Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

NUMA-ICTM: Uma Versão Paralela do ICTM Explorando Estratégias de Alocação de Memória para Máquinas NUMA

Márcio Bastos Castro

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes
Co-orientador: Prof. Dr. Marilton Sanchotene de Aguiar

Porto Alegre
2009

Dados Internacionais de Catalogação na Publicação (CIP)

C355n Castro, Márcio Bastos
NUMA-ICTM : uma versão paralela do ICTM explorando estratégias de alocação de memória para máquinas NUMA / Márcio Bastos Castro.
Porto Alegre, 2009.
81 f.

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

1. Informática. 2. Processamento de Alto Desempenho. 3. Arquitetura de Computador.
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 "**NUMA-ICTM: Uma Versão Paralela do ICTM Explorando Estratégias de Alocação de Memória para Máquinas NUMA**", apresentada por Márcio Bastos Castro, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 16/01/09 pela Comissão Examinadora:

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

Prof. Dr. Marilton Sanchotene de Aquiar – Co-orientador — UFPel

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

Prof. Dr. Adenauer Corrêa Yamin – — UCPel

Homologada em 03/03/09, conforme Ata No. 003/09. pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
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

Dedico este trabalho aos meus pais e avós, pelo incentivo e apoio em todos os momentos de minha vida.

Agradecimentos

À minha família, pelo carinho, educação e apoio em todos os momentos e decisões que tomei. Especialmente aos meus pais, Teresinha e Antônio, por terem me dado todo o suporte necessário, permitindo-me assim, conquistar mais este objetivo em minha vida. Sem este suporte certamente eu não teria chegado até aqui. Espero ter retribuído, com mais este título, todo o investimento recebido em educação. Igualmente agradeço aos meus avós Aida e Francisco, os quais sempre me apoiaram e torceram por mim, que Deus esteja sempre com vocês!

Aos colegas do Grupo de Modelagem de Aplicações Paralelas (GMAP), em especial ao Thiago Nunes e Mateus Raeder, os quais considero grandes amigos, pela força, apoio e troca de conhecimento desde a época de Graduação. Aos demais colegas que conheci na Graduação com os quais tive o prazer de compartilhar alguns momentos durante o Mestrado. Também agradeço aos colegas que conheci durante o Mestrado, em especial ao Márcio Dorn, um cara muito inteligente e dedicado, com o qual aprendi bastante durante estes dois anos.

Aos professores da PUCRS, em especial ao Prof. Dr. Luiz Gustavo Leão Fernandes (Gãs), pela orientação neste trabalho e conhecimento transmitido ao longo dos anos. Em muitos momentos a relação orientador-orientando foi deixada de lado, transparecendo assim uma grande amizade, o que acredito ter sito muito importante para que juntos pudéssemos crescer.

Aos amigos e amigas de Rio Grande, com os quais dividi momentos inesquecíveis como churrascos, jantas, festas e praias. Um agradecimento especial aos amigos do "QG": obrigado pela amizade, pelos conselhos e por todo o apoio. Espero que a nossa amizade continue sempre forte!

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pela Bolsa Integral de Mestrado, no pagamento de mensalidades para minha manutenção e custeio das taxas escolares.

Resumo

Na Geofísica, a subdivisão apropriada de uma região em segmentos é extremamente importante. O ICTM (*Interval Categorizer Tesselation Model*) é uma aplicação capaz de categorizar regiões geográficas utilizando informações extraídas de imagens de satélite. O processo de categorização de grandes regiões é considerado um problema computacionalmente intensivo, o que justifica a proposta e desenvolvimento de soluções paralelas com intuito de aumentar sua aplicabilidade. Recentes avanços em arquiteturas multiprocessadas caminham em direção a arquiteturas do tipo NUMA (*Non-uniform Memory Access)*, as quais combinam a eficiência e escalabilidade das máquinas MPP (*Massively Parallel Processing*) com a facilidade de programação das máquinas SMP (*Symmetric Multiprocessors*). Neste trabalho, é apresentada a NUMA-ICTM: uma solução paralela do ICTM para máquinas NUMA explorando estratégias de alocação de memória. Primeiramente, o ICTM é paralelizado utilizando-se somente OpenMP. Posteriormente, esta solução é otimizada utilizando-se a interfade MAI (*Memory Affinity Interface*), a qual proporciona um melhor controle sobre a alocação de dados em memória em máquinas NUMA. Os resultados mostram que esta otimização permite importantes ganhos de desempenho sobre a solução paralela que utiliza somente OpenMP.

Palavras-chave: ICTM, Computação de Alto Desempenho, NUMA, Libnuma, OpenMP, MAI.

Abstract

In Geophysics, the appropriate subdivision of a region into segments is extremely important. ICTM (Interval Categorizer Tesselation Model) is an application that categorizes geographic regions using information extracted from satellite images. The categorization of large regions is a computational intensive problem, what justifies the proposal and development of parallel solutions in order to improve its applicability. Recent advances in multiprocessor architectures lead to the emergence of NUMA (Non-Uniform Memory Access) machines, which combine the efficiency and scalability of MPP (Massively Parallel Processing) machines with the programming facility of the SMP (Symmetric Multiprocessors) machines. In this work, NUMA-ICTM is presented: a parallel solution of ICTM for NUMA machines exploiting memory placement strategies. First, ICTM is parallelized using only OpenMP. After, the OpenMP solution is improved using the MAI (Memory Affinity Interface) library, which allows a control of memory allocation in NUMA machines. The results show that the optimization of memory allocation leads to significant performance gains over the pure OpenMP parallel solution.

Keywords: ICTM, High Performance Computing, NUMA, Libnuma, OpenMP, MAI.

Lista de Figuras

Figura 1 Conceito geral do ICTM multi-camada. 20
Figura 2 Extração de dados de uma das camada das imagens obtidas do satélite. . 21
Figura 3 Processo de categorização de uma camada *c* qualquer do modelo. . . . 22
Figura 4 Implementação da matriz de estados do modelo teórico em quatro matrizes. 23
Figura 5 Esquema genérico de um multiprocessador UMA. 25
Figura 6 Esquema genérico de um multiprocessador NUMA. 25
Figura 7 Exemplo do problema de *false sharing*. 27
Figura 8 Modelo *fork-join* de execução da API OpenMP. 31
Figura 9 Arquitetura Opteron com 8 processadores *dual core*. 34
Figura 10 Arquitetura Itanium 2 com 16 processadores. 34
Figura 11 Exemplo de uma máscara de *bits*. 37
Figura 12 Política *first-touch*. 38
Figura 13 Estrutura geral de uma aplicação paralela que utiliza *threads* em conjunto com a interface MAI. 41
Figura 14 Arquivo de configuração da interface MAI. 42
Figura 15 Leitura e escrita das matrizes durante o processo de categorização do ICTM. 46
Figura 16 Divisão do trabalho entre as *threads*. 47
Figura 17 Desempenho da solução OpenMP-ICTM na arquitetura Opteron. 51
Figura 18 Desempenho da solução OpenMP-ICTM na arquitetura Itanium 2. . . . 52
Figura 19 Influência do caso de estudo no desempenho do OpenMP-ICTM (arquitetura Opteron). 53
Figura 20 Influência do caso de estudo no desempenho do OpenMP-ICTM (arquitetura Itanium 2). 54
Figura 21 A política *bind_block* aplicada ao OpenMP-ICTM. 57
Figura 22 A política *cyclic* aplicada ao OpenMP-ICTM. 57
Figura 23 Desempenho das políticas *bind_block* e *bind_all* - Opteron. 59
Figura 24 Desempenho das políticas *cyclic* e *cyclic_block* - Opteron. 60
Figura 25 Desempenho das políticas *bind_block* e *bind_all* - Itanium 2. 61
Figura 26 Desempenho das políticas *cyclic* e *cyclic_block* - Itanium 2. 62
Figura 27 Eficiência média da política *cyclic*. 63
Figura 28 Eficiência média da política *bind_block*. 64
Figura 29 Proposta de integração do NUMA-ICTM com o HPC-ICTM (*cluster* de máquinas NUMA). 70

Lista de Tabelas

Tabela 1 Casos de estudo utilizados para avaliar o desempenho das soluções propostas neste trabalho. 49
Tabela 2 Ganho de desempenho da solução NUMA-ICTM em relação à solução OpenMP-ICTM (média dos *speed-ups* de 2 a 16 processadores). 68
Tabela 3 Ganho de desempenho da solução NUMA-ICTM em relação à solução OpenMP-ICTM (média dos *speed-ups* de 8 a 16 processadores). 68
Tabela 4 Ganho de desempenho da solução NUMA-ICTM em relação a solução OpenMP-ICTM (média dos *speed-ups* com somente 16 processadores). 69

Lista de Siglas

ICTM	Interval Categorizer Tesselation Model	14
UMA	Uniform Memory Access	14
NUMA	Non-Uniform Memory Access	14
MPP	Massively Parallel Processing	14
SMP	Symmetric Multiprocessing	14
MPI	Message Passing Interface	15
INRIA	Institut National de Recherche en Informatique et en Automatique	17
GMAP	Grupo de Modelagem de Aplicações Paralelas	17
MESCAL	Middleware Efficiently SCALable	17
API	Application Program Interface	17
OpenMP	Open Multiprocessing	17
GMFC	Grupo de Pesquisa de Matemática e Fundamentos da Computação	19
ACI	Autômatos Celulares Intervalares	19
CT-PETRO	Fundo Setorial do Petróleo e Gás Natural	19
FMC2	Fundamentos Matemáticos da Computação: Modelos e Aplicações de Computações Intervalares	19
CT-INFO	Fundo Setorial para Tecnologia da Informação	19
SIG	Sistemas de Informações Geográficas	19
CPU	Central Processing Unit	24
API	Application Programming Interface	29
Pthreads	POSIX Threads	29
SMI	Shared Memory Interface	32
SCI	Scalable Coherent Interface	32
AMD	Advanced Micro Devices	33
GHz	Gigahertz	33
MB	Megabyte	33
GB	Gigabyte	33

KB	Kilobyte	33
GCC	GNU Compiler Collection	33
FSS	FAME Scalability Switch	34
ICC	Intel C Compiler	34
SUSE	Software und System Entwicklung	36
LIG	Laboratoire d'Informatique de Grenoble	40
RAM	Random Access Memory	42

Sumário

1 Introdução 14
1.1 Trabalhos relacionados 15
1.2 Motivação 16
1.3 Objetivos 17
1.4 Estrutura do volume 18

2 O modelo ICTM 19
2.1 Contexto e aplicabilidade do ICTM 19
2.2 Visão geral 20
2.3 Processo de categorização 21

3 Contexto NUMA 24
3.1 Conceitos básicos 24
3.1.1 Localidade de dados 26
3.1.2 Escalonamento de processos 28
3.2 Bibliotecas para multiprocessadores 29
3.2.1 Threads 29
3.2.2 OpenMP 30
3.2.3 Demais bibliotecas 32
3.3 Máquinas NUMA utilizadas neste trabalho 33
3.4 Discussão 35

4 Afinidade de threads e memória 36
4.1 NUMA API 36
4.1.1 Afinidade de threads 37
4.1.2 Afinidade de memória 38
4.2 Memory Interface Library (MAI) 39
4.2.1 Visão geral 40
4.2.2 Principais funções 42
4.3 Discussão 44

5 OpenMP-ICTM 45
5.1 Distribuição do trabalho entre threads 45
5.2 Avaliação de desempenho 49
5.2.1 Casos de estudo 49
5.2.2 Resultados 50
5.3 Discussão 52

6 NUMA-ICTM 55
6.1 Integração com a MAI 55
6.2 Avaliação de desempenho 58
6.2.1 Arquitetura Opteron 58
6.2.2 Arquitetura Itanium 2 61
6.3 Discussão 63

7 Conclusão 65
7.1 NUMA-ICTM x HPC-ICTM 65
7.2 NUMA-ICTM x OpenMP-ICTM 66
7.3 Avaliação da interface MAI 69
7.4 Trabalhos futuros 70

Referências 72

APÊNDICE A – Médias e desvio padrão na arquitetura Opteron 76

APÊNDICE B – Médias e desvio padrão na arquitetura Itanium 2 . . . 79

1 Introdução

A Geofísica é uma ciência voltada à compreensão da estrutura, composição e dinâmica do planeta Terra, sob a ótica da Física. Consiste basicamente na aplicação de conhecimentos da Física ao estudo da Terra. Neste contexto, sob o ponto de vista dos pesquisadores desta área, a subdivisão apropriada de uma área geográfica em segmentos é extremamente importante, visto que é possível extrapolar os resultados obtidos em determinadas partes de um segmento específico já estudadas anteriormente para outras partes deste mesmo segmento ainda não analisadas [1]. Isto permite um bom entendimento de um segmento como um todo, sem a necessidade de analisá-lo completamente.

O ICTM (*Interval Categorizer Tessellation Model*) [2] é um modelo multi-camada desenvolvido para categorização de regiões geográficas utilizando informações extraídas de imagens de satélite. Porém, a categorização de grandes regiões requer um alto poder computacional. Além disso, o processo de categorização como um todo resulta em um uso intensivo de memória. Conseqüentemente, estas duas características principais motivam o desenvolvimento de uma solução paralela para esta aplicação com intuito de categorizar grandes regiões de forma mais rápida.

Em arquiteturas UMA (*Uniform Memory Access*) tradicionais, o computador possui somente um controlador de memória, o qual é compartilhado por todos os processadores. Esta única conexão com a memória muitas vezes se torna o gargalo do sistema quando muitos processadores acessam a memória ao mesmo tempo. Este problema se mostra ainda pior em arquiteturas com um grande número de processadores, nas quais um único controlador de memória não é satisfatoriamente escalável. Portanto, estes tipos de arquiteturas podem não suprir os requisitos básicos para determinados tipos de aplicações. O ICTM é um exemplo deste tipo de aplicação, onde resultados melhores poderão ser obtidos em arquiteturas que ao mesmo tempo ofereçam uma memória com alta capacidade de armazenamento e reduzam o problema da disputa da mesma entre os processadores.

Em arquiteturas NUMA (*Non-Uniform Memory Access*) o sistema é dividido em múltiplos nodos [3]. Esta divisão, tem como principal objetivo aumentar a escalabilidade de arquiteturas UMA. Este tipo de arquitetura se caracteriza pelo uso de hierarquias de memória as quais são vistas pelo desenvolvedor como sendo uma única memória global. Arquiteturas do tipo NUMA combinam a eficiência e escalabilidade das arquiteturas MPP (*Massively Parallel Processing*) com a facilidade de programação das arquiteturas SMP (*Symmetric Multiprocessors*) [4]. Porém, devido ao fato de a memória ser dividida em blocos, o tempo necessário para realizar um acesso a ela é condicionado pela "distância" entre o processador (o qual acessa a memória) e o

bloco de memória (no qual o dado a ser acessado está fisicamente alocado).

Para que seja possível utilizar ao máximo os recursos oferecidos pelas arquiteturas NUMA é importante que sejam considerados alguns fatores. O primeiro deles é a questão referente aos diferentes tempos de acesso à memória. Uma utilização incorreta da memória fará com que o desenvolvedor não obtenha os benefícios oferecidos por ela. Outro fator importante é arquitetura alvo, para a qual será paralelizada a aplicação. É necessário que o desenvolvedor tenha conhecimento da forma com que ela está organizada para que então seja possível realizar um bom uso da mesma. Por fim, além do conhecimento geral da aplicação a ser paralelizada é importante também conhecer os mecanismos e estratégias utilizadas pela aplicação ao acessar os dados armazenados na memória.

1.1 Trabalhos relacionados

No trabalho realizado por Silva et al. [5], os autores apresentaram uma proposta de paralelização do ICTM para *clusters* utilizando-se o padrão MPI (*Message Passing Interface*) [6]. Neste trabalho, os autores exploraram três possibilidades de decomposição do problema, são elas:

- Camadas: cada processo paralelo calcula uma determinada camada do modelo;
- Funções: cada processo paralelo calcula uma determinada fase do modelo;
- Domínios: cada processo paralelo calcula uma parte da região que será analisada.

Estas três possibilidades de decomposição do problema são bastantes distintas e algumas delas exigem grande comunicação entre os processos. Tendo em vista que esta solução foi proposta para *clusters*, os autores optaram pela **decomposição em camadas**, pois esta mostrou-se uma forma simples e direta de paralelizar o problema. Além disso, como cada camada pode ser processada de forma individual, não haverá grandes necessidades de trocas de mensagens entre os processos.

Na prática, a decomposição em camadas foi implementada utilizando-se o modelo mestre-escravo. O processo mestre é responsável por ler o arquivo de dados das camadas, criar nc tarefas (o número total de tarefas é igual ao número de camadas - nc), realizar a distribuição inicial das tarefas e aguardar os resultados. Os processos escravos somente são responsáveis por processar as tarefas (categorizar camadas) e informar o processo mestre quando o processamento for finalizado. À medida em que os resultados forem sendo enviados pelos escravos, caso hajam mais tarefas a serem processadas, o mestre enviará novamente mais trabalho para os escravos ociosos.

Os resultados obtidos com a aplicação deste modelo apresentam interessantes ganhos de desempenho. Esta conclusão foi obtida através da análise dos resultados obtidos em dois dife-

rentes *clusters*. Porém, considerando o fato de que cada processo escravo irá calcular uma dada camada do modelo, o tamanho máximo desta camada (em termos de espaço físico a ser armazenado em memória) está limitado pela quantidade de memória disponível no nodo em que este processo estará sendo executado. Como conseqüência disto, grandes regiões não poderão ser categorizadas utilizando este método de decomposição do problema, visto que é pouco comum dispor-se de um *cluster* onde cada nodo possua uma memória principal da ordem de dezenas de GB.

Por outro lado, em um outro trabalho realizado por Silva et al. [7], os autores propuseram uma extensão do trabalho anteriormente citado para o ambiente de grades computacionais. No caso de grades computacionais, existem diversos ambientes que oferecem serviços e controle de tarefas a serem distribuídas na grade. Um destes ambientes é o *OurGrid* [8–10], que foi utilizado no referido trabalho.

Os resultados deste trabalho mostraram o desempenho da solução para grades computacionais ao utilizar as três formas de composição do problema: em camadas, domínios e funções. A solução para grades permitiu com que regiões maiores pudessem ser processadas, mostrando também ganhos interessantes de desempenho.

Além das três formas de decomposição do problema, duas abordagens relacionadas a localização dos dados foram propostas. A primeira utilizou dados geográficos centralizados, enquanto a segunda utilizou dados geográficos distribuídos. A segunda solução é mais apropriada para grades computacionais, visto que através da distribuição dos dados há uma redução drástica da comunicação entre nodos da grade. Todavia, para que esta solução mostre um ganho de desempenho considerável, os dados precisam estar previamente armazenados nos nodos da grade.

1.2 Motivação

As motivações para realização deste trabalho surgiram da análise das características do ICTM e suas propostas de paralelização anteriormente citadas, da identificação da importância do ICTM no contexto da Geofísica e da possibilidade de avaliar os benefícios ao utilizar-se a MAI (*Memory Affinity Interface*) [11]: uma biblioteca específica para programação para máquinas NUMA.

Os trabalhos descritos na Seção 1.1 popuseram diferentes formas de paralelizar o ICTM e mostraram interessantes ganhos de desempenho. Entretanto, estas propostas possuem limitações, especialmente quanto ao fato de utilizar regiões extremamente grandes a serem categorizadas. Tendo em vista o exposto e considerando as características do ICTM, é bastante provável que esta aplicação possa ser melhor adaptada para arquiteturas com memória compartilhada. Através do uso de máquinas NUMA é possível utilizar não somente os benefícios oferecidos por uma arquitetura com memória compartilhada, mas também a possibilidade de

desenvolver uma solução paralela que apresente bom desempenho com a utilização de muitos processadores. O uso de uma memória compartilhada permite o desenvolvimento de outras formas de paralelização do problema, pois o custo de comunicação neste tipo de arquitetura é bastante inferior em comparação com os obtidos em *clusters*.

O ICTM é uma aplicação que auxilia no estudo de regiões geográficas e pode ser aplicado em diversas áreas da Geofísica. Porém, a solução seqüencial limita-se a somente análises de regiões relativamente pequenas, o que muitas vezes não é interessante no ponto de vista de pesquisadores da área. Arquiteturas de alto desempenho aparecem como uma forma de reduzir esta limitação, possibilitando a categorização de grandes áreas geográficas.

Uma das arquiteturas de alto desempenho que está se tornando difundida nos últimos anos é representada pelas máquinas do tipo NUMA. Porém, por introduzirem o conceito de "distância" através dos diferentes tempos de acesso à memória, são arquiteturas mais complexas em comparação com arquiteturas do tipo UMA. Para tratar com esta maior complexidade, o INRIA (*Institut National de Recherche en Informatique et en Automatique*) em Grenoble (França), está desenvolvendo uma interface inovadora denominada MAI [11]. A utilização desta interface na solução paralela do ICTM proposta neste trabalho servirá também para avaliar o ganho de desempenho e os benefícios ao utilizá-la, servindo como um caso de estudo onde a interface foi aplicada à paralelização de uma aplicação real. Isto somente foi possível devido a uma cooperação entre o GMAP (Grupo de Modelagem de Aplicações Paralelas), o qual o autor desta dissertação pertence, e o grupo MESCAL (*Middleware Efficiently SCALable*), localizado na Universidade de Grenoble (França).

1.3 Objetivos

O objetivo principal deste trabalho é apresentar uma solução paralela do ICTM para máquinas NUMA explorando estratégias de alocação de memória. Primeiramente, será apresentada a paralelização do ICTM através do uso da API (*Application Program Interface*) OpenMP (*Open Multiprocessing*) [6], denominada OpenMP-ICTM). OpenMP provê um modelo portável e escalável para desenvolvedores de aplicações paralelas para arquiteturas com memória compartilhada.

Porém, esta API não foi originalmente desenvolvida para arquiteturas com acesso não-uniforme à memória. Tendo isto em vista, através da utilização da MAI (*Memory Affinity Interface*), diferentes políticas de memória serão aplicadas na solução paralela proposta neste trabalho com o intuito de otimizar a utilização das arquiteturas NUMA (solução denominada NUMA-ICTM). A avaliação dos benefícios ao utilizar-se a interface MAI servirá como um objetivo secundário deste trabalho.

Para a avaliação de desempenho da solução proposta serão utilizadas duas arquiteturas NUMA com características bastante distintas. Com isto, será possível não somente analisar-

se o impacto das arquiteturas no desempenho da solução paralela, mas também como será seu comportamento sob o uso de diferentes políticas de memória.

1.4 Estrutura do volume

Este trabalho possui a seguinte estrutura:

- **Capítulo 2:** apresenta uma visão geral do ICTM contendo seu contexto e aplicabilidade, o processo de categorização e detalhes de implementação;
- **Capítulo 3:** contextualiza arquiteturas NUMA, mostrando alguns conceitos básicos importantes, bibliotecas para implementação de paralelismo neste tipo de arquitetura e descreve as duas máquinas NUMA que serão utilizadas para avaliar o desempenho da solução proposta neste trabalho;
- **Capítulo 4:** apresenta o conceito de afinidade e como ela pode ser atingida através do uso de uma API de mais baixo nível (NUMA API) [3]. Além disso, a interface MAI é apresentada, surgindo como uma alternativa de mais alto nível para atingir a afinidade de memória e *threads*;
- **Capítulo 5:** descreve a primeira solução paralela do ICTM denominada OpenMP-ICTM. Nesta solução, somente a API OpenMP é utilizada e nenhuma atenção é dada à alocação de dados em memória. Posteriormente, uma análise de desempenho desta solução é feita considerando-se as duas arquiteturas alvo. Com isto, mostra-se a importância de utilizar afinidade de *threads* e memória com intuito de utilizar ao máximo as arquiteturas alvo.
- **Capítulo 6:** apresenta a solução NUMA-ICTM, a qual é resultante da otimização da solução OpenMP-ICTM com afinidade de *threads* e memória. Primeiramente, é mostrado de que forma a interface MAI pode ser aplicada à solução OpenMP-ICTM através de quatro políticas de memória que ela implementa. Após, é apresentada uma análise de desempenho desta nova solução nas duas arquiteturas alvo.
- **Capítulo 7:** apresenta as considerações finais deste trabalho. Primeiramente, uma comparação entre a solução proposta e os trabalhos relacionados é feita. Após, uma comparação final entre a solução OpenMP-ICTM e NUMA-ICTM é apresentada, discutindo-se o ganho geral obtido com a solução otimizada. A seguir, uma avaliação da interface MAI é feita, discutindo-se os benefícios alcançados ao utilizá-la. Finalmente, uma proposta de trabalho futuro é descrita, onde propõe-se a união da solução apresentada nos trabalhos relacionados com a solução proposta neste trabalho.

