valor hipotético. Então esses dois valores expressam a diferença entre os acessos à memória local e a memória remota.

É importante, salientar que esses valores dependem das tecnologias envolvidas na construção do sistema NUMA.

2.1 Conceitos

Alguns conceitos referentes à gerência e localidade de dados na memória e escalonamento de processos pelo *kernel* precisam ser apresentados de modo a possibilitar um melhor entendimento dos demais conceitos e fundamentos referentes aos Sistemas NUMA.

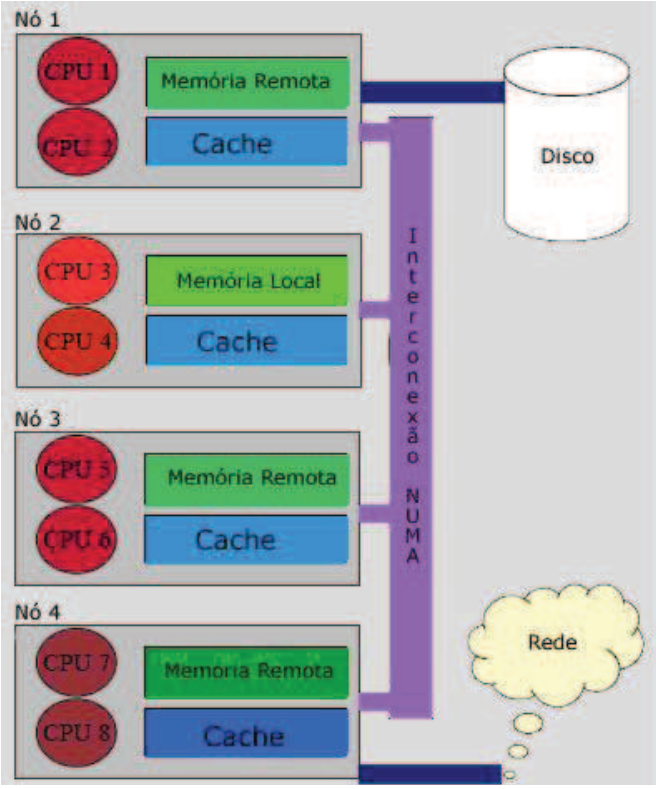

Figura 2.1: Esquema Máquina NUMA.

2.1.1 Gerência de Memória no Unix

Um processo Unix, em uma arquitetura de 32bits pode, teoricamente, ocupar até quatro gigabytes. Entretanto, o espaço em memória física (RAM) é um recurso escasso e o sistema operacional deve dividi-lo entre os vários processos em execução. Quando é necessário carregar um novo processo em memória, o sistema operacional traz do disco apenas as porções necessárias à sua execução e não o processo inteiro. Se ao executar, o processo acessa uma posição de memória que não está carregada em memória, o sistema operacional bloqueia esse processo, realiza a carga da porção que está faltando e, na seqüência, torna o processo novamente apto a executar. Esse esquema de gerenciamento de memória desvincula o tamanho lógico do processo do espaço disponível em RAM, já que o sistema operacional mantém em memória física apenas as porções necessárias à execução do programa. Isso é a base do que é denominado de memória virtual [11].

Existem duas técnicas fundamentais para definir as porções de memória: segmentação e paginação. Na segmentação, o espaço de endereçamento do processo é dividido em porções de tamanhos variáveis denominados de segmentos. Já na paginação, essas porções possuem tamanhos fixos e são denominados de páginas. A maioria dos sistemas operacionais emprega paginação por seu gerenciamento ser mais simples. A paginação é uma técnica de gerenciamento de memória que consiste em dividir tanto o espaço de endereçamento lógico do processo quanto o espaço físico em memória RAM em porções de tamanho fixo denominadas de página e quadros (*frames*), respectivamente. O sistema operacional mantém uma lista de quadros livres e os aloca a páginas à medida que os processos vão sendo carregados em memória (paginação por demanda). Qualquer página pode ser

alocada a qualquer quadro. A vinculação entre páginas e quadros é feita a partir de uma tabela de páginas.

2.1.2 Alocação de Memória

Sempre que um processo é criado, é definido um espaço de endereçamento lógico que pode ser visto como uma espécie de ambiente de execução contendo código, dados e pilha. O procedimento de criação de processos em Unix é baseado nas chamadas de sistema *fork* e *exec*. Um processo, denominado de processo pai, quando executa a chamada de sistema fork cria um novo processo, atribuindo um novo *pid* (*process identifier*) e um espaço de endereçamento lógico, uma cópia quase exata do seu próprio espaço de endereçamento [11]. Na prática, isso é feito com que o processo filho receba uma cópia da tabela de páginas do processo pai. Entretanto, na maioria das vezes o processo filho é criado para executar um código diferente do processo pai. Assim, na seqüência o processo filho executa a chamada de sistema *exec* para substituir as cópias das páginas do processo pai por páginas próprias provenientes de um arquivo executável. A partir desse ponto, os processos pai e filho, se diferenciam. Nesse ponto, ocorre uma alocação de memória para as páginas do processo filho.

Os processos, uma vez em execução, podem alocar memória adicional a partir da área de *heap* de seu espaço de endereçamento lógico. Isso é feito adicionando novas páginas ao seu espaço de endereçamento virtual, ou seja, aumentando o número de entradas em sua tabela de páginas, através da chamada de sistema malloc. Entretanto, normalmente, no Unix, a vinculação de uma nova página (endereço lógico) a um quadro (memória real) só acontece quando essa página for acessada pela primeira vez. Essa política é conhecida como ***first touch*** [29]. Posto de outra forma, ao se realizar uma primitiva *malloc* se obtém um espaço de endereçamento lógico proporcional à quantidade solicitada pelo *malloc*, porém a memória física correspondente só será realmente alocada quando for feito o primeiro acesso a ela.

2.1.3 Localidade de Dados

É o principal mecanismo utilizado em *caches* para agrupar posições contíguas em blocos, quando um processo referencia pela primeira vez um ou mais bytes em memória, o bloco completo é transferido da memória principal para a *cache*. Desta forma, se outro dado pertencente a este bloco for referenciado posteriormente, este já estará presente na memória *cache*, não sendo necessário buscá-lo novamente na memória principal. Portanto, sabendo-se o tamanho dos blocos utilizados pela *cache* e a forma com que os dados são armazenados pelo compilador é possível desenvolver uma aplicação paralela que possa melhor utilizar essa característica.

Blocos são utilizados em *cache* devido a características básicas em programas seqüenciais: localidade espacial e localidade temporal [28]. Na primeira, um determinado endereço foi referenciado, então há uma grande chance de um endereço próximo a esse também ser referenciado em um curto espaço de tempo. Já na localidade temporal, leva-se em conta que um programa é normalmente

composto por um conjunto de laços. Cada laço poderá acessar um grupo de dados de forma repetitiva. Por causa disto, se um endereço de memória foi referenciado, há também a chance deste ser referenciado novamente em pouco tempo.

2.1.4 O problema do Falso Compartilhamento

As características de localidade de dados descritas podem representar uma grande desvantagem quando utilizadas em sistemas multiprocessados. O problema é que mais de um processador pode necessitar de partes diferentes de um bloco. Se um determinado processador escreve somente em uma parte de um bloco em sua *cache* (somente alguns bytes), cópias deste bloco inteiro nas *caches* dos demais processadores devem ser atualizadas ou invalidadas, dependendo da política de coerência de *cache* utilizada. Este problema é conhecido como Falso Compartilhamento [28], podendo reduzir o desempenho de uma aplicação paralela. A Figura 2.2 busca mostrar um possível esquema, para exemplificar esse problema.

falsocompartilhamento.jpg

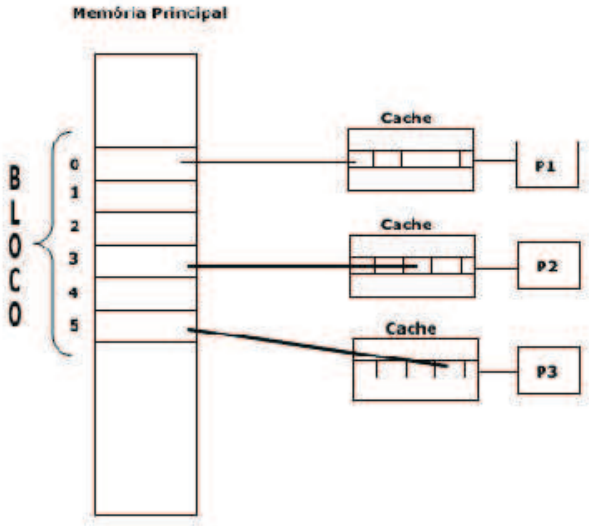

Figura 2.2: Falso Compartilhamento.

Em máquinas NUMA, estas questões referentes à localidade dos dados e falso compartilhamento se agravam, pois os dados podem pertencer à memória remota, ou seja, a memória de um nó vizinho, assim o tempo de acesso à memória vai ser onerado pelo Fator NUMA do sistema.

Existem mecanismos que permitem ao programador definir em que módulo de memória um determinado dado deverá ser armazenado. Em grande parte das arquiteturas NUMA, um dado é armazenado no nó em que acessou primeiro. Outra estratégia possível é permitir que o sistema operacional mantenha esse controle sobre a localidade dos dados.

Sabendo destas características é importante que o programador tenha conhecimento dos cuidados a serem tomados para o desenvolvimento de aplicações paralelas em máquinas NUMA de modo a conseguir um bom desempenho.

2.1.5 Escalonamento

O mecanismo responsável por determinar qual processo executará em um determinado processador e por quanto tempo deverá permanecer no processador é denominado escalonador. O objetivo deste mecanismo é permitir que a carga do sistema possa ser balanceada da melhor maneira entre todas as CPUs do sistema. Tratando-se de arquiteturas paralelas o escalonamento pode ser tratado em dois níveis: aplicação ou como entidade externa a aplicação [11].

No primeiro caso, a solução paralela para um problema é desenvolvida levando-se em consideração as características da arquitetura alvo, como por exemplo, número de processadores, rede de interconexão e memória disponível. O mecanismo de escalonamento estará presente dentro da aplicação, distribuindo as tarefas a serem executadas de forma a melhor utilizar o sistema.

O segundo caso, o esforço na programação é reduzido, porém também é comum o desempenho nesse caso ser pior. Isso ocorre porque o mecanismo de escalonamento tem conhecimento do sistema, mas não da aplicação.

Quando se trata de máquinas NUMA, há uma questão importante relacionada ao balanceamento de carga. Se um processo *p1*, em um determinado processador localizado em um nó *n1*, aloca dados em memória e os utiliza com freqüência, é bastante provável que esses dados estejam alocados na memória local do nó *n1*. Contudo, para manter o sistema balanceado o processo *p1* pode ser enviado para outro processador. Assim se esse outro processador estiver em outro nó que não o *n1*, todo o acesso que o processo *p1* necessitar fazer à memória que está alocada no nó *n1* terá uma maior latência, prejudicando certamente o desempenho do processo.

Nesses casos, é importante o programador avaliar o sistema NUMA utilizado para saber qual a melhor alternativa na seqüência de execução do programa. Pode-se seguir utilizando os dados armazenados na memória do nó, então, remoto. Ou pode-se, fazer a migração das páginas de memória do nó original para o nó em que o processo está sendo executado. Portanto, aplicações paralelas em máquinas NUMA somente terão resultados ótimos levando em consideração todos os detalhes referentes ao escalonamento e o balanceamento de carga do sistema.

2.2 Memória em Máquinas NUMA

Em sistemas NUMA, a memória é gerenciada separadamente para cada nó. Essencialmente, existe um *pool* de páginas em cada nó. Assim, cada nó tem uma *thread* de *swap* que é responsável pelo *memory reclaim* do nó [18]. Cada faixa de endereço de memória no nó é chamada de zona de alocação. A estrutura da zona de alocação contém listas que possuem as páginas disponíveis. Também existem listas informando quais as páginas estão ativas e inativas de modo a auxiliar no processo de limpeza ou recuperação da memória.

Se uma requisição de alocação é feita para um sistema NUMA, então é necessário decidir em qual *pool* e em qual nó será alocada a memória. Normalmente, é utilizado o padrão do processo que esta fazendo a requisição e é alocada a memória no nó local. No entanto, pode-se utilizar funções de alocação específicas, que serão vistas adiante, podendo alocar a memória em nós diferentes [5].

2.2.1 Alocação Eficiente

A Alocação de Memória em sistemas NUMA lida com novas considerações na comparação com um sistema com tempo de acesso à memória uniforme. A existência de nós eficientes significa que o sistema possui muitas partes com mais ou menos autonomia, assim os processos podem rodar melhor quando os dados são alocados levando em consideração essas características. Pode-se classificar como nós eficientes, aqueles que possuem o processo rodando e os dados alocados na memória do mesmo nó.

2.2.2 Localidade Ótima: Nó Local

A localidade ótima de alocação de memória para o processo se dá ao alocar a memória no nó local do processo. Nesse caso o processo terá o menor custo possível no sistema NUMA para acesso aos dados da memória e não terá tráfego na interconexão NUMA. No entanto, essa estratégia de alocação somente será a melhor se o processo continuar acessando os dados de maneira regular e local.

2.2.3 Aplicação Multi-Nó

Uma situação mais complexa ocorre quando a aplicação precisa de mais recursos que um nó pode oferecer. Quando não é possível alocar toda a memória necessária para a aplicação num mesmo nó, então é necessário fazer uma avaliação quanto ao desempenho da aplicação, levando-se em conta a forma como a aplicação utiliza esse recurso [18].

Aplicações multi-nó podem ter poucas *thread*s, porém essas *thread*s podem utilizar mais memória física que o nó tem disponível. Nesse caso, mais memória pode ser alocada automaticamente de outros nós do sistema NUMA. É verdade que o desempenho dessa aplicação sofrerá perdas em função de aumentar o tempo de acesso aos dados que estão em memória remota.

2.2.4 Migração de Memória

A necessidade de migrar páginas de memória entre nós aparece quando um processo que estava executando em um determinado nó é movido pelo escalonador do sistema operacional para outro nó. Nessa situação, para evitar o tempo de acesso à memória remota pode ser justificado migrar as páginas de memória alocadas para esse processo para a memória local do nó [18].

Outro momento em que a migração de páginas de memória se mostra vantajosa é para refazer o balanceamento de acesso à memória entre todos os nós que estão acessando dados na memória de maneira entrelaçada. Em uma aplicação rodando sobre vários nós, utilizando cada *thread* os dados em sua memória local. Após o encerramento de uma *thread*, os dados daquele nó devem ser acessados pelas outras *threads*. Nesse caso, é melhor migrar as páginas do nó que já encerrou o processamento para os demais nós, conforme a necessidade da aplicação.

Para ocorrer a migração de páginas, todas as referências a essas páginas devem ser desfeitas temporariamente e depois movidas para o novo endereço das páginas. Se não for possível remover todas as referências, a tentativa de migração é abortada e as antigas referências são re-estabelecidas. Esse tipo de problema pode ocorrer se existirem vários processos que acessam a mesma área de memória.

2.3 CPUsets

CPUsets provê um mecanismo para associar um conjunto de CPUs e memória de um nó a um conjunto de tarefas. Assim, é restringido a dada aplicação utilizar apenas os recursos disponibilizados pelo *CPUset* corrente. Ele forma uma hierarquia aninhada dentro de um sistema de arquivos virtual, e essas informações são necessárias para a gerência do sistema.

Cada tarefa tem um apontador para um *CPUset*, e múltiplas tarefas podem referenciar o mesmo *CPUset*. Requisições de tarefas utilizam a chamada de sistema *sched_setaffinity* para incluir CPUs na máscara de afinidade. Também são utilizadas as chamadas de sistemas *mbind* e *set_mempolicy* para configurar a afinidade de alocação de memória.

O escalonador não irá escalonar tarefas para CPUs que não estiverem no vetor de *cpus_allowed* e o alocador de páginas do *kernel* não irá alocar memória de outro nó que não esteja no vetor *mems_allowed*. Se o *CPUset* é CPU ou memória exclusivo, nenhum outro *CPUset* poderá referenciar aqueles recursos já uma vez referenciados. *CPUset* é valioso para o gerenciamento de um sistema de memória que é composto por partes em nós separados. No caso, os sistemas NUMA, se encaixam bem nessa definição. Isso, porque nesses sistemas é extremamente importante ter domínio de onde serão alocadas as páginas de memória e em quais CPUs serão executadas as tarefas.

2.4 Considerações Finais

Este capítulo apresentou as principais características das máquinas NUMA, informando detalhes do funcinamento destes equipamentos, ressaltando as vantagens desta arquitetura. Também foram apresentados alguns conceitos como: gerência de memória, alocação de memória, localidade de dados e escalonamento, que serão necessário para o entendimento de algumas escolhas feitas durante este trabalho. Ainda foram apresentadas o modo de classificar memória em máquinas NUMA e também os recursos de *CPUset*. Neste capítulo foram discutidos alguns conhecimentos necessários para se entender o funcionamento de uma arquitetura NUMA. No capítulo seguinte serão apresentadas as principais ferramentas necessárias para a utilização dos recursos em uma arquitetura NUMA.

3. Ferramentas para Utilização de Arquiteturas NUMA

As limitações de escalabilidade que os servidores SMPs possuem em função do problema de disputa entre os processadores por acesso ao barramento, quando do acesso à memória justificaram alternativas como as arquiteturas NUMA. No entanto, para a utilização correta das melhorias trazidas por esse novo sistema foi necessário a criação de recursos nos ambientes de modo a viabilizar esse processo. Os primeiros passos, foram a criação da Biblioteca *Libnuma* e do *Numactl* que são as partes mais relevantes da *NUMA API* [17], que possibilitam ao programador especificar os parâmetros necessários de maneira que a aplicação faça a alocação dos recursos de maneira correta.

Como a utilização dessas ferramentas vem crescendo, justamente pelo crescimento das plataformas NUMA no mercado, existem algumas propostas de novas bibliotecas ou *interfaces* para facilitarem a utilização da *NUMA API*, visto que a utilização das funções disponibilizadas por ela requer um grande esforço de programação. A MAi (*Memory Affinity Interface*), desenvolvida em um laboratório na França [21], busca oferecer uma *interface* mais simples para a implementação de aplicações que devem utilizar os recursos das arquiteturas NUMAs. A seguir, será feita uma apresentação da NUMA API, assim como da MAi.

3.1 NUMA API

Os sistemas Unix possuem maneiras de escalonar os processos para as unidades de processamento disponíveis na arquitetura. No caso do Linux, esse processo tradicionalmente é feito utilizando as chamadas de sistema *sched_set_affinity* e *schedutils* [17]. NUMA API estende essas chamadas de sistema permitindo que os programas possam especificar em qual nó a memória será alocada e ou em qual CPU será colocada para executar determinada *thread*. NUMA API é normalmente distribuída em pacotes e segundo [17] já está disponível no SUSE desde a versão Enterprise 9. NUMA API consiste de diferentes componentes: existe uma parte que trabalha junto ao *kernel* gerenciando as políticas de memória por processos ou especificando o mapeamento da memória. Essa parte é controlada por três novas chamadas de sistema. Tem-se a *libnuma*, que é uma biblioteca compartilhada no espaço de usuário do sistema, que pode ser ligada a aplicação, e é a *interface* recomendada para utilização das políticas NUMA. Tem-se também a *numactl*, um utilitário de linha de comando, que permite controlar as políticas NUMA para uma determinada aplicação e também os processos que essa aplicação gerar. Como utilitários auxiliares têm-se o *numastat*, que coleta as estatísticas sobre alocação de memória, e o *numademo*, que mostra os efeitos das diferentes políticas no sistema.