2 O modelo ICTM

A proposta inicial de um modelo para categorização de áreas geográficas foi apresentado em [1]. Esta proposta deu origem a diversos outros trabalhos. Estes trabalhos foram idealizados com o intuito de generalizar cada vez mais o modelo inicial. Os projetos criados pelo GMFC (Grupo de Pesquisa de Matemática e Fundamentos da Computação) denominados ACI (Autômatos Celulares Intervalares) com Aplicações em Topografia, no contexto do fundo setorial CT-PETRO (Fundo Setorial do Petróleo e Gás Natural), e FMC2 (Fundamentos Matemáticos da Computação: Modelos e Aplicações de Computações Intervalares), no contexto do fundo setorial CT-INFO (Fundo Setorial para Tecnologia da Informação), possibilitaram, entre outros resultados, a criação de uma versão seqüencial para a categorização de áreas geográficas denominada ICTM (*Interval Categorizer Tesselation Model*).

Neste capítulo é apresentado o funcionamento geral do ICTM. O processo de categorização utilizado pelo ICTM é composto por diversas etapas e envolve a manipulação e o processamentos de diversas matrizes de dados. O entendimento do processo de categorização servirá como base para, posteriormente, definir a estratégia de paralelização para máquinas NUMA. Além disso, são apresentadas as etapas mais custosas do processo de categorização e a influência de alguns parâmetros no desempenho da versão seqüencial.

2.1 Contexto e aplicabilidade do ICTM

O emprego da computação no processamento de dados geográficos vem crescendo ao longo dos últimos anos. No início da década de 80, os primeiros SIGs (Sistemas de Informações Geográficas) começaram a ser oferecidos comercialmente. Em poucas palavras, um SIG é um conjunto de aplicativos, *hardwares*, procedimentos de entrada e saída de dados, entre outros. O objetivo principal é fornecer funções de coleta, tratamento e apresentação de informações [12].

Os SIGs podem ser aplicados em diferentes áreas e por isso são muito importantes. Na logística, podem ser aplicados para escolha da melhor rota a ser seguida por caminhões para a distribuição de produtos de uma empresa. Na agricultura, através do suporte para plantação e colheita extraído através de previsões climáticas. Estes são alguns dos inúmeros exemplos de aplicação dos SIGs.

A relação entre os SIGs e o ICTM está na aplicação destes na análise de terrenos para as ciências ambientais. Pode-se dizer que o ICTM implementa uma funcionalidade de um SIG,

pois sua função é extrair informações geográficas (como por exemplo vindas de imagens de satélite) e gerar outras informações geográficas. Uma outra questão importante é o aspecto inovador do modelo ICTM. Os resultados da categorização gerados pelo ICTM ainda não são produzidos pelos SIGs existentes [13].

2.2 Visão geral

O ICTM 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. O conceito geral do ICTM é mostrado na Figura 1.

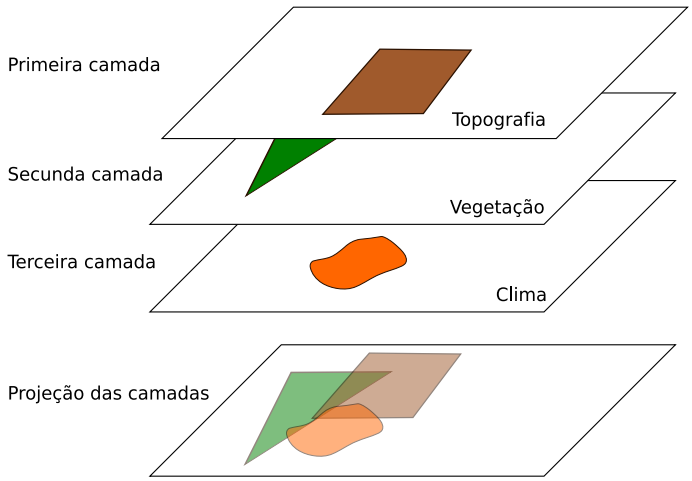

Figura 1 – Conceito geral do ICTM multi-camada. A camada base representa uma projeção das categorizações das camadas em questão.

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 (Figura 2). 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.

O conceito de tesselações pode ser entendido como uma generalização do conceito de autô-

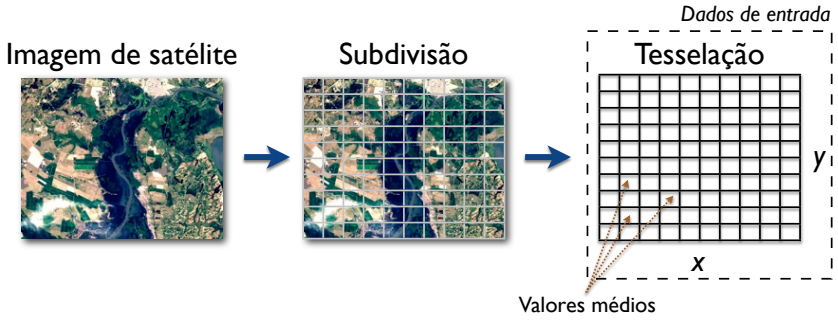

Figura 2 – Uma imagem de satélite é utilizada para extração dos dados de uma camada. Estes dados são então representados em uma tesselação e servem de entrada para o ICTM.

matos celulares [14]. 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 [15]. 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 será dada uma explicação mais aprofundada do processo de categorização e os impactos no desempenho da solução seqüencial quando são alterados parâmetros importantes do modelo, tais como a dimensão das matrizes e o raio.

2.3 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. A tesselação mostrada na Figura 2 é representada na prática como um conjunto de matrizes com n_r linhas e n_c colunas.

É possível dividir o processo de categorização em duas fases: a fase de preparação e a fase de categorização. A Figura 3 apresenta o processo de categorização de uma camada c qualquer do ICTM.

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**. Normalmente, imagens fotografadas por satélites armazenam muitas informações as quais muitas vezes são irrelevantes. Desta forma, para cada subdivisão da tesselação de entrada são extraídos os valores médios e armazenados na Matrix Absoluta.

A categorização prossegue então para a próxima etapa, onde os dados serão simplificados.

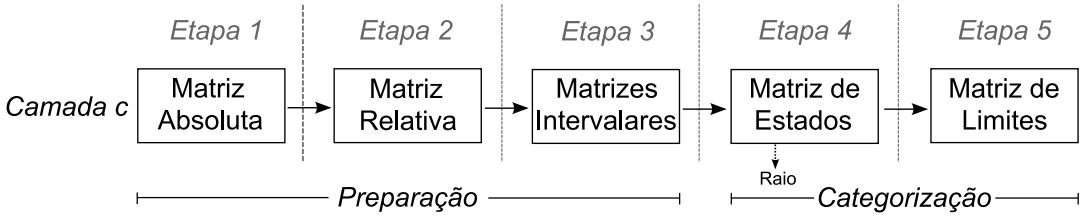

Figura 3 – Processo de categorização de uma camada c qualquer do modelo. Esta categorização é composta por um conjunto de etapas divididas em duas fases.

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**.

Como os dados extraídos de imagens de satélite são muito exatos, os erros contidos na Matriz Relativa são resultado da discretização da região em células da tesselação. Por este motivo, técnicas da Matemática Intervalar [16] são utilizadas para controlar os erros associados aos valores das células (vantagens do uso de intervalos para resolver problemas semelhantes podem ser vistas em [1] e [15]). Portanto, duas **Matrizes Intervalares** são criadas contendo os valores intervalares para as coordenadas x e y, representando assim, a terceira etapa da fase de preparação.

Terminada a fase de preparação, as Matrizes Intervalares são utilizadas como dados de entrada para a fase de categorização. Esta é a principal fase do modelo, capaz de categorizar as células de acordo com a suas características em comum. O processo de categorização compreende duas etapas. Da mesma forma que a fase de preparação, cada etapa da fase de categorização depende dos resultados obtidos na etapa anterior.

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**.

O processo de comparação de cada célula com as suas vizinhas leva em consideração a propriedade que está sendo representada pela camada. Para tanto, esta propriedade é representada por uma função e esta é utilizada para comparar cada célula com a sua vizinhança. Desta forma, esta etapa se caracteriza por analisar a **monotonicidade** da função que mapeia a propriedade representada na camada nas quatro direções [1]. No modelo teórico, estas informações são representadas através de quatro registradores de monotonicidade – $reg.n$ (norte), $reg.s$ (sul), $reg.l$ (leste) and $reg.o$ (oeste) – indicando o comportamento da célula em relação às quatro direções.

Para as células que não fazem parte da borda: $reg.X = 0$, se existe uma função de aproximação não-crescente entre a célula e seus vizinhos na direção X; caso contrário, $reg.X = 1$.

No caso de células que fazem parte da borda nas direções norte, sul, leste e oeste, $reg.n = 0$, $reg.s = 0$, $reg.l = 0$ and $reg.o = 0$, respectivamente.

Sejam $w_{reg.n} = 1$, $w_{reg.s} = 2$, $w_{reg.l} = 4$ and $w_{reg.o} = 8$ pesos associados aos registradores de declividade. A Matriz de Estados é então definida como uma matriz de dimensão $n_r \times n_c$, onde cada valor representa o estado da célula correspondente, calculado pela Equação 2.1. Desta forma, cada célula poderá assumir somente um estado do intervalo de valores $estados_{celula} = [0..15]$.

$$estado_{celula} = (1 \times reg.n) + (2 \times reg.s) + (4 \times reg.l) + (8 \times reg.o) \tag{2.1}$$

Na prática (implementação do ICTM), a Matriz de Estados do modelo teórico foi representada por **4 matrizes separadas**, onde cada uma representa o comportamento de cada célula com as suas vizinhas em uma única direção (Figura 4). Portanto, durante o processo de comparação de cada célula em uma direção, somente uma matriz estará sendo escrita em memória, melhorando o desempenho da aplicação.

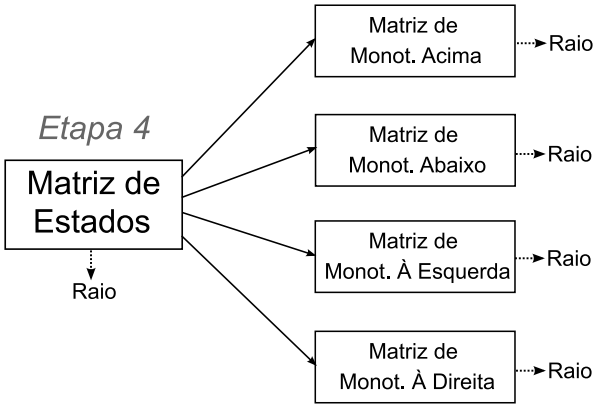

Figura 4 – A Matriz de Estados do modelo teórico é implementada em quatro matrizes separadas (uma para cada direção).

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.

A versão seqüencial do modelo ICTM foi implementada utilizando-se a liguagem C++ e utiliza a biblioteca OpenGL [17] para visualização gráfica dos resultados. Porém, neste trabalho será utilizado somente o módulo de categorização, sendo este o alvo da paralelização deste trabalho.

3 Contexto NUMA

Os últimos anos foram marcados pelo grande investimento em estudos sobre microprocessadores. Cada vez mais nos deparamos com novas arquiteturas de computadores e, principalmente, com a diminuição considerável do tamanho dos *chips*. Porém, a velocidade de processamento de uma máquina não está somente relacionada ao poder de processamento dos microprocessadores. Um microprocessador somente conseguirá processar dados de forma rápida se estes dados puderem ser acessados também de forma rápida na memória.

A tecnologia atual não permite a construção de memórias com grande capacidade de armazenamento que acompanhem a velocidade dos microprocessadores. Porém, deseja-se que cada vez mais seja possível construir memórias que, ao mesmo tempo, possuam maior capacidade de armazenamento e menor tempo de acesso (latência).

Estudos sobre novas tecnologias de microprocessadores continuam avançando. Atualmente, ao invés de continuar investindo em formas de aumentar a freqüência dos processadores, as empresas mudaram sua estratégia e estão investindo em arquiteturas com mais de um núcleo de processamento. Arquiteturas mais avançadas com mais de um processador, que antigamente eram somente utilizadas em áreas específicas da Física, Geografia e Biologia, estão presentes nos computadores pessoais através da tecnologia *multi-core* [18].

3.1 Conceitos básicos

Um computador convencional consiste, em poucas palavras, em um processador, também chamado de CPU (*Central Processing Unit*), executando um programa que está armazenado em uma memória principal centralizada. Normalmente, o tempo de acesso à memória pelo processador é uniforme, ou seja, depende da latência (custo de comunicação) com a memória. O processador é conectado à memória primária e ao sistema de I/O através de um barramento. A memória *cache* ajuda a manter o processador ocupado através da redução do tempo de acesso aos dados (usufruindo das características de localidade temporal e espacial) [19].

Um multiprocessador com memória centralizada é uma extensão simples da arquitetura com um único processador. Neste caso, processadores são adicionados e conectados ao mesmo barramento compartilhando os dados localizados na memória primária. Este tipo de sistema é denominado, pelo ponto de vista do tempo de acesso à memória, uma arquitetura do tipo UMA (Figura 5).

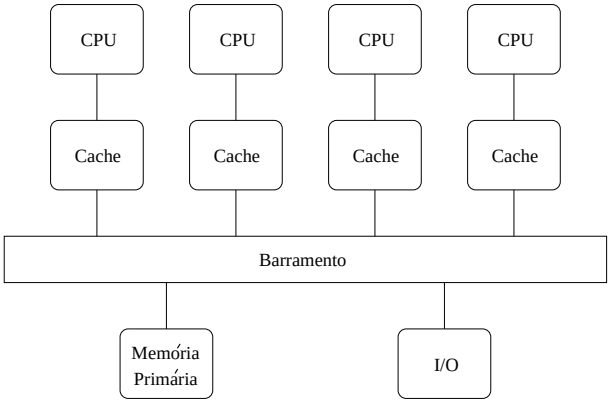

Figura 5 – Esquema genérico de um multiprocessador com memória centralizada, onde o tempo de acesso à memória é uniforme.

O problema dos multiprocessadores com memória centralizada, também conhecidos por máquinas SMP, é que a largura de banda do barramento tipicamente limita o número de processadores conectados a ele. À medida em que são adicionados mais processadores ao mesmo barramento ocorre uma grande disputa para a utilização do mesmo, fazendo com que este tipo de arquitetura não tenha bom desempenho com muitos processadores [20].

Em multiprocessadores, a comunicação entre os processadores é feita através de dados compartilhados. Dizemos que um dado é **compartilhado** quando este é utilizado por mais de um processador. Por outro lado, um dado é dito **privado** quando este é acessado somente por um único processador.

A alternativa para aumentar o limite do número de processadores em um sistema com memória central compartilhada é a distribuição desta memória, dividindo-a em blocos, entre os processadores. Desta forma, cria-se um sistema onde o acesso a uma memória local é muito mais rápido do que ao acesso a uma memória remota. Considera-se **acesso local** quando um processador busca um dado ocasionado a leitura no bloco de memória que está diretamente conectado a ele. Por outro lado, o **acesso remoto** ocorre quando o dado está localizado fisicamente em outro bloco de memória (o qual está mais próximo de outro processador). Neste caso, o tempo de acesso é maior e é realizado através de uma rede de interconexão.

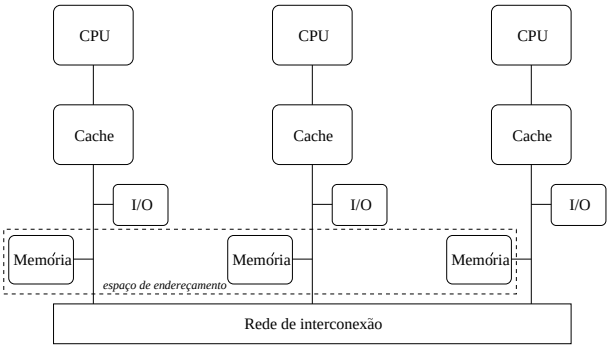

Figura 6 – Esquema genérico de um multiprocessador com memória distribuída, onde o tempo de acesso à memória é variável.

Um multiprocessador com memória distribuída emprega um único espaço de endereçamento, permitindo com que cada processador possa endereçar toda a memória. Além disso, o mesmo endereço em diferentes processadores faz referência à mesma posição em memória. Este tipo de sistema é também denominado multiprocessador NUMA, pois o tempo de acesso à memória varia consideravelmente, dependendo se o endereço que está sendo referenciado está localizado na memória "próxima" ao processador ou não. A Figura 6 mostra a idéia genérica de um sistema com memória distribuída (NUMA).

Arquiteturas NUMA introduzem a noção de **distância** entre componentes do sistema, como por exemplo, CPUs, memória e I/O. A métrica utilizada para determinar a distância varia, porém, a quantidade de "saltos" (*hops*) é uma métrica popular e bastante utilizada. Estes termos significam essencialmente o mesmo daqueles utilizados em redes de interconexão. No exemplo da Figura 6, se um determinado dado não está localizado no módulo de memória mais "próximo" do processador ocorrerá o uso da rede de interconexão para que o dado possa ser buscado em outro bloco.

Uma métrica muito importante que permite avaliar a relação entre o tempo de acesso local e remoto de uma arquitetura NUMA é o **fator NUMA**. O fator NUMA (FN) é a razão entre o tempo médio necessário para buscar um determinado dado armazenado no bloco de memória distate de um processador (t_{remoto}) e tempo médio necessário para buscar este mesmo dado no bloco de memória próximo a este mesmo processador (t_{local}). A Equação 3.1 demonstra como é feito o cálculo do fator NUMA.

$$FN = \frac{t_{remoto}}{t_{local}} \tag{3.1}$$

Basicamente, a forma com que os processadores e blocos de memória são organizados em uma arquitetura NUMA e o fator NUMA permitem uma boa compreensão geral da arquitetura. Além disso, fornecem subsídios ao desenvolvedor para que este possa definir suas estratégias de paralelização de aplicações para este tipo de arquitetura.

3.1.1 Localidade de dados

O mecanismo principal utilizado em *caches* agrupa posições de memória 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 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 estas características.

Blocos são utilizados em *caches* devido a características básicas em programas seqüenci-

ais: localidade espacial e temporal [19]. Se um determinado endereço foi referenciado, há uma grande chance do endereço seguinte também ser referenciado em um curto espaço de tempo (localidade espacial). Por isto, ao invés de somente trazer um único dado da memória principal para a *cache*, um bloco de dados é copiado, pois há uma grande probabilidade de que os dados contíguos a ele também sejam utilizados em curto espaço de tempo. Por outro lado, um programa é também 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 (localidade temporal).

A maior desvantagem em utilizar-se blocos de dados contíguos em multiprocessadores é que diversos processadores podem necessitar de partes diferentes de um bloco, como por exemplo *bytes* diferentes. 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. Esta questão é conhecida por *false sharing* (Figura 7) [21], podendo reduzir o desempenho de uma aplicação paralela.

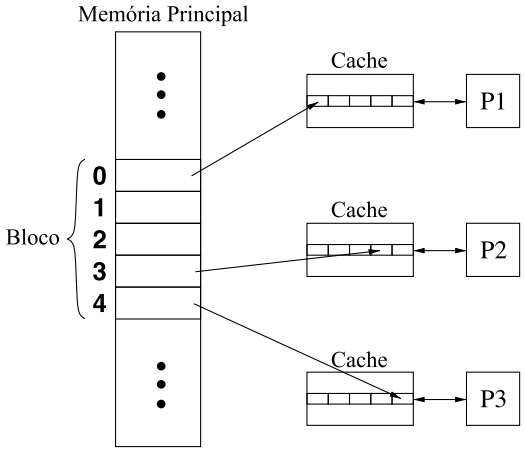

Figura 7 – Exemplo do problema de *false sharing* onde três processadores buscam dados distintos que estão armazenados no mesmo bloco.

Nesta figura, três processadores acessam dados que estão presentes em um mesmo bloco. Por isto, cada processador possui, em sua *cache*, uma cópia deste bloco. Ocorrerá *false sharing*, pois cada processador acessa *bytes* diferentes neste bloco, fazendo com que todo o bloco seja invalidado ou atualizado nas *caches* dos demais processadores. Por questões de simplificação, os módulos de memória foram omitidos na figura.

Em máquinas NUMA, estas questões referentes à localidade dos dados se agravam. Em máquinas UMA, se um endereço em memória referenciado por um processador não está disponível em sua *cache*, este deverá ser trazido da memória principal. O tempo desperdiçado para esta operação será uniforme, visto que a memória é centralizada neste tipo de arquitetura. Porém, em arquiteturas NUMA, este endereço poderá pertencer a um módulo de memória que não está

próximo do processador, havendo assim um desperdício de tempo ainda maior.

Existem mecanismos que permitem o programador definir em que módulo de memória um determinado dado deverá ser armazenado. Em grande parte das arquiteturas NUMA, um dado é armazenado no nodo em que o acessou primeiro: política denominada *first-touch* (ver Seção 4.1.2). Uma outra estratégia possível, ao invés de dar o controle ao programador, é permitir que o próprio sistema operacional mantenha este controle sobre a localidade dos dados. Neste caso, o sistema operacional poderá alterar a localidade dos dados à medida que o programa é executado. Isto pode ser feito de duas formas: através da migração ou replicação de páginas. Na primeira, existe somente uma cópia de cada página de memória, podendo esta ser movida entre os nodos. Na segunda, uma página poderá ter diversas cópias, cada uma residindo em um nodo diferente do sistema.

Tendo em vista as questões expostas, é importante que o desenvolvedor possua conhecimento sobre os mecanismos utilizados no gerenciamento de memória em máquinas NUMA. Tais fatores influenciam nas estratégias utilizadas na paralelização de aplicações neste tipo de arquitetura influenciando consideravelmente o desempenho das mesmas [22].

3.1.2 Escalonamento de processos

O mecanismo responsável por determinar qual processo executará em um determinado processador e por quanto tempo deverá permanecer é denominado escalonador. O objetivo principal deste mecanismo é permitir que a carga do sistema possa ser balanceada, distribuindo as tarefas a serem realizadas entre os processadores. Um sistema possuirá um bom balanceamento de carga se, na maior parte do tempo, todos os processadores estiverem trabalhando.

Em se tratando de arquiteturas paralelas, o escalonamento pode ser analisado em dois níveis [23]. No primeiro, é possível tratar o mecanismo de escalonamento em nível de aplicação. Neste caso, a solução paralela para um problema é desenvolvida levando-se em consideração as características da arquitetura alvo, como por exemplo, número de processadores e suas freqüências, rede de interconexão, memória disponível, entre outras. O mecanismo de escalonamento estará presente dentro da aplicação, distribuindo as tarefas a serem executadas de forma a melhor utilizar a arquitetura alvo. Como o mecanismo é implementado dentro da aplicação, se esta for executada em uma arquitetura diferente, poderá resultar em uma perda de desempenho, necessitando muitas vezes modificá-la.

Por outro lado, o mecanismo de escalonamento poderá ser tratado como uma entidade externa à aplicação. Neste caso, o esforço de programação é reduzido, porém, muitas vezes os resultados não serão tão bons quando se espera. Isto se deve ao fato de que o mecanismo de escalonamento tem conhecimento da arquitetura mas não sobre o comportamento da aplicação que está sendo executada. Por isto, muitas vezes as decisões tomadas pelo escalonador não serão adequadas. Em multiprocessadores, a entidade externa às aplicações que implementa os

mecanismos de escalonamento é o sistema operacional.

Quando tratamos de máquinas multiprocessadas NUMA, há uma questão importante relacionada ao balanceamento de carga. Se um processo p, em um determinado processador localizado em um nodo n, aloca dados em memória e os utiliza com freqüência, é bem provável que este dado esteja fisicamente localizado no módulo de memória do nodo n (próximo à este processador). Porém, se por algum motivo o sistema acaba por ficar desbalanceado, ou seja, existem processadores inativos enquanto outros estão com muito trabalho, o algoritmo de balanceamento de carga pode optar por migrar este processo p para outro nodo do sistema. Neste caso, o tempo de acesso ao módulo de memória será comprometido, visto que a memória alocada estará ainda no nodo n.

Existem diversos estudos sobre a migração de processos em máquinas NUMA com o objetivo de avaliar em quais casos é importante a migração de páginas em memória [24–26]. Mas, esta migração de páginas poderá ser muito custosa dependendo do sistema e da freqüência com que é realizada [27].

Portanto, uma solução paralela somente terá ótimos resultados se estas questões referentes ao escalonamento e balanceamento de carga forem levadas em consideração. Entender como o sistema operacional escalona os processos ajudará o desenvolvedor a definir a melhor maneira de atribuir *threads* a processadores e quando será necessário utilizar mecanismos de migração.

3.2 Bibliotecas para multiprocessadores

Basicamente, existem dois mecanismos mais utilizados para implementar paralelismo em arquiteturas com memória compartilhada: *threads* e a API (*Application Programming Interface*) OpenMP. Nesta seção serão apresentados estes dois mecanismos mais comuns, além de outros também utilizados, porém, com menor freqüência.

3.2.1 Threads

Threads é um mecanismo muito utilizado para implementar programação concorrente em sistemas com memória compartilhada. São conhecidas como processos leves, sendo diferentes de processos usuais, pois compartilham a mesma área de memória. Normalmente, ao desenvolver uma aplicação utilizando *threads*, regiões de memória compartilhada são utilizadas para comunicação entre processos leves. A idéia principal é criar processos concorrentes de forma a melhor utilizar os recursos da arquitetura.

Existem diversas implementações de *threads* disponíveis para diferentes sistemas operacionais. Uma das implementações mais utilizadas em sistemas Linux é a Pthreads (POSIX Threads) [28]. Trata-se de uma implementação do padrão IEEE POSIX 1003.1c de 19995 para

manipulação de *threads*.

Pthreads especifica uma API para lidar com a maior parte das ações requeridas por *threads*. Estas ações incluem a criação e destruição de *threads*, espera por término do processamento de *threads* e interação entre elas. Além disso, estão também disponíveis diversos mecanismos para o tratamento de regiões críticas como *mutexes*, variáveis condicionais e semáforos.

Para fazer o uso de paralelismo utilizando Pthreads, os desenvolvedores precisam necessariamente escrever seu código especificamente para esta API. Isto significa que é necessário incluir bibliotecas, declarar estruturas de dados da biblioteca Pthreads e invocar funções específicas. Basicamente, este processo não é muito diferente em outras APIs que implementam o mecanismo de *threads*.

Mesmo a biblioteca Pthreads sendo consideravelmente simples e portável, esta sofre de uma séria limitação assim como as demais APIs que implementam este mecanismo: a necessidade de utilização de código bastante específico. Em outras palavras, para que o paralelismo possa ser incluído em aplicações as quais originalmente não foram feitas para tanto, muitas vezes são necessárias modificações drásticas no código fonte original. Tarefas que deveriam ser relativamente simples como por exemplo a paralelização de uma estrutura de laço necessitam de modificações bastante importantes como a criação de *threads* e controle da divisão de trabalho entre elas. Nada disto será automatizado, ficando assim a cargo do desenvolvedor.

Devido ao fato de ser necessário inserir uma quantidade considerável de código específico para realizar operações relativamente pouco complexas, os desenvolvedores vêm cada vez mais procurando por alternativas mais simples.

3.2.2 OpenMP

OpenMP (*Open Multiprocessing*) é uma API para programação paralela em multiprocessadores. O padrão OpenMP consiste em um conjunto de diretivas de compilação e uma biblioteca de funções suporte que auxiliam o compilador a gerar códigos *multi-thread*, fazendo o uso de múltiplos processadores em um sistema multiprocessado com memória compartilhada. OpenMP funciona em conjunto com as linguagens Fortran, C e C++.

A idéia do padrão OpenMP é modificar o menos possível o código fonte original de uma aplicação que realizava um processamento seqüencial. Desta forma, diretivas permitem a criação automática de *threads* em partes do código que possam ser paralelizadas. Através do uso de diretivas, o desenvolvedor informa para o compilador que determinado bloco de código deve ser paralelizado. Esta biblioteca utiliza o modelo *fork-join*, criando *threads* para executar determinado processamento em paralelo e, posteriormente, destruindo-as ao final do processamento. A Figura 8 demonstra o funcionamento geral da biblioteca.

Normalmente, blocos de código que requerem um maior poder computacional são compostos por laços onde há um grande processamento a ser realizado. Nos casos em que as ite-

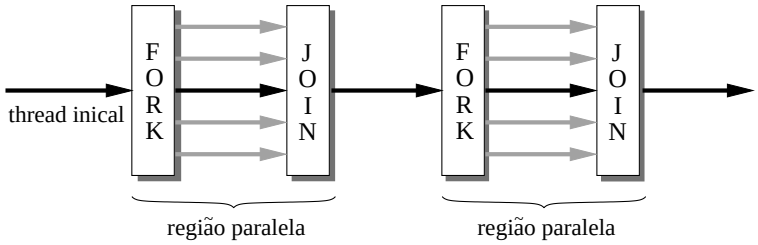

Figura 8 – Modelo de execução da API OpenMP: permite a criação e destruição automática de *threads* em blocos de código a serem paralelizados.

rações de um laço são independentes, a biblioteca OpenMP oferece uma diretiva que permite paralelizá-lo facilmente. Através do uso desta diretiva, o desenvolvedor indica ao compilador que um determinado laço terá suas iterações executadas em paralelo. Para explicar melhor seu uso, será tomado como exemplo o laço abaixo expresso em C:

```
for (i=0; i < tamanho; i++) computa(vetor[i]);
```

Imaginando que a função *computa* fará um grande processamento considerando o valor armazenado na posição *i* do vetor, a paralelização deste laço poderá ser feita utilizando uma diretiva `omp parallel for` da seguinte forma:

```
#pragma omp parallel for
for (i=0; i < tamanho; i++) computa(vetor[i]);
```

Durante a execução deste laço, a *thread* inicial criará *threads* adicionais e todas trabalharão em conjunto para cobrir todas as iterações do laço. O número de *threads* a serem criadas é definido pelo usuário através da função *omp_set_num_threads()*. Quando todas as iterações do laço tiverem sido executadas somente a *thread* inicial continuará executando o restante do código.

É importante notar que nem todo laço poderá ser paralelizado. Para que o compilador possa transformar este laço seqüencial em um laço paralelo é preciso ser possível determinar o número de iterações do laço em tempo de compilação. Devido a este fato, o laço não poderá conter testes condicionais que façam com que a sua execução seja interrompida prematuramente (como por exemplo *return* ou *break*).

Além da primitiva mostrada existem diversas outras que permitem paralelizar porções do código, como por exemplo criar regiões onde todas as *threads* executam o mesmo código em paralelo. Desta forma, através do uso de condicionais (*if-else*) é possível atribuir tarefas distintas para *threads*. Juntamente com as primitivas de paralelização de laços é possível também incluir funções de redução. Estas funções são bastante úteis quando o laço possui algum tipo de variável incremental. Através do uso destas funções a biblioteca OpenMP realiza uma determi-

nada operação definida pelo desenvolvedor para reduzir os resultados parciais computados por cada *thread* em um resultado global ao final do laço.

Como pode ser visto, a biblioteca OpenMP oferece grande facilidade ao programador no desenvolvimento de uma aplicação paralela para uma máquina multiprocessada com memória centralizada. Grandes benefícios podem ser obtidos ao utilizar esta biblioteca, como por exemplo, a simplicidade de programação. Trata-se de uma biblioteca que torna questões de mais baixo nível transparentes para o desenvolvedor.

3.2.3 Demais bibliotecas

Esta seção apresenta outras bibliotecas que podem ser utilizadas para paralelização de aplicações para máquinas NUMA. Porém, estes são casos mais específicos que dependem de características da rede de interconexão e da forma com que a arquitetura alvo é construída.

SMI

A biblioteca SMI (*Shared Memory Interface*) [29] baseia-se em um modelo de programação que utiliza regiões compartilhadas de memória. O foco principal desta biblioteca é permitir o programador lidar com todos os efeitos resultantes de uma máquina NUMA. Levando-se em consideração a tendência de construir máquinas NUMA através do agrupamento de diversas máquinas multiprocessadas (e.g., SMPs), há a necessidade de existir uma biblioteca capaz de lidar com estas diferenças de tempos de acesso e alocação de regiões compartilhadas de uma forma mais alto nível.

Um exemplo de rede de interconexão que permite o agrupamento de máquinas multiprocessadas formando uma máquina NUMA é a SCI (*Scalable Coherent Interface*). Este é um padrão de interconexão que especifica *hardware* e protocolos para conectar nodos em uma rede de alto desempenho [30]. A grande diferença entre SCI e redes de interconexão como, por exemplo, Myrinet [31] reside na forma como a comunicação é realizada. Em SCI, a comunicação não é baseada em troca de mensagens. Toda a comunicação dá-se por comunicação implícita através de acessos remotos à memória. Desta forma, cada nodo pode mapear para seu próprio espaço de endereçamento segmentos remotos de memória pertencentes a qualquer outro nodo, atuando como se tais segmentos fossem locais. Toda a comunicação real é feita de forma transparente pelo *hardware* e protocolos de comunicação, que se responsabilizam por leituras e escritas remotas.

MPI

Diferentemente das bibliotecas anteriormente descritas, as quais subentendiam a existência de uma memória compartilhada, MPI é um padrão que define um modelo para troca de men-

sagens em sistemas onde não existe memória compartilhada. A idéia básica é abstrair detalhes de baixo nível, permitindo ao desenvolvedor focar em detalhes da aplicação. Uma execução de uma aplicação em MPI consiste em um conjunto de processos que se comunicam utilizando métodos que permitem enviar e receber mensagens, agrupar e sincronizar processos [32]. Uma biblioteca que implementa o padrão MPI muito conhecida e utilizada é a MPICH [33].

Mesmo sendo originalmente utilizada para comunicação entre processos em arquiteturas onde não há memória compartilhada (e.g., *clusters*), esta biblioteca também pode ser utilizada em máquinas NUMA. Neste caso, os vários processos serão criados e a comunicação é feita localmente, porém, por troca de mensagens. A biblioteca MPICH possui otimizações para troca de mensagens entre processos que estão sendo executados em um mesmo nodo, não havendo a necessidade do dado ser transmitido pela rede de interconexão. Diversos estudos foram realizados com o objetivo de propor modificações no padrão MPI de forma a obter melhor desempenho em máquinas multiprocessadas [34–38].

O uso desta biblioteca torna-se interessante em *clusters* de máquinas multiprocessadas. Neste caso, cada nodo do *cluster* é uma máquina NUMA com vários processadores. É possível utilizar o padrão MPI para realizar comunicações dos processos entre os nodos do *cluster* e outra biblioteca (originalmente desenvolvida para programação paralela em multiprocessadores) para paralelizar o processamento dentro do nodo.

3.3 Máquinas NUMA utilizadas neste trabalho

Os experimentos deste trabalho foram realizados em duas máquinas NUMA bastante distintas. A primeira arquitetura é composta por 8 processadores *dual core* AMD® Opteron de 2.2 GHz e 2 MB de memória *cache* em cada processador. A máquina é organizada em 8 nodos e possui no total 32 GB de memória principal. A Figura 9 mostra um esquema desta arquitetura.

A memória principal é dividida em 8 blocos (4 GB de memória em cada bloco) e o tamanho de uma página é de 4 KB. Cada nodo possui 3 conexões as quais são utilizadas para comunicar com os demais nodos e com os controladores de entrada e saída (no caso dos nodos 0 e 1). Estas conexões resultam em diferentes latências para acessos remotos (**fator numa de 1.2 à 1.5**). O sistema operacional utilizado nesta arquitetura é uma distribuição Debian do Linux versão 2.6.23-1-amd64 com suporte à NUMA (chamadas de sistema e a API numactl). O compilador disponível é o GCC (*GNU Compiler Collection*). De agora em diante esta arquitetura será referenciada pelo nome de **Opteron**.

A segunda máquina NUMA utilizada neste trabalho é composta por 16 processadores Itanium 2 de 1.6 GHz e 9 MB de *cache* L3 em cada processador. A máquina é organizada em 4 nodos de 4 processadores e possui no total 64 GB de memória principal. Esta memória é dividida em 4 blocos (16 GB de memória em cada nodo) e o tamanho de uma página é 64 KB. Os nodos são interconectados utilizando-se um *switch* próprio denominado FSS (*FAME Scala-*

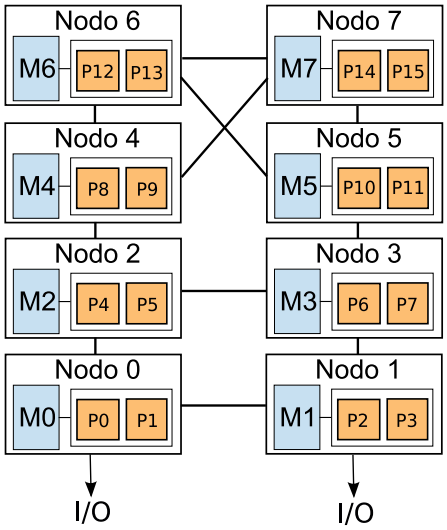

Figura 9 – Arquitetura **Opteron** com 8 processadores *dual core*. Os processadores são organizados em 8 nodos (cada um com um bloco de memória de 4 GB) e é caracterizada pelo baixo fator NUMA.

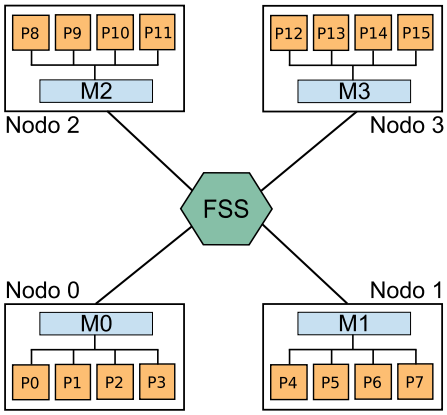

Figura 10 – Arquitetura **Itanium 2** com 16 processadores. Os processadores são organizados em 4 nodos (cada um com um bloco de memória de 16 GB) e é caracterizada pelo alto fator NUMA.

bility Switch), o qual é um *backplane* desenvolvido pela empresa Bull. Esta conexão resulta em diferentes tempos de acesso à memória (**fator NUMA de 2 à 2.5**). A Figura 10 demonstra a idéia geral da organização desta arquitetura.

O sistema operacional utilizado é uma distribuição Red Hat do Linux (versão 2.6.18) com suporte a arquiteturas NUMA (chamadas de sistema e a API numactl). O compilador disponível nesta arquitetura é o ICC (*Intel C Compiler*). De agora em diante será utilizado o nome **Itanium 2** para fazer referência a esta arquitetura.

3.4 Discussão

Este capítulo apresentou alguns conceitos importantes que devem ser levados em consideração ao desenvolver aplicações paralelas para máquinas NUMA. Além disso, foram apresentadas algumas bibliotecas que possibilitam a implementação de paralelismo neste tipo de arquitetura. Por fim, foram apresentadas duas máquinas NUMA com características bastante distintas. Estas máquinas serão utilizadas para avaliar o desempenho da solução proposta neste trabalho.

O grande problema das bibliotecas apresentadas anteriormente, com exceção da SMI, está relacionado ao fato de que nenhuma delas oferece diretivas ou mecanismos que permitem ao programador obter um maior controle sobre a localidade dos dados em memória. Na teoria, máquinas NUMA podem ser tratadas como sendo máquinas SMP (onde a memória é centralizada), ignorando-se as diferenças entre tempo de acesso local e remoto à memória. Em alguns casos, dependendo das características da aplicação e da arquitetura alvo, é possível que bons resultados sejam obtidos sem maiores controles sobre localidade dos dados. Porém, na maioria das vezes há um impacto relativamente grande e o desempenho ideal não é obtido.

A biblioteca SMI oferece serviços para trabalhar com o conceito de localidade (dados remotos ou locais). Porém, esta é uma biblioteca que foi implementada para ser utilizada juntamente com a rede de interconexão SCI. Isto não torna a biblioteca portável para diversos tipos de arquiteturas NUMA.

A afinidade de memória é muito importante e normalmente traz benefícios consideravelmente grandes. O Capítulo 4 tratará exatamente sobre este tema: de que forma é possível atingir a afinidade de memória utilizando-se outros mecanismos em conjunto com bibliotecas que não implementam tais funcionalidades, como por exemplo, Pthreads e OpenMP.

Neste trabalho optou-se pela utilização da biblioteca OpenMP devido aos seguintes fatores:

- Controle de *threads*: o controle sobre a criação e destruição de *threads* é feito automaticamente pela API;
- Modelo *fork-join*: é um modelo que se adapta facilmente às características do ICTM. O ICTM é composto por uma seqüência de etapas dependentes, onde cada etapa poderá ser paralelizada internamente;
- Facilidade de paralelização de laços: todas as etapas do ICTM seqüencial possuem uma estrutura básica de laços aninhados para realizar o processamento. Este tipo de estrutura pode ser facilmente paralelizado utilizando-se diretivas específicas da API.

A utilização de duas arquiteturas NUMA bastante distintas mostra-se muito importante. Neste contexto, o fator NUMA apresenta-se como uma propriedade importante para diferenciar arquiteturas deste tipo. A escolha de duas arquiteturas com processadores diferentes e fatores NUMA contrastantes permitirá uma avaliação mais completa do comportamento da solução paralela proposta neste trabalho.

4 Afinidade de threads e memória

Um sistema operacional que oferece suporte a máquinas NUMA é desenvolvido com o intuito de automaticamente alocar memória no bloco mais próximo de cada processo que está sendo executado em um determinado nodo de uma máquina NUMA (política denominada *first-touch*). Porém, o que acontecerá com processos que disparam diversas *threads* através de diversos nodos da máquina? Como os desenvolvedores podem assegurar-se de que a memória está sendo utilizada de forma otimizada por estas *threads*?

Um dos conceitos mais importantes para desenvolvedores de aplicações paralelas para máquinas NUMA é o conceito de **afinidade**. A afinidade deve ser entendida como uma forma de "associação". Existem dois tipos de afinidade: a afinidade de memória e a afinidade de *thread*. A afinidade de memória significa dizer que um certo conjunto de endereços de memória é fisicamente mapeado para um bloco de memória local em um nodo específico da máquina NUMA. Ou seja, há uma espécie de associação destes endereços a um certo bloco da memória principal. Isto significa dizer que, quando algum destes endereços é requisitado por quaisquer processadores, o dado deverá ser buscado no bloco de memória o qual ele está fisicamente alocado.

Da mesma forma, afinidade de *thread* significa dizer que uma determinada *thread* será executada somente em um conjunto particular de processadores/núcleos. Tipicamente, este conjunto pertence a um mesmo nodo de máquina NUMA.

Através do uso explícito de afinidade, os desenvolvedores podem assegurar-se de que cada *thread* estará acessando dados locais os quais estão armazenados no bloco de memória mais próximo a elas. Portanto, o desempenho geral da aplicação pode ser melhorado.

4.1 NUMA API

A partir da versão 2.6 do *kernel*, o sistema operacional Linux introduziu chamadas de sistema que permitem atribuir *threads* para processadores específicos. A NUMA API (libnuma) extende esta funcionalidade, permitindo especificar em qual nodo a memória deverá ser alocada [3]. A fim de permitir que programas possam tirar um maior proveito de arquiteturas NUMA, esta API captura informações sobre a arquitetura, oferecendo especificações sobre a topologia. Atualmente esta API está disponível na distribuição SUSE® Linux Enterprise Server 9 para processadores AMD® 64 e para a família de processadores Intel® Itanium. Todavia, é possível instalá-la em qualquer distribuição do Linux através do pacote **numactl** [3].

A libnuma é a API recomendada para controlar afinidade de *threads* e afinidade de regiões de memória em máquinas NUMA e para isto, oferece uma interface aos desenvolvedores. A afinidade é obtida através da utilização de políticas que modificam a maneira com que as *threads* são escalonadas e forma com que a memória pode ser alocada. As políticas sobre *threads* e regiões de memória são aplicadas utilizando-se um conjunto distinto de funções oferecidas pela API. A seguir serão explicadas como estas políticas poderão ser aplicadas.

4.1.1 Afinidade de threads

Para que uma *thread* possa ser atribuída a um processador/núcleo específico ou migrar entre processadores/núcleos pertencentes a um conjunto determinado, a NUMA API oferece a função `sched_setaffinity()`. Esta função recebe dois parâmetros: o identificador da *thread* e uma máscara.

A especificação dos processadores/núcleos os quais esta *thread* poderá ser executada é passado como parâmetro através de uma máscara de *bits*, onde cada *bit* representa o identificador único do processador o qual poderá executá-la. A máscara é configurada utilizando-se a função `CPU_SET()` que recebe dois parâmetros: o identificador do processador ou núcleo e a máscara. A Figura 11 exemplifica o funcionamento da máscara de *bits*.

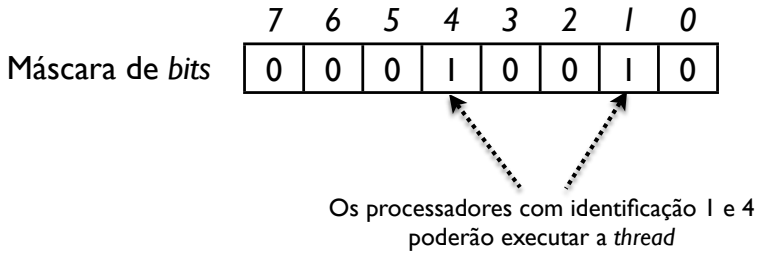

Figura 11 – Exemplo de uma máscara de *bits*. Neste caso, as *threads* poderão ser executadas tanto no processador 1 quanto no 4 (o número total de processadores neste exemplo é 8).

No exemplo da Figura 11, a função `CPU_SET()` foi utilizada para configurar a máscara, configurando os *bits* relacionados aos processadores 1 e 4. Neste caso, quando uma máscara com mais de um *bit* em 1 é passada para a função `sched_setaffinity()`, o desenvolvedor indica que a *thread* em questão poderá ser executada em quaisquer processadores configurados nela (neste caso estão configurados os processadores 1 e 4). Portanto, fica a cargo do sistema operacional escaloná-la e migrá-la livremente entre estes processadores quando o algoritmo de escalonamento o achar necessário.

Caso o desenvolvedor deseje fixar uma *thread* em apenas um processador ou núcleo, somente o *bit* correspondente deverá ser configurado na máscara. Isto indica que o escalonador do sistema operacional não poderá migrá-la em hipótese alguma.

4.1.2 Afinidade de memória

As políticas de memória podem ser definidas por processo ou por região de memória. A política por processo (*process policy*) é aplicada a todas as alocações de memória realizadas no contexto de um processo. Por outro lado, políticas definidas por região de memória, também chamadas de *virtual memory area policy*, permitem que processos possam determinar uma política para um bloco em memória em seu espaço de endereçamento.