A principal tarefa da *NUMA API* é a gerência das políticas NUMA. As políticas podem ser aplicadas aos processos ou a áreas de memória. São suportadas quatro diferentes políticas:

- ***default***: aloca no nó local;

- ***bind***: aloca num conjunto especificado de nós;
- ***interleve***: entrelaça a alocação de memória num conjunto de nós;
- ***preferred***: tenta alocar primeiro num determinado nó.

A diferença entre as políticas *bind* e *preferred* é que a primeira vai falhar diretamente, quando o nó não conseguir alocar a memória, enquanto que a segunda, no caso de falha, vai tentar alocar em outro nó. As políticas podem ser por processos ou por regiões de memória, levando sempre em consideração que os processos filhos sempre herdam as políticas dos processos pais. E também que a política sempre é aplicada a toda a memória alocada no contexto do processo.

3.1.1 NUMACTL

NUMACTL é uma ferramenta de linha de comando para rodar aplicações com determinadas políticas NUMA especificadas. Essa ferramenta possibilita que seja definido um conjunto de políticas sem que a aplicação precise ser modificada e compilada novamente.

A seguir são apresentados alguns exemplos retirados do [17] que mostram alguns comandos:

numactl cpubind=0 membind=0,1 ./programa

Nesse caso, é executado o *programa* na CPU do nó 0, e apenas é alocada memória dos nós 0 e 1.

numactl preferred=1 numactl show

Aqui é especificado a política preferred para o nó 1 depois mostrado o resultado.

numactl interleave=all numbercruncher

Executa um programa (*numbercruncher*) que tem uso intensivo de memória, de modo a alocar a memória de maneira entrelaçada entre todos nos nós disponíveis.

Algumas opções do comando numactl são importantes e merecem destaque. Muitas dessas opções requerem uma máscara para especificar o nó (nodemask), isto é, uma lista separada por vírgulas de números dos nós. O comando *numactl hardware*, mostra a lista de nós disponíveis no sistema e *numactl show*, mostra estado dos processos correntes e seus filhos.

As políticas mais utilizadas são as destinadas aos processos:

- ***membind=nodemask***: aloca memória apenas nos nós especificados;
- ***interleave=nodemask***: entrelaça toda alocação de memória entre os nós especificados;
- ***cpubind=nodemask***: executa os processos apenas nas CPUs especificadas na mascara de nós;
- ***preferred=node***: aloca memória preferencialmente no nó especificado.

A seguir, será apresentada uma maneira de gerenciar a alocação de memória compartilhada com o *numactl*, visto que até agora foi visto apenas a opções que tratam da alocação de processos.

Algumas aplicações podem ter seu desempenho melhorado quando utilizam a melhor política de alocação de memória, levando em consideração suas características. Por exemplo, imagine uma aplicação multiprocessada, que esteja alocando um único segmento de memória que é compartilhada por todos os processos. Se essa alocação for feita de modo entrelaçada entre as memórias de todos os nós, é bastante razoável imaginar que o desempenho irá melhorar. Assim, segue um exemplo das opções que podem ser passadas para o *numactl* de modo a escolher uma política de alocação de memória.

numactl length=1G fila=/dev/shm/interleaved interleave=all

O comando acima configura um tmpfs, filesystem temporário, de um gigabyte para entrelaçar a memória sobre todos os nós do sistema. Muitas outras opções podem ser encontradas na *Man Pages do Linux* sobre *numactl*.

3.1.2 LIBNUMA

O *numactl* pode fazer o controle de políticas NUMA, porém sempre que aplicada à política ela é válida para todo o processo. Já a *libnuma*, que é uma biblioteca compartilhada, pode aplicar políticas para área de memória especifica ou até mesmo para uma *thread*.

A libnuma pode ser ligada aos programas e então oferecer ao programador uma API para utilização das políticas NUMA, a biblioteca também é considerada a melhor maneira de se utilizar os recursos da NUMA API. Um exemplo de compilação de um programa com a libnuma é:

cc ... lnuma

As funções e macros da NUMA API são declaradas no arquivo de *include numa.h*. Antes de qualquer função da libnuma ser utilizada, o programa deve chamar a função *numa_available()*, quando essa função retorna um valor negativo, significa que o suporte a NUMA API não está disponível. A seguir, deve-se chamar a função *numa_max_node()*, essa função descobre e informa o número de nós disponíveis no sistema.

Algumas outras funções serão apresentadas sucintamente a seguir, com exemplos de utilização.

A *libnuma* armazena numa estrutura de dados abstrata chamada *nodemask_t*, que representa o conjunto de nós que pode ser gerenciado pela API. Cada nó do sistema tem um único número, o maior número corresponde ao valor retornado pela função *numa_max_node()*. A máscara de nós é inicializada com a função *nodemask_zero()*.

Um único nó pode ser configurado na máscara com as funções: *nodemask_set()* e *nodemask_clr()*. A função *nodemask_equal*, compara duas máscaras. E a função *nodemask_isset* testa se um bit esta configurado na máscara.

Existem duas máscaras de nós pré-definidas: *numa_all_nodes* e *numa_no_nodes*, que respectivamente representam uma máscara com todos os nós e outra sem nós.

A libnuma possui funções para a alocação de memória com políticas especificadas. Essas funções de alocação são relativamente lentas e não devem ser utilizadas para a alocação de porções pequenas da memória [17]. Quando uma alocação for solicitada e não for possível, então será retornado o valor *NULL*. Toda a memória alocada com as funções da família *numa_alloc()* deve ser liberada

com a função *numa_free()*. A função *numa_alloc_onnode* aloca memória no nó especificado. Por padrão, a função ira tentar alocar a memória no nó, caso tenha problema poderá tentar alocar a memória em outro nó, a menos que a função *numa_set_strict()*, tenha sido chamada. Para se ter a informação da quantidade de memória que o nó tem disponível pode ser chamada a função *numa_node_size()*. A função *numa_alloc_interleaved()* aloca a memória de maneira entrelaçada entre todos os nós do sistema.

É importante ressaltar que nem sempre a alocação de memória entre todos os nós garante um bom desempenho a aplicação, em muitas arquiteturas NUMA o ideal é a alocação entre apenas alguns nós vizinhos, nesse caso pode-se utilizar a função *numa_alloc_interleaved_subset()*. Ainda tem-se a função *numa_alloc_local()*, a qual aloca memória no nó local e a função *numa_alloc()* que aloca a memória conforme a política do sistema como um todo.

A biblioteca libnuma também pode gerenciar as políticas de processos. A função *numa_set_interl eaved_mask()* habilita entrelaçamento para a *thread* corrente, assim todas as outras alocações serão feitas utilizando-se do round-robin entrelaçado entre os nós contidos na máscara.

A função *numa_set_prefered()* configura o nó como preferencial para a *thread* corrente. Assim, é tentado alocar a memória no nó local, caso tenha-se um problema é então alocado a memória em outro nó. A função *numa_set_membind()* configura a alocação de memória de maneira estrita aos nós definidos na máscara. A função *numa_set_localalloc()* configura a política de alocação para o padrão local.

As políticas apresentadas até aqui, especificam os nós onde serão alocadas as memórias. Outra característica da libnuma é alocar as *threads* nos nós conforme especificado. Isso é feito com a função *numa_run_on_node()*, que especifica com a *numa_run_on_node_mask* em quais nós as *threads* serão executadas. A *numa_bind* é outra função de configuração para a alocação de processos.

A função *numa_get_run_node_mask()* retorna a máscara com os nós para os quais os processos correntes estão sendo alocados. Ainda é possível obter informações sobre o ambiente no qual os processos estão sendo executados.A função *numa_node_to_cpus()* retorna o número de CPUs de todas as CPUs no nó. O tratamento de erros da *libnuma* é relativamente simples. Isso porque os erros de configuração das políticas podem ser ignorados, o único problema é que as aplicações irão ser executadas com menor desempenho. Quando um erro ocorre é chamada a função *numa_error()*, e como padrão ela imprime um mensagem de erro para o *stderr*, quando a variável global *numa_exit_on_error* esta configurada e ocorre um erro da libnuma no programa, então o programa é finalizado.

A *libnuma* possui um número bastante extenso de funcionalidades, muitas outras funções podem ser necessárias durante a utilização da biblioteca. Portanto é sempre importante ter acesso a documentação da *NUMA API*.

3.2 MAi - Memory Affinity Interface

Esta *interface* de afinidade de memória foi desenvolvida pela estudante de doutorado Christiane Pousa Ribeiro sob a orientação do Prof. Dr. Jean-François Méhaut no LIG (Laboratoire d'Informatique de Grenoble, ligado ao INRIA (*Institut Nacional de Recherche en Informatique et en Automatique*) em Grenoble. O objetivo é permitir maior facilidade ao desenvolvedor na gerência da Afinidade de Memória em Arquiteturas NUMA. A unidade de afinidade na MAi é um vetor (*array*) de aplicações paralelas. O conjunto de políticas de memória da MAi podem ser aplicadas a esses vetores de maneira simples. Assim, as funções em alto nível implementadas pela MAi reduzem o trabalho do desenvolvedor, se comparados com as funções disponibilizadas pela NUMA API conforme visto na seção 3.1.2.

A biblioteca possui sete políticas de memória: *cyclic*, *cyclic_block*, *prime_mapp*, *skew_mapp*, *bind_all*, *bind_block* e *random*. Na política *cyclic* as páginas de memória que contém os vetores são colocados na memória física formando um circulo com os blocos de memória. A diferença entre *cyclic* e *cyclic_block* é a quantidade de páginas de memórias que são utilizadas. Na *prime_mapp* as páginas de memórias são alocadas em nós de memória virtual. O número de nós de memória virtual é escolhido pelo usuário e tem que ser primo. Essa política tenta espalhar as páginas de memória pelo sistema NUMA. Na *skew_mapp*, as páginas de memórias são alocadas utilizando o modo *round-robin*, por todo o sistema NUMA. A cada volta do round-robin é feito um deslocamento com o número de nós. Ambos as políticas *prime_mapp* e *skew_mapp*, podem ser utilizadas com blocos, assim como cyclic_block, sendo um bloco um conjunto de linhas e colunas de um *array*. Em *bind_all* e *bind_block* a política de páginas de memória aloca os dados em nós especificados pelo desenvolvedor. A diferença entre os dois, é que o *bind_block* aloca os blocos de memória com os processo/*threads* que estão fazendo uso deles. A última política é a *random*. Nessas políticas as páginas de memórias são alocadas em uma distribuição aleatória e uniforme. O principal objetivo dessas políticas é diminuir a contenção de acesso à memória das arquiteturas NUMA.

3.2.1 Detalhes da Implementação

A MAi é implementada em C para Linux baseados em sistemas NUMA. Programadores que quiserem utilizar a MAi devem utilizar o modelo de programação de Memória Compartilhada. As bibliotecas de Memória Compartilhada que podem ser utilizadas são Pthreads e OpenMP para C/C++. Com pré-requisito para a MAi tem-se a NUMA API.

A Figura 3.1 mostra a arquitetura da *interface* MAi. Porém, para garantir afinidade de memória, todo o gerenciamento de memória deve ser feito utilizando-se as estruturas da MAi. Essas estruturas são carregadas na inicialização da *interface*, função *mai_init()*, e são usadas durante a execução da aplicação.

O arquivo de configuração servirá para informar a MAi sobre quais os blocos de memória utilizar para alocar dados de acordo com a política especificada e também quais processadores ou núcleos utilizar para executar *threads*.

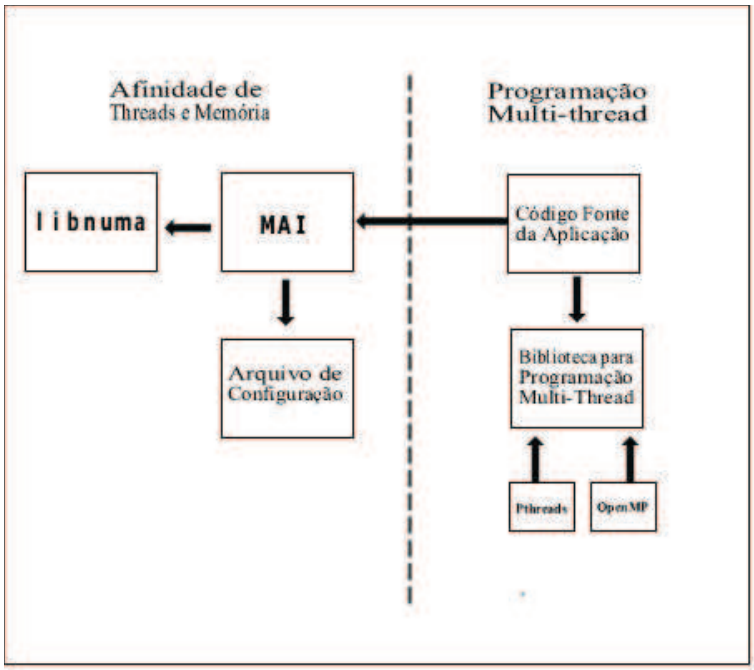

Figura 3.1: Diagrama de Implementação da MAi [21]

3.2.2 Funções da Interface

Para desenvolver aplicações usando a MAi os programadores precisam apenas de algumas funções de alto nível. Essas funções são divididas em cinco grupos: sistema, alocação, memória, *threads* e estatísticas:

- ***Funções de Sistema***: as funções de sistema são divididas em funções para configurar a *interface* e funções para obter algumas informações sobre a plataforma. A funções de configuração podem ser chamadas no inicio (*mai_init()*) e no final (*mai_final()*) do programa. A função *mai_init()* é responsável por inicializar os Ids dos nós e *threads* que irão ser usados para políticas de memória e *threads*. Essa função recebe como parâmetro o arquivo de configuração da MAi, que contém as informações sobre os blocos de memória e processadores/*cores* da arquitetura. Se não for passado o arquivo como parâmetro, a MAi escolhe quais blocos de memória e processadores/*cores* usará. A função *mai_final()* é usada para liberar quais dados alocados pelo programador. Listagem das principais funções de sistema:

  ```
  void mai_init (char filename [] ) ;
  void mai_final ( ) ;
  int mai_get_num_nodes ( ) ;
  int mai_get_num_threads ( ) ;
  int mai_get_num_cpu ( ) ;
  unsigned long * mai_get_nodes_id ( ) ;
  unsigned long * mai_get_cpus_id ( ) ;
  ```

```
void mai_show_nodes ( ) ;
void mai_show_cpus ( ) ;
```

- ***Funções de Alocação***: essas funções permitem ao programador alocar vetores (*arrays*) de tipos primitivos da linguagem C (*char,int,Double e float*) e tipos não primitivos como estruturas. A função precisa do número de itens e do tipo em C. O retorno é um ponteiro para os dados alocados. A memória é sempre alocada seqüencialmente Essa estratégia permite um melhor desempenho e faz com que seja mais fácil definir uma política de memória para o vetor. Listagem das principais funções de alocação:

```
void * mai_alloc_1D( int nx , size_t size_item , int type ) ;
void * mai_alloc_2D( int nx , int ny , size_t size_item , int type ) ;
void * mai_alloc_3D( int nx , int ny , int nz , size_t size_item , int type ) ;
void * mai_alloc_4D( int nx , int ny , int nz , int nk , size_t size_item , int type ) ;
void mai_free_array ( void *p ) ;
```

- ***Funções de Políticas de Memória***: as funções desse tipo permitem ao programador selecionar a política de memória para a aplicação e migrar dados entre um nó e outro. Listagem das principais funções de política de memória:

```
void cyclic (void *p) ;
void cyclic_block(void *p , intblocksize);
void bind_all(void *p);
void bind_block(void *p);
void migrate_pages(void *p,int np ,unsigned long node);
```

- ***Funções de Políticas de Threads***: essas funções permitem ao desenvolvedor atribuir ou migrar um processo/*thread* para um *CPU/core* no sistema NUMA. As funções *bind_thread()* usam o id da *thread* e a máscara da *CPU/core* para fazer a atribuição. Listagem das principais funções de políticas de *threads*:

```
void bind_threads();
void set_thread_id_omp();
void set_threads_id_posix(unsigned int *id);
```

- ***Funções Estatísticas***: as funções estatísticas permitem ao programador coletar algumas informações sobre a execução da aplicação no sistema NUMA. Existem informações sobre memória e *threads*, tanto sobre alocação quanto sobre migração e também informações de sobrecarga de migrações. Listagem as principais funções estatísticas:

```
void mai_print_pagenodes ( unsigned long * pageaddrs , int size);
```

```
int mai_number_page_migration ( unsigned long * pageaddrs , int size) ;
double mai_get_time_pmigration ( ) ;
void mai_print_threadcpus ( ) ;
int mai_number_thread_migration ( unsigned int * threads , int size) ;
double mai_get_time_tmigration ( ) .
```

Com a utilização da MAi é possível obter o melhor desempenho das máquinas NUMA, com um menor esforço no desenvolvimento da aplicação. Pois suas funções exigem uma menor e mais simples codificação em comparação as funções da *libnuma*.

3.3 Considerações Finais

Neste capítulo foi apresentado as principais ferramentas para a utilização dos recursos em uma máquina NUMA. A *NUMA API* concentra o *numactl* e a *libnuma*. O *numactl* é uma ferramenta de linha de comando que possibilita que programas já compilados se beneficiem dos recursos NUMA. Já a libnuma é uma biblioteca que deve ser "ligada"durante o processo de compilação de um programa, dessa forma o programa pode ser construído para utilizar os recursos NUMA conforme a sua necessidade. Foi desenvolvido uma *interface* chamada de MAi, que possibilita que os desenvolvedores utilizem os recursos NUMA em seus programas de maneira menos trabalhosa. No próximo capítulo serão apresentados informações sobre programação híbrida.

4. Programação Híbrida: MPI e OpenMP

Conforme mencionado, *clusters* de nós multiprocessados de memória compartilhada estão se tornando cada vez mais populares na computação de alto desempenho. Assim, o foco de programação está se deslocando da computação de nós que possuem arquiteturas de memória distribuída, para arquiteturas com compartilhamento de memória. Estas arquiteturas incluem uma grande variedade de sistemas, desde arquiteturas em que a interligação do nó fornece um único espaço de endereçamento de memória, até sistemas com capacidade de acesso à memória remota direta (*RDMA - Remote Direct Memory Access*).

O modelo de programação híbrido MPI e OpenMP pode ser explorado com sucesso em cada um destes sistemas. De fato, os princípios de códigos paralelos híbridos são adequados para sistemas que vão desde um nó SMP até grandes *clusters* com nós SMPs, ou mesmo para máquinas NUMA, e ainda para *clusters* de máquinas NUMA. No entanto, a otimização do código híbrido é bastante diferente em cada classe de sistema. No restante deste capítulo, serão consideradas as características específicas do MPI e do OpenMP, descrevendo os modelos que podem ser adotados para a sua utilização em conjunto.

4.1 MPI

MPI (*Message-Passing Interface*) [12] é uma especificação de uma biblioteca de *interface* de troca de mensagens. MPI representa um paradigma de programação paralela no qual os dados são movidos de um espaço de endereçamento de um processo para o espaço de endereçamento de outro processo, através de operações de troca de mensagens.

As principais vantagens do estabelecimento de um padrão de troca de mensagens são a portabilidade e a facilidade de utilização. Em um ambiente de comunicação de memória distribuída ter a possibilidade de utilizar rotinas em mais alto nível ou abstrair a necessidade de conhecimento e controle das rotinas de passagem de mensagens, *e.g. sockets*, é um benefício bastante razoável. E como a comunicação por troca de mensagens é padronizada por um fórum de especialistas, é correto assumir que será provido pelo MPI eficiência, escalabilidade e portabilidade.

MPI possibilita a implementação de programação paralela em memória distribuída em qualquer ambiente. Além de ter como alvo de sua utilização as plataformas de *hardware* de memórias distribuídas, MPI também pode ser utilizado em plataformas de memória compartilha como as arquiteturas SMPs e NUMA. E ainda, torna-se mais aplicável em plataformas híbridas, como *clusters* de máquinas NUMA.

4.2 OpenMP

OpenMP [20] é uma API (*Application Program Interface*) para programação em C/C++, Fortran entre outras linguagens, que oferece suporte para programação paralela em computadores que possuem uma arquitetura de memória compartilhada. O modelo de programação adotado pelo OpenMP é portável e escalável, podendo ser utilizado numa gama de plataformas que variam desde um computador pessoal até supercomputadores. OpenMP é baseado em diretivas de compilação, rotinas de bibliotecas e variáveis de ambiente.

O OpenMP provê um padrão suportado por quase todas as plataformas ou arquiteturas de memória compartilhada. Um detalhe interessante e importante é que esta compatibilidade é conseguida utilizando-se um conjunto simples de diretivas de programação. Em alguns casos, o paralelismo é implementado usando 3 ou 4 diretivas. É, portanto, correto afirmar que o OpenMP possibilita a paralelização de um programa sequencial de forma amigável.

Como o OpenMP é baseado no paradigma de programação de memória compartilhada, o paralelismo consiste então de múltiplas *threads*. Assim, pode-se dizer que OpenMP é um modelo de programação paralelo explícito, oferecendo controle total ao programador.

4.3 Programação Híbrida MPI + OpenMP

A principal motivação para utilizar programação híbrida MPI e OpenMP é tirar partido das melhores características de ambos os modelos de programação, mesclando a paralelização explícita de grandes tarefas com o MPI com a paralelização de tarefas simples com o OpenMP. Em alto nível, o programa está hierarquicamente estruturado como uma série de tarefas MPI, cujo código sequencial está enriquecido com diretivas OpenMP para adicionar *multithreading* e aproveitar as características da presença de memória compartilhada e multiprocessadores dos nós. As Figuras 4.1,4.2 e 4.3 mostram respectivamente: (i) uma maneira de explorar o hardware disponível de um *cluster* SMP de 2-CPUs para uma decomposição somente com MPI, (ii) para uma decomposição somente com OpenMP, (iii) para uma decomposição hierárquica híbrida com MPI e OpenMP.

De fato, pode-se observar que na Figura 4.3 será obtido um maior número de processos sendo executado paralelamente. Baseado nesta afirmação é possível inferir que uma aplicação híbrida terá sempre melhor desempenho que aplicações que utilizam apenas MPI ou aplicações que utilizam apenas OpenMP. No entanto, não é verdadeira essa assertiva. Sempre será necessário analisar a aplicação de modo a associar o melhor modelo de programação para a situação, buscando então os melhores resultados possíveis.

4.4 Modelos de Programação Paralela em Plataformas Híbridas

Nessa seção serão apresentados alguns modelos de programação possíveis para plataformas híbridas, tentando deixar mais evidente a importância do casamento correto entre o modelo de programação escolhido para cada plataforma.

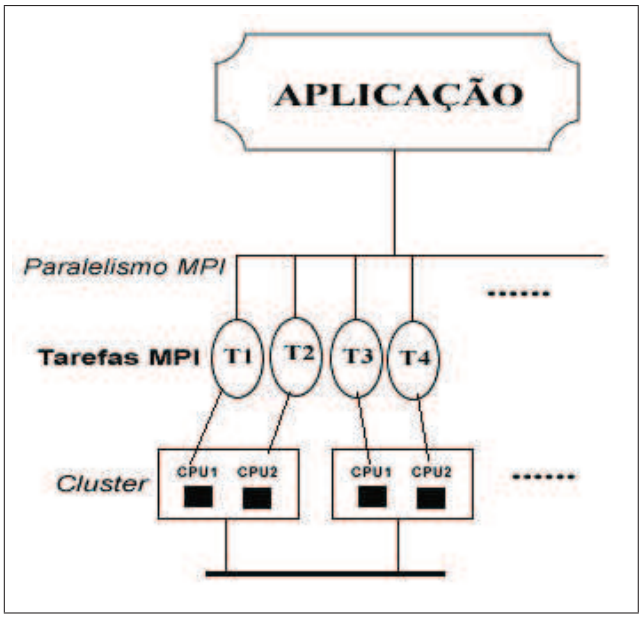

Figura 4.1: Decomposição apenas MPI.

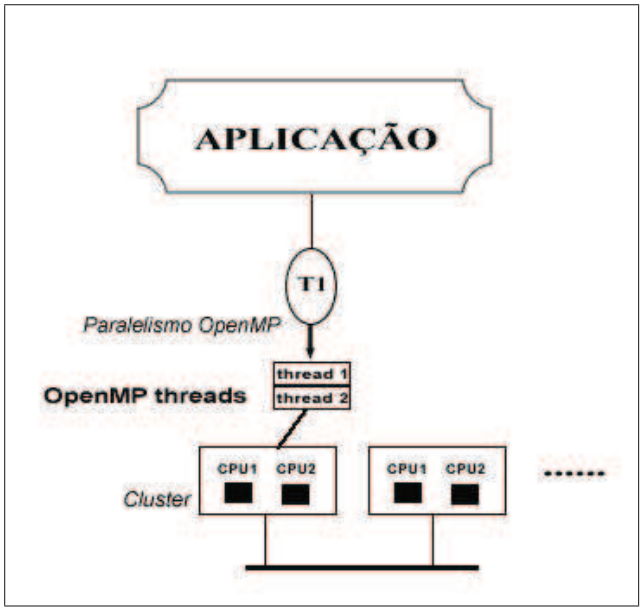

Figura 4.2: Decomposição apenas OpenMP.

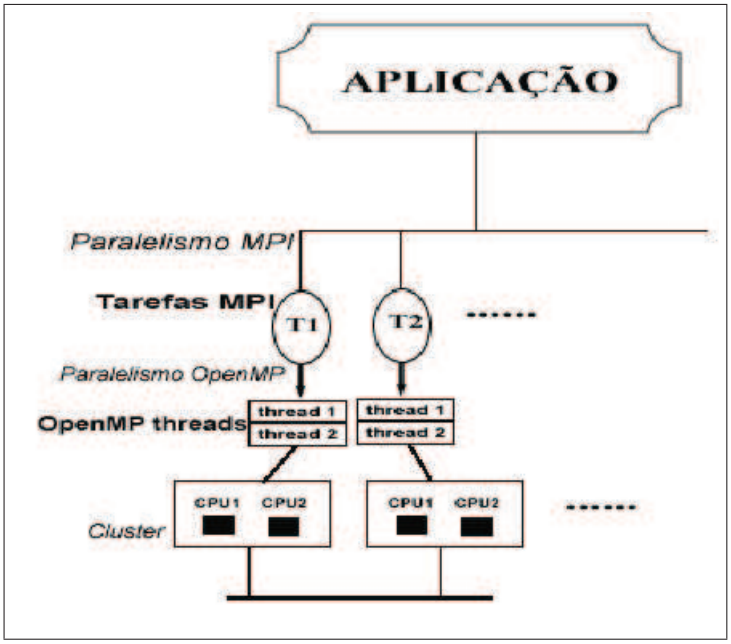

Figura 4.3: Decomposição Híbrida com MPI e OpenMP.

A Figura 4.4 [24] mostra uma taxonomia de modelos de programação paralela para plataformas híbridas. O autor adicionou a nomenclatura OpenMP Puro porque tecnologias como *Intel Cluster OpenMP* [15] permitem o uso de OpenMP para a paralelização em *clusters* com memória distribuída, porém a análise desta ferramenta está fora do escopo deste trabalho. Esta visão ignora os detalhes sobre como exatamente as *threads* e os processos de um programa híbrido serão mapeadas de forma hierárquica no *hardware*. Os problemas de incompatibilidade que podem ocorrer serão discutidos adiante na seção 4.5.1.

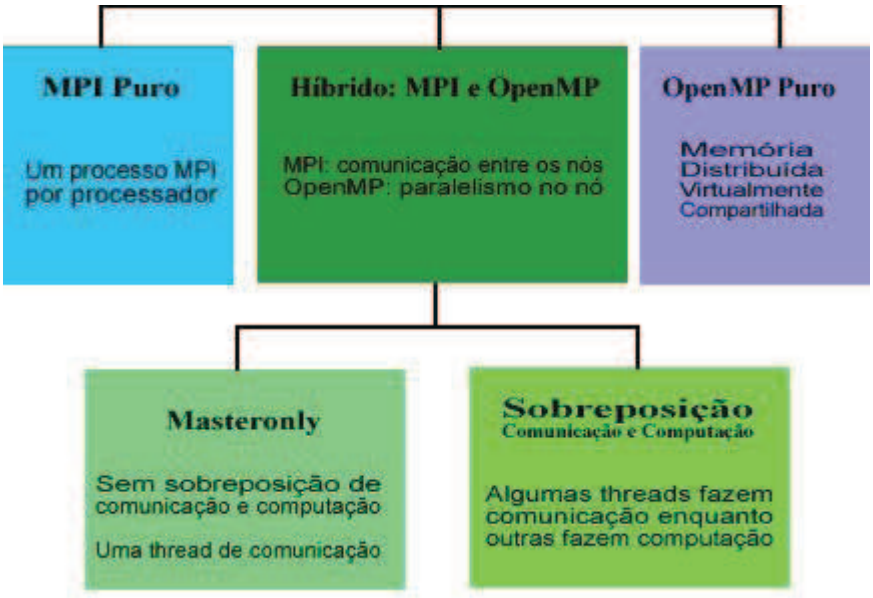

Figura 4.4: Modelos de Programação para Plataformas Híbridas. [24]

4.4.1 MPI Puro

Do ponto de vista do programador, o MPI puro ignora o fato do nó possuir vários núcleos compartilhando a memória. Esse modelo pode ser empregado de imediato sobre todos os tipos de *clusters* sem alterações no código. Além disso, não é necessária para a biblioteca MPI que as demais camadas de *software* suportem aplicações *multithread*, o que simplifica a implementação. Por outro lado, um modelo de programação MPI puro implicitamente assume que a troca de mensagens é o paradigma a ser usado para todos os níveis de paralelismo disponível na aplicação. Além disso, toda comunicação entre os processos no mesmo nó vai ocorrer através das camadas de *software* do MPI, o que aumenta a sobrecarga da rede. Mesmo sendo esperado que a biblioteca seja capaz de usar atalhos através da memória compartilhada, para a comunicação dentro do nó. Essas otimizações geralmente estão fora da influência do programador.

4.4.2 Híbrido *Masteronly*

O modelo híbrido *masteronly* utiliza um processo MPI por nó e *multithread* OpenMP sobre os *cores* do nó, sem chamadas MPI dentro de regiões paralelas. Assim, apenas a *thread* principal faz chamadas MPI, sendo responsável então, pela comunicação. As demais *threads* apenas realizam computação. Uma amostra simples da organização do programa seria o pseudocódigo do modelo híbrido *masteronly* 4.1.

Como não há transmissão de mensagens dentro do nó, as otimizações do MPI não são necessárias. Naturalmente, as partes do OpenMP devem ser otimizadas para a arquitetura do nó pelo programador, por exemplo, empregando cuidados com a localidade dos dados na memória em nós NUMA, ou usando mecanismos de afinidade de *threads* [8].

Existem, no entanto, alguns problemas ligados com o modo *masteronly*:

Algoritmo 4.1: Pseudocódigo: Modelo híbrido *masteronly*

```
for (i=1 ... N) do
  #pragma omp paralelo
  (
  / * Código numérico * /
  )
  / * Na thread mestre * /
  MPI_Send (envia dados em massa para as áreas de dados em outros nós)
  MPI_R ecv (dados dos vizinhos)
end for
```

- Todas as outras *threads* estarão ociosas durante a fase de comunicação da *thread* principal, o que poderá levar a um forte impacto de sobrecarga de comunicação;

- A capacidade de comunicação entre os nós pode ficar subutilizada, em função de apenas uma *thread* fazer chamadas MPI.

4.4.3 Híbrido com Sobreposição de Comunicação e Processamento

Uma maneira de evitar subutilização das *threads* de computação durante a comunicação MPI é separar uma ou mais *threads* para fazer a comunicação em paralelo. Um exemplo de estrutura do programa seria o pseudocódigo do modelo híbrido com sobreposição 4.2.

Algoritmo 4.2: Pseudocódigo: Sobreposição de comunicação e computação

```
if (my_thread_ID <...)  then
  / * threads de comunicação * /
  / * dados para comunicação* /
  MPI_Send (envia dados)
  MPI_Recv (recebe dados)
else
  / * threads de computação * //
  / * Executar o código que não depende de dados que estão sendo trocados* /
end if
/ * todas as threads: * /
/ * Executar o código que precisa dos dados * /
```

Uma possível razão para usar mais de uma *thread* de comunicação pode surgir se uma única *thread* não conseguir saturar a rede de comunicação entre os nós. Há, contudo, uma relação custo/benefício a ser gerenciada, porque quanto mais *threads* utilizadas para troca de mensagens, serão menos *threads* disponíveis para processamento do problema. Também é necessário considerar a complexidade da programação para utilizar esse modelo.

4.4.4 OpenMP Puro em *Clusters*

Algumas pesquisas têm investido na implementação de *software* para compartilhamento virtual de memória distribuída o que permite programação similar à utilizada para memória compartilhada em sistemas de memória distribuída. Desde 2006 a Intel oferece o *Cluster OpenMP* [15] que é um sistema que permite o uso de programas feitos com OpenMP em um *cluster*. O mecanismo de software utilizado pelo *Cluster OpenMP* é conhecido como memória compartilhada distribuída, DSM (*Distributed Shared Memory*) ou DVSM (*Distributed Virtual Shared Memory*).

Com este sistema, o OpenMP torna-se um modelo de programação possível para *clusters*. É, em certa medida, um modelo híbrido, sendo idêntico ao OpenMP dentro de um nó de memória compartilhada, mas empregando um protocolo sofisticado que mantém automaticamente páginas de memória c̈ompartilhadasc̈oerentes entre os nós. Com *Cluster OpenMP*, ocorrem frequentemente sincronização das páginas por todos os nós. Isso pode potencialmente tornar-se mais custoso do que utilizar MPI [24].

4.5 Vantagens e Desvantagens da Programação Híbrida

Pelo menos na teoria, as vantagens da programação híbrida MPI e OpenMP são interessantes. A técnica permite obter uma divisão de tarefas com MPI, fazendo o processamento nos nós usando OpenMP em máquinas de memória compartilhada. No entanto, nem sempre uma versão híbrida apresenta desempenho superior a uma versão feita com os recursos do MPI puro [4]. Contudo, pode-se adaptar programas que já estão desenvolvidos em MPI para verificar as melhorias obtidas com a utilização do OpenMP nos nós multiprocessados.

Provavelmente, a melhor maneira de entender se um projeto de código híbrido pode ser bem sucedido é fazer uma análise cuidadosa das características do problema. Após isso, é preciso mapeá-lo conforme a arquitetura disponível para resolução do problema. Situações com bom potencial para o projeto de código híbrido incluem o seguinte:

- Programas que tenham uma má escala com o aumento de tarefas (processos) MPI;
- Programas MPI com balanceamento de carga mal ajustados (escolher os pontos certos onde as *threads* OpenMP podem ser implementadas);
- Problemas de paralelismo com grão-fino, mais adequado para uma utilização de OpenMP do que MPI;
- Programas com dados replicados, quando se utiliza OpenMP, é possível atribuir um único exemplar dos dados em cada nó, para ser acessado por todas as *threads* do nó local;
- Programas MPI, com restrições sobre o número de tarefas;
- Programas que serão executados em máquinas com comunicação entre os nós de baixo desempenho (rede de baixa capacidade).

Nos demais casos, os códigos híbridos são passíveis de ter igual ou até pior desempenho do que uma solução MPI puro. Isto deve-se à algumas ineficiências ligada à abordagem híbrida. De fato, um programa decomposto hierarquicamente tem as chamadas MPI realizadas fora de regiões paralelas. Isto significa que sempre que uma chamada MPI é necessária, terá apenas uma *thread* ativa por nó, e isso é claramente uma fonte de sobrecarga. Adicionais sobrecargas são devidas a causas mais sutis. Entre estas, deve-se mencionar possível aumento dos custos de comunicação, devido à impossibilidade de uma *thread* utilizar toda a largura de banda da interconexão do nó. Na prática, a abordagem híbrida é conveniente sempre que as vantagens se sobressaem. Mais detalhes podem ser encontrados em [4].

4.5.1 Cuidados com a Incompatibilidade

Os modelos de programação em *clusters* de nós multiprocessados têm vantagens, mas também sérias desvantagens relativas aos problemas de incompatibilidade entre o modelo híbrido de programação e a arquitetura híbrida, a seguir temos algumas situações que precisam sempre ser verificadas:

- Com MPI puro, para minimizar a comunicação entre os nós, exige-se que o domínio de aplicação coincida com a topologia do *hardware*;
- MPI puro também introduz a comunicação entre os nós, que no caso de nós de SMPs pode ser omitido se for utilizado programação híbrida;
- Por outro lado, programação MPI e OpenMP não é capaz de utilizar a largura de banda completa do nó, quando se utiliza apenas uma *thread* para responder pela comunicação;
- Utilizando modelo *masteronly*, todas as *threads* que não fazem comunicação ficam paradas sem executar processamento no momento da troca de mensagens entre os nós.
- Tempo de CPU também é desperdiçado, se todos os processadores de um nó SMP comunicarem ao mesmo tempo, pois alguns processadores já são capazes de saturar a largura de banda do nó.

  A sobreposição de comunicação e computação é uma maneira para tentar obter uma utilização melhor do *hardware*, mas:

- Requer um grande esforço do programador para separar no programa as *threads* responsáveis pela comunicação e as *threads* responsáveis pelos dados;
- *Threads* de comunicação e *thread* de computação devem ser balanceadas.

Mais informações detalhadas sobre problemas são apresentadas em [23].

4.6 Considerações Finais

Este capítulo apresentou a dupla MPI e OpenMP como uma das soluções mais citadas para utilização em programação híbrida. Também foram apresentados os principais modelos de programação para arquiteturas híbridas segundo [24]. Por fim, foram listadas algumas vantagens e desvantagens da programação híbrida. No capítulo seguinte será apresentado o caso de estudo e a aborgagem proposta para os testes deste trabalho. Ainda serão descritos os recursos computacionais utilizados.