Para que as políticas de memória possam ser utilizadas de maneira correta é necessário primeiramente utilizar a chamada de sistema `mmap()`. Esta chamada de sistema permite reservar um espaço de memória virtual a ser utilizado posteriormente. Este espaço será constituído, após a alocação física dos dados, por um conjunto de páginas de memória. O tamanho de uma página de memória poderá variar de acordo com a arquitetura que está sendo utilizada (como visto na Seção 3.3).

As páginas de uma região de memória virtual reservada através da função `mmap()` que ainda não foram acessadas pela primeira vez são chamadas de *untouched*. A alocação física destas páginas só ocorrerá no momento em que algum dado pertencente a elas for escrito. A política básica implementada pelo sistema operacional é denominada *first-touch*, onde o escalonador realizará a alocação física no bloco de memória mais próximo do processador o qual realizou o primeiro acesso. A Figura 12 demostra o funcionamento desta política.

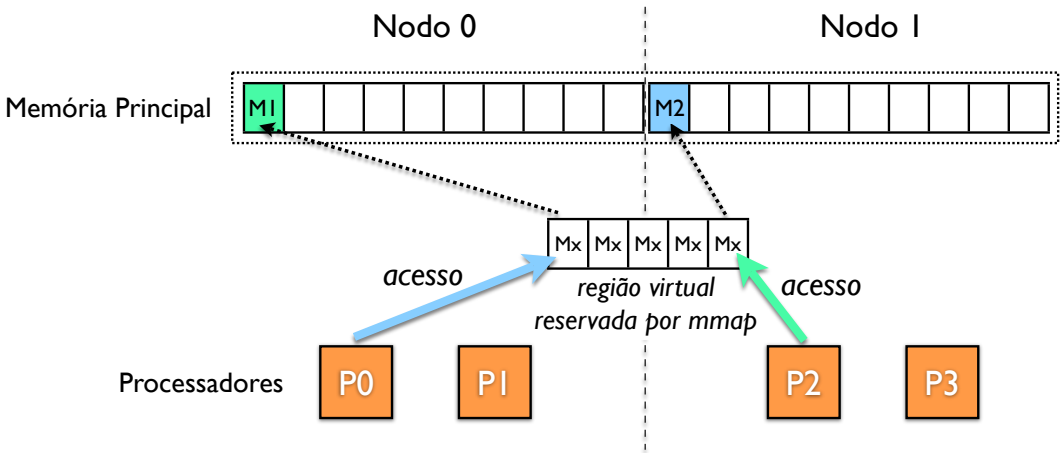

Figura 12 – Política *first-touch*: a página é alocada próxima ao processador o qual a requisitou primeiro.

No exemplo da Figura 12, uma arquitetura NUMA hipotética com 4 processadores divididos em dois nodos é utilizada para mostrar o funcionamento da política *first-touch*. Uma região de memória composta por 5 páginas é reservada através da chamada de sistema `mmap()`. O processador $P0$ acessa algum dado armazenado em uma página pela primeira vez. Como este processador está localizado no nodo 0, a página é então fisicamente alocada no bloco de memória pertencente a este nodo. Da mesma forma, o processador $P2$ realiza um acesso a outra página, sendo esta alocada no bloco de memória localizado no nodo 1, o qual pertence o

processador $P2$.

A princípio esta política parece ser suficiente para controlar as alocações de memória em máquinas NUMA. Porém, dependendo de como uma aplicação paralela realiza o acesso à memória, muitas vezes esta política não mostra bons resultados. Além disso, muitas vezes os algoritmos implementados pelo escalonador do sistema operacional fazem com que *threads* sejam migradas entre processadores. Nestes casos, o acesso que anteriormente era local pode deixar de o ser, pois a *thread* que o acessava pode não estar mais sendo executada no mesmo nodo. Portanto, não somente é necessário ter-se conhecimento sobre a política de memória mais adequada para uma aplicação paralela que é executa em uma arquitetura NUMA, mas também é importante manter-se um controle sobre a localização de *threads* em processadores.

A aplicação de uma política de memória é feita através da função `mbind()`. Esta função trabalha sobre páginas de memória, ou seja, uma política é sempre aplicada à uma página ou a um conjunto de páginas que foram anteriormente reservadas com o uso da função `mmap()`. Para que a política tenha o efeito desejado, é necessário que a função `mbind()` seja utilizada sobre uma ou um conjunto de páginas *untouched*, por isto mostra-se necessário o uso da função `mmap()` ao invés da função `malloc()`[1]. Caso a página ou o conjunto de páginas já tenha sido utilizado, a política aplicada não terá efeito, visto que a política padrão *first-touch* já foi aplicada quando as páginas foram acessadas pela primeira vez.

A NUMA API oferece quatro tipos de políticas de memória, são elas:

- *default*: é a política padrão (*first-touch*), onde o dado será alocado no nodo o qual fez o primeiro acesso;
- *bind*: aloca um conjunto de páginas em um conjunto específico de nodos (utiliza uma máscara de *bits* para especificar os nodos);
- *interleave*: entrelaça alocações de páginas em um conjunto de nodos (utiliza uma máscara de *bits* para especificar os nodos);
- *preferred*: aloca preferencialmente em um nodo específico.

A diferença entre a política *bind* e *preferred* está no fato de que a primeira falhará ao tentar alocar no nodo específico caso não haja espaço necessário para isto. Por outro lado, a segunda alocará em qualquer outro nodo caso não haja espaço no nodo requisitado.

4.2 Memory Interface Library (MAI)

Diversas propostas anteriores utilizaram-se de algoritmos específicos, mecanismos e ferramentas para alocação de dados em memória, migração e replicação de páginas para garantir a

[1]Não há garantia de que, após a alocação de dados através da função *malloc()*, todas as páginas de memória alocadas estejam *untouched*.

afinidade de memória de um processo pesado em máquinas NUMA [4,39–43]. Nestas soluções, a afinidade de memória é aplicada através de políticas de memória para o processo como um todo, não sendo possível especificar diferentes políticas para os diferentes padrões de acesso à memória realizados por um processo pesado composto por diversas *threads*. Isto pode resultar muitas vezes em um desempenho não ideal, pois a afinidade de memória está relacionada às variáveis e objetos de uma aplicação.

A interface MAI surgiu com o intuito de permitir que diferentes políticas de memória possam ser aplicadas a diferentes dados em um mesmo processo pesado. Através de sua utilização, é possível obter-se um controle mais refinado da afinidade de memória e *threads* em máquinas NUMA. Esta biblioteca utiliza como base a NUMA API (*libnuma*), ou seja, funciona como uma interface de mais alto nível que esconde as chamadas de sistema e a utilização de máscaras de *bits*, o que muitas vezes necessitam de maiores conhecimentos dos desenvolvedores. Além de ser mais amigável, também implementa novas políticas de memória não oferecidas pela NUMA API.

Além das funções para controlar políticas de memória, a MAI também oferece funções para migração de páginas. Estas funções podem ser muito úteis para aplicações com padrões de acesso à memória bastante irregulares. Há também funções que retornam informações estatísticas sobre a execução da aplicação, como por exemplo o número de migrações de *threads* entre processadores ou de páginas de memória.

Esta interface está sendo 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 National de Recherche en Informatique et en Automatique*) em Grenoble, estando sob os direitos da *GNU Public License v.2*. É implementada em C e foi desenvolvida para utilização em máquinas NUMA com sistema operacional Linux. O modelo de programação com memória compartilhada é um pré-requisito para o uso da interface. Além disto, como esta interface baseia-se na utilização da NUMA API, o pacote *numactl* precisa necessariamente estar instalado.

4.2.1 Visão geral

Como dito anteriormente, a MAI foi desenvolvida para ser utilizada em conjunto com um modelo de programação com memória compartilha. Portanto, assume-se que o desenvolvedor utilizará *threads* para processar tarefas em paralelo. Atualmente existem duas bibliotecas para programação *multi-thread* suportadas pela interface: Pthreads e OpenMP. Isto significa dizer que a interface é capaz de utilizar funções específicas destas bibliotecas para controlar a localização de *threads* em máquinas multiprocessadas.

A estrutura de uma aplicação paralela que utiliza *threads* em conjunto com a interface MAI é mostrada na Figura 13.

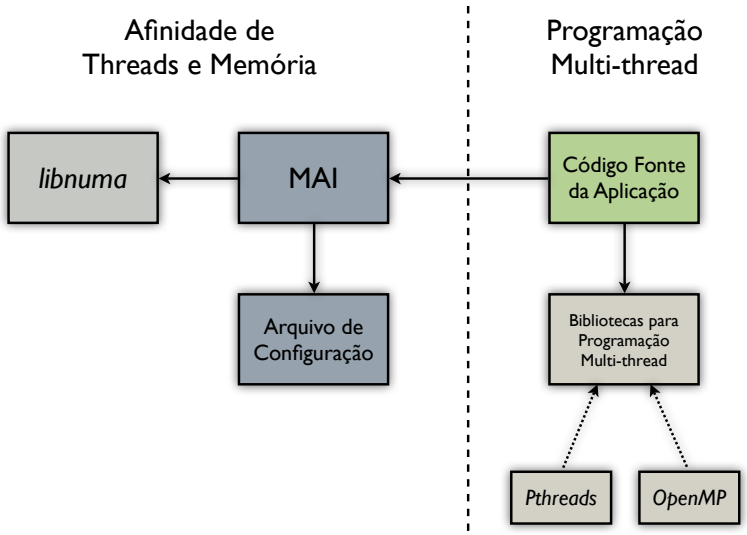

Figura 13 – Estrutura geral de uma aplicação paralela que utiliza *threads* em conjunto com a interface MAI.

O lado direito da Figura 13 descreve a estrutura geral de uma aplicação paralela *multi-thread*. Em poucas palavras, o desenvolvedor escreve código fonte de sua aplicação e inclui uma biblioteca que oferece serviços de criação, controle e destruição de *threads* (as bibliotecas Pthreads e OpenMP foram utilizadas como exemplo). Estas *threads* irão então executar processamentos em paralelo, utilizando os processadores disponíveis na arquitetura alvo. Para que a afinidade de *threads* e memória possa ser adicionada, basta que o desenvolvedor inclua a interface MAI em seu código *multi-thread*. A interface, por sua vez, necessitará de um arquivo de configuração o qual irá informá-la a respeito de quais blocos de memória serão utilizados para alocar dados pelas políticas de memória e quais processadores irão executar as *threads*. Esta figura mostra claramente que a interface MAI é tratada como um serviço adicional, oferecendo funcionalidades para que o desempenho desejado possa ser obtido em máquinas NUMA.

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*. A Figura 14 mostra a idéia geral da estrutura do arquivo de configuração. Primeiramente deverão ser especificados a quantidade de nodos da arquitetura NUMA a ser utilizada, seguida por seus identificadores únicos (neste caso os identificadores são numerados de 0 à $n-1$). Os nodos são tratados pela interface como sendo compostos por um bloco de memória e um conjunto de processadores ou núcleos. Da mesma forma, deverá ser especificado a quantidade de processadores ou núcleos (em caso de arquiteturas *multi-core*) a ser utilizada, seguida pelos identificadores únicos dos processadores ou núcleos (também numerados neste exemplo de 0 à $n-1$).

A criação deste arquivo de configuração poderá ser feita de forma dinâmica, ou seja, através de regras implementadas pelo próprio desenvolvedor a aplicação poderá criá-lo automaticamente no início de sua execução. Supondo que um dos parâmetros da aplicação paralela seja o

```
#especificação dos nodos NUMA
n
0
1
...
n-1
#especificação dos processadores (ou núcleos)
n
0
1
...
n-1
```

Figura 14 – Arquivo de configuração da MAI. Através dele são especificados os nodos NUMA e os processadores (ou núcleos) a serem utilizados.

número de *threads* que irão executar em paralelo, o próprio desenvolvedor poderá utilizar este parâmetro para criar o arquivo de configuração com os identificadores dos processadores ou núcleos que serão utilizados para executar tais *threads*. Isto se mostra muito útil, permitindo com que a própria aplicação paralela possa configurar a MAI de forma a obter o melhor desempenho possível.

A seguir serão descritas as funções oferecidas pela MAI para controlar as políticas de memória, *threads*, migração de páginas, entre outras.

4.2.2 Principais funções

As funções implementadas pela MAI podem ser divididas em cinco grupos: funções de sistema, alocação, políticas de memória, políticas de *threads* e estatísticas.

Funções de Sistema: são divididas em funções para configurar a interface e funções para coletar informações da máquina NUMA. As funções `init(char *filename)` e `final()` permitem inicializar a interface com o arquivo de configuração especificado e finalizar a aplicação desalocando dados em memória, respectivamente. As demais funções possuem finalidades diversas de coleta de informações.

Listagem das principais funções de sistema:

```
void init(char *filename);
void final();
int get_num_nodes();
int get_num_threads();
```

Funções para Alocação de Memória: realizam o mapeamento de uma determinada área da memória RAM física em uma memória virtual, permitindo a aplicação de políticas de memória

nesta área. A estrutura a ser mapeada pode ser um vetor, matrizes bi-dimensionais ou tri-dimensionais com tipos primitivos do C ou estruturas de dados. Estas funções possuem como base o uso da chamada de sistema `mmap()`, tornando transparente os detalhes de baixo nível.

Listagem das principais funções para alocação de memória:

```
void* alloc_1D(int nx, int typesize, int type);
void* alloc_2D(int nx, int ny, int typesize, int type);
void* alloc_3D(int nx, int ny, int nz, int typesize, int type);
```

Funções de Controle de Políticas de Memória: são as funções que permitem aplicar as políticas de memória. As políticas *cyclic* e *cyclic_block* distribuem as páginas de uma região de memória virtual entre os nodos (especificados no arquivo de configuração) de forma circular. A política *cyclic* distribui página à página, enquanto a política *cyclic_block* permite distribuir grupos de páginas de tamanho `blocksize`. A política *bind_all* realiza a alocação física dos dados em quaisquer um dos nodos especificados no arquivo de configuração. Porém, quem determina qual página deverá ser alocada em cada um destes nodos é o sistema operacional. Por outro lado, a política *bind_block* divide uma região virtual em n partes, onde n é o número de nodos especificados no arquivo de configuração. É possível também realizar a migração de páginas através da função `migrate_pages`.

Listagem das principais funções para o controle de políticas de memória:

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

Funções de Controle de Políticas de *Threads*: permitem associar *threads* à um processador ou a um conjunto de processadores. Quando uma *thread* é associada a somente um processador garante-se que esta não irá migrar. As funções `set_thread_id_omp()` (caso a biblioteca seja OpenMP) e `set_threads_id_posix()` (caso a biblioteca seja Pthreads) quando executadas por cada *thread*, configuram a MAI de forma que esta *thread* execute em somente um processador ou núcleo (*thread* 0 será executada no processador/núcleo 0, *thread* 1 será executada no processador/núcleo 1, e assim sucessivamente). A função `bind_threads()` aplica a configuração feita por uma das duas funções anteriores.

Listagem das principais funções para o controle de políticas de *threads*:

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

Funções para Exibição de Estatísticas: exibem informações a respeito da execução da aplicação. As funções `number_page_migration()` e `number_thread_migration()` retornam o número de migrações de páginas de uma região de memória virtual e de *threads*,

respectivamente. As funções `get_time_pmigration()` e `get_time_tmigration` retornam o tempo desperdiçado com migrações de páginas e *threads*, respectivamente.

Listagem das principais funções para a exibição de estatísticas:

```
int number_page_migration(unsigned long *pageaddrs, int size);
int number_thread_migration(unsigned int *threads, int size);
double get_time_pmigration();
double get_time_tmigration();
```

4.3 Discussão

Este capítulo apresentou um dos conceitos mais importantes para o desenvolvimento de aplicações paralelas para máquinas NUMA: o conceito de afinidade. Além disso, foi apresentado como a afinidade pode ser aplicada utilizando-se duas bibliotecas: NUMA API e MAI. As funções de gerenciamento de memória juntamente com as funções de gerenciamento de localização de *threads* permitem que o controle sobre dados e processos possa ser realizado pelo desenvolvedor. Com a utilização destas funcionalidades é possível construir aplicações paralelas que utilizem melhor os recursos de máquinas NUMA.

Porém, este controle se dá através do uso de funções e chamadas de sistema de baixo nível pela NUMA API. A utilização de máscaras de *bits* para especificar processadores (no caso das funções para controlar afinidade de *threads*) e nodos (no caso das funções para controlar afinidade de memória) não é uma forma muito prática e exige um conhecimento maior do desenvolvedor. Além disso, a função `mbind()` exige do desenvolvedor o conhecimento sobre páginas de memória que armazenam uma determinada região de memória virtual, visto que a aplicação das políticas se dará sempre neste nível de abstração. Portanto, a aplicação de diferentes políticas de memória a uma mesma região virtual exige do desenvolvedor cálculos específicos para que isto seja feito em termos de páginas de memória. Caso contrário, as políticas não serão aplicadas corretamente.

Tendo em vista o exposto, mostra-se necessário o uso de uma interface capaz de abstrair estas questões de mais baixo nível, facilitando assim aplicação e o controle de políticas de memória e *threads*. A MAI fornece este alto nível de abstração, além de incluir novas políticas e a possibilidade de realizar a migração de páginas de memória. Além disto, esta interface possui suporte às bibliotecas OpenMP e Pthreads, tornando-se simples o processo de integração. Devido a isto, a interface MAI será utilizada neste trabalho para a aplicação das políticas de memória e *threads*.

5 OpenMP-ICTM

O primeiro passo para paralelização do ICTM para máquinas NUMA foi baseado na utilização da biblioteca OpenMP. Como dito na Seção 3.4, a escolha desta biblioteca se deu por diversas razões. A razão principal está relacionada a simplicidade de uso: o código seqüencial pode ser paralelizado com poucas modificações, pois o controle de criação e destruição de *threads* é feito pela API. Além disso, OpenMP utiliza o modelo *fork-join*, o qual consiste no uso de diversas regiões paralelas em meio a porções de código seqüencial. Portanto, este modelo pode ser facilmente aplicado ao ICTM seqüencial, onde cada etapa do processo de categorização pode ser paralelizada.

O principal objetivo deste capítulo é descrever uma proposta de uma solução paralela para o ICTM sem levar em consideração a localidade de *threads* e de dados em memória, denominada OpenMP-ICTM. Desta forma, será possível avaliar a necessidade de utilização de mecanismos de afinidade para aumentar o desempenho da solução em máquinas NUMA. Para isto, primeiramente será descrito como foi feita a paralelização das etapas do processo de categorização do ICTM. Após, uma descrição dos casos de estudo utilizados para avaliar o desempenho da solução paralela serão apresentados. Finalmente, uma análise de desempenho desta solução será mostrada.

5.1 Distribuição do trabalho entre threads

O processo de categorização do ICTM inicia com a leitura dos dados de entrada. Estes dados são então armazenados na Matriz Absoluta para então serem computados ao longo do processo de categorização. Neste trabalho, a paralelização será focada nas demais etapas do processo, pois estas necessitam de um alto poder computacional ao categorizar grandes áreas geográficas.

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. A Figura 15 mostra como as etapas utilizam as matrizes durante o processamento.

A computação realizada **em cada uma das etapas** mostradas na Figura 15 pode ser entendida, de um modo estrutural, como sendo composta por dois laços de repetição encadeados. Estes laços serão responsáveis pela computação de todas as células das matrizes em cada etapa.

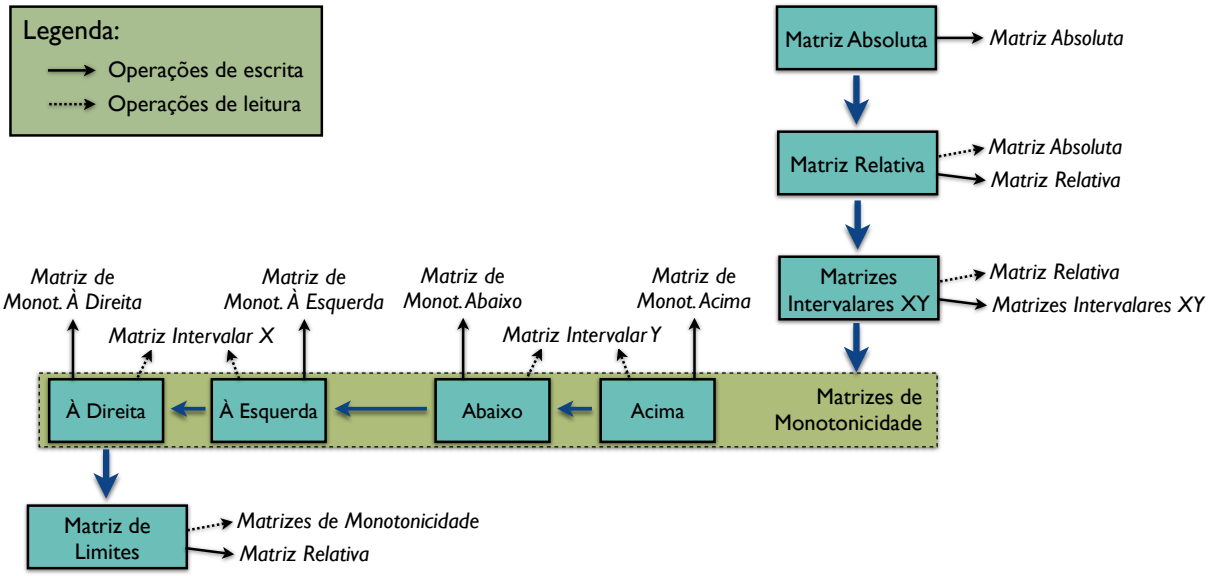

Figura 15 – Leitura e escrita das matrizes durante o processo de categorização do ICTM. Dependendo da etapa, diferentes matrizes são consultadas.

Porém, para o cálculo das matrizes de monotonicidade o parâmetro raio é levado em consideração. Desta forma, a cada uma destas etapas é adicionada mais uma estrutura de laço, pois cada 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.

O encadeamento de laços em cada etapa pode ser paralelizado utilizando-se uma diretiva OpenMP denominada `omp parallel for` (descrita anteriormente na Seção 3.2.2). Ao aplicar esta diretiva no laço mais externo, o desenvolvedor informa a biblioteca OpenMP que as iterações deste laço deverão ser quebradas em t partes, onde t corresponde ao número de *threads* criadas pela biblioteca OpenMP:

```
#pragma omp parallel for
for (i = 0; i < linhas); i++) {
  for (j = 0; j < colunas); j++) {
    //computação específica de uma etapa
  }
}
```

A diretiva *omp parallel for* é responsável pela criação das *threads*, divisão do trabalho entre elas e destruição (após o término do laço mais externo). No caso específico do ICTM, a divisão do trabalho entre as *threads* é realizada da seguinte forma: cada *thread* será responsável por processar um conjunto de linhas da matriz. A Figura 16 ilustra esta forma de divisão do trabalho entre as *threads* através do uso da diretiva *omp parallel for* no laço mais externo de cada etapa do processo de categorização.

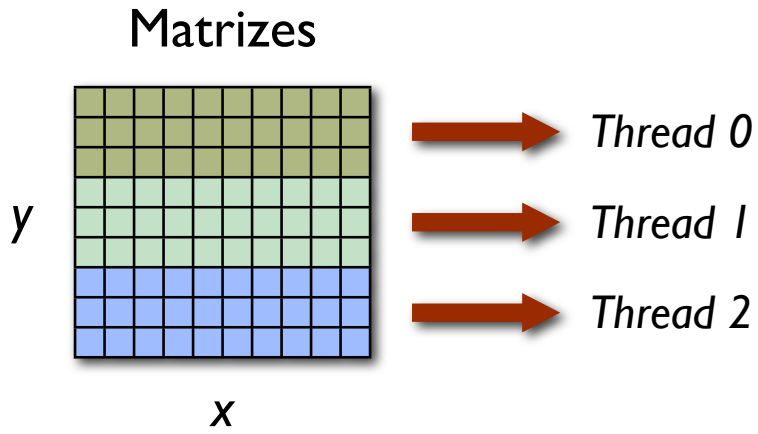

Figura 16 – Divisão do trabalho entre as *threads*. A primitiva `omp parallel for` atua sobre o laço mais externo, quebrando o processamento da matriz em conjuntos de linhas.