5. ABORDAGEM PROPOSTA

Neste capítulo será apresentada a aplicação que foi utilizada como caso de estudo para validar e entender os resultados obtidos neste trabalho sobre programação híbrida em *clusters* de máquinas NUMA. Também será apresentada a abordagem proposta para o desenvolvimento da solução paralela na sua versão híbrida, utilizando MPI e OpenMP. Ainda será utilizada a biblioteca MAi de modo a fazer uso dos recursos da *libnuma* para obter melhor desempenho em *clusters* de máquinas NUMA explorando afinidade de memória.

5.1 Modelo ICTM

Nesta seção será apresentada uma definição do ICTM de forma mais geral, qual seja, um modelo geral, baseado em tesselações, capaz de produzir uma categorização confiável de sub-regiões de um espaço de características geométricas, podendo analisar múltiplas características conhecidas em suficientemente muitos pontos [2]. A categorização determinada por cada característica é efetuada em uma camada do modelo, gerando diferentes subdivisões da região analisada. Por exemplo, uma região pode ser analisada conforme sua topografia, vegetação, demografia, dados econômicos, etc.

O conjunto de pontos analisados pode pertencer a um espaço multi-dimensional, determinando, assim, o caráter multi-dimensional de cada camada. Uma categorização global pode ser alcançada, a partir da categorização de cada camada, mediante um procedimento de projeção. Esta categorização global determinará uma subdivisão mais confiável e com maior significância, combinando as análises efetuadas para cada característica. Este tipo de projeção permite análises bastante interessantes sobre a dependência mútua destas características.

Cada característica da aplicação está representada em uma camada do modelo ICTM. Assim, devido às características paralelas, as subdivisões em cada camada também podem ocorrer de forma independente.

Os dados que servem como entrada para o ICTM são extraídos de imagens de satélites, nas quais as informações são referenciadas por pontos correspondentes as coordenadas de latitude e longitude. A região geográfica é então representada por uma tesselação regular. Esta tesselação é determinada pela subdivisão da área total em sub-áreas retangulares suficientemente pequenas. Cada uma destas sub-áreas representa uma célula da tesselação. Esta subdivisão é feita de acordo com o tamanho da célula, o qual é estabelecido por um analista geofísico ou ecologista e está diretamente associado ao grau de refinamento dos dados de entrada [2]. O conceito de tesselações pode ser entendido como uma generalização do conceito de autômatos celulares. Ou seja, assim como os autômatos celulares, as tesselações são malhas de células idênticas e discretas, onde cada uma delas possui um estado. Este estado é determinado localmente a partir dos estados das células vizinhas.

Com intuito de minimizar e controlar os erros oriundos da discretização da região em células da tesselação, o ICTM utiliza Matemática Intervalar. O uso de intervalos traz diversos benefícios ao

processo de categorização, porém sua utilização faz com que ocorra um aumento da necessidade de poder computacional para o processamento da categorização.

A seguir, tem-se uma explicação do processo de categorização e os impactos no desempenho quando são alterados parâmetros importantes do modelo, tais como a dimensão das matrizes e o raio.

5.1.1 Processo de Categorização

O processo de categorização é realizado em cada camada do modelo de entrada de forma seqüencial. Portanto, todas as camadas passam pelo mesmo processo de categorização para que, posteriormente, estes resultados possam ser projetados em uma camada base. A categorização de cada camada é composta por diversas etapas seqüenciais, onde cada uma utiliza os resultados obtidos na etapa anterior. É possível dividir o processo de categorização em duas fases: a fase de preparação e a fase de categorização. A Figura 5.1 apresenta o processo de categorização de uma camada qualquer do ICTM.

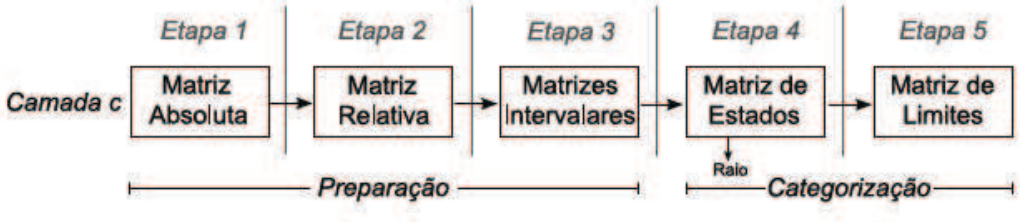

Figura 5.1: Processo de Categorização.

A fase de preparação compreende três etapas seqüenciais. A primeira delas involve a leitura dos dados de entrada (extraídos de imagens de satélite) e estes são armazenados em uma matriz denominada Matrix Absoluta. A Matriz Absoluta é normalizada através da divisão dos valores de cada célula pela célula que possui o maior valor, criando-se assim a Matriz Relativa. Após são utilizadas técnicas de Matemática Intervalar para criar as Matrizes Intervalares.

A primeira etapa desta fase de categorização será responsável por construir a Matriz de Estados. Esta matriz representará a relação do comportamento de cada célula com seus vizinhos em cada uma das quatro direções: norte, sul, leste e oeste. O processamento é feito por direção, ou seja, primeiro cada célula é comparada com seus vizinhos na direção norte, posteriormente cada célula é comparada com seus vizinhos na direção sul e assim sucessivamente. O número de células vizinhas a serem utilizadas para determinar a relação de cada célula com as demais é parametrizável. A este parâmetro é dado o nome de **raio**. Finalmente, a última etapa compreende a criação da Matriz de Limites. Nesta matriz é armazenada a informação de quais células são consideradas limítrofes entre regiões de características distintas. Com isto é possível identificar quais grupos de células possuem características em comum.

É importante salientar que a categorização de somente uma camada que representa uma grande região possui um custo computacional muito alto. Este custo está relacionado basicamente a dois

parâmetros: a dimensão da tesselação e o número de vizinhos a serem analisados. Quanto maior a região e maior o raio, maior será a necessidade de poder computacional.

Como o modelo ICTM foi bastante discutido nos trabalhos [26] e [7] já sob uma análise voltada para a paralelização da aplicação, entende-se salutar referenciá-los.

5.2 ICTM Híbrido

A metodologia que está sendo utilizada para o desenvolvimento da aplicação híbrida é baseada na estrutura de programação hierárquica. Explorando a troca de mensagens para distribuição de tarefas entre os nós do *cluster* e explorando a programação de memória compartilhada dentro do nó. No primeiro momento, será analisada a aplicação utilizando *OpenMP* para explorar o paralelismo intra nó. Já em um segundo momento, será utilizando também a *interface* MAi para que seja obtido melhoras com relação a alocação de memória, visto que a MAi utiliza-se da *libnuma*.

A Figura 5.2 ilustra uma matriz com as informações de uma camada do ICTM dividida em blocos. A versão híbrida do ICTM fará com que cada nó do *cluster* receba informações destes blocos para realizar o processamento. Este processamento requer que os nós troquem informações das bordas dos blocos, e isto poderia ser feito utilizando MPI. Porém, o trabalho [26] utilizou-se desse expediente e obteve resultados modestos em função do grande número de trocas de mensagens necessário para chegar ao final do processamento. Neste ponto é relevante salientar que quanto maior o raio maior é a quantidade de informação que dever ser trocada.

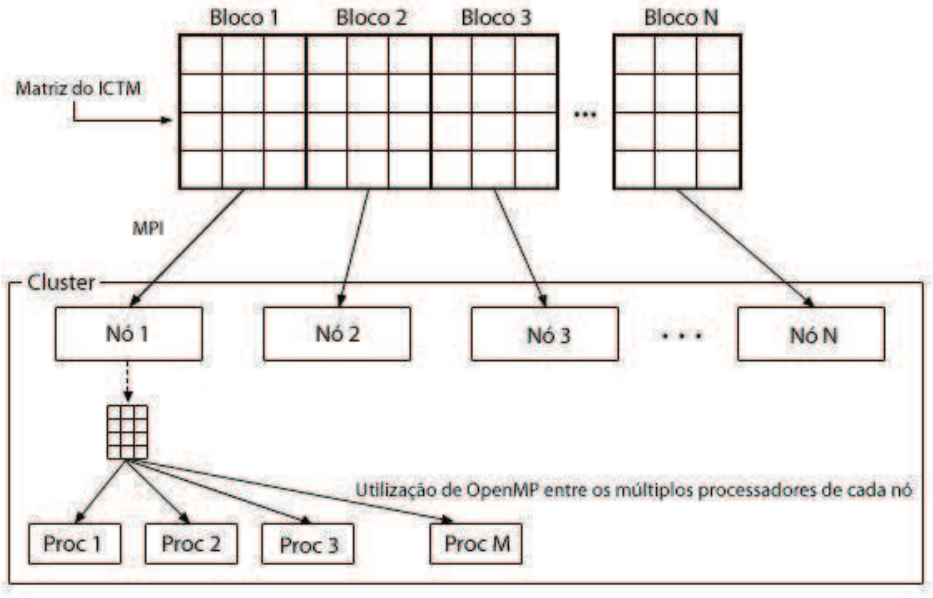

Figura 5.2: Divisão dos blocos da matriz do ICTM utilizando hierarquia de memória.

Com base nesta experiência, foi decidido considerar que cada nó irá processar além do bloco que lhe for destinado, também uma borda correspondente ao raio definido no início da execução. Isso vale tanto para a direita quanto para a esquerda, tendo os devidos cuidados com o primeiro e o último blocos, conforme ilustra a Figura 5.3. Dessa maneira se terá uma computação redundante

nos nós, mas isso é aceitável, visto que do contrário seria necessário uma troca de mensagens num volume muito grande, o que colocaria em risco a viabilidade da implementação híbrida. E como o raio representa um pequeno percentual a ser acrecido, o retrabalho certamente será recompensado.

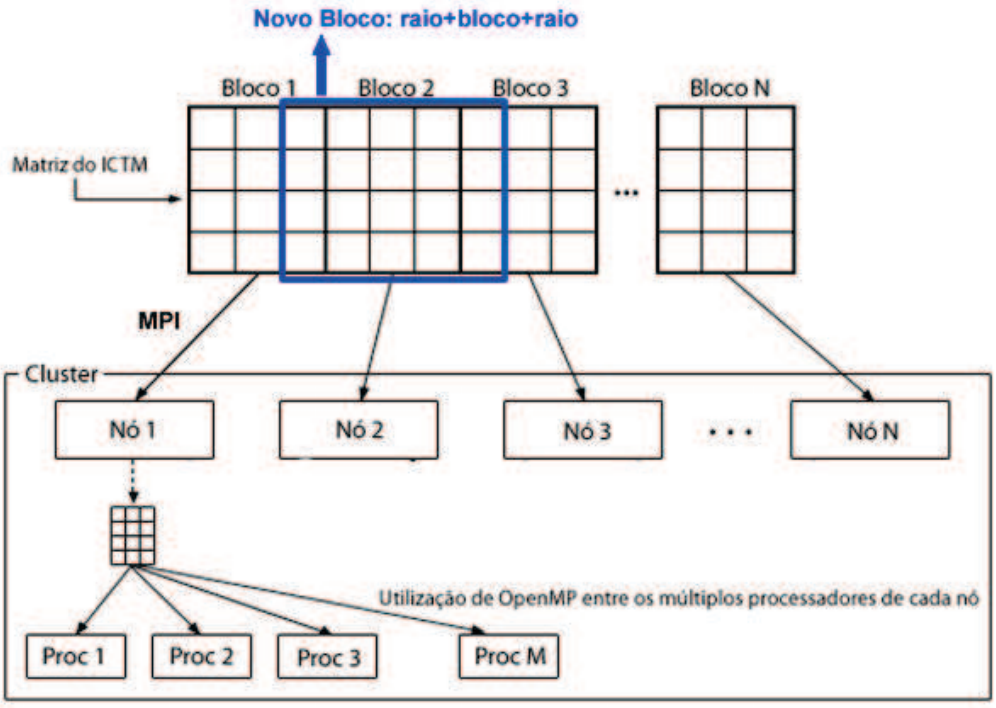

Figura 5.3: Novo Bloco criado com os raios.

Já o processamento do bloco em cada um dos nós multiprocessados será paralelizado utilizando OpenMP. Essa implementação OpenMP é fortemente baseada na implementação do trabalho [7]. Inclusive tem-se uma versão utilizando apenas OpenMP e outra versão utilizando também a *interface* MAi, buscando melhores resultados com afinidade de memória, visto que os *clusters* possuem recursos NUMA.

Sendo assim, será utilizado o modelo de programação paralelo mestre/escravo. Onde um dos nós ficará responsável por dividir as tarefas entre os demais nós. Este nó é o mestre e os demais nós são chamados de escravos. O processo mestre após enviar as informações do bloco que deve ser processado para cada um dos escravos, ficará aguardando o retorno do escravo informando a finalização do processo. Após todos os blocos serem processados pelos escravos, então o processo mestre finaliza a aplicação.

5.3 Recursos Computacionais

A aplicação foi executada no *cluster* **Atlantica**, que está localizado no LAD - Laboratório de Alto Desempenho da PUCRS. O LAD é um laborátorio que provê recursos computacionais de alto desempenho para os grupos de pesquisa da PUCRS.

O *cluster* Atlantica é composto por 10 máquinas Dell PowerEdge R610. Cada máquina possui dois processadores *Inter Xeon Quad-Core E5520 2.27GHZ Hyper-Threading*, totalizando 16 *cores* por nó, 16GB de memória RAM e 150 GB de capacidade de armazenamento em disco. Os nós estão interligados por duas redes *Gigabits-Ethernet* chaveadas, uma para a comunicação entre os nós e uma para a gerência. O processador *Intel Xeon Processor E5520* possui 8M *cache*, *clock* de 2.27 GHz e 5.86 GT/s de Intel QPI.

O Intel QPI (*QuickPath Interconnect*) é a parte chave da nova arquitetura da Intel, *QuickPath Architecture*. O QPI foi lançado, no primeiro semestre de 2009 [14], para substituir o FSB (*Front Side Bus*) ou barramento frontal, que vinha sendo utilizado nos processadores mais antigos da Intel. O FSB consiste em um barramento compartilhado que liga o processador ao *chipset*, conforme figura 5.4. Como o FSB não é utilizado apenas para a comunicação entre os *cores* e a memória, mas também é utilizado para comunicação entre os *cores*, ele acaba engargalando o acesso a memória, prejudicando o desempenho do sistema. O problema se multiplica conforme aumento o número de processadores em uma máquina.

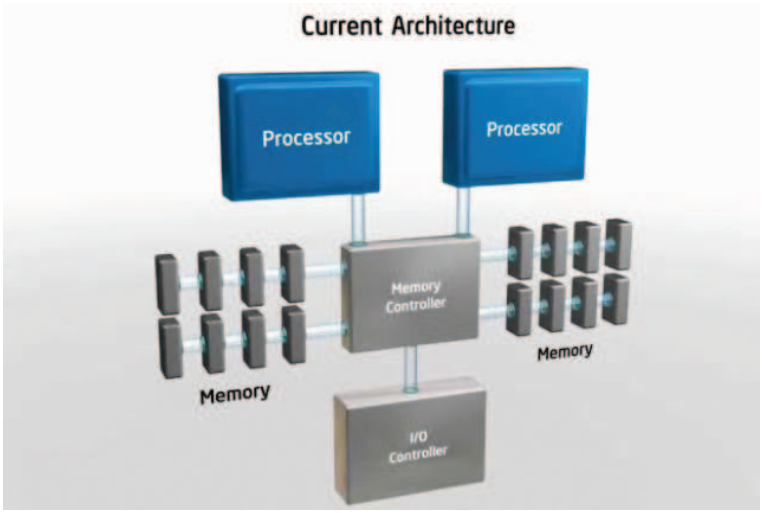

Figura 5.4: Arquitetura utilizando o FSB [14].

Sendo assim o *QuickPath Interconnect* veio substituir o FSB. O QPI são *links* independentes que operam a 4.8 ou 6.4 GT/s, com a transmissão de 16 *bits* de dados em cada direção por ciclo, resultando em um barramento de 9.6 ou 12.8 GB/s em cada direção (25.6 GB/s) por linhas de dados.

Como a memória é agora acessada diretamente pelo controlador de memória, este *link* fica inteiramente disponível para o tráfego de *I/O*. Ao utilizar dois processadores, cada processador passa a se comunicar com o *chipset* através de uma linha independente e uma terceira linha de dados é implantada para coordenar a comunicação entre os dois, conforme figura 5.5. Esse é o caso da arquitetura que esta sendo utilizada para realizar os testes deste trabalho.

Ao usar 4 processadores ou mais, possibilidade que deverá ser bem explorada no caso dos servidores de alto desempenho, são incluídos barramentos adicionais, que fazem com que cada processador tenha acesso direto a todos os demais, conforme a figura 5.6, que exemplifica uma arquitetura

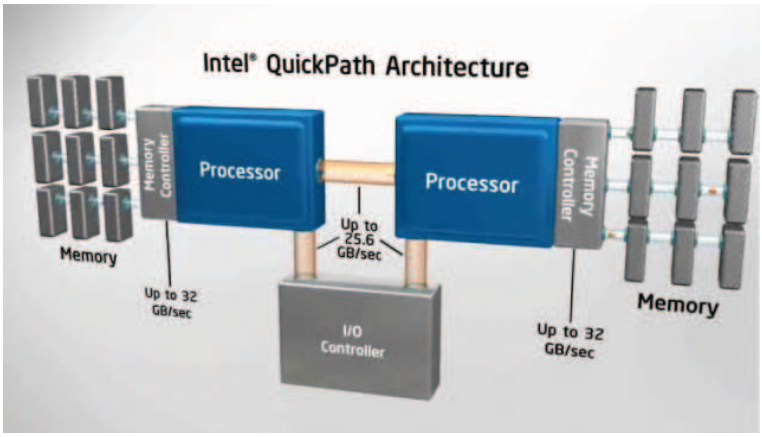

Figura 5.5: Arquitetura utilizando QPI [14].

com 4 processadores.

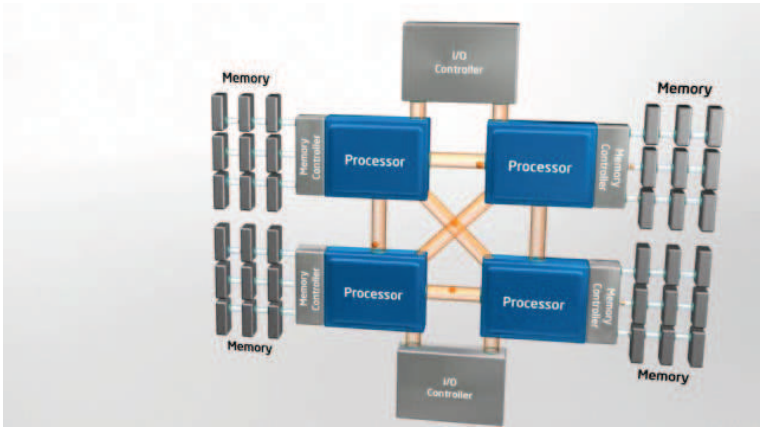

Figura 5.6: Arquitetura com 4 processadores utilizando QPI [14].

Portanto, com essa nova arquitetura os processadores Intel adquiriram características de máquinas NUMA, ou seja, cada um dos processadores possui um tempo não uniforme para o acesso à memória.

5.4 Considerações Finais

Neste capítulo foi descrita a aplicação que será utilizada como caso de estudo para este trabalho:*ICTM - Interval Categorizer Tesselation-based Model*. Este capítulo, ainda, apresentou a abordagem utilizada para a implementação da solução do ICTM híbrido. Então será utilizado o modelo de programação paralelo mestre/escravo. Cada escravo irá receber um determinado bloco para processar através do MPI. Após o processamento destes blocos utilizando OpenMP os processos escravos informam o processo mestre que terminaram suas tarefas. Após todas as tarefas concluídas, o processo mestre finaliza a aplicação. Ainda foram apresentados os recursos computacionais que serão utilizados para a realização dos testes. Já no capítulo seguinte serão apresentados os resultados obtidos nos testes, bom como uma análise sobre estes resultados.

6. RESULTADOS OBTIDOS

De um modo geral, cada etapa do processo de categorização realiza um determinado tipo de computação sobre a matriz relacionada a etapa. Para isto, são consultados valores referentes a matrizes anteriormente criadas. Conforme descrito na seção 5.1.1 que mostra como as etapas utilizam as matrizes durante o processamento.

Porém, para o cálculo das matrizes de monotonicidade o parâmetro raio é levado em consideração. Desta forma, a cada uma destas etapas a célula da matriz é comparada com r vizinhos, onde r varia de 1 até o valor do raio passado como parâmetro. Logo, a utilização de valores maiores para o raio aumenta ainda mais a quantidade de processamento nestas etapas de monotonicidade.

A divisão do trabalho entre as *threads* é realizada da seguinte forma: cada *thread* será responsável por processar um conjunto de linhas da matriz. Desta forma, o trabalho foi divido entre os nós considerando apenas a divisão por colunas. Logo se uma matriz possui uma dimensão 10.000×10.000 e está sendo paralelizada por um *cluster* com 4 nós, cada um destes nós irá processar um bloco de 10.000×2.500, conforme descrito na seção 5, é verdade que é necessário considerar o raio que será somado as estremidades desse bloco. Esta solução não apresenta problemas para as etapas de monotonicidade, onde a vizinhança é consultada para o processamento de cada célula, pois estas etapas somente consultam dados das Matrizes Intervalares. Logo, não há dependência mútua entre etapas de cálculo das Matrizes de Monotonicidade.

De acordo com as características do ICTM, basicamente dois parâmetros possuem influência imediata no desempenho da solução seqüencial: a dimensão das matrizes e o raio utilizado nas etapas de cálculo da monotonicidade. Portanto, os casos de estudo apresentados aqui foram baseados na modificação destes parâmetros com o intuito de avaliar a solução híbrida proposta neste trabalho.

É importante salientar que os dados de entrada utilizados neste trabalho não são dados reais oriundos de imagens de satélite. Devido a dificuldade destes dados serem obtidos, optou-se por criar casos de testes com valores aleatórios. Isto é possível devido ao fato de que o poder de processamento necessário para categorizar uma determinada camada do ICTM está vinculado aos parâmetros citados anteriormente (dimensão das matrizes e raio). Logo, a utilização de dados aleatórios, ao invés de reais, para a geração da Matriz Absoluta não altera o tempo de processamento durante o processo de categorização de regiões de mesma dimensão e raio. Os casos de teste apresentados aqui sempre utilizarão somente uma única camada no modelo.

O trabalho [7] utilizou matriz de vários tamanhos como: 4.800×4.800, 6.700×6.700, 9.400×9.400 e 13.300×13.300 e raios considerando 20, 40 e 80 vizinhos. Como estamos considerando uma solução híbrida que tem por característica ser escalável, iremos trabalhar com matrizes maiores: 10.000×10.000, 15.000×15.000 e 20.000×20.000, no entanto iremos utilizar os mesmo valores de raio: 20, 40 e 80, pois já são valores adequados para os testes.

Duas importantes medidas de qualidade de programas paralelos serão utilizadas para avaliar o desempenho das soluções apresentadas neste trabalho: *speed-up* e eficiência. Todos os resultados

foram baseados na utilização de médias, onde cada caso de estudo, com a mesma configuração de parâmetros, foi executado 5 vezes, excluindo-se o pior e melhor tempo de execução. Estas médias apresentaram um desvio padrão bastante pequeno, pois todos os experimentos foram realizados com acesso exclusivo aos nós do *cluster*. Inclusive foram colocados nas tabelas valores com duas casas decimais, então os casos em que valores de desvio estão como 0, é porque são menores de 0,01.

Um outro ponto bastante relevante a salientar-se é com relação às politicas de memória utilizadas pelas versões que fazem uso dos recursos NUMA da arquitetura. Será utilizada a política que obteve os melhores resultados no trabalho [7], a saber, a **política cíclica**, ou *round-robin* ou *cyclic*.

6.1 ICTM-OpenMP e ICTM-NUMA

Nesta seção serão apresentados os resultados obtidos após a execução da aplicação ICTM-OPENMP e ICTM-NUMA como foram chamadas no trabalho [7], a primeira utiliza-se apenas do OpenMP para obter paralelismo em um unico nó, já a segunda utiliza-se também dos recursos NUMA do nó. Foram necessários alguns ajustes para a execução na arquitetura apresentada. É importante salientar que não houve nenhuma alteração no código, no entanto a aplicação precisou ser compilada novamente para execução.

A tabela 6.1 e a tabela 6.2 apresentam os resultados obtidos após a execução da aplicação ICTM-OpenMP, ou seja, a aplicação que utiliza o **OpenMP** para explorar o paralelismo em apenas **um** nó. Valores com as matrizes **20.000×20.000** não estão apresentados porque com esses valores os nós começaram a utilizar a área de *swap*, o que inviabiliza a análise do tempo. Portanto, os valores apresentados nas tabelas representam o tempo em segundos necessário para o processamento de cada caso de teste.

Tabela 6.1: Resultados ICTM-OpenMP - Matriz 10.000×10.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	75.12	0.02	129.98	0.01	248.6	0.01
2	38.12	0.16	65.69	0.01	124.97	0.13
4	19.23	0.01	33.04	0.05	63.46	0.04
8	13.19	0.23	19.61	0.02	35.73	0.20
16	11.56	0.05	18.83	0.29	33.34	0.05

Tabela 6.2: Resultados ICTM-OpenMP - Matriz 15.000×15.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	173.98	0.02	301.22	0.08	564.97	0.11
2	87.35	0.23	151.23	0.03	282.77	0.02
4	44.26	0.13	76.09	0.02	150.48	0.08
8	23.08	0.04	44.21	1.87	77.43	0.20
16	24.12	0.03	40.48	0.12	72.72	0.13

Já as tabelas 6.3 e 6.4 apresentam os resultados obtidos após a execução da aplicação ICTM-NUMA, ou seja, a aplicação que utiliza o **OpenMP + Libnuma e MAi** para explorar o paralelismo em apenas **um** nó.

Tabela 6.3: Resultados ICTM-NUMA - Matriz 10.000×10.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	75.23	0.00	130.45	0.01	251.42	0.05
2	37.87	0.00	65.52	0.00	125.93	0.05
4	18.95	0.00	32.8	0.01	64.02	0.20
8	9.51	0.00	16.68	0.01	34.44	0.00
16	9.09	0.02	16.61	0.01	31.24	0.03

Tabela 6.4: Resultados ICTM-NUMA - Matriz 15.000×15.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	169.23	0.04	295.93	0.05	582.91	1.76
2	85.27	0.01	148.68	0.02	288.73	0.70
4	42.68	0.00	74.76	0.03	160.06	0.05
8	21.55	0.14	42.17	0.01	83.10	0.00
16	21.51	0.02	39.46	0.01	74.84	0.00

Portanto, temos quatro tabelas que apresentam tempos para os cálculos. Em uma análise apenas sobre essas tabelas já podemos confirmar os resultados obtivos pelo trabalho [7], mesmo que estejamos executando essa aplicação em um ambiente muito diferente daquele utilizado pelo trabalho anterior.

Para se comparar alguns números pode-se verificar o tempo da tabela 6.1 para 16 *cores* e raio 20 que é de 11.56 segundos, bater com o correspondente valor da tabela 6.3, que é de 9.09. Neste caso tem-se um ganho de aproximadamente 27%.

As figuras 6.1 e 6.2 apresentam os *speedups* obtidos com base nos resultados obtidos com a versão ICTM-OpenMP. Nestas figuras pode-se observar que o *speed-up* até 8 *cores* é bastante bom, contudo com 16 *cores* esse valor distância-se bastante do valor ideial. Isto está relacionado diretamente com a arquitetura utilizada, ou seja, possivelmente se enfrenta um gargalo de acesso à memória quando os 16 *cores* precisam acessar à memória, visto que é utilizada o *hyperthreading* dos processadores.

Já as figuras 6.3 e 6.4 apresentam os *speedups* obtidos com base nos resultados da versão ICTM-NUMA. Nestes gráficos também foram adicionados os valores de *speed-up* da versão ICTM-OpenMP para uma comparação direta. Então, é possível perceber os ganhos obtidos com a versão ICTM-NUMA. No entanto, o gargalo quando se utiliza os 16 *cores* continua. É verdade que se tem uma melhora, mas ainda está distante do valor ideal.

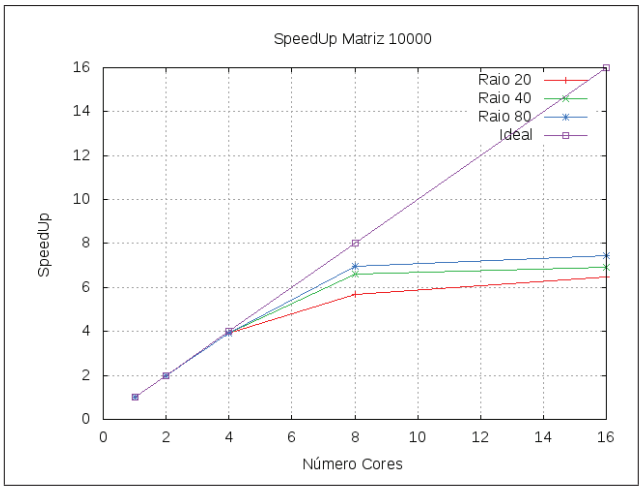

Figura 6.1: *Speed-up* ICTM-OpenMP Matriz 10.000×10.000.

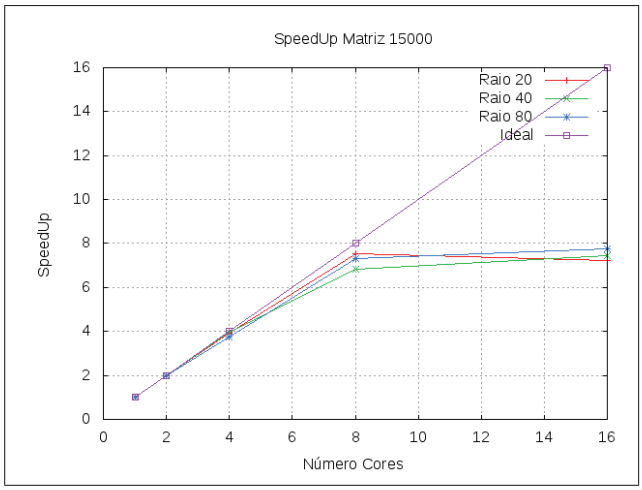

Figura 6.2: *Speed-up* ICTM-OpenMP Matriz 15.000×15.000.

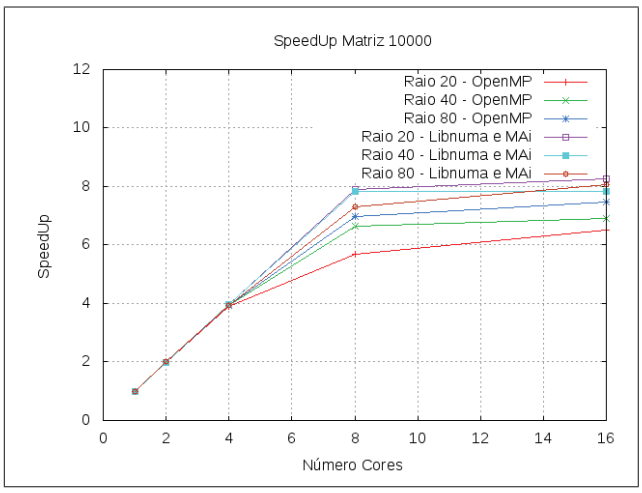

Figura 6.3: *Speed-up* ICTM-NUMA Matriz 10.000×10.000.

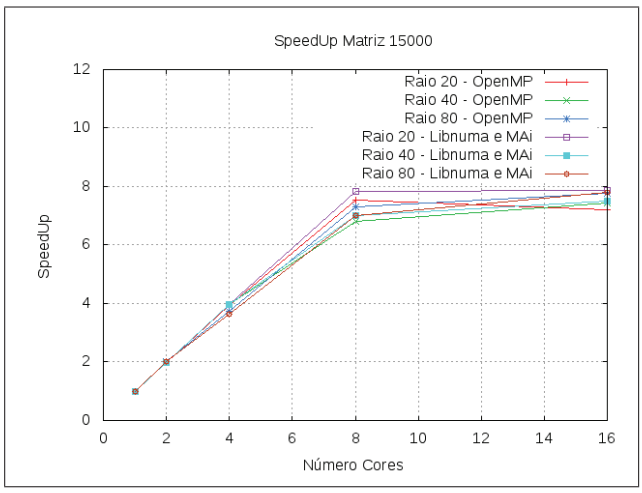

Figura 6.4: *Speed-up* ICTM-NUMA Matriz 15.000×15.000.

6.1.1 Análise dos Resultados

Nesta seção será feita uma análise dos resultados obtidos com a execução das versões ICTM-OpenMP e ICTM-NUMA, com base nos gráficos de eficiência que são apresentados nas figuras 6.5, 6.6, 6.7 e 6.8.

A versão ICTM-OpenMP apresentou valores de eficiência para 2, 4 e 8 *cores* muito bons, no entanto com 16 *cores*, como era de se esperar os valores foram modestos, mas isso pode-se associar a arquitetura do ambiente utilizado. No entanto, pode-se inferir após a análise dos gráficos de eficiência da versão ICTM-OpenMP, que quanto maior o raio, ou seja, quanto maior o processamento melhor é a eficiência. Isso deve-se ao fato de que o tempo disperdiçado para acesso à memória é melhor "'disperso"'. Já comparando-se com os gráficos da versão ICTM-NUMA está afirmativa confirma-se, ou seja, na versão ICTM-NUMA quanto menor o raio, maior a eficiência, isto deve-se ao fato de o

acesso à memória ser melhor executado, visto que está sendo explorada a questão de afinidade de memória.

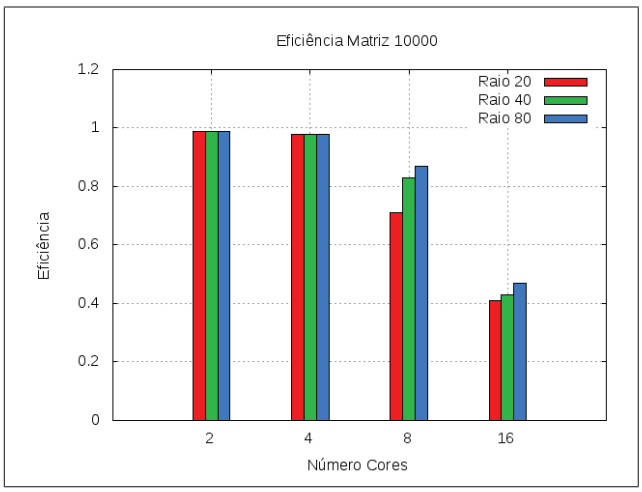

Figura 6.5: Eficiência ICTM-OpenMP Matriz 10.000×10.000.

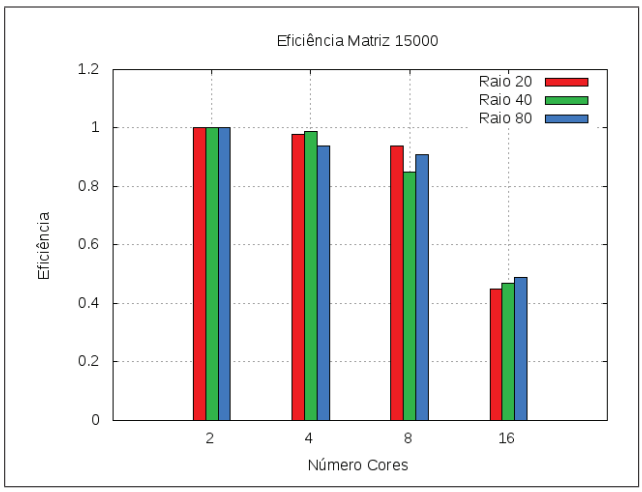

Figura 6.6: Eficiência ICTM-OpenMP Matriz 15.000×15.000.

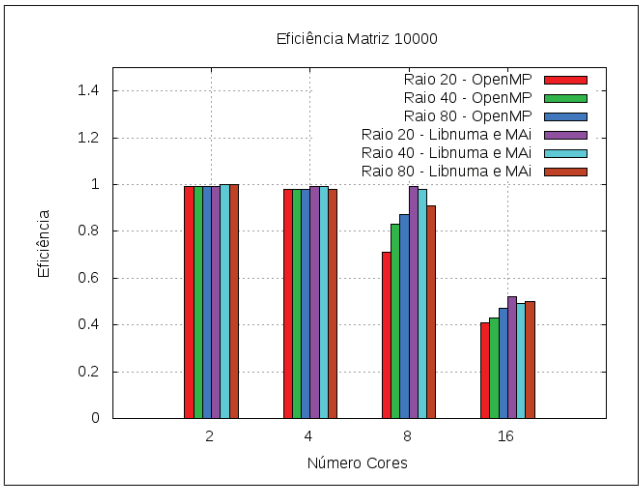

Figura 6.7: Eficiência ICTM-NUMA Matriz 10.000×10.000.

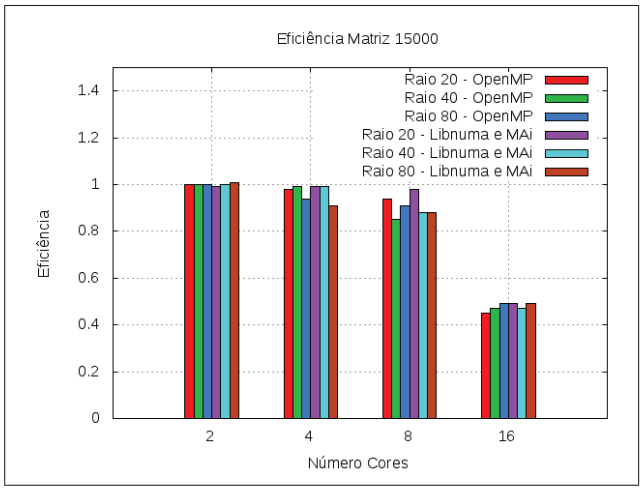

Figura 6.8: Eficiência ICTM-NUMA Matriz 15.000×15.000.

6.2 ICTM Híbrido MPI + OpenMP

Nesta seção serão apresentados os resultados obtidos após a execução da aplicação ICTM Híbrido MPI + OpenMP. Estes resultados serão comparados com os valores obtidos anteriormente na seção 6.1. Nesta implementação o trabalho é distribuído pelo processo mestre para os processos escravos utilizando o MPI. Dentro de cada nó os blocos referentes aquele nó serão processados paralelamente utilizando o OpenMP para isso. Com base no trabalho [26], sabe-se que o melhor resultado obtido

foi quando o número de blocos a ser processados foi igual ao número de escravos, então neste trabalho será utilizado o mesmo expediente.