No exemplo da Figura 16, as matrizes possuem 9 linhas e 10 colunas. Ao executar a solução paralela com 3 *threads*, a biblioteca OpenMP dividirá as iterações dos laços em 3 partes. Esta será a forma de divisão das matrizes durante todo o processo de categorização.

Esta solução não apresenta problemas para as etapas de monotonicidade, onde a vizinhança é consultada para o processamento de cada célula, pois como dito anteriormente, estas etapas somente consultam dados das Matrizes Intervalares. Logo, não há dependência mútua entre etapas de cálculo das Matrizes de Monotonicidade.

De um modo geral, estas primitivas foram adicionadas nos laços de cada etapa do processo de categorização, com exceção da etapa de construção da Matriz Relativa. Na solução seqüencial, dois procedimentos são feitos nesta etapa: (i) busca pelo célula com maior valor e (ii) divisão de todas as células pela célula de maior valor.

Para a solução paralela, utilizou-se a mesma estratégia apresentada anteriormente, onde a matriz é dividida em conjuntos de linhas e cada *thread* será responsável por processar somente as linhas do respectivo conjunto. Porém, não é possível utilizar somente a primitiva `omp parallel` **`for`** devido a necessidade de encontrar-se o maior valor.

Logo, a seguinte estratégia foi adotada: (i) cada *thread* encontrará a célula com maior valor local (para o seu conjunto de linhas), armazenando este valor em um vetor indexado pelo identificador único de cada *thread*, após, (ii) cada *thread* consultará este vetor afim de descobrir o maior valor global (maior dos maiores locais) e finalmente, (iii) cada *thread* ficará responsável por normalizar as células do seu conjunto de linhas.

A API OpenMP oferece a diretiva `omp parallel` que permite criar uma região paralela onde todas as *threads* executarão o mesmo código. Dentro desta região paralela, é possível utilizar a diretiva `omp` **`for`** para que, neste ponto, todas as *threads* trabalhem em conjunto para dividir o processamento de uma estrutura de laço. Estas primitivas serão utilizadas para paralelizar a etapa de construção da Matriz Relativa, de acordo com as estratégia anteriormente descrita.

O primeiro conjunto de linhas da listagem de código abaixo é referente ao procedimento (i), onde cada *thread* encontrará a célula com maior valor local. O segundo conjunto de linhas diz respeito ao procedimento (ii), onde cada *thread* encontrará o maior valor global (que será o mesmo para todas as *threads*). Finalmente, o último conjunto de linhas é responsável por dividir o processo de normalização entre as *threads*, dividindo os valores das células pelo maior valor encontrado em (ii). Ao final, a Matriz Relativa terá todas as suas células normalizadas:

```
#pragma omp parallel private(i, j, contMaior, threadId, maior)
{
    threadId = omp_get_thread_num();
    maiores[threadId] = -1;

    #pragma omp for
    for(i=0; i < _rows; i++)
      for(j=0; j < _columns; j++)
        if (_abs[i][j] > maiores[threadId])
          maiores[threadId] = _abs[i][j];

    maior = maiores[0];
    for(contMaior=1; contMaior < _numThreads; contMaior++)
      if(maiores[contMaior] > maior)
        maior = maiores[contMaior];

    #pragma omp for
    for(i=0; i < _rows; i++)
      for(j=0; j < _columns; j++)
        _rel[i][j] = _abs[i][j] / maior;
}
```

A instrução opcional `private` utilizada em conjunto com a diretiva `omp parallel` permite informar ao compilador que as variáveis `i`, `j`, `contMaior`, `threadId` e `maior` serão privadas, ou seja, cada uma destas variáveis poderá assumir valores diferentes em cada *thread*. A variável `threadId` é responsável por armazenar o identificador único da *thread*, assumindo assim um valor diferente para cada *thread* que estará executando esta região paralela. A obtenção deste identificador é feita através da função `omp_get_thread_num()` oferecida pela API OpenMP.

Acredita-se que esta solução para a etapa de construção da Matriz Relativa e a utilização da diretiva `omp parallel for` para as demais etapas do processo de categorização seja uma forma elegante e pouco evasiva de paralelizar o ICTM. Todavia, a alocação de memória nos

nodos NUMA e a migração de *threads* é feita de acordo com as regras do *kernel* do Linux, não sendo possível controlá-las utilizando somente diretivas da biblioteca OpenMP.

5.2 Avaliação de desempenho

Nesta seção será apresentado a avaliação de desempenho da paralelização do ICTM através do uso da biblioteca OpenMP (OpenMP-ICTM). Primeiramente, serão descritos os casos de estudo escolhidos para avaliar a solução. Após, será feita uma análise dos resultados obtidos nas duas arquiteturas alvo (Seção 3.3). No final, será feita uma discussão sobre os resultados e algumas conclusões serão apontadas. Uma listagem completa dos resultados, onde mostra-se a média dos *speed-ups* com raio 20, 40 e 80 e seu desvio padrão, é feita nos Apêndices A e B.

5.2.1 Casos de estudo

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 paralela 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 estudo 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.

A solução proposta neste trabalho divide o processamento entre as *threads* dentro de cada etapa do processo de categorização de uma camada. Devido a este fato, os casos de teste apresentados aqui sempre utilizarão somente uma única camada no modelo. A Tabela 1 mostra os parâmetros escolhidos para a realização dos testes.

Tabela 1 – Casos de estudo utilizados para avaliar o desempenho das soluções propostas neste trabalho.

Nome	Dimensão das matrizes	Uso de memória	Raio
Caso 1	4.800 x 4.800	1 GB	20, 40 e 80
Caso 2	6.700 x 6.700	2 GB	20, 40 e 80
Caso 3	9.400 x 9.400	4 GB	20, 40 e 80
Caso 4	13.300 x 13.300	8 GB	20, 40 e 80

As dimensões das matrizes foram calculadas considerando-se a quantidade de memória necessária para armazená-las. Além disso, com o intuito de compreender a influência do raio na solução paralela, optou-se por realizar os testes com três valores para o raio: 20 (pequeno), 40 (médio) e 80 (grande).

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 10 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 às máquinas NUMA.

5.2.2 Resultados

Esta seção apresenta os resultados obtidos com a solução paralela do ICTM utilizando-se a biblioteca OpenMP. Diferentes experimentos foram realizados variando-se os 4 casos de estudo descritos anteriormente. Além disso, foram utilizadas as duas arquiteturas NUMA descritas na Seção 3.3.

A medição dos tempos de execução do ICTM paralelo utilizando-se a biblioteca OpenMP foi feita individualmente em cada etapa, excluindo-se o tempo necessário de leitura dos dados de entrada e escrita na Matriz Absoluta (parte não paralelizada). O tempo de execução final compreende a soma dos tempos individuais de todas as demais etapas do processo de categorização de uma camada.

Arquitetura Opteron

Os resultados após a execução dos experimentos na arquitetura Opteron são mostrados na Figura 17. Ao analisar os gráficos de *speed-up* de cada caso individualmente é possível perceber que os melhores resultados foram obtidos sempre utilizando-se um raio maior (80). A utilização de poucos processadores (de 2 até 6) resulta em um ganho de desempenho bastante próximo do ideal. Porém, à medida com que são utilizados mais processadores o ganho de desempenho cai de forma considerável. Isto acontece devido ao fato de serem utilizados mais nodos, aumentando-se a quantidade de acessos remotos aos dados.

Ao comparar os resultados com os diferentes casos de estudo pode-se perceber um comportamento interessante das curvas de *speed-up*: a medida que se aumenta a dimensão das matrizes, as curvas tendem a se aproximar umas das outras. No Caso 1, a utilização de matrizes pequenas aliada a raios pequenos resultam em pouco ganho de desempenho. Porém, a medida que a dimensão das matrizes aumenta (demais casos) o ganho de desempenho utilizando-se raios menores aumenta. Isto faz com que as curvas de *speed-up* utilizando-se raios 20 e 40 se aproximem da curva com raio 80.

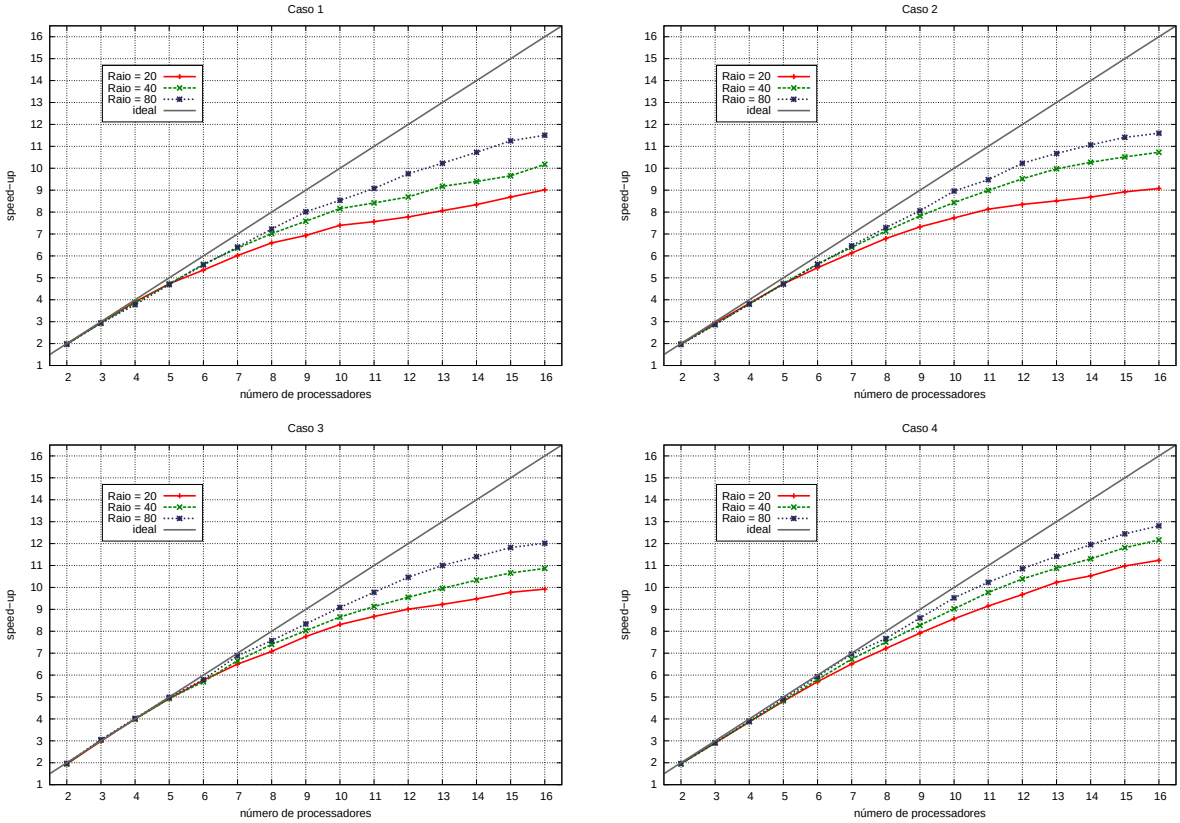

Figura 17 – Desempenho da solução OpenMP-ICTM na arquitetura Opteron.

O baixo fator NUMA (de 1.2 à 1.5) desta arquitetura permite que, mesmo aumentando-se significantemente a probabilidade de ocorrerem acessos remotos à memória (raio maior e matrizes de dimensão maior), o desempenho da solução continue aumentando.

Arquitetura Itanium 2

Os resultados dos experimentos na arquitetura Itanium 2 são mostrados na Figura 18. Em oposição aos resultados obtidos com a arquitetura Opteron, à medida em que o casos de estudo maiores são utilizados, percebe-se uma redução considerável de desempenho. O fator de *speed-up* máximo obtido no Caso 1 foi de aproximadamente 12,5 enquanto o máximo no Caso 4 foi de 11.

Comparando os resultados das duas arquiteturas é possível concluir principalmente duas diferenças. A primeira está relacionada à proximidade das curvas de *speed-up* à curva ideal com poucos processadores. Neste caso, mesmo com poucos processadores existe uma distância entre elas e a curva ideal (em oposição a curvas muito próximas da ideal na arquitetura Opteron). A segunda está relacionada à influência do raio no desempenho: mesmo com poucos processadores é possível observar diferenças de *speed-up* ao utilizar raios de tamanho diferente (em oposição às curvas sobrepostas à ideal na arquitetura Opteron).

Além das informações apontadas anteriormente, há uma peculiaridade interessante quando são comparados os 4 gráficos da Figura 18. O aumento do caso de estudo implica em uma me-

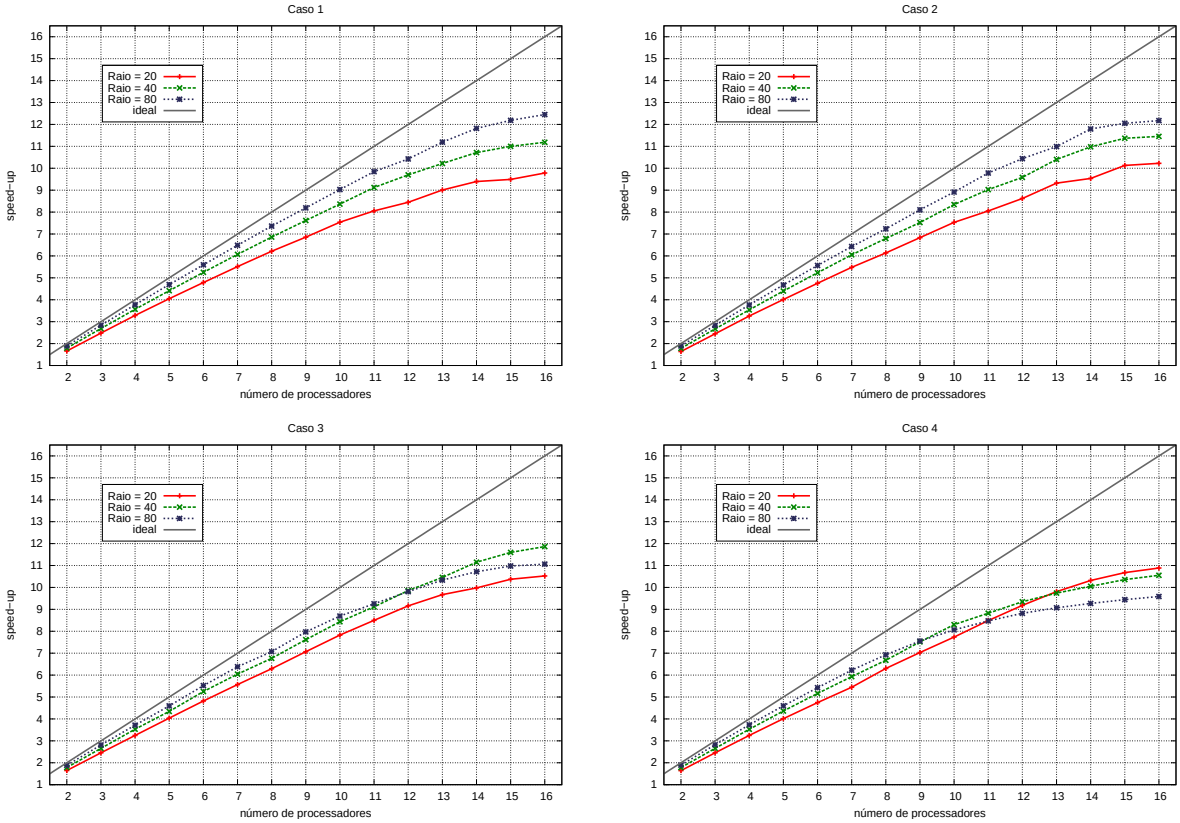

Figura 18 – Desempenho da solução OpenMP-ICTM na arquitetura Itanium 2.

lhora do desempenho das curvas que representam o uso de raios pequenos. De forma análoga, isto implica em uma redução de desempenho das curvas que representam o uso de raios maiores. Isto acontece principalmente com o uso de raios extremos: o desempenho utilizando-se raio 20 (pequeno) aumenta quando o tamanho do caso de estudo é aumentado enquanto o desempenho utilizando-se raio 80 (grande) diminui. Isto faz com que as curvas se cruzem (casos de estudo 3 e 4). Isto se deve principalmente ao alto fator NUMA desta arquitetura (de 2 à 2.5). A utilização de valores maiores para o raio faz com que mais acessos à memória sejam feitos. Como nenhum controle sobre a localidade dos dados é feito, há uma maior chance de ocorrerem acessos remotos. Como o fator NUMA é alto, há um maior impacto no desempenho geral da solução paralela.

5.3 Discussão

Este capítulo apresentou a proposta inicial de uma solução paralela do ICTM para máquinas NUMA (OpenMP-ICTM) utilizando-se somente a biblioteca OpenMP. Neste caso, as máquinas NUMA foram consideradas como sendo UMA, pois nenhum controle específico sobre a localidade de dados em memória foi feito.

Após a análise dos resultados do OpenMP-ICTM é possível concluir que este apresentou um

interessante ganho de desempenho. Porém, as medidas de *speed-up* e eficiência apontam que esta solução possui limitações. O maior *speed-up* obtido foi de aproximadamente 12,8 na arquitetura Opteron utilizando-se 16 processadores enquanto que na arquitetura Itanium 2 o fator foi de aproximadamente 12,5. O impacto do fator NUMA pode ser observado principalmente ao submeter os testes na arquitetura Itanium 2. O aumento da dimensão das matrizes juntamente com a utilização de raios maiores resultaram em desempenhos piores.

Uma análise geral da eficiência da solução OpenMP-ICTM em ambas arquiteturas é mostrada nas Figuras 19 e 20. Nestas, são mostradas as eficiências médias para cada caso de estudo. Isto foi feito através do cálculo de uma curva de *speed-up* média para cada caso de estudo, que conterá a média dos *speed-ups* relativos aos três valores de raios (20, 40, 80). É importante ressaltar que estes *speed-ups* médios apresentaram um desvio padrão pequeno (estes valores estão descritos nos Apêndices A e B).

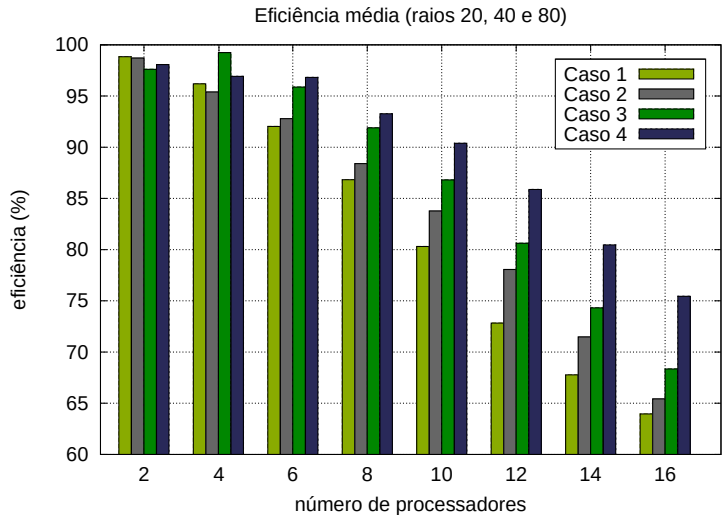

Figura 19 – Influência do caso de estudo no desempenho do OpenMP-ICTM (arquitetura Opteron).

A Figura 19 mostra as eficiências médias obtidas com a solução OpenMP-ICTM na arquitetura Opteron. Os resultados mostram que com poucos processadores a solução apresenta uma eficiência parecida, independentemente do caso de estudo (dimensão das matrizes). Porém, a medida em que são inseridos mais processadores, as diferenças entre as eficiências dos quatro casos de estudo comparados aumentam. Estas diferenças se dão pelo aumento da eficiência dos casos de estudo maiores quando comparados aos casos de estudo menores. A utilização de um número maior de processadores faz com que casos de estudo que utilizam matrizes maiores possuam também eficiências melhores. Novamente, salienta-se que isto ocorre devido ao baixo fator NUMA desta arquitetura.

De forma análoga, a Figura 20 mostra as eficiências médias obtidas com a solução OpenMP-ICTM na arquitetura Itanium 2. Em oposição ao comportamento observado na arquitetura anterior (Opteron), a utilização de mais processadores implica em uma redução significativa da eficiência dos casos de estudo maiores. A má localização dos dados em memória aliada ao

fato desta arquitetura possuir um alto fator NUMA impacta drasticamente nesta redução da eficiência.

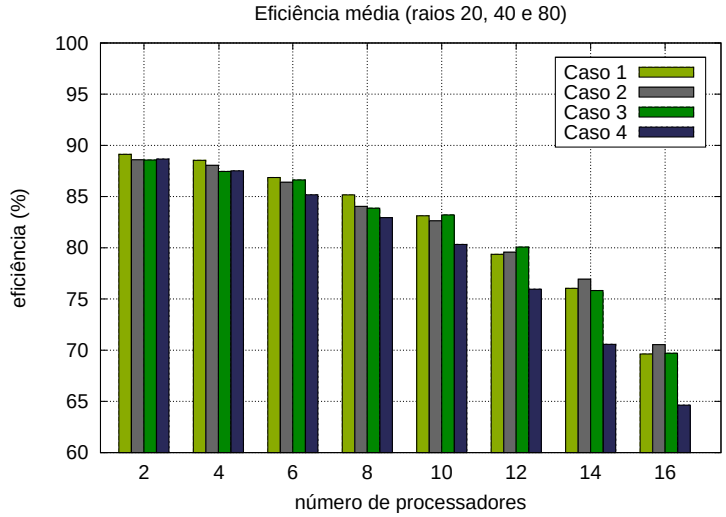

Figura 20 – Influência do caso de estudo no desempenho do OpenMP-ICTM (arquitetura Itanium 2).

As diretivas oferecidas pela biblioteca OpenMP não oferecem mecanismos de controle sobre a localidade dos dados e posicionamento de *threads*. É evidente que, com um melhor controle sobre a afinidade de memória e *threads*, é possível reduzir a interferência dos acessos não uniformes à memória, tornando-se possível extrair significativos ganhos de desempenho em máquinas NUMA.

6 NUMA-ICTM

Em arquiteturas NUMA torna-se importante reduzir o impacto do fator NUMA sobre a aplicação que está sendo paralelizada. A redução deste fator pode ser feita considerando dois tipos de otimizações: latência e largura de banda. Manter os dados mais próximos do processador que os utilizam reduz a latência, reduzindo assim o número de acessos remotos. Por outro lado, é possível reduzir a disputa de acesso entre os processadores ao mesmo bloco de memória, distribuindo-se os dados entre estes blocos de memória, otimizando-se assim a largura de banda.

Estas duas estratégias serão exploradas através da utilização de diferentes políticas de memória. A alocação de dados em memória será feita com o uso da interface MAI, através da aplicação de políticas de memória implementadas por ela. Isto possibilitará o desenvolvimento de uma nova solução paralela para o problema, denominada NUMA-ICTM.

Primeiramente será descrito como a interface MAI foi integrada a OpenMP-ICTM (apresentada na Seção 5). Após, uma análise de desempenho desta nova solução será discutida, mostrando-se os resultados obtidos. Finalmente, uma discussão geral sobre os resultados será feita onde serão apontadas algumas informações adicionais que puderam ser extraídas dos resultados.