As tabelas 6.5 e 6.6 apresentam os resultados obtidos após a execução da aplicação ICTM Híbrido com o MPI + OpenMP utilizando 4 nós do *cluster*, ou seja, a aplicação explora o paralelismo entre os nós do *cluster* utilizando MPI e intra nó utilizando apenas OpenMP. Por sua vez, as tabelas 6.7 e 6.8 apresentam os resultados utilizando 6 nós do *cluster*.

Tabela 6.5: Resultados ICTM Híbrido MPI + OpenMP - 4 Nós e Matriz 10.000×10.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	25.16	0.09	43.57	0.10	82.93	0.14
2	12.81	0.08	22.03	0.08	41.75	0.11
4	6.57	0.08	11.20	0.09	21.15	0.12
8	5.21	0.71	8.22	1.14	14.47	1.58
16	5.45	0.17	7.70	0.17	12.57	0.20

Tabela 6.6: Resultados ICTM Híbrido MPI + OpenMP - 4 Nós e Matriz 15.000×15.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	57.79	0.13	100.35	0.15	187.99	0.25
2	29.36	0.12	50.66	0.12	94.42	0.16
4	14.94	0.13	25.61	0.13	47.60	0.23
8	10.31	1.55	16.35	1.87	26.70	1.71
16	10.43	0.27	15.63	0.21	26.72	0.25

Tabela 6.7: Resultados ICTM Híbrido MPI + OpenMP - 6 Nós e Matriz 10.000×10.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	15.11	0.12	26.23	0.13	49.95	0.44
2	7.75	0.10	13.32	0.09	25.17	0.12
4	4.05	0.08	6.84	0.08	12.79	0.12
8	3.67	0.43	5.73	0.66	9.56	1.10
16	4.24	0.18	5.58	0.19	8.40	0.21

Tabela 6.8: Resultados ICTM Híbrido MPI + OpenMP - 6 Nós e Matriz 15.000×15.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	34.68	0.19	60.38	0.22	112.89	0.42
2	17.66	0.14	30.53	0.17	56.92	0.68
4	9.05	0.13	15.51	0.12	28.70	0.15
8	7.30	0.83	11.53	1.54	18.79	2.71
16	7.78	0.54	10.77	0.27	17.18	0.30

Aqui, temos quatro tabelas que apresentam tempos para os cálculos utilizando a versão híbrida com 4 e 6 nós do *cluster*. Em uma análise apenas sobre essas tabelas já podemos verificar a redução de tempo obtida para resolução do problema se comparadas as tabelas obtidas com a versão ICTM-OpenMP.

Para se comparar alguns números pode-se verificar o tempo da tabela 6.1 para 8 *cores* e raio 20 que é de 13.19 segundos, comparar com os correspondentes valores das tabelas 6.5 e 6.7, que são respectivamente 5.21 segundos para 4 nós e 3.67 segundos para 6 nós. Ou seja, com 4 nós ficou aproximadamente 2.5 vezes mais rápida e com 6 nós ficou aproximadamente 3.6 vezes mas rápida.

As figuras 6.9 e 6.10 apresentam os *speedups* obtidos com base nos resultados obtidos com a versão ICTM-OpenMP. Nestas figuras pode-se observar que o *speed-up* até 8 *cores* é bastante bom, contudo com 16 *cores* é um pouco pior. Isto está relacionado diretamente com a arquitetura utilizada, ou seja, possivelmente se enfrenta um gargalo de acesso à memória quando os 16 *cores* precisam acessar à memória em cada um dos nós, fazendo uso do recurso de *hyperthreading* dos processadores.

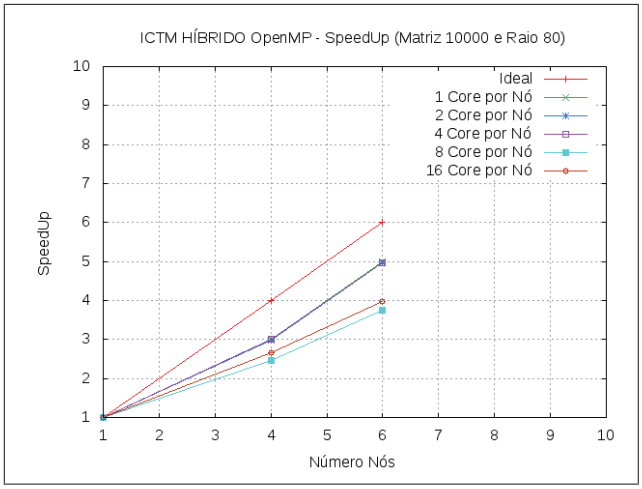

Figura 6.9: *Speed-up* ICTM Híbrido MPI + OpenMP - Matriz 10.000×10.000.

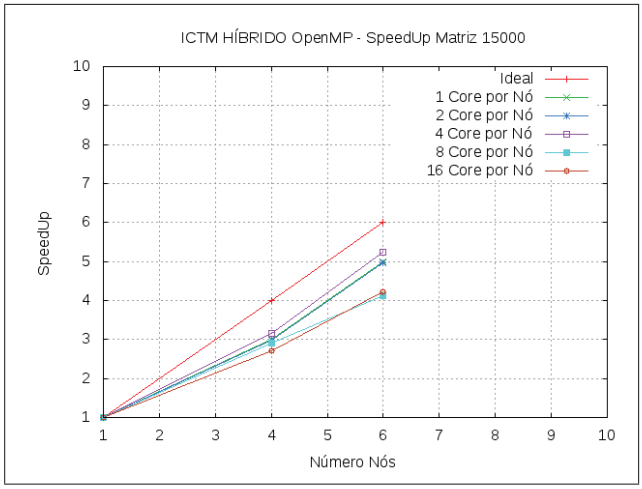

Figura 6.10: *Speed-up* ICTM Híbrido MPI + OpenMP - Matriz 15.000×15.000.

A seguir serão apresentados os *speedups* e as eficiências levando em consideração o número total de *cores* utilizados durante os testes. Portanto, será considerado o número de nós utilizados multiplicado pelo número de *cores* utilizado intra nó. A tabela 6.9 apresenta o *speed-up* e a eficiência relacionado a execução utilizando 4 nós, por sua vez, a tabela 6.10 considera a utilização de 6 nós. As figuras 6.11 e 6.12 apresentam gráficos relativos aos valores destas tabelas.

Os *speedups* obtidos quando se utiliza até 8 *cores* por nó, ou seja, no caso com 4 nós até 32 *cores*, são bastante bons, já os *speedups* obtidos quando se utiliza 16 *cores* por nó é bastante modesto. No estanto, já era esperado um resultado deste tipo visto que quando se utiliza os 16 *cores* no nó, percebe-se um gargalo no acesso à memória, em função da utilização do recurso de *hyperthreading* dos processadores. Contudo, é relevante salientar que mesmo com essa queda no *speed-up* tem-se bons resultados quando analisá-se o tempo para os cálculos.

Cores	*Speed-up*	Eficiência
1	1	1
4	3	0.75
8	5.95	0.74
16	11.76	0.73
32	17.18	0.54
64	19.78	0.31

Tabela 6.9: *Speed-up* e Eficiência ICTM Híbrido MPI + OpenMP - 64 *Cores*

Cores	*Speed-up*	Eficiência
1	1	1
6	4.98	0.83
12	9.88	0.82
24	19.44	0.81
48	26.01	0.54
96	29.61	0.31

Tabela 6.10: *Speed-up* e Eficiência ICTM Híbrido MPI + OpenMP - 96 *Cores*

6.2.1 Análise dos Resultados

Nesta seção será feita uma análise dos resultados obtidos com a execução da versão híbrida MPI + OpenMP, com base nos gráficos de eficiência que são apresentados nas figuras 6.13 e 6.14, esses valores de eficiência para 4 e 6 nós são muito bons, todos ficando acima de 60% de eficiência. No entanto, é importante salientar que a melhor eficiência foi obtida quando cada nó utilizoun 4 *cores*.

Já as figuras 6.15 e 6.16 apresentam a eficiência considerando o número total de *cores*. E nestes casos os valores de eficiência são bons quando se utiliza até 4 *cores* por nó. Com 8 e 16 cores, a eficiência cai para baixo de 60%.

6.3 ICTM Híbrido MPI + OpenMP + Libnuma e MAi

Nesta seção serão apresentados os resultados obtidos após a execução da aplicação ICTM Híbrido MPI + OpenMP + Libnuma e MAi. Estes resultados serão comparados com os valores obtidos nas duas seções anteriores. Nesta implementação o trabalho é distribuído pelo processo mestre para os processos escravos utilizando o MPI. Dentro de cada nó os blocos referentes aquele nó serão processados paralelamente utilizando o OpenMP + Libnuma e MAi, para explorar a afinidade de memória da arquitetura onde estão sendo realizados os testes. Ademais, o processo é bastante similar ao ICTM Híbrido MPI + OpenMP.

As tabelas 6.11 e 6.12 apresentam os resultados obtidos após a execução da aplicação ICTM Híbrido com o MPI + OpenMP + Libnuma e MAi utilizando 4 nós do *cluster*, ou seja, a aplicação explora o paralelismo entre os nós do *cluster* utilizando MPI e intra nó utilizando apenas OpenMP. Por sua vez, as tabelas 6.13 e 6.14 apresentam os resultados utilizando 6 nós do *cluster*.

Nestas quatro tabelas que apresentam tempos para os cálculos utilizando a versão híbrida MPI +

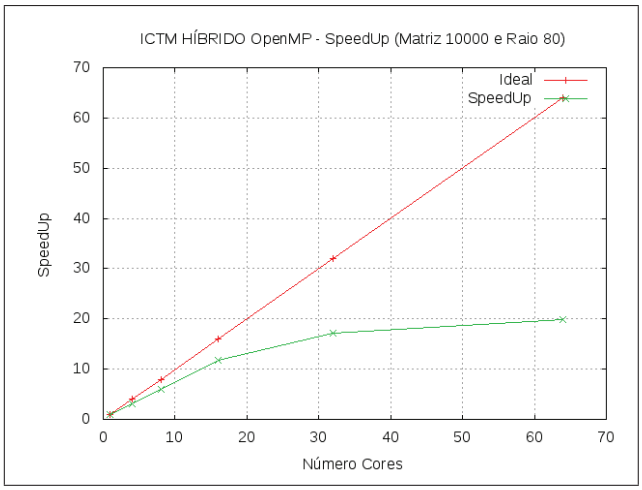

Figura 6.11: *Speed-up* ICTM Híbrido MPI + OpenMP - 64 *cores*.

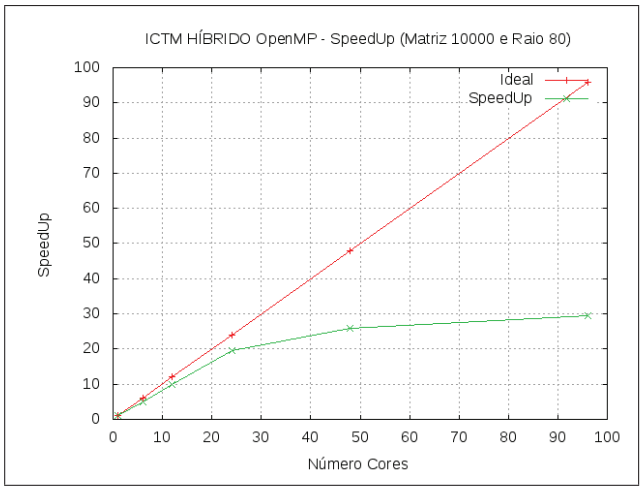

Figura 6.12: *Speed-up* ICTM Híbrido MPI + OpenMP - 96 *cores*.

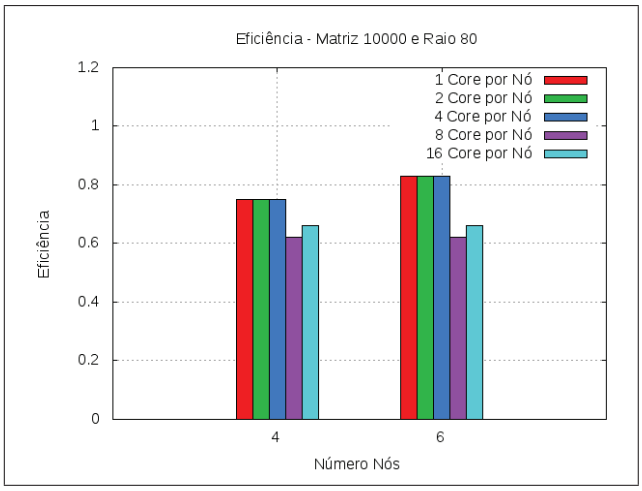

Figura 6.13: Eficiência ICTM Híbrido MPI + OpenMP Matriz 10.000×10.000.

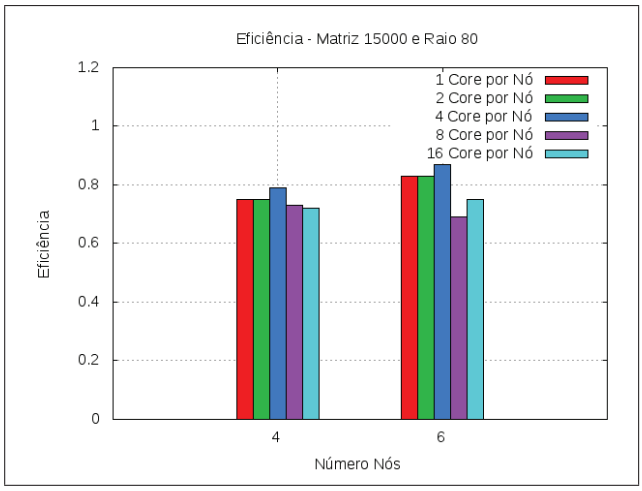

Figura 6.14: Eficiência ICTM Híbrido MPI + OpenMP Matriz 15.000×15.000.

Tabela 6.11: Resultados ICTM Híbrido MPI + OpenMP + NUMA - 4 Nós e Matriz 10.000×10.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	25.09	0.01	43.49	0.00	83.61	0.01
2	12.58	0.01	21.82	0.03	41.89	0.00
4	6.32	0.01	10.94	0.01	20.98	0.00
8	3.18	0.01	5.49	0.01	10.52	0.00
16	3.02	0.00	5.32	0.00	10.14	0.01

OpenMP + Libnuma e MAi com 4 e 6 nós do *cluster*. Em uma análise apenas sobre essas tabelas já podemos verificar a redução de tempo obtida para resolução do problema se comparadas as tabelas obtidas com a versão híbrida com MPI + OpenMP.

Para se comparar alguns números pode-se verificar o tempo da tabela 6.6 para 16 *cores* e

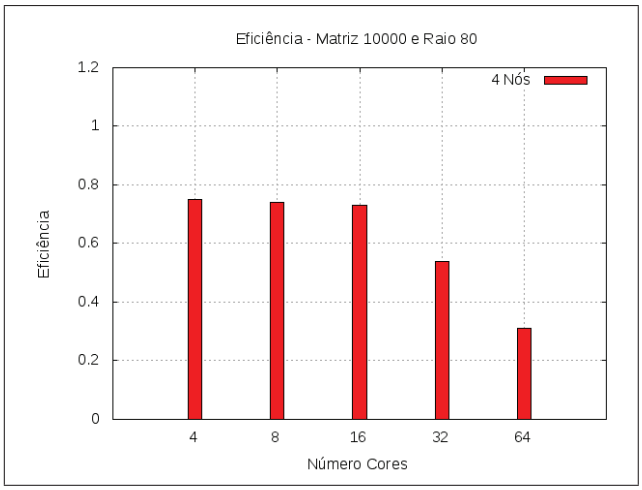

Figura 6.15: Eficiência ICTM Híbrido MPI + OpenMP - 64 *cores*.

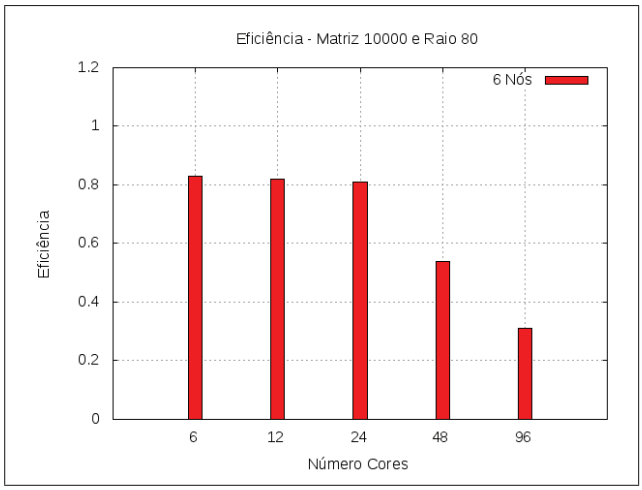

Figura 6.16: Eficiência ICTM Híbrido MPI + OpenMP - 96 *cores*.

Tabela 6.12: Resultados ICTM Híbrido MPI + OpenMP + NUMA - 4 Nós e Matriz 15.000×15.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	56.57	0.03	98.57	0.03	189.41	0.00
2	28.48	0.05	49.46	0.06	94.86	0.00
4	14.27	0.02	24.76	0.02	47.82	0.05
8	7.18	0.00	12.45	0.01	24.46	0.00
16	6.8	0.00	12.09	0.02	24.35	0.02

Tabela 6.13: Resultados ICTM Híbrido MPI + OpenMP + NUMA - 6 Nós e Matriz 10.000×10.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	15.22	0.02	26.24	0.00	50.2	0.00
2	7.66	0.00	13.19	0.01	25.16	0.01
4	3.82	0.01	6.61	0.00	12.59	0.00
8	1.93	0.00	3.32	0.00	6.31	0.00
16	1.83	0.00	3.21	0.00	5.95	0.01

Tabela 6.14: Resultados ICTM Híbrido MPI + OpenMP + NUMA - 6 Nós e Matriz 15.000×15.000

	Raio 20		Raio 40		Raio 80	
Cores	Média	Desvio	Média	Desvio	Média	Desvio
1	34.06	0.08	59.16	0.01	113.49	0.00
2	17.15	0.01	29.73	0.02	56.84	0.00
4	8.55	0.02	14.89	0.01	28.45	0.00
8	4.30	0.00	7.48	0.00	14.29	0.01
16	4.10	0.00	7.32	0.14	13.79	0.02

raio 20 que é de 10.43 segundos, e comparar com o correspondente valor da tabela 6.12, que 6.8 segundos. Ou seja, a utilização dos recursos NUMA, neste caso específico correspondeu a um ganho de aproximadamente 53%.

As figuras 6.17 e 6.18 apresentam os *speedups* obtidos com base nos resultados obtidos com a versão híbrida MPI + OpenMP + Libnuma e MAi. Nestas figuras pode-se observar que os *speedups* são bastante bons. Como se obteve ganhos em relação versão híbrida que utiliza apenas OpenMP, consequentemente isso reflete-se nos *speedups*.