6.1 Integração com a MAI

Após a proposta de uma solução utilizando-se somente diretivas OpenMP para paralelizar o modelo ICTM para máquinas NUMA, foram adicionadas funções específicas da biblioteca MAI para controlar políticas de memória e *threads*. Basicamente, estas modificações foram realizadas no código correspondente ao processo de inicialização, onde as matrizes são alocadas. Quatro grupos de funções foram utilizadas:

- **Funções de Sistema:** foram utilizadas as funções de inicialização da interface `init()` (com passagem de arquivos de configuração) e função de finalização `final()`;
- **Funções de Alocação de Memória:** a função `malloc()` foi substituída pela função `alloc_2D()` para a alocação das matrizes bi-dimensionais;
- **Funções para o controle de políticas de *threads*:** duas funções foram utilizadas para

posicionar *threads* em processadores ou núcleos específicos: `bind_threads()` em conjunto com a função `set_thread_id_omp();`

- **Funções para o controle de políticas de memória:** as quatro funções que permitem modificar as políticas aplicadas durante a alocação física de dados nos blocos de memória foram utilizadas: `cyclic()`, `cyclic_block()`, `bind_block()` e `bind_all()`.

Os arquivos de configuração da interface MAI foram criados de acordo com a plataforma NUMA e o número de *threads* a serem utilizadas em cada execução. No caso da máquina Opteron, cada *thread* é atribuída a um núcleo (pois trata-se de uma arquitetura com processadores *dual core*), enquanto na máquina Itanium 2, cada uma é atribuída a um processador.

Como dito anteriormente, o controle de afinidade de *threads* foi feito através do uso de duas funções da interface MAI. Estas funções foram utilizadas da seguinte forma:

```
#pragma omp parallel
  set_thread_id_omp();
bind_threads();
```

A diretiva `omp parallel` cria uma região paralela onde todas as *threads* executam o mesmo código – neste caso a função `set_thread_id_omp()`. Ao executar esta função, cada *thread* individualmente armazenará seu identificador único em uma estrutura interna da interface MAI. Este identificador é obtido através da chamada de uma função específica da biblioteca OpenMP utilizada pela função `get_thread_id_omp()`. Após a atribuição dos identificadores de cada *thread* à estrutura interna, somente a *thread* principal executará a função `bind_threads()`. Esta função aplicará as configurações armazenadas nesta estrutura interna da MAI. Com isto, é possível se assegurar que não ocorrerão migrações entre processadores ou núcleos, visto que cada *thread* será associada a somente um processador ou núcleo. O controle sobre *threads* permite a utilização de afinidade de memória para melhor controlar a localização dos dados.

A alocação das matrizes na solução OpenMP-ICTM foi feita utilizando-se a função padrão de alocação de dados do C: `malloc()`. Como descrito na Seção 4.2, a interface MAI oferece uma função para a alocação de matrizes denominada `alloc_2D()`. Esta função permite criar um mapeamento da memória RAM física em uma memória virtual. Devido a similaridade entre estas duas funções, poucas modificações foram necessárias para adaptar a antiga forma de alocação das matrizes. Mais uma vez salienta-se que o uso da função `alloc_2D()` é obrigatório para que seja possível, posteriormente, especificar a política de memória a ser utilizada na alocação das matrizes.

A Figura 21 mostra visualmente como a política de memória *bind_block* é aplicada às matrizes do ICTM. Nesta figura, as células das matrizes foram agrupadas em termos de páginas de memória, visto que todas as políticas são aplicadas a este tipo de estrutura. No caso específico desta política, os dados armazenados nas matrizes são fisicamente alocados nos blocos

de memória de acordo com a distribuição do trabalho feita pela primitiva `omp parallel for` (solução OpenMP-ICTM descrita na Seção 5.1). Devido a isto, cada *thread* irá acessar, principalmente, páginas de memória que estão fisicamente armazenadas no mesmo nodo, reduzindo-se assim o número de acessos remotos.

Considerando-se a matriz com 9 linhas mostrada na figura e supondo-se que a máquina NUMA possui **três processadores em cada nodo**, cada *thread* que estará sendo executada em cada processador será responsável por computar somente uma linha da matriz. Esta linha estará fisicamente alocada no bloco de memória pertencente ao nodo onde a *thread* está sendo executada. Isto permite a redução de ocorrências de *false-sharing* e, mesmo quando ocorram, o tempo desperdiçado para atualização das *caches* dos demais processadores será menor, visto que os dados estarão armazenados no bloco de memória próximo a eles.

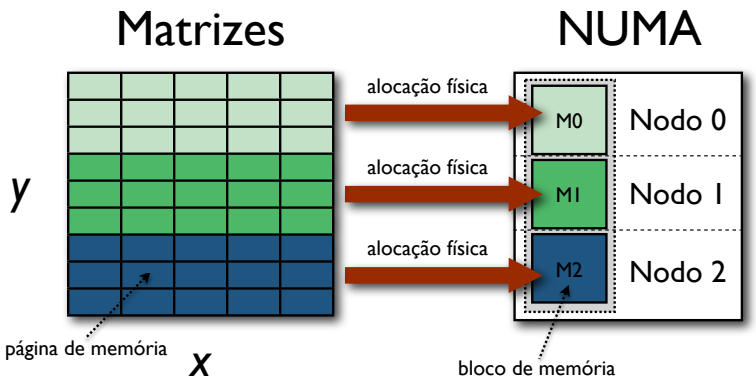

Figura 21 – A política *bind_block* aplicada ao OpenMP-ICTM. Esta política permite com que as linhas das matrizes acessadas pelas *threads* estejam fisicamente alocada nos blocos de memória próximos a elas.

De forma similar, a política *bind_all* poderá ser aplicada especificando-se somente os nodos os quais possuem *threads* processando. Porém, a localidade física de cada página será determinada pelo *kernel* do Linux e não pelo desenvolvedor, como na política *bind_block*.

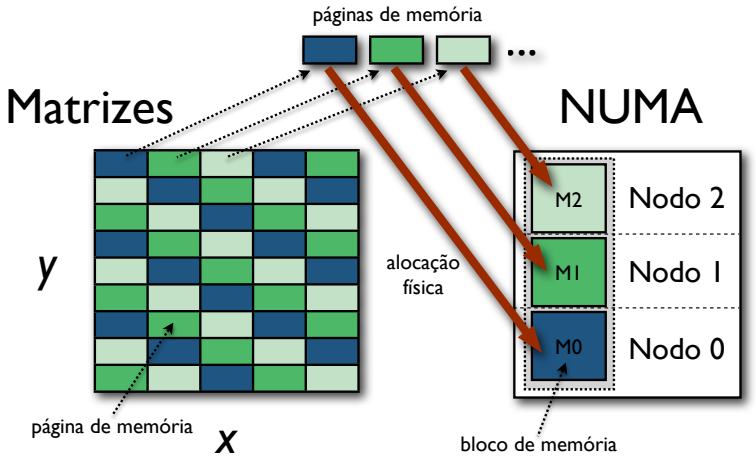

Figura 22 – A política *cyclic* aplicada ao OpenMP-ICTM. Esta política distribuirá as páginas de forma circular entre os nodos da máquina NUMA.

Ao aplicar a política *cyclic*, as páginas de memória serão fisicamente alocadas da forma mostrada na Figura 22. Estas páginas de memória são distribuídas através dos nodos da máquina NUMA através de um processo cíclico: a primeira página de cada matriz será fisicamente armazenada no bloco de memória pertencente ao Nodo 0, a segunda página no bloco de memória pertencente ao Nodo 1, a terceira página no bloco de memória pertencente ao Nodo 2, repetindo-se este processo para as demais páginas. Um comportamento similar ocorre quando a política *cyclic_block* é aplicada. Porém, ao invés da distribuição ser feita página a página, o processo distribuirá conjuntos de páginas.

A esta nova solução paralela do ICTM otimizada para a utilização de afinidade de memória deu-se o nome de NUMA-ICTM. Nesta solução foram adicionadas as quatro políticas implementadas pela interface MAI. A política de memória a ser aplicada é especificada através de um parâmetro de entrada, permitindo-se que esta solução possa ser facilmente configurada para diferentes tipos de máquinas NUMA. A inclusão de afinidade de memória permite com que esta solução utilize melhor os recursos das máquinas NUMA, como será visto na seção a seguir.

6.2 Avaliação de desempenho

Esta seção apresenta uma análise de desempenho da solução paralela do ICTM que explora estratégias de alocação de memória (NUMA-ICTM). Da mesma forma apresentada na Seção 5.2.2, diferentes experimentos foram realizados variando-se os 4 casos de estudo escolhidos. Novamente, as duas arquiteturas NUMA descritas na Seção 3.3 foram utilizadas para a realização de todos os experimentos.

A forma de medição dos tempos de execução da NUMA-ICTM não foi alterada, ou seja, são tomados os tempos de execução de cada etapa do processo de categorização, excluindo-se o tempo necessário para a leitura dos dados de entrada e escrita na Matriz Absoluta. Portanto, o tempo de execução final compreende a soma dos tempos individuais de todas as demais etapas do processo de categorização.

A seguir, são apresentados os resultados obtidos por arquitetura NUMA comparando-se as diferentes políticas de memória e seus impactos no desempenho final da solução proposta. Uma listagem completa dos resultados, onde mostra-se a média dos *speed-ups* com raio 20, 40 e 80 e seu desvio padrão, é feita nos Apêndices A e B.

6.2.1 Arquitetura Opteron

Os resultados apresentados nesta seção são oriundos da realização de experimentos na arquitetura Opteron. Foram realizados experimentos com as quatro políticas implementadas pela interface MAI, sendo os resultados divididos em duas partes: comparação entre as duas polí-

ticas do tipo *bind* (*bind_block* e *bind_all*) e comparação entre as duas políticas do tipo *cyclic* (*cyclic* e *cyclic_block*).

Políticas bind_all e bind_block

A Figura 23 apresenta um quadro comparativo dos *speed-ups* obtidos ao utilizar as políticas *bind_block* e *bind_all* na arquitetura Opteron. Cada curva representa o *speed-up* de uma política utilizando-se um determinado valor para o raio. Logo, seis curvas serão apresentadas em cada caso de estudo (duas políticas e três valores de raio para cada uma).

Ao analisar os resultados da Figura 23 conclui-se que a aplicação destas políticas resultou em um ganho de desempenho considerável comparado ao resultado da solução OpenMP-ICTM (5.2.2). No geral, a política *bind_block* apresentou melhores resultados. Porém, ao comparar curvas de mesmo raio (entre políticas diferentes) percebe-se que a diferença de desempenho não é tão significativa em diversos casos.

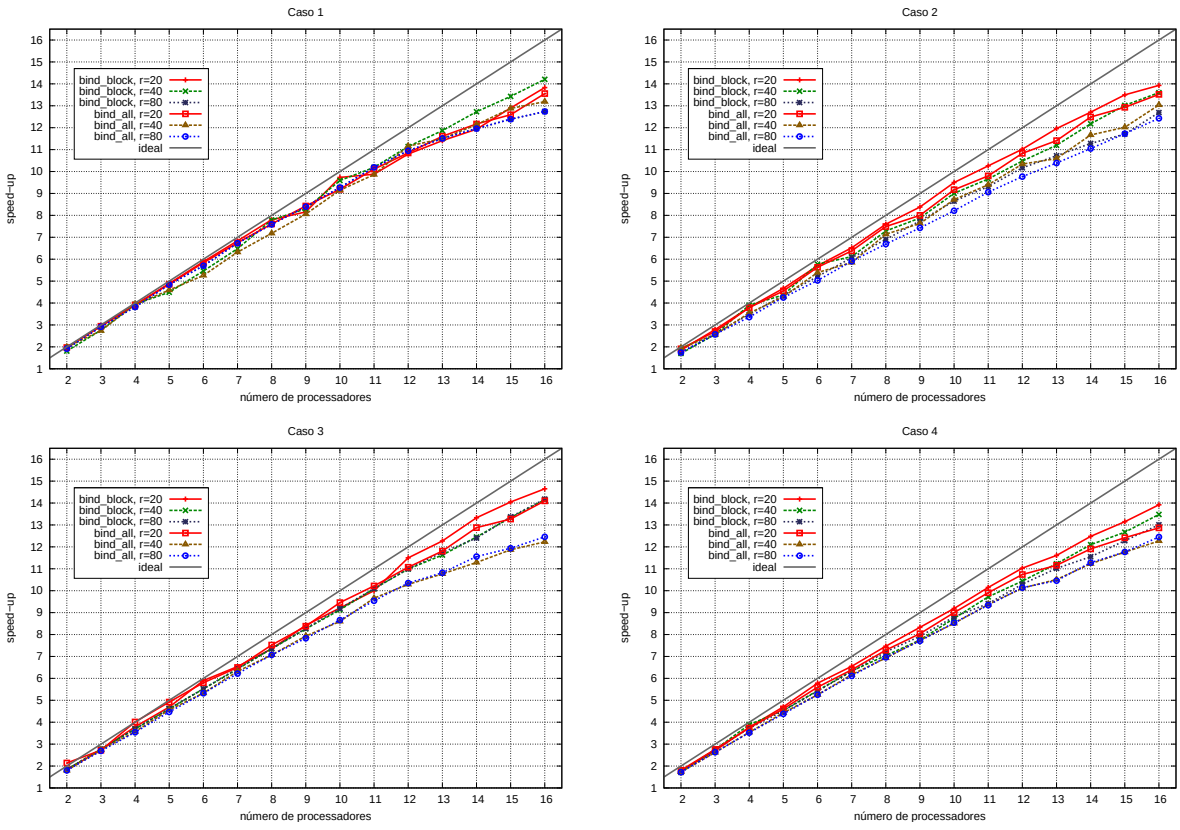

Figura 23 – Desempenho das políticas *bind_block* e *bind_all* - Opteron.

Como explicado na Seção 6.1, a diferença entre as políticas *bind_block* e *bind_all* está no fato de que na segunda o *kernel* do Linux será responsável por determinar onde cada página será alocada. Como esta arquitetura possui um baixo fator NUMA, não ocorrem grandes diferenças de desempenho em relação a estas duas políticas. Isto pode ser verificado também pela proximidade entre as curvas nos gráficos de *speed-up*.

Políticas cyclic e cyclic_block

O desempenho das políticas *cyclic* e *cyclic_block* pode ser observado na Figura 24. Em contraste com os resultados nesta arquitetura, onde as políticas *bind_block* e *bind_all* foram aplicadas, aqui percebe-se um maior ganho de desempenho. Além disso, os resultados mostram curvas mais lineares e bastante próximas à ideal. À medida em que o caso de estudo é aumentado percebe-se uma pequena perda de desempenho, porém ainda assim os ganhos são superiores aos da solução OpenMP-ICTM.

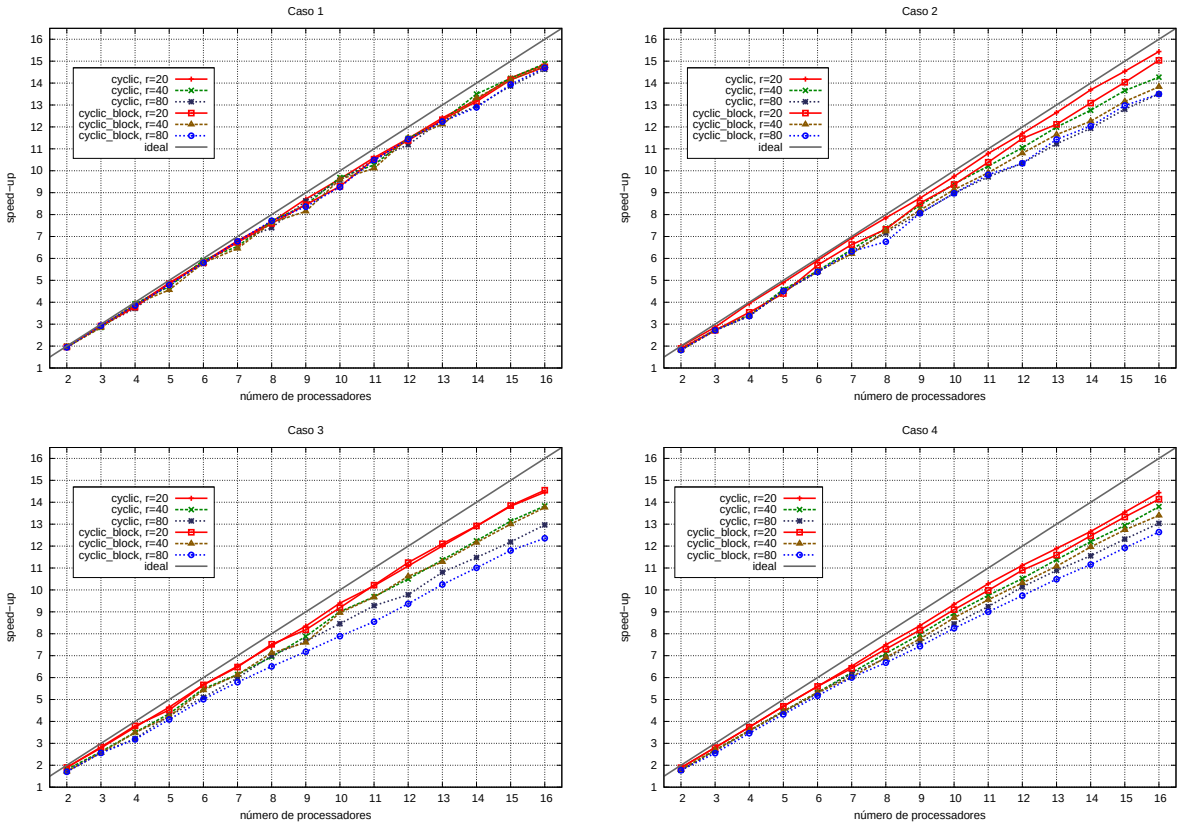

Figura 24 – Desempenho das políticas *cyclic* e *cyclic_block* - Opteron.

O baixo fator NUMA desta arquitetura permite a utilização de uma estratégia que otimize a largura de banda. Isto é feito pela política *cyclic*, onde as páginas de memória serão distribuídas entre os blocos de memória da arquitetura através de um processo circular. Isto faz com que seja possível reduzir a disputa de acesso entre os processadores ao mesmo bloco de memória, pois os dados a serem acessados por processadores de um mesmo nodo não necessariamente estarão armazenados neste nodo.

Como dito na Seção 6.1, a política *cyclic_block* distribui conjuntos de páginas de forma cíclica (e não página à página como na política *cyclic*). Nos resultados mostrados na Figura 24 a política *cyclic_block* foi parametrizada para distribuir conjuntos compostos por 10 páginas. Como pode ser observado, os resultados foram piores quando comparados a política *cyclic*. Diversos outros experimentos com 20, 30, 40 e 50 páginas por conjunto foram realizados e

apresentaram resultados ainda piores. Isto pode ser interpretado da seguinte forma: uma granulosidade fina é mais apropriada para esta aplicação e esta arquitetura.

6.2.2 Arquitetura Itanium 2

Os resultados apresentados nesta seção foram extraídos de experimentos na arquitetura Itanium 2. Novamente, estes estão divididos em duas partes: comparação entre as duas políticas do tipo *bind* e comparação entre as duas políticas do tipo *cyclic*.

Políticas bind_all e bind_block

A comparação dos *speed-ups* obtidos com a aplicação das políticas *bind_all* e *bind_block* é mostrada na Figura 25. A política *bind_block* apresentou um desempenho melhor e curvas bem próximas da ideal. Devido ao alto fator NUMA desta arquitetura, é possível que um maior ganho de desempenho seja obtido quando os dados são armazenados próximos aos processadores que os utilizam. Desta forma, diminui-se o custo de comunicação decorrente dos acessos remotos (dados em blocos de memória distantes). A variação do valor do raio não mostrou interferir de forma significativa no desempenho da política *bind_block*, o que pode ser comprovado pela proximidade das curvas com raio 20, 40 e 80.

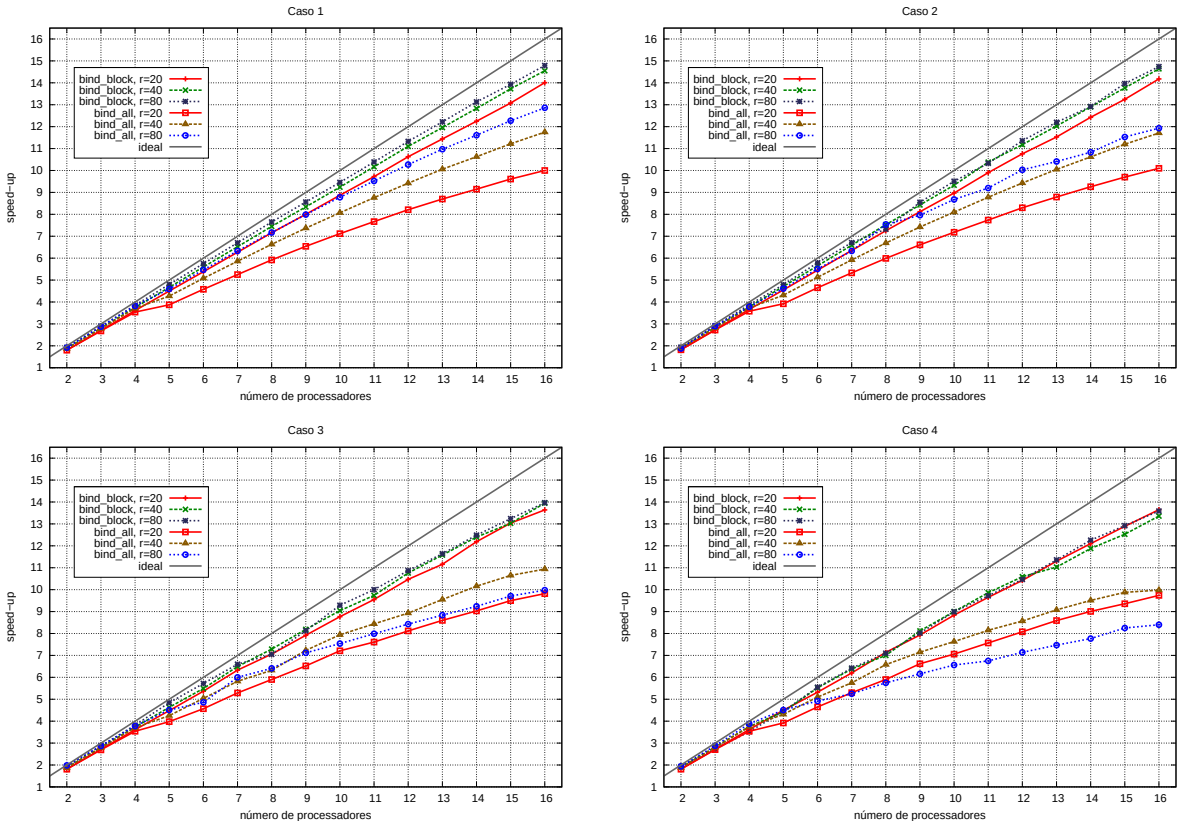

Figura 25 – Desempenho das políticas *bind_block* e *bind_all* - Itanium 2.

Diferentemente dos resultados da arquitetura Opteron, aqui é possível observar uma diferença significativa de desempenho entre as políticas *bind_block* e *bind_all*. Novamente o fator NUMA é determinante: como a política *bind_all* deixa a cargo do *kernel* do Linux a alocação física das páginas, caso estas não estejam próximas dos processadores que as acessam, um grande tempo será desperdiçado para acessá-las em blocos de memória remotos, impactando assim no desempenho. À medida em que o caso de estudo é aumentado a diferença de desempenho entre as políticas *bind_block* e *bind_all* também aumenta, pois há uma maior chance do *kernel* alocar os dados em blocos de memória distantes.

Políticas cyclic e cyclic_block

Os *speed-ups* após a aplicação das políticas *cyclic* e *cyclic_block* na arquitetura Itanium 2 são mostrados na Figura 26. No geral, as políticas *cyclic* e *cyclic_block* apresentaram resultados melhores que a política *bind_all*. Porém, é visível que esta estratégia de distribuição de páginas de memória não é a mais adequada para esta arquitetura com alto fator NUMA. Devido a isto, ambas as políticas mostraram desempenho inferior à política *bind_block*.