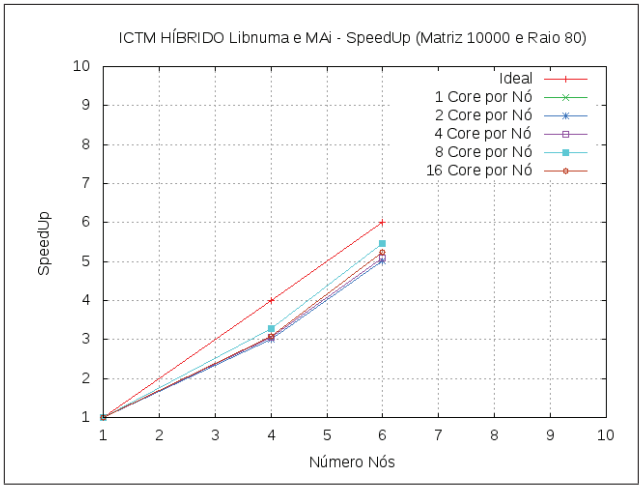

Figura 6.17: *Speed-up* ICTM Híbrido MPI + OpenMP + NUMA - Matriz 10.000×10.000.

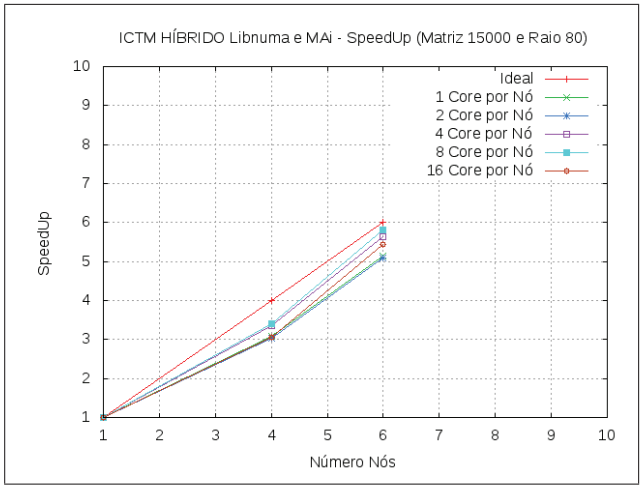

Figura 6.18: *Speed-up* ICTM Híbrido MPI + OpenMP + NUMA - Matriz 15.000×15.000.

A seguir serão apresentados os *speedups* e as eficiências levando em consideração o número total de *cores* utilizados durante os testes. Portanto, será considerado o número de nós utilizados multiplicado pelo número de *cores* utilizado intra nó. A tabela 6.15 apresenta o *speed-up* e a eficiência relacionado a execução utilizando 4 nós, por sua vez, a tabela 6.16 considera a utilização de 6 nós. As figuras 6.19 e 6.20 apresentam gráficos relativos aos valores destas tabelas.

Os *speedups* obtidos quando se utiliza até 8 *cores* por nó, ou seja, no caso com 4 nós até 32 *cores*, são bastante bons, já os *speedups* obtidos quando se utiliza 16 *cores* por nó é bastante modesto, mas ainda assim são valores melhores que aqueles obtidos pela versão híbrida com MPI + OpenMP. No estanto, já era esperado um resultado deste tipo visto que quando se utiliza os 16 *cores* no nó, percebe-se um gargalo no acesso à memória, como já foi afirmado anteriormente. Contudo, é relevante salientar que mesmo com essa queda no *speed-up* tem-se bons resultados quando analisá-se o tempo para os cálculos.

Tabela 6.15: *Speed-up* e Eficiência ICTM Híbrido MPI + OpenMP + NUMA - 64 *cores*

Cores	*Speed-up*	Eficiência
1	1	1
4	3.01	0.75
8	6	0.75
16	11.98	0.75
32	23.9	0.75
64	24.79	0.39

Tabela 6.16: *Speed-up* e Eficiência ICTM Híbrido MPI + OpenMP + NUMA - 96 *cores*

Cores	*Speed-up*	Eficiência
1	1	1
6	5.01	0.83
12	9.99	0.83
24	19.97	0.83
48	39.83	0.83
96	42.29	0.44

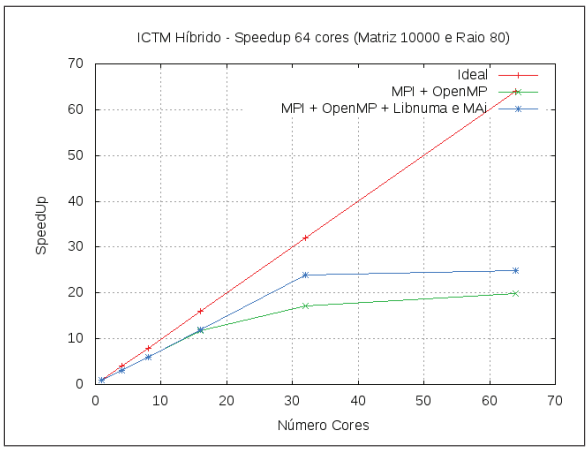

Figura 6.19: *Speed-up* ICTM Híbrido MPI + OpenMP + NUMA - 64 *cores*.

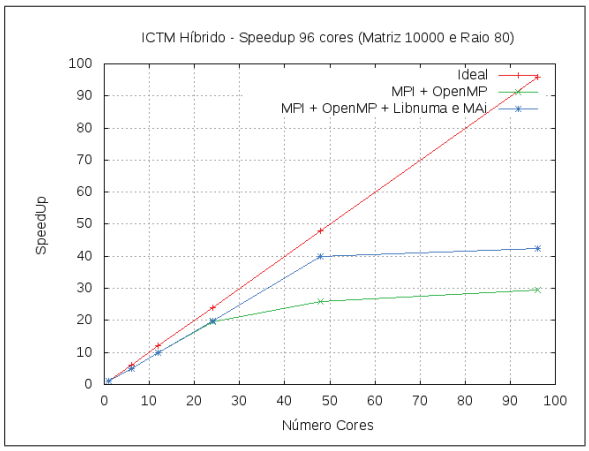

Figura 6.20: *Speed-up* ICTM Híbrido MPI + OpenMP + NUMA - 96 *cores*.

6.3.1 Análise dos Resultados

Esta seção apresenta uma análise dos resultados obtidos com a execução da versão híbrida MPI + OpenMP + Libnuma e MAi, com base nos gráficos de eficiência que são apresentados nas figuras 6.21 e 6.22, esses valores de eficiência para 4 e 6 nós são muito bons, todos ficando acima dos valores apresentados pela versão híbrida MPI + OpenMP.

Já as figuras 6.23 e 6.24 apresentam a eficiência considerando o número total de *cores*. E nestes casos os valores de eficiência são bons quando se utiliza até 8 *cores* por nó, o que não acontecia com a outra versão híbrida, que já com 8 *cores* apresentava uma eficiência ruim. Mas com 16 *cores*, a eficiência cai bastante.

6.4 Considerações Finais

Neste capítulo foram apresentados os resultados obtidos após os inúmeros testes realizados. Num primeiro momento foram apresentados os resultados obtidos com a execução do ICTM-OpenMP e ICTM-NUMA, que foram aplicações desenvolvidas para rodar em apenas um nó NUMA. Após foi apresentado os resultados da versão ICTM híbrida com MPI + OpenMP. Esta versão apresentou os ganhos esperados quando se utiliza o MPI para paralelizar uma aplicação. E por fim, foram apresentados os resultados obtidos com a versão ICTM híbrido MPI + OpenMP + Libnuma e MAi.

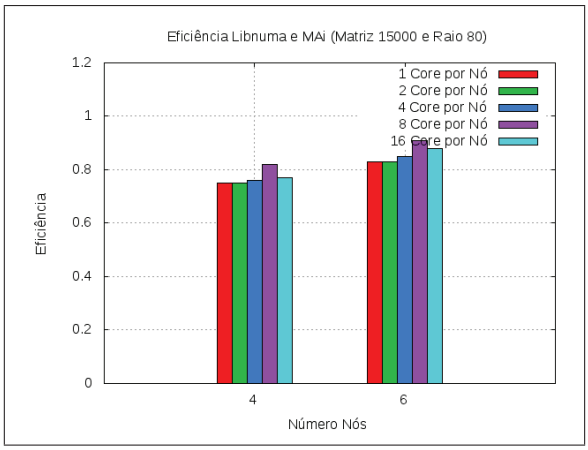

Figura 6.21: Eficiência ICTM Híbrido MPI + OpenMP + NUMA - Matriz 10.000×10.000.

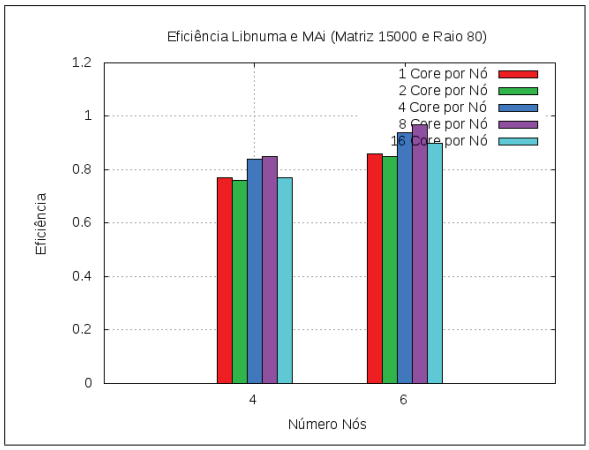

Figura 6.22: Eficiência ICTM Híbrido MPI + OpenMP + NUMA - Matriz 15.000×15.000.

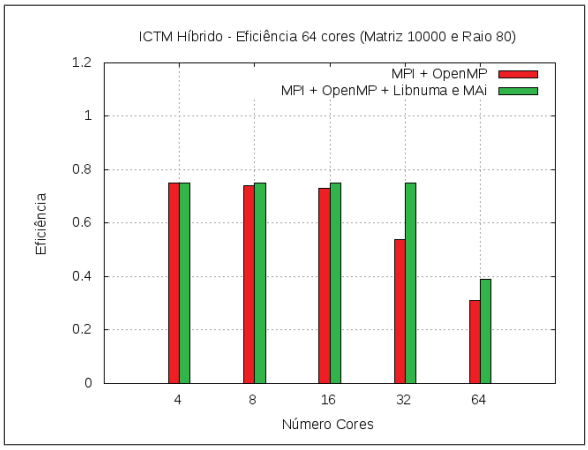

Figura 6.23: Eficiência ICTM Híbrido MPI + OpenMP + NUMA - 64 *cores*.

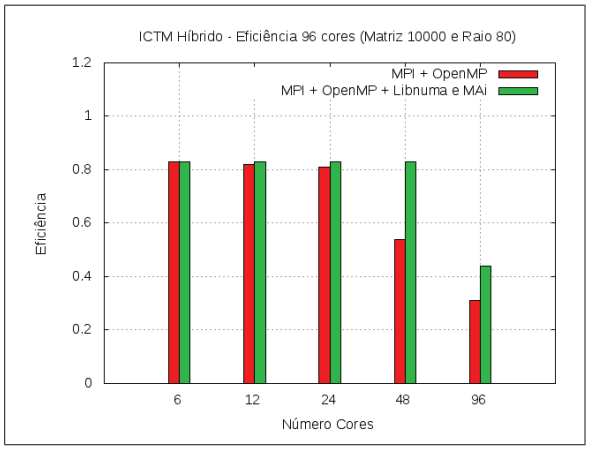

Figura 6.24: Eficiência ICTM Híbrido MPI + OpenMP + NUMA - 96 *cores*.

Esta última, mostrou os ganhos obtidos quando se utiliza os recursos NUMA da arquitetura, ou seja, utiliza a afinidade de memória. No capítulo seguinte serão apresentados comparativos entre as versões e ganhos obtidos com maiores detalhes. Além da possibilidade de escalar a aplicação para matriz de maiores dimensões.

7. CONCLUSÃO E TRABALHOS FUTUROS

Este capítulo tem por objetivo concluir a análise dos resultados obtidos neste trabalho. Num primeiro momento, serão feitos comparativos entre as versões ICTM-OpenMP e ICTM-NUMA para destacar as características NUMA da arquitetura utilizada para os testes. Em seguida, serão apresentados os ganhos obtidos com as versões híbridas. Ainda será apresentada a escalabilidade permitida pelas versões híbridas. E por fim, tem-se o objetivo de destacar-se algumas boas práticas para a programação de aplicações híbridas utilizando MPI + OpenMP + Libnuma e MAi.

7.1 Comparativo dos Resultados

As tabelas 7.1 e 7.2 apresentam um comparativo entre a versão ICTM-OpenMP e ICTM-NUMA considerando uma matriz com a dimensão 10.000×10.000 e 15.000×15.000, respectivamente, destacando o ganho obtido pela versão que utiliza os recursos NUMA. As tabelas também destacam o número de *cores* utilizados.

Os resultados mostram que quanto maior o número de *cores* utilizados maiores são os ganhos da versão ICTM-NUMA sobre a versão ICTM-OpenMP, chegando a ter um ganho de 38.69% comparados os tempos das duas aplicações utilizando 8 *cores*, e tendo o ICTM como parâmetro um raio 20.

Analisando as tabelas, percebe-se também que quanto maior o raio menores são os ganhos da versão ICTM-NUMA. Isso ocorre porque com um raio grande a aplicação tem uma maior demanda por processamento e isso acaba mascarando ganho referente ao tempo de acesso à memória. Por isso, que quando tem-se um raio 20 os ganhos são os maiores, pois o acesso à memória é muito mais frequente.

Tabela 7.1: Comparativo dos Resultados ICTM-OpenMP e ICTM-NUMA - Matriz 10.000

	Raio 20			Raio 40			Raio 80		
Cores	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma
1	75.12	75.23	-0.15%	129.98	130.45	-0.36%	248.6	251.42	-1.12%
2	38.12	37.87	0.64%	65.69	65.52	0.27%	124.97	125.93	-0.76%
4	19.23	18.95	1.47%	33.04	32.8	0.74%	63.46	64.02	-0.87%
8	13.19	9.51	38.69%	19.61	16.68	17.60%	35.73	34.44	3.73%
16	11.56	9.09	27.18%	18.83	16.61	13.35%	33.34	31.24	6.74%

Tabela 7.2: Comparativo dos Resultados ICTM-OpenMP e ICTM-NUMA - Matriz 15.000

	Raio 20			Raio 40			Raio 80		
Cores	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma
1	173.98	169.23	2.81%	301.22	295.93	1.79%	564.97	582.91	-3.08%
2	87.35	85.27	2.43%	151.23	148.68	1.71%	282.77	288.73	-2.06%
4	44.26	42.68	3.71%	76.09	74.76	1.78%	150.48	160.06	-5.98%
8	23.08	21.55	7.10%	44.21	42.17	4.85%	77.43	83.1	-6.82%
16	24.12	21.51	12.11%	40.48	39.46	2.58%	72.72	74.84	-2.84%

As figuras 7.1 e 7.2 são relevantes para demonstrar os ganhos obtidos pelo paralelismo nesta aplicação. Também é possível fazer uma comparação entre gráficamente entre as duas versões que rodam em um único nó.

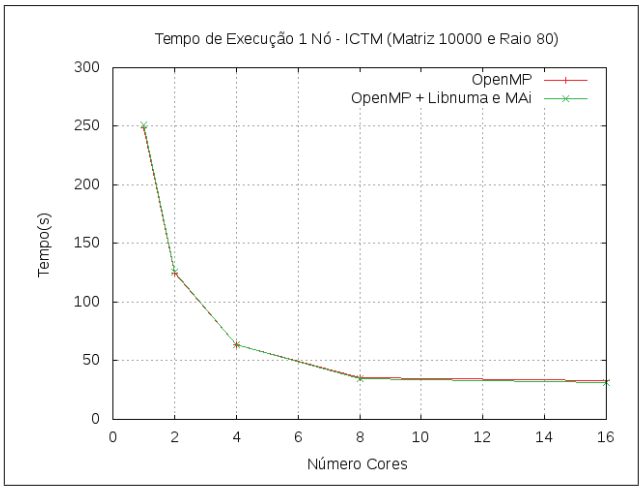

Figura 7.1: Tempos ICTM-OpenMP e ICTM-NUMA - Matriz 10.000 e Raio 80.

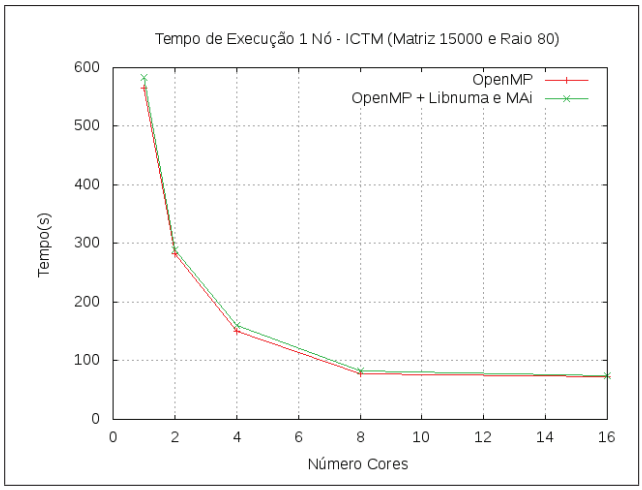

Figura 7.2: Tempos ICTM-OpenMP e ICTM-NUMA - Matriz 15.000 e Raio 80.

As tabelas 7.3, 7.4,7.5, 7.6, 7.7, 7.8 apresentam um comparativo entre as versões híbridas utilizando MPI + OpenMP e MPI + OpenMP + Libnuma e MAi. Nas tabelas a primeira é citada como OpenMP e a segunda como Libnuma. São apresentadas as 6 tabelas sendo 3 referentes a execuções em 4 nós, diferenciando a dimensão das matrizes: 10.000x10.000, 15.000x15.000 e 20.000x20.000. As tabelas destacam o número de *cores* utilizados, em cada nó para os testes.

Os resultados mostram que quanto maior o número de *cores* utilizados maiores são os ganhos da versão híbrida que utiliza os recursos NUMA, chegando a ter um ganho de 80.54% comparados os tempos das duas aplicações utilizando o mesmo número de nós, mesmo número de *cores*, e tendo o ICTM como parâmetro um raio 20.

Analisando as tabelas, percebe-se também que quanto maior o raio menores são os ganhos da versão híbrida NUMA. Isso ocorre porque com um raio grande a aplicação tem uma maior demanda por processamento e isso acaba mascarando o tempo de acesso à memória. Por isso, que quando tem-se um raio 20 os ganhos são os maiores, pois o acesso à memória é muito mais frequente. Essa mesma conclusão pode ser feita com relação a dimensão da Matriz, quanto maior a dimensão da matriz, menores são os ganhos obtidos com a utilização dos recursos NUMA.

Tabela 7.3: Comparativo dos Resultados ICTM Híbrido 4 Nós - Matriz 10.000×10.000

	Raio 20			Raio 40			Raio 80		
Cores	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma
1	25.16	25.09	0.28%	43.57	43.49	0.17%	82.93	83.61	-0.82%
2	12.81	12.58	1.84%	22.03	21.82	0.98%	41.75	41.89	-0.32%
4	6.57	6.32	4.02%	11.2	10.94	2.44%	21.15	20.98	0.80%
8	5.21	3.18	64.03%	8.22	5.49	49.72%	14.47	10.52	37.54%
16	5.45	3.02	80.54%	7.7	5.32	44.80%	12.57	10.14	23.89%

As figuras 7.3 e 7.4 são relevantes para demonstrar os ganhos obtidos pela versão híbrida que utiliza além do MPI + OpenMP, também a Libnuma e MAi. O gráfico considera apenas a execução que ocorreu no *cluster* utilizando 6 nós. Porém destaca o tempo em segundos comparando com o número de *cores* em cada um dos nós.