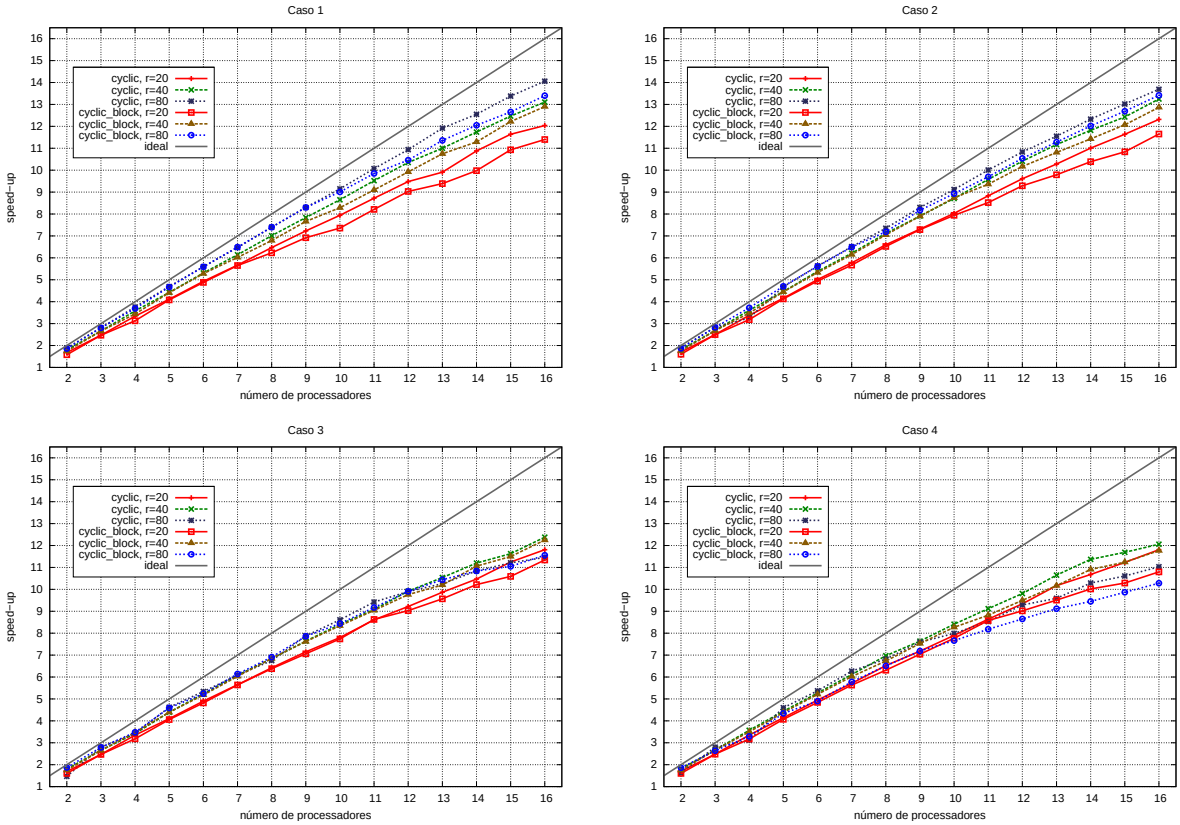

Figura 26 – Desempenho das políticas *cyclic* e *cyclic_block* - Itanium 2.

Novamente, o uso de uma granulosidade maior resultou em desempenhos piores. Além disso, o aumento do tamanho do caso de estudo impactou drasticamente no desempenho destas políticas nesta arquitetura. Quanto maior a quantidade de dados, maiores serão as chances de

ocorrerem acessos remotos devido à alocação de páginas de memória distantes dos processadores que as acessam.

6.3 Discussão

Este capítulo apresentou a otimização da solução OpenMP-ICTM através da utilização de afinidade (NUMA-ICTM). A afinidade de *threads* e memória foi aplicada utilizando-se a interface MAI, explorando-se as quatro políticas de memória que ela implementa. Uma análise de desempenho foi feita comparando-se os resultados nas duas arquiteturas NUMA descritas na Seção 3.3. Os experimentos realizados demonstraram que determinados tipos de políticas de memória são mais adaptadas para máquinas com alto fator NUMA (como por exemplo a política *bind_block*). Neste tipo de política, é possível alocar fisicamente páginas de memória em blocos específicos de forma a reduzir a distância entre dados e processadores os quais os acessam. Por outro lado, políticas cíclicas (como por exemplo a *cyclic*) são mais indicadas para arquiteturas com baixo fator NUMA, onde uma estratégia que prioriza largura de banda pode resultar em um melhor desempenho.

Uma avaliação geral em termos de eficiência da mesma forma apresentada na seção que discutiu os resultados da solução OpenMP-ICTM (Seção 5.3) é mostrada nas Figuras 27 e 28. Porém, como neste capítulo foram realizados experimentos com as quatro políticas de memória em ambas arquiteturas, optou-se por apresentar somente a eficiência média da solução NUMA-ICTM com a política que apresentou melhores resultados em cada arquitetura. Novamente foi observado um pequeno desvio padrão no cálculo da média dos *speed-ups* e estes podem ser vistos nos Apêndices A e B.

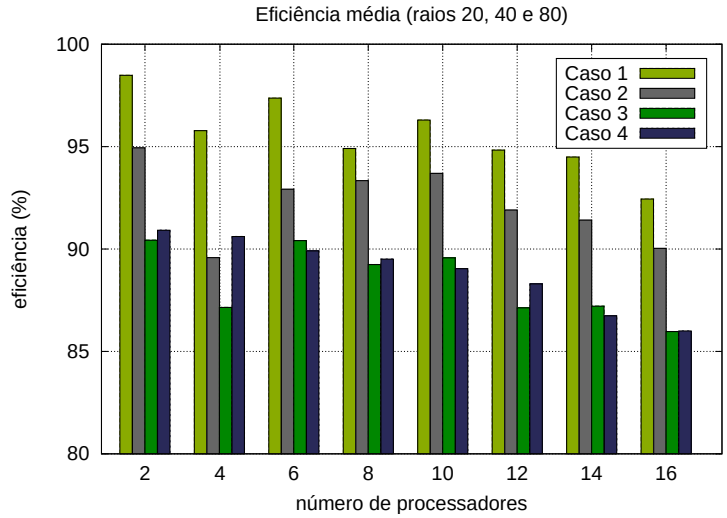

Figura 27 – Eficiência média da política *cyclic* - melhor política para a arquitetura Opteron.

A Figura 27 apresenta a eficiência média da solução NUMA-ICTM na arquitetura Opte-

ron. Neste caso, a política de memória que apresentou melhores resultados foi a *cyclic*. Como dito anteriormente, esta estratégia permite um aumento do número de acessos concorrentes a diferentes blocos de memória, reduzindo a disputa de acesso ao mesmo bloco.

Os resultados indicam que os maiores ganhos foram obtidos com casos de estudo menores (onde há um grande aumento da eficiência). Todavia, pode-se perceber também ganhos nos casos de estudo maiores. Outra consideração importante a ser feita está relacionada ao número de processadores: ao aumentar o número de processadores, a importância da utilização de uma política de memória que permita um melhor uso da arquitetura também aumenta. Isto pode ser percebido ao comparar as eficiências médias da solução OpenMP-ICTM e NUMA-ICTM a partir de 8 processadores. Nesta última, há uma atenuação da perda de desempenho de 8 a 16 processadores (a solução OpenMP-ICTM apresenta uma queda considerável de desempenho neste intervalo). Isto mostra que a solução NUMA-ICTM é mais escalável.

A Figura 28 apresenta a eficiência média da solução NUMA-ICTM na arquitetura Itanium 2. Neste caso, a política de memória que apresentou melhores resultados foi a *bind_block*. O problema do alto fator NUMA desta arquitetura é atenuado mantendo-se os dados próximos aos processadores que os acessam.

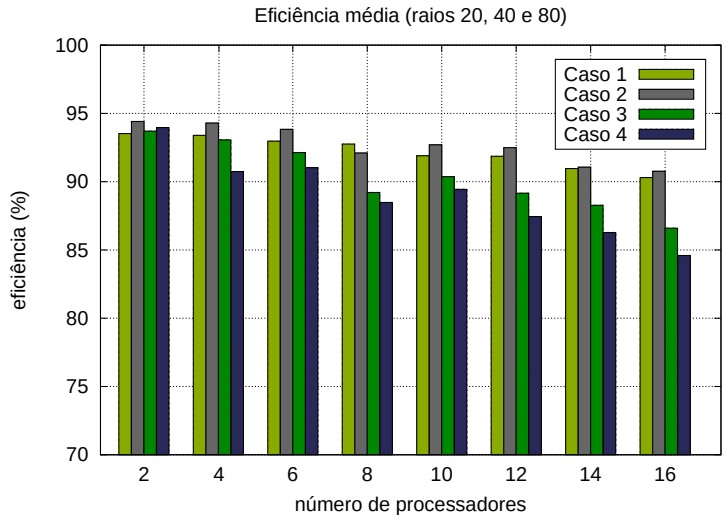

Figura 28 – Eficiência média da política *bind_block* - melhor política para a arquitetura Itanium 2.

Como esperado, o ganho de desempenho da solução NUMA-ICTM na arquitetura Itanium 2 foi mais significativo. Devido ao alto fator NUMA desta arquitetura, aumenta-se a importância de aplicar uma política que mantenha os dados mais próximos dos processadores que os acessam. Em comparação com a solução OpenMP-ICTM nesta arquitetura, percebe-se aqui um grande aumento da escalabilidade, principalmente quando são utilizados mais do que 8 processadores.

7 Conclusão

Este trabalho apresentou a NUMA-ICTM: uma solução paralela do ICTM para máquinas NUMA explorando estratégias de alocação de memória. Para isto, uma primeira solução denominada OpenMP-ICTM foi desenvolvida utilizando-se OpenMP (uma API desenvolvida para implementação de aplicações paralelas para multiprocessadores com memória centralizada). Nesta solução, as arquiteturas alvo (NUMA) foram vistas como sendo UMA, ou seja, ignorou-se completamente o conceito de "distância" no acesso à memória, característica marcante deste tipo de arquitetura. O objetivo principal deste primeiro passo foi de comprovar a necessidade de utilização de estratégias de alocação de memória. Posteriormente, esta solução foi otimizada incluindo-se a interface MAI, a qual permite aplicar políticas de memória e *threads* de uma forma mais alto nível que a NUMA API. A esta nova solução deu-se o nome de NUMA-ICTM, a qual permite a utilização de diferentes políticas de memória. O comportamento de cada política (desempenho) foi analisado em duas arquiteturas NUMA bastante distintas, mostrando-se que através da afinidade de memória é possível obter-se resultados mais próximos do ideal.

A seguir, será feito um paralelo entre a solução apresentada neste trabalho e os trabalhos relacionados (HPC-ICTM). Posteriormente, uma comparação final entre a solução NUMA-ICTM e a solução OpenMP-ICTM será feita, mostrando-se o quanto, na média, a solução NUMA-ICTM é melhor que a OpenMP-ICTM. Finalmente, uma breve avaliação da interface MAI será feita, onde serão mostrados os prós e contras de utilizá-la. A última seção deste capítulo tratará sobre trabalhos futuros.

7.1 NUMA-ICTM x HPC-ICTM

A solução descrita neste trabalho (NUMA-ICTM) e as soluções descritas em [5] e [7] (HPC-ICTM) apresentam propostas diferenciadas de paralelizar o ICTM. A principal diferença se dá na escolha das arquiteturas alvo. O HPC-ICTM foi proposto para multicomputadores, ou seja, são arquiteturas caracterizadas pelo não compartilhamento de memória (a comunicação se dá através de trocas de mensagens entre processos). Por outro lado, a NUMA-ICTM foi proposta para multiprocessadores: arquiteturas caracterizadas pelo compartilhamento de memória.

A escolha da arquitetura alvo é de grande importância e irá definir as estratégias a serem utilizadas no processo de paralelização da aplicação. No contexto de multicomputadores, existe uma grande preocupação em reduzir-se o número de mensagens a serem trocadas entre pro-

cessadores, pois este é um fator importante e diretamente relacionado ao desempenho final da solução paralela. Devido a isto, busca-se estratégias que permitem a realização de processamentos independentes em paralelo, reduzindo-se assim a necessidade por troca de informações.

Uma das grandes vantagens da solução HPC-ICTM é a de necessitar um tipo de arquitetura de mais baixo custo: *clusters*. Desta forma, é possível que para alguns casos um menor investimento seja necessário, trazendo resultados bastante interessantes. Porém, dependendo das propriedades da região a ser categorizada (nesta solução o número de camadas do modelo, dimensão da região e tamanho do raio) é provável que a decomposição em camadas não seja a melhor alternativa. A decomposição em camadas exige pouca troca de mensagens mas somente será vantajosa se o modelo de entrada possuir diversas camadas (o número de processos escravos é igual ao número de camadas do modelo). Já na decomposição por funções, devido a dependência entre etapas do processo de categorização, somente 4 processos poderão executar funções do modelo em paralelo. Por fim, na decomposição por domínios, caso a dimensão da região seja muito grande, uma maior quantidade de troca de mensagens será necessária.

Por outro lado, a solução NUMA-ICTM apresenta, como uma de suas grandes vantagens, a possibilidade de tratar regiões extremamente grandes com um importante ganho de desempenho. Diferentemente da solução HPC-ICTM, esta define somente um modelo de decomposição do problema que oferece ganhos independentemente das características da região de entrada. A categorização de regiões extremamente grandes se torna possível devido à grande disponibilidade de memória oferecida pelas máquinas do tipo NUMA. Além disso, este tipo de arquitetura é uma tendência forte atualmente e seus conceitos estão sendo aplicados em *desktops*. Como desvantagem desta solução aponta-se o alto investimento necessário em uma máquina deste porte. Em comparação com *clusters*, este tipo de arquitetura é muito mais custosa, porém muitas vezes necessária para a computação de determinados tipos de problema. Acredita-se que o ICTM é uma aplicação que pode ser melhor adaptada para arquiteturas NUMA devido à quantidade de dados a serem armazenados, à disponibilidade dos mesmos durante o processo de categorização e por exigir grande capacidade de processamento.

7.2 NUMA-ICTM x OpenMP-ICTM

O desempenho de uma solução paralela de uma aplicação em máquinas NUMA está basicamente relacionado a dois fatores: as características da máquina e da aplicação. As principais características que descrevem uma máquina NUMA são o fator NUMA, o número de processadores em cada nodo e o tamanho da memória. Máquinas que possuem um baixo fator NUMA normalmente possuem uma maior quantidade de nodos e poucos processadores por nodo. Isto permite a possibilidade de explorar diversos acessos simultâneos a diferentes blocos de memória, reduzindo-se assim a disputa de diversos processadores a um mesmo bloco. Por outro lado, máquinas com alto fator NUMA são normalmente organizadas em uma estrutura com poucos

nodos e muitos processadores por nodo, tendo-se em vista que o tempo de acesso a um bloco de memória distante é muito maior.

As características da aplicação alvo também influenciam no desempenho da solução paralela. Quanto mais acessos a dados forem necessários, maior a importância a ser dada para uma melhor distribuição dos mesmos nos diversos blocos de memória. Outro fator importante é o padrão de acesso aos dados, que pode ser regular ou irregular. Aplicações com um padrão de acesso regular normalmente apresentam bons resultados em máquinas com alto fator NUMA, visto que a regularidade do acesso a dados permite uma melhor distribuição dos dados, deixando-os mais próximos dos processadores que os utilizam. Um exemplo de uma aplicação com padrão de acesso regular aos dados é o ICTM. Como sabe-se de antemão a forma com que os dados serão acessados, pode-se definir a melhor localização dos mesmos de forma a extrair ao máximo os benefícios oferecidos pela máquina alvo. Isto foi comprovado pelos resultados obtidos na arquitetura Itanium 2, onde a NUMA-ICTM mostrou resultados bem mais expressivos em comparação com a solução OpenMP-ICTM. Por outro lado, aplicações com um padrão de acesso irregular podem apresentar resultados melhores em arquiteturas com baixo fator NUMA, pois o custo do acesso a dados distantes não é muito maior.

Mostra-se importante salientar também o fato de que não existe uma melhor política de memória. A escolha da política a ser aplicada está relacionada às características da máquina alvo e da aplicação. Neste contexto, o fator NUMA e o padrão de acesso aos dados são informações importantes que devem ser levadas em consideração para a escolha da política mais adequada.

Para que possa ser feita uma análise de quanto (na média) a solução NUMA-ICTM é melhor que a OpenMP-ICTM, o seguinte procedimento foi feito:

1. Média dos resultados com raio 20, 40 e 80: foi calculado um *speed-up* médio dos experimentos com raio 20, 40 e 80 para cada combinação NUMA-ICTM $\times$ Arquitetura $\times$ Caso de Estudo $\times$ Política e OpenMP-ICTM $\times$ Arquitetura $\times$ Caso de Estudo;

2. Média dos *speed-ups* da solução OpenMP-ICTM: foi feita a média dos *speed-ups* de 2 à 16 processadores para a solução OpenMP-ICTM para cada caso de estudo em cada arquitetura, resultando em 8 *speed-ups* médios (4 casos de estudo e 2 arquiteturas);

3. Média dos *speed-ups* da solução NUMA-ICTM: foi feita a média dos *speed-ups* de 2 à 16 processados para a solução NUMA-ICTM para cada caso de estudo e política de memória em cada arquitetura, resultando em 32 *speed-ups* médios (4 casos de estudo, 4 políticas e 2 arquiteturas).

A comparação dos resultados entre as duas soluções foi feita utilizando-se a seguinte fórmula, onde A significa "arquitetura", C significa "caso de estudo" e P significa "política":

$$Ganho_{A,C,P} = \frac{(SpMedNUMA\_ICTM_{A,C,P} - SpMedOpenMP\_ICTM_{A,C}) * 100}{SpMedOpenMP\_ICTM_{A,C}} \quad (7.1)$$

Esta equação fornecerá a seguinte informação: quanto (na média) a solução NUMA-ICTM é melhor que a solução OpenMP-ICTM em termos percentuais. Um percentual negativo para uma combinação Arquitetura X Caso de Estudo X Política de memória indica que a solução NUMA-ICTM apresentou resultados piores que a solução OpenMP-ICTM para esta combinação.

Salienta-se que estas médias servirão somente para a comparação entre as duas soluções de forma bastante aproximada, dando assim uma idéia geral de quanto a solução otimizada é melhor. A Tabela 2 apresenta o resultado da aplicação da Equação 7.1 $\forall(A, C, P)$, onde $SpMedNUMA\_ICTM_{A,C,P}$ e $SpMedOpenMP\_ICTM_{A,C}$ representam a média dos *speed-ups* **de 2 a 16 processadores** das soluções NUMA-ICTM e OpenMP-ICTM, respectivamente.

Tabela 2 – Ganho de desempenho da solução NUMA-ICTM em relação à solução OpenMP-ICTM (média dos *speed-ups* de 2 a 16 processadores).

Políticas	**Caso 1**		**Caso 2**		**Caso 3**		**Caso 4**	
	Opteron	Itanium 2	Opteron	Itanium 2	Opteron	Itanium 2	Opteron	Itanium 2
bind_all	16,89%	-1,34%	7,08%	-2,22%	5,70%	-7,87%	-2,24%	-9,01%
bind_block	20,49%	**15,01%**	10,56%	**15,36%**	11,08%	**12,15%**	1,69%	**15,09%**
cyclic	**23,73%**	6,86%	**10,58%**	6,77%	**11,72%**	1,11%	**1,86%**	4,61%
cyclic_block	18,20%	3,21%	8,82%	4,28%	9,11%	0,08%	0,43%	0,21%

Como pode ser visto na Tabela 2, a política *bind_block* apresentou melhores resultados na arquitetura Itanium 2 (ganhos entre $12,15\%$ a $15,36\%$), enquanto a política *cyclic* foi a melhor para a arquitetura Opteron (ganhos entre $1,86\%$ e $23,73\%$). A política *bind_all* em alguns casos apresentou resultados piores que a solução OpenMP-ICTM (representados na tabela por valores negativos).

Como observado anteriormente, a melhora de desempenho da solução NUMA-ICTM é mais significante à medida em que são utilizados mais processadores. Isto é comprovado pela Tabela 3, onde apresenta-se o resultado da aplicação da Equação 7.1 $\forall(A, C, P)$, porém, as médias dos *speed-ups* utilizadas compreendem somente o intervalo **de 8 a 16 processadores**.

Tabela 3 – Ganho de desempenho da solução NUMA-ICTM em relação à solução OpenMP-ICTM (média dos *speed-ups* de 8 a 16 processadores).

Políticas	**Caso 1**		**Caso 2**		**Caso 3**		**Caso 4**	
	Opteron	Itanium 2	Opteron	Itanium 2	Opteron	Itanium 2	Opteron	Itanium 2
bind_all	21,83%	-1,52%	11,09%	-2,99%	9,14%	-9,69%	-0,63%	-10,93%
bind_block	26,25%	**17,40%**	14,92%	**17,41%**	15,86%	**13,60%**	3,64%	**17,87%**
cyclic	**30,36%**	8,59%	**15,42%**	8,23%	**16,75%**	1,74%	**4,32%**	5,83%
cyclic_block	29,39%	4,27%	13,50%	5,21%	14,17%	0,47%	1,70%	0,92%

Como esperado, considerando-se somente os resultados de 8 a 16 processadores o ganho de desempenho em relação a solução OpenMP-ICTM foi maior para as políticas com melhor resultado em cada arquitetura (entre $4,32\%$ e $30,36\%$ na arquitetura Opteron e entre $13,60\%$ e

$17,87\%$ na arquitetura Itanium 2). Isto comprova a melhor escalabilidade da solução NUMA-ICTM, a qual apresenta melhores resultados quando um maior número de processadores é utilizado.

Por fim, a Tabela 4 apresenta o quanto a solução NUMA-ICTM é melhor que a OpenMP-ICTM quando considera-se somente os resultados **com 16 processadores**. Novamente, os resultados são superiores visto que com muitos processadores a solução NUMA-ICTM apresenta um desempenho bem mais interessante.

Tabela 4 – Ganho de desempenho da solução NUMA-ICTM em relação a solução OpenMP-ICTM (média dos *speed-ups* com somente 16 processadores).

Políticas	**Caso 1**		**Caso 2**		**Caso 3**		**Caso 4**	
	Opteron	Itanium 2	Opteron	Itanium 2	Opteron	Itanium 2	Opteron	Itanium 2
bind_all	28,65%	3,54%	24,13%	-0,35%	18,22%	-8,12%	3,81%	-9,35%
bind_block	37,67%	**29,68%**	28,08%	**28,67%**	31,01%	**24,23%**	11,61%	**30,85%**
cyclic	**44,53%**	17,31%	**31,72%**	15,97%	**31,72%**	6,57%	**13,99%**	12,49%
cyclic_block	44,07%	12,80%	10,95%	12,05%	29,17%	5,04%	10.95%	5,85%

No geral, a utilização de políticas de memória trouxe grandes benefícios. Os maiores ganhos puderam ser observados quando um maior número de processadores foi utilizado. A queda drástica de desempenho apresentada pela solução OpenMP-ICTM a partir de 8 processadores não foi observada na solução NUMA-ICTM. A aplicação de políticas adequadas para as arquiteturas alvo e características do ICTM possibilitou uma melhor utilização dos recursos, resultando em expressivos ganhos de desempenho.

7.3 Avaliação da interface MAI

Um dos fatores os quais motivaram a realização deste trabalho foi a possibilidade da utilização e avaliação da MAI: uma interface inovadora que está sendo desenvolvida no INRIA para controle de políticas de memória em arquiteturas NUMA. Acredita-se que o trabalho aqui apresentado serviu como um caso de estudo onde a interface pôde ser aplicada a um problema real. De acordo com os resultados mostrados, a utilização da interface trouxe inúmeros benefícios, aumentando-se o desempenho da aplicação alvo.