Tabela 7.4: Comparativo dos Resultados ICTM Híbrido 4 Nós - Matriz 15.000×15.000

	Raio 20			Raio 40			Raio 80		
Cores	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma
1	57.79	56.57	2.15%	100.35	98.57	1.82%	187.99	189.41	-0.75%
2	29.36	28.48	3.10%	50.66	49.46	2.42%	94.42	94.86	-0.46%
4	14.94	14.27	4.72%	25.61	24.76	3.41%	47.6	47.82	-0.46%
8	10.31	7.18	43.50%	16.35	12.45	31.38%	26.7	24.46	9.17%
16	10.43	6.8	53.31%	15.63	12.09	29.23%	26.72	24.35	9.74%

Tabela 7.5: Comparativo dos Resultados ICTM Híbrido 4 Nós - Matriz 20.000×20.000

	Raio 20			Raio 40			Raio 80		
Cores	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma
1	100.31	100.71	-0.39%	174.18	174.99	-0.46%	334.94	335.83	-0.26%
2	51.04	50.65	0.76%	88.01	87.75	0.30%	168.31	168.15	0.09%
4	25.69	25.36	1.33%	44.21	43.98	0.53%	84.6	85.13	-0.62%
8	16.98	12.74	33.29%	27.17	22.13	22.80%	48.34	45.43	6.41%
16	16.95	12.13	39.71%	26.64	21.73	22.57%	46.33	41.64	11.27%

Tabela 7.6: Comparativo dos Resultados ICTM Híbrido 6 Nós - Matriz 10.000×10.000

	Raio 20			Raio 40			Raio 80		
Cores	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma
1	15.11	15.22	-0.73%	26.23	26.24	-0.02%	49.95	50.2	-0.51%
2	7.75	7.66	1.10%	13.32	13.19	0.98%	25.17	25.16	0.04%
4	4.05	3.82	6.04%	6.84	6.61	3.43%	12.79	12.59	1.53%
8	3.67	1.93	90.78%	5.73	3.32	72.83%	9.56	6.31	51.39%
16	4.24	1.83	131.32%	5.58	3.21	73.86%	8.4	5.95	41.22%

Tabela 7.7: Comparativo dos Resultados ICTM Híbrido 6 Nós - Matriz 15.000×15.000

	Raio 20			Raio 40			Raio 80		
Cores	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma
1	34.68	34.06	1.83%	60.38	59.16	2.06%	112.89	113.49	-0.54%
2	17.66	17.15	2.98%	30.53	29.73	2.71%	56.92	56.84	0.15%
4	9.05	8.55	5.92%	15.51	14.89	4.13%	28.7	28.45	0.88%
8	7.3	4.3	69.65%	11.53	7.48	54.17%	18.79	14.29	31.54%
16	7.78	4.1	89.64%	10.77	7.32	47.07%	17.18	13.79	24.57%

Tabela 7.8: Comparativo dos Resultados ICTM Híbrido 6 Nós - Matriz 20.000×20.000

	Raio 20			Raio 40			Raio 80		
Cores	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma	OpenMP	Libnuma	Ganho Libnuma
1	60.25	60.48	-0.39%	104.63	105.03	-0.38%	200.86	201.19	-0.16%
2	30.6	30.47	0.44%	52.83	52.78	0.08%	100.93	100.75	0.18%
4	15.6	15.27	2.18%	26.73	26.4	1.23%	50.97	50.43	1.05%
8	11.86	7.67	54.59%	18.15	13.24	37.06%	32.91	25.92	26.97%
16	12.12	7.29	66.35%	17.75	12.78	38.86%	29.59	24.73	19.62%

Já a figura 7.5 apresenta um comparativo entre *speedups* comparando aquelas situações que apresentaram o menor tempo para a execução da aplicação.

7.2 Escalabilidade ICTM Híbrido

O ICTM é uma aplicação que demanda muita memória, então um grande problema enfrentado pelos trabalhos anteriores foi com relação a dimensão da matriz. Quanto maior a dimensão maior o consumo de memória, até se chegar ao limite físico de memória RAM da máquina.

Com a versão híbrida do ICTM surgiu a oportunidade de escalar a aplicação. Neste trabalho encontrou-se o limite de execução no ambiente, em um unico nó, em uma matriz com dimensões de 15.000x15.000. Afirma-se isso, levando em consideração que a utilização de *swap* prejudica muito o desempenho da aplicação.

A tabela 7.9 apresenta valores obtidos após aumentar a dimensão da matriz e também os números de nós na solução. Sendo assim, chegou-se a processar matrizes com dimensão 45.000x45.000.

Um outro ponto relevante, é que mesmo com esse aumento significativo da matriz, não aumentou

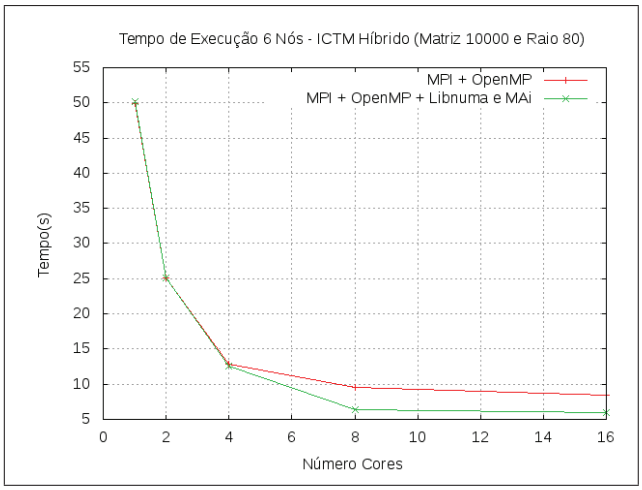

Figura 7.3: Tempos ICTM Híbrido - Matriz 10.000 e Raio 80.

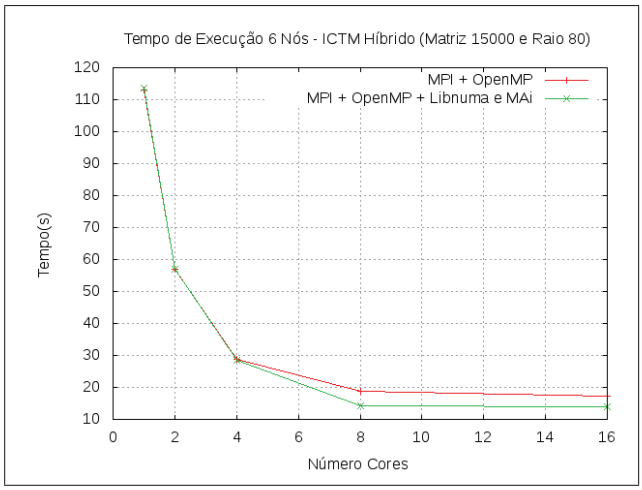

Figura 7.4: Tempos ICTM Híbrido - Matriz 15.000 e Raio 80.

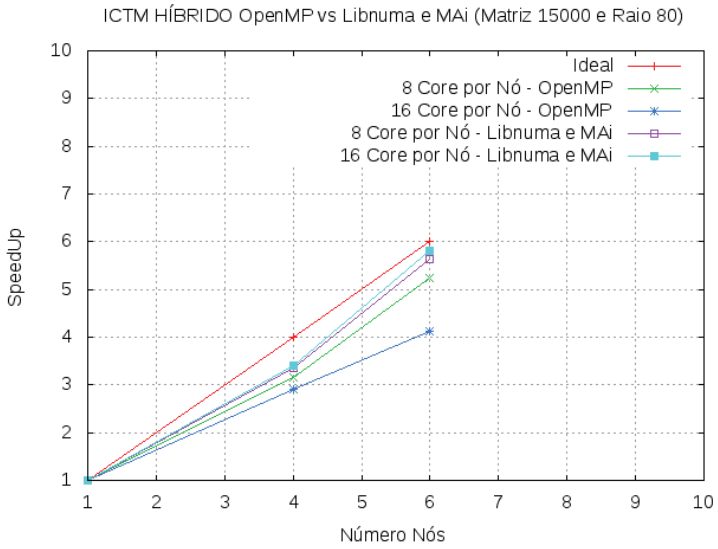

Figura 7.5: *Speed-up* comparativo ICTM Híbrido.

significativamente o tempo necessário para a execução da aplicação. O que leva a conclusão que aumento-se o número de nós pode-se aumentar ainda mais a dimensão da matriz. O que possibilita dizer que o ICTM híbrido é escalável.

Tabela 7.9: Escalabilidade do ICTM Híbrido - Raio 40

	3 Nós Matriz 25000		4 Nós Matriz 30000		5 Nós Matriz 35000		6 Nós Matriz 40000		7 Nós Matriz 45000	
Cores	Media	Desvio	Media	Desvio	Media	Desvio	Media	Desvio	Media	Desvio
4	95.35	0.08	99.78	0.06	101.04	0.04	100.3	0.06	103.69	0.01
8	50.11	0.02	51.24	0.04	52.28	0.05	50.99	0.17	52.27	0.06
16	48.67	0.79	49.79	0.04	52.34	0.01	51.56	0.06	52.3	0.07

7.3 Comparação com os Trabalhos Anteriores

Como descrito anteriormente, já foram desenvolvidos, no Programa de Pos-graduação em Ciência da Computação da PUCRS, dois trabalhos utilizando a aplicação ICTM para estudo de programação de alto desempenho.

O trabalho [26] utilizou apenas o paradigma de troca de mensagens para paralelizar o ICTM, através de MPI. O ambiente de execução utilizado era um *cluster* com 40 máquinas com processadores *Pentium* III e 256 MB memória. Os resultados obtidos por este trabalho foram modestos, o melhor *speed-up* obtido foi de **3.02**, quando utilizados os **40** processadores, tendo a matriz dividida em 39 blocos. Sendo assim, tinha-se uma máquina como o mestre e 39 escravos processando os blocos. Esse pequeno desempenho é explicado pelo grande número de mensagens trocadas entre os processos para o cálculo das bordas. De outro modo, o ICTM Híbrido utilizou a estratégia de processar além do bloco especificado também dois pequenos blocos referentes aos raios, fornecendo então os dados necessários para o processamento total, sem que seja necessário uma troca constante de mensagens dos nós atráves da rede utilizando MPI. A escolha desta estratégia viabilizou a implementação do ICTM Híbrido, que alcançou um **speed-up** de **39.83** ao utilizar **48** *cores*.

O trabalho [7] utilizou OpenMP para paralelizar o ICTM. O ambiente de execução utilizado contava com uma máquina NUMA com 16 processadores e 16 Gb de memória. Este trabalho obteve resultados muito bons chegando a um *speed-up* de 15 quando utilizava a política de alocação cíclica. No entanto, este trabalho tinha problema de escalabilidade. Como era executado em apenas uma máquina os dados do ICTM não podiam ultrapassar os 16 Gb disponíveis. O ICTM Híbrido consegue apresentar escalabilidade para a aplicação, não ficando restrito a apenas uma máquina, além de manter os bons resultados referentes a afinidade de memória que o trabalho apresentou.

7.4 Boas Práticas Programação Híbrida

O levantamento de boas práticas para a programação híbrida não é simples de ser feito, pois não existe uma regra que irá valer para todas as situações. Assim, como as demais soluções para programação de alto desempenho, é necessário analisar o problema, analisar a arquitetura que se tem disponível e tentar mapear o modelo de programação mais adequado para a situação.

No entanto algumas boas práticas podem ser destacadas:

- **Análise da Aplicação**: A aplicação deve ter características que possibilitem sua divisão de forma hierárquica. Tem que ser possível dividir o problema em tarefas grandes, o que pode ser chamado de grão grosso, para ser enviados aos nós, e depois é necessário ser possível paralelizar essa tarefa dentro do nó, que poderia ser chamado de grão fino;

- **Mapeamento da aplicação com a arquitetura utilizada**: O modelo adotado deve ser mapeado diretamente para a arquitetura disponível. Se as redes de interconexão dos nós não possuirem bom desempenho, é possível considerar um sobrecarga no processamento intra nó para evitar o gargalo da rede;

- **Uso extensivo dos recursos de processamento**: No entanto, se o poder de processamento intra nó for modesto, deve-se utilizar o máximo possível de todos os nós mesmo utilizando a rede de interconexão;

- **Adaptar o desenvolvimento para a arquitetura**: Dentro do nó é importante explorar as características específicas da arquitetura, principalmente se for uma arquitetura NUMA.

Portanto, é importante reforçar que não existe um passo-a-passo a ser seguido para se obter bons ganhos em programação híbrido. Contudo, as possibilidades de mapear o problema para a arquitetura e se obter bons resultados são muito grandes.

7.5 Trabalhos Futuros

A programação híbrida é um modelo de programação que cada vez mais terá aplicabilidade nas arquiteturas de *clusters* com nós multiprocessados. No entanto, recursos NUMA que até pouco tempo existiam apenas em computadores com alto custo, agora estão disponíveis em computadores com custo bastante acessíveis, quando se fala de computação de alto desempenho. Inclusive, máquinas com arquiteturas NUMA estão sendo vendidos desde computadores para usuários domésticos até para computadores para empresas com fins comerciais.

Com esta situação, acredita-se que as novas gerações de solução para alto desempenho serão arquiteturas com recursos NUMA e caracteristicos para um mapeamentos para aplicações híbridas.

Então trabalhos futuros serão necessários para avaliar o desempenho de aplicações com características diferentes das características do ICTM. Pois, o ICTM é uma aplicação regular que possui uma grande demanda por memória, no entanto será necessário testar aplicações irregulares e/ou que tenham características de processamento diferentes do ICTM.

Referências

[1] ACPI. Advanced Configuration and Power Interface Specificication. Revision 3.0, September 2, 2004. Extracted from http://www.acpi.info/DOWNLOADS/ACPIspec30.pdf, 16 de Janeiro de 2010.

[2] Aguiar, M. S. The multi-layered interval categorizer tesselation-based model. In *GeoInfo'04: Proceedings of the 6th Brazilian Symposium on Geoinformatics* (Campos do Jordão, São Paulo, Brazil, 2004), Instituto Nacional de Pesquisas Espaciais, pp. 437–454.

[3] Ashworth, M., Bush, I. J., Guest, M. F., Sunderland, A. G., Booth, S., Hein, J., Smith, L., Stratford, K., and Curioni, A. HPCx: towards capability computing: Research articles. *Concurrency and Computation: Practice and Experience 17* (August 2005), 1329–1361.

[4] Aversa, R., Di Martino, B., Rak, M., Venticinque, S., and Villano, U. Performance prediction through simulation of a hybrid MPI/OpenMP application. *Parallel Comput. 31* (October 2005), 1013–1033.

[5] Bolosky, W. J., Scott, M. L., Fitzgerald, R. P., Fowler, R. J., and Cox, A. L. NUMA policies and their relation to memory architecture. *SIGPLAN Not. 26* (April 1991), 212–221.

[6] Cappello, F., and Etiemble, D. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks, 2000.

[7] Castro, M. NUMA-ICTM: Uma versão paralela do ICTM explorando estratégias de alocação de memória para máquinas NUMA. Dissertação de mestrado, Pontifícia Universidade do Rio Grande do Sul. Porto Alegre, Brasil, 2009.

[8] Castro, M., Fernandes, L. G., Pousa, C., Mehaut, J.-F., and de Aguiar, M. S. NUMA-ICTM: A parallel version of ICTM exploiting memory placement strategies for NUMA machines. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 1–8.

[9] Chorley, M. J., Walker, D. W., and Guest, M. F. Hybrid message-passing and shared-memory programming in a molecular dynamics application on multicore clusters. *International Journal of High Performance Computing Applications 23* (August 2009), 196–211.

[10] Corrêa, M., Zorzo, A., and Scheer, R. Operating system multilevel load balancing. In *Proceedings of the 2006 ACM Symposium on Applied Computing* (New York, NY, USA, 2006), ACM, pp. 1467–1471.

[11] de Oliveira, R. S., Carissimi, A. S., and Toscani, S. S. *Sistemas Operacionais.* Série Didática da UFRGS. Sagra-Luzzato. Vol. 11, 2004.

[12] Forum, M. MPI: A Message-Passing Interface Standard Version 2.1. Extracted from www.mpi-forum.org/docs/mpi21-report.pdf, 16 de Janeiro de 2010.

[13] Henty, D. S. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)* (Washington, DC, USA, 2000), Supercomputing '00, IEEE Computer Society.

[14] Intel. An Introduction to the Intel QuickPath Interconnect. Extracted from http://www.intel.com/technology/quickpath/introduction.pdf, 16 de Janeiro de 2010.

[15] Intel. Cluster OpenMP for Intel Compilers. Extracted from http://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers/, 16 de Janeiro de 2010.

[16] Jost, G., Jin, H., Mey, D. A., and Hatay, F. F. Comparing the OpenMP, MPI, and hybrid programming paradigm on an SMP cluster, 2003.

[17] Kleen, A. A NUMA API for LINUX. Extracted from www.halobates.de/numaapi3.pdf, 16 de Janeiro de 2010.

[18] Lameter, C. Local and remote remory: Memory in Linux/NUMA System. Extracted from http://kernel.org/pub/linux/kernel/people/christoph/pmig/numa memory.pdf, 16 de Janeiro de 2010.

[19] Lusk, E., and Chan, A. Early experiments with the OpenMP/MPI hybrid programming model. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism* (Berlin, Heidelberg, 2008), IWOMP 2008, pp. 36–47.

[20] OpenMP. OpenMP Application Program Interface Version 3.0 Complete Specifications. Extracted from http://www.openmp.org/mp-documents/specs30.pdf, 16 de Janeiro de 2010.

[21] Pousa Ribeiro, C., and Méhaut, J.-F. MAi: Memory Affinity Interface. Relatório de Pesquisa RT-0359, INRIA, 2010.

[22] Rabenseifner, R., Hager, G., and Jost, G. Hybrid MPI and OpenMP parallel programming. Extracted from https://fs.hlrs.de/projects/rabenseifner/publ/sc10_hybrid_final_corrected.pdf, 13–19 de Novembro de 2010.

[23] Rabenseifner, R., Hager, G., and Jost, G. Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proceedings of the Cray Users Group Conference 2009* (Atlanta, GA, USA, 2009), CUG 2009.

[24] Rabenseifner, R., Hager, G., and Jost, G. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (Washington, DC, USA, 2009), pp. 427–436.

[25] Ribeiro, C. P., Mehaut, J.-F., Carissimi, A., Castro, M., and Fernandes, L. G. Memory affinity for hierarchical shared memory multiprocessors. In *Proceedings of the 2009 21st International Symposium on Computer Architecture and High Performance Computing* (Washington, DC, USA, 2009), SBAC-PAD '09, pp. 59–66.

[26] Silva, R. K. S. HPC-ICTM: um modelo de alto desempenho para categorização de Áreas geográficas. Dissertação de mestrado, Pontifícia Universidade do Rio Grande do Sul. Porto Alegre, Brasil, 2006.

[27] Smith, L., and Bull, M. Development of mixed mode MPI/OpenMP applications. *Sci. Program. 9* (August 2001), 83–98.

[28] Torrellas, J., Lam, M. S., and Hennessy, J. L. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers Vol. 43* (1994), 651–663.

[29] Vahalia, U. *UNIX Internals: The New Frontiers.* Prentice Hall, 1995.