Acredita-se que os mesmos resultados obtidos neste trabalho também poderiam ter sido alcançados utilizando-se a NUMA API. Porém, devido ao baixo nível de abstração (máscaras de *bits*, chamadas de sistema, entre outras peculiaridades) um maior tempo seria necessário para que resultados de mesma qualidade fossem conquistados. A NUMA API é bastante eficaz, porém exige um maior conhecimento dos programadores. Como exemplos dos benefícios da MAI destaca-se:

- Funções para alocação de memória: abstraem a utilização da chamada de sistema `mmap()`.

Como possuem parâmetros similares a função `malloc()` torna-se simples adaptar a alocação de dados da solução original para estas oferecidas pela interface;

- Funções de Controle de Políticas de Memória: forma prática de aplicar diferentes políticas de memória. Abstraem máscaras de *bits* necessárias para especificar em quais blocos de memória as políticas serão aplicadas;
- Funções de Controle de Políticas de *Threads*: da mesma forma que as funções de controle de políticas de memória, abstraem máscaras de *bits* necessárias para especificar em quais processadores cada *thread* deverá ser executada.

A utilização de arquivos de configuração facilita a especificação da arquitetura alvo, como por exemplo quais processadores e nodos deverão ser utilizados, reduzindo drasticamente o número de linhas a serem codificadas. Infelizmente, a migração de processos não foi utilizada neste trabalho, pois o ICTM possui um padrão de acesso a dados regular. Logo, esta funcionalidade da interface não pode ser testada.

7.4 Trabalhos futuros

Levando-se em conta os benefícios da solução HPC-ICTM (para *clusters* e *grids*) e aqueles apresentados neste trabalho (NUMA-ICTM), pontua-se como uma possibilidade de trabalho futuro a integração destas duas soluções, oferecendo assim uma proposta para *cluster* de máquinas NUMA. Neste contexto, o padrão MPI poderia ser utilizado para a comunicação entre diferentes nodos do *cluster* enquanto a API OpenMP em conjunto com a interface MAI poderiam ser utilizadas para a paralelização interna em cada nodo.

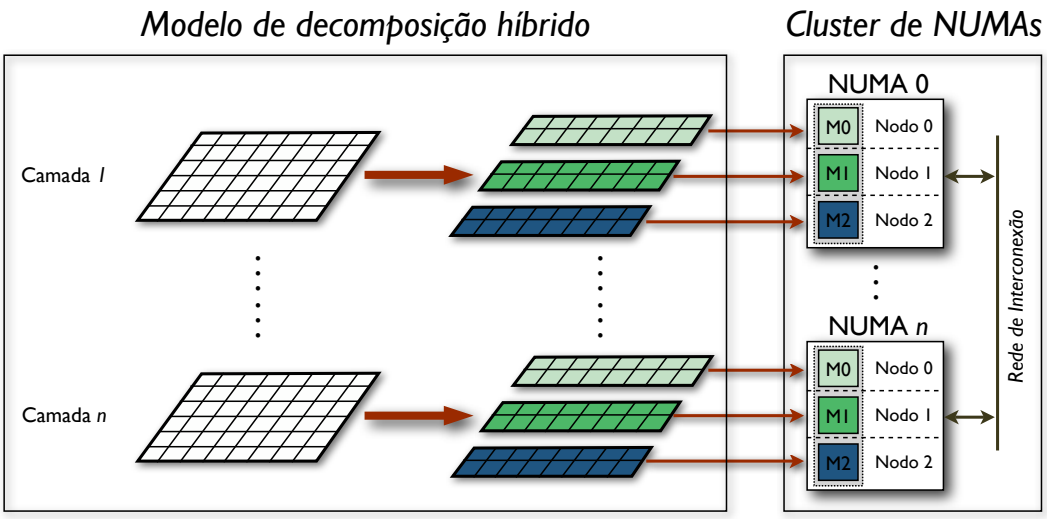

Figura 29 – Proposta de integração do NUMA-ICTM com o HPC-ICTM (*cluster* de máquinas NUMA).

Uma estratégia possível seria a utilização da decomposição em camadas proposta pelos trabalhos relacionados. Neste caso, cada processo escravo processaria uma determinada camada do modelo, paralelizando-o da forma apresentada neste trabalho. A Figura 29 mostra uma idéia geral desta proposta de integração do HPC-ICTM com a NUMA-ICTM. Acredita-se que com esta integração seja possível aumentar ainda mais os casos de estudo de entrada, sendo possível assim, a categorização de regiões geográficas ainda maiores.

Referências

[1] COBLENTZ, D. et al. Towards Reliable Sub-Division of Geological Areas: Interval Approach. In: *NAFIPS '00: Proceedings of the 19th International Meeting of the North American Fuzzy Information Processing Society*. Atlanta, GA, USA: IEEE Computer Society, 2000. p. 368–372.

[2] AGUIAR, M. S. de et al. 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: Instituto Nacional de Pesquisas Espaciais, 2004. p. 437–454.

[3] A NUMA API for LINUX. Disponível em: http://www.novell.com/collateral/4621437/4621437.pdf. Acesso em: 5 de nov. 2008.

[4] MU, T. et al. Interactive Locality Optimization on NUMA Architectures. In: *SOFTVIS '03: Proceedings of the ACM 2003 Symposium on Software Visualization*. San Diego, CA, USA: ACM, 2003. p. 133–141.

[5] SILVA, R. K. S. et al. HPC-ICTM: a Parallel Model for Geographic Categorization. In: *JVA '06: Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing*. Sofia, Bulgaria: IEEE Computer Society, 2006. p. 143–148.

[6] QUINN, M. J. *Parallel Programming in C with MPI and OpenMP*. New York, NY: McGraw-Hill, 2004. 544 p.

[7] SILVA, R. K. S. et al. Extending the HPC-ICTM Geographical Categorization Model for Grid Computing. In: *PARA '06: Proceedings of the Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop*. Umeå, Sweden: Springer, 2006. (Lecture Notes in Computer Science, v. 4699), p. 850–859.

[8] ANDRADE, N. et al. Discouraging Free Riding in a Peer-to-Peer CPU-Sharing Grid. In: *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*. Honolulu, Hawaii, USA: IEEE Computer Society, 2004. p. 129–137.

[9] BRASILEIRO, F. et al. Bridging the High Performance Computing Gap: the OurGrid Experience. In: *CCGRID '07: Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*. Rio de Janeiro, RJ, Brazil: IEEE Computer Society, 2007. p. 817–822.

[10] NÓBREGA-JÚNIOR, N.; ASSIS, L.; BRASILEIRO, F. Scheduling CPU-Intensive Grid Applications Using Partial Information. In: *ICPP '08: Proceedings of the 37th International Conference on Parallel Processing*. Portland, Oregon, USA: IEEE Computer Society, 2008. p. 262–269.

[11] RIBEIRO, C. P.; MÉHAUT, J.-F. *MAI: Memory Affinity Interface*. Technical Report 0359, INRIA, 2008.

[12] MCMASTER, R. B.; USERY, E. L. *A Research Agenda for Geographic Information Science*. USA: CRC Press, 2004. 414 p.

[13] SILVA, R. K. S. *HPC-ICTM: Um Modelo de Alto Desempenho para Categorização de Áreas Geográficas*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio Grande do Sul, 2006.

[14] WOLFRAM, S. *Cellular Automata and Complexity: Collected Papers*. USA: Westview Press, 1994. 610 p.

[15] KEARFOTT, R. B.; KREINOVICH, V. *Applications of Interval Computations*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1996. 458 p.

[16] MOORE, R. E. *Methods and Applications of Interval Analysis*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1979. 190 p.

[17] WRIGHT, R. S.; LIPCHAK, B. *OpenGL SuperBible*. 3rd edition. ed. Indianapolis, Indiana, USA: Sams, 2004. 1200 p.

[18] RAMANATHAN, R. M. *Intel Multi-Core Processors: Leading the Next Digital Revolution*. Santa Clara, CA, USA: Technology@Intel Magazine, 2006.

[19] SNIR, M.; YU, J. *On the Theory of Spatial and Temporal Locality*. Technical Report UIUCDCS-R-2005-2611, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[20] DONGARRA, J. et al. *The Sourcebook of Parallel Computing*. San Francisco, CA, USA: Morgan Kaufmann, 2002. 842 p.

[21] TORRELLAS, J.; LAM, H. S.; HENNESSY, J. L. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, IEEE Computer Society, Washington, DC, USA, v. 43, n. 6, p. 651–663, 1994.

[22] BHUYAN, L. N.; WANG, H.; IYER, R. Impact of CC-NUMA Memory Management Policies on the Application Performance of Multistage Switching Networks. *IEEE Transactions on Parallel & Distributed Systems*, IEEE Press, Piscataway, NJ, USA, v. 11, n. 3, p. 230–246, 2000.

[23] LEUNG, J.; KELLY, L.; ANDERSON, J. H. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Boca Raton, FL, USA: CRC Press, 2004. 1224 p.

[24] SQUILLANTE, M. S.; NELSON, R. D. Analysis of Task Migration in Shared-Memory Multiprocessor Scheduling. In: *SIGMETRICS '91: Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. San Diego, California, USA: ACM, 1991. p. 143–155.

[25] WILSON, K. M.; AGLIETTI, B. B. Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C. In: *SC '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. Denver, CO, USA: IEEE Computer Society, 2001. p. 98–150.

[26] LAROWE, J. R. P.; ELLIS, C. S. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, ACM, New York, NY, USA, v. 9, n. 4, p. 319–363, 1991.

[27] CORREA, M. L. M. *Algoritmo de Construção de Hierarquia de Domínios de Escalonamento Multinível para Máquinas NUMA*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio Grande do Sul, 2005.

[28] NICHOLS, B.; BUTTLAR, D.; FARRELL, J. P. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. Sebastopol, CA, EUA: O'Reilly & Associates, 1996. 267 p.

[29] DORMANNS, M. et al. A Programming Interface for NUMA Shared-Memory Clusters. In: *HPCN Europe '97: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*. Vienna, Austria: Springer-Verlag, 1997. (Lecture Notes in Computer Science, v. 1225), p. 698–707.

[30] CORPORATE Institute of Electrical and Electronics Engineers, Inc. Staff. *IEEE Standard for Scalable Coherent Interface, Science: IEEE Std. 1596-1992*. New York, NY, USA: IEEE Standards Office, 1993.

[31] MYRICOM Home Page. Disponível em: http://www.myri.com. Acesso em: 30 out. 2007.

[32] PACHECO, P. S. *Parallel Programming with MPI*. San Francisco, CA, EUA: Morgan Kaufmann, 1997. 418 p.

[33] MPICH2 Home Page. Disponível em: http://www-unix.mcs.anl.gov/mpi/mpich. Acesso em: 20 out. 2007.

[34] CLAUSS, C.; PÖPPE, M.; BEMMERL, T. Optimising MPI Applications for Heterogeneous Coupled Clusters with MetaMPICH. In: *PARALEC '04: Proceedings of the IEEE International Conference on Parallel Computingin Electrical Engineering*. Dresden, Germany: IEEE Computer Society, 2004. p. 7–12.

[35] BUNTINAS, D.; MERCIER, G.; GROPP, W. Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem. In: *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*. Singapore: IEEE Computer Society, 2006. p. 521–530.

[36] CHAI, L.; HARTONO, A.; PANDA, D. K. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In: *CLUSTER '06: Proceedings of the 2006 IEEE International Conference on Cluster Computing*. Barcelona, Spain: IEEE Computer Society, 2006. p. 1–10.

[37] SQUYRES, J. M.; LUMSDAINE, A. A Component Architecture for LAM/MPI. In: *PVM/MPI '03: 10th European PVM/MPI Users' Group Meeting*. Venice, Italy: Springer, 2003. (Lecture Notes in Computer Science, v. 2840), p. 379–387.

[38] GABRIEL, E. et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: *PVM/MPI '04: Proceedings of the 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary: Springer Berlin, 2004. (Lecture Notes in Computer Science, v. 3241), p. 97–104.

[39] MARATHE, J.; MUELLER, F. Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems. In: *PPOPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*. New York, NY, USA: ACM, 2006. p. 90–99.

[40] BELLOSA, F.; STECKERMEIER, M. The Performance Implications of Locality Information Usage in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, Academic Press, Inc., Orlando, FL, USA, v. 37, n. 1, p. 113–121, August 1996.

[41] ANTONY, J.; JANES, P.; RENDELL, A. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In: *HiPC '06: Proceedings of the 13th International Conference, High Performance Computing*. Bangalore, India: Springer, 2006. (Lecture Notes in Computer Science, v. 4297), p. 338–352.

[42] BIRCSAK, J. et al. Extending OpenMP for NUMA Machines. In: *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. Dallas, Texas, United States: IEEE Computer Society, 2000. p. 48–48.

[43] VERGHESE, B. et al. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In: *ASPLOS-VII: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, Massachusetts: ACM, 1996. p. 279–289.

APÊNDICE A – Médias e desvio padrão na arquitetura Opteron

OpenMP-ICTM

NP	Caso 1		Caso 2		Caso 3		Caso 4	
	média	**desvio**	**média**	**desvio**	**média**	**desvio**	**média**	**desvio**
2	1.98	0.01	1.97	0.01	1.95	0.02	1.96	0.02
3	2.94	0.02	2.89	0.04	2.99	0.04	2.91	0.00
4	3.85	0.07	3.82	0.02	3.97	0.04	3.88	0.02
5	4.72	0.02	4.73	0.01	4.90	0.05	4.85	0.06
6	5.52	0.14	5.57	0.09	5.75	0.05	5.81	0.12
7	6.26	0.21	6.33	0.17	6.68	0.19	6.73	0.23
8	6.95	0.32	7.07	0.25	7.35	0.25	7.46	0.23
9	7.51	0.54	7.74	0.37	8.04	0.29	8.26	0.34
10	8.03	0.58	8.38	0.62	8.68	0.39	9.04	0.48
11	8.35	0.76	8.87	0.68	9.20	0.56	9.72	0.54
12	8.74	0.99	9.37	0.95	9.68	0.74	10.31	0.59
13	9.16	1.08	9.72	1.11	10.06	0.89	10.84	0.59
14	9.49	1.20	10.01	1.21	10.40	0.96	11.26	0.71
15	9.86	1.29	10.28	1.26	10.75	1.03	11.75	0.74
16	10.23	1.25	10.47	1.28	10.94	1.05	12.07	0.79

NUMA-ICTM: política *bind_all*

NP	Caso 1		Caso 2		Caso 3		Caso 4	
	média	**desvio**	**média**	**desvio**	**média**	**desvio**	**média**	**desvio**
2	1.96	0.04	1.87	0.11	1.91	0.20	1.75	0.05
3	2.86	0.11	2.61	0.08	2.72	0.03	2.67	0.08
4	3.88	0.06	3.57	0.22	3.71	0.26	3.60	0.12
5	4.76	0.15	4.37	0.16	4.65	0.24	4.48	0.12
6	5.60	0.31	5.35	0.30	5.48	0.27	5.37	0.20
7	6.60	0.24	6.04	0.29	6.35	0.14	6.23	0.15
8	7.46	0.24	7.11	0.40	7.22	0.26	7.06	0.20
9	8.30	0.21	7.69	0.29	8.04	0.29	7.83	0.18
10	9.20	0.06	8.70	0.48	8.90	0.48	8.68	0.28
11	10.05	0.17	9.42	0.37	9.81	0.36	9.54	0.32
12	10.99	0.16	10.31	0.53	10.57	0.44	10.33	0.35
13	11.54	0.06	10.80	0.54	11.14	0.58	10.71	0.39
14	12.09	0.11	11.73	0.72	11.91	0.85	11.47	0.38
15	12.62	0.25	12.22	0.63	12.36	0.79	11.99	0.38
16	13.17	0.41	13.00	0.55	12.93	1.02	12.53	0.31

NUMA-ICTM: política *bind_block*

NP	Caso 1		Caso 2		Caso 3		Caso 4	
	média	**desvio**	**média**	**desvio**	**média**	**desvio**	**média**	**desvio**
2	1.92	0.09	1.76	0.08	1.84	0.01	1.75	0.03
3	2.89	0.11	2.66	0.12	2.76	0.02	2.74	0.03
4	3.93	0.01	3.73	0.22	3.73	0.06	3.79	0.09
5	4.76	0.23	4.46	0.18	4.64	0.04	4.59	0.10
6	5.74	0.24	5.56	0.31	5.65	0.22	5.55	0.20
7	6.71	0.20	6.24	0.25	6.47	0.06	6.40	0.13
8	7.79	0.01	7.27	0.35	7.35	0.01	7.24	0.22
9	8.40	0.30	8.00	0.33	8.32	0.09	8.00	0.30
10	9.67	0.08	9.05	0.42	9.18	0.04	8.92	0.24
11	10.11	0.21	9.74	0.48	10.06	0.05	9.76	0.37
12	11.03	0.20	10.56	0.43	11.16	0.29	10.59	0.39
13	11.93	0.55	11.29	0.62	11.89	0.34	11.29	0.30
14	12.52	0.53	12.06	0.72	12.73	0.52	12.04	0.47
15	13.34	0.40	12.74	0.93	13.58	0.40	12.70	0.43
16	14.09	0.22	13.41	0.64	14.33	0.28	13.47	0.45

NUMA-ICTM: política *cyclic*

NP	Caso 1		Caso 2		Caso 3		Caso 4	
	média	**desvio**	**média**	**desvio**	**média**	**desvio**	**média**	**desvio**
2	1.97	0.03	1.81	0.09	1.90	0.09	1.82	0.07
3	2.90	0.04	2.67	0.11	2.77	0.08	2.72	0.08
4	3.83	0.11	3.49	0.27	3.58	0.31	3.62	0.10
5	4.82	0.08	4.42	0.20	4.67	0.21	4.52	0.15
6	5.84	0.08	5.42	0.30	5.58	0.29	5.39	0.18
7	6.71	0.14	6.21	0.28	6.55	0.33	6.31	0.18
8	7.59	0.17	7.14	0.27	7.47	0.35	7.16	0.32
9	8.59	0.14	7.97	0.35	8.42	0.35	7.99	0.37
10	9.63	0.06	8.96	0.47	9.37	0.39	8.90	0.45
11	10.46	0.17	9.72	0.46	10.24	0.53	9.76	0.52
12	11.38	0.17	10.46	0.66	11.03	0.68	10.60	0.49
13	12.33	0.07	11.40	0.60	11.96	0.72	11.39	0.50
14	13.23	0.27	12.21	0.72	12.80	0.88	12.14	0.56
15	14.11	0.21	12.92	1.02	13.67	0.86	12.94	0.61
16	14.79	0.14	13.75	0.75	14.41	0.98	13.76	0.70

NUMA-ICTM: política *cyclic_block = 10*

NP	Caso 1		Caso 2		Caso 3		Caso 4	
	média	**desvio**	**média**	**desvio**	**média**	**desvio**	**média**	**desvio**
2	1.95	0.09	1.77	0.08	1.84	0.01	1.82	0.03
3	2.90	0.11	2.65	0.12	2.72	0.02	2.68	0.03
4	3.85	0.01	3.49	0.22	3.44	0.06	3.59	0.09
5	4.73	0.23	4.28	0.18	4.47	0.04	4.49	0.10
6	5.80	0.24	5.37	0.31	5.49	0.22	5.36	0.20
7	6.66	0.20	6.13	0.25	6.38	0.06	6.15	0.13
8	7.65	0.01	7.05	0.35	7.11	0.01	6.97	0.22
9	8.32	0.30	7.65	0.33	8.26	0.09	7.79	0.30
10	9.39	0.08	8.68	0.42	9.16	0.04	8.69	0.24
11	10.38	0.21	9.48	0.48	10.04	0.05	9.51	0.37
12	11.43	0.20	10.41	0.43	10.87	0.29	10.32	0.39
13	12.23	0.55	11.21	0.62	11.73	0.34	11.05	0.30
14	13.12	0.53	12.03	0.72	12.46	0.52	11.87	0.47
15	14.08	0.40	12.88	0.93	13.39	0.40	12.66	0.43
16	14.74	0.22	13.55	0.64	14.13	0.28	13.39	0.45

APÊNDICE B – Médias e desvio padrão na arquitetura Itanium 2

OpenMP-ICTM

NP	Caso 1		Caso 2		Caso 3		Caso 4	
	média	**desvio**	**média**	**desvio**	**média**	**desvio**	**média**	**desvio**
2	1.78	0.12	1.77	0.13	1.77	0.12	1.77	0.12
3	2.67	0.18	2.65	0.19	2.64	0.18	2.65	0.18
4	3.54	0.24	3.52	0.25	3.50	0.24	3.50	0.24
5	4.38	0.32	4.36	0.33	4.33	0.28	4.32	0.30
6	5.21	0.41	5.18	0.41	5.20	0.35	5.11	0.34
7	6.02	0.49	5.99	0.48	6.00	0.41	5.87	0.39
8	6.81	0.58	6.72	0.55	6.71	0.39	6.64	0.31
9	7.56	0.67	7.49	0.64	7.55	0.46	7.37	0.29
10	8.31	0.75	8.26	0.69	8.32	0.45	8.03	0.29
11	9.01	0.90	8.95	0.87	8.96	0.40	8.59	0.20
12	9.52	1.00	9.55	0.91	9.61	0.39	9.12	0.27
13	10.14	1.10	10.24	0.85	10.16	0.42	9.54	0.41
14	10.65	1.21	10.77	1.14	10.62	0.59	9.88	0.54
15	10.89	1.35	11.18	0.97	10.98	0.61	10.16	0.64
16	11.14	1.33	11.29	0.99	11.15	0.67	10.34	0.68

NUMA-ICTM: política *bind_all*

NP	Caso 1		Caso 2		Caso 3		Caso 4	
	média	**desvio**	**média**	**desvio**	**média**	**desvio**	**média**	**desvio**
2	1.87	0.06	1.87	0.04	1.89	0.08	1.88	0.06
3	2.78	0.10	2.81	0.09	2.78	0.09	2.79	0.09
4	3.69	0.14	3.70	0.10	3.68	0.12	3.72	0.17
5	4.24	0.36	4.28	0.35	4.25	0.27	4.15	0.20
6	5.05	0.45	5.09	0.43	4.82	0.24	4.69	0.39
7	5.82	0.54	5.86	0.50	5.71	0.37	5.43	0.27
8	6.58	0.63	6.74	0.78	6.21	0.27	6.14	0.55
9	7.30	0.73	7.33	0.68	6.96	0.38	6.81	0.65
10	7.99	0.83	7.99	0.75	7.56	0.37	6.98	0.69
11	8.65	0.93	8.58	0.75	8.01	0.41	7.49	0.70
12	9.30	1.03	9.25	0.88	8.50	0.41	7.86	0.84
13	9.91	1.14	9.75	0.85	9.00	0.50	8.38	0.83
14	10.46	1.24	10.24	0.85	9.47	0.60	8.76	0.89
15	11.03	1.34	10.81	0.97	9.95	0.61	9.16	0.83
16	11.54	1.45	11.25	1.00	10.25	0.61	9.38	0.85

NUMA-ICTM: política *bind_block*

NP	Caso 1		Caso 2		Caso 3		Caso 4	
	média	**desvio**	**média**	**desvio**	**média**	**desvio**	**média**	**desvio**
2	1.87	0.06	1.89	0.05	1.87	0.05	1.88	0.05
3	2.80	0.09	2.83	0.08	2.80	0.08	2.75	0.06
4	3.74	0.12	3.77	0.11	3.72	0.11	3.63	0.11
5	4.66	0.15	4.69	0.13	4.65	0.18	4.46	0.03
6	5.58	0.18	5.63	0.17	5.53	0.18	5.46	0.12
7	6.50	0.22	6.55	0.18	6.48	0.14	6.34	0.11
8	7.42	0.24	7.37	0.14	7.14	0.14	7.08	0.07
9	8.30	0.28	8.36	0.22	8.08	0.15	8.02	0.10
10	9.19	0.30	9.27	0.27	9.04	0.26	8.94	0.09
11	10.09	0.34	10.20	0.26	9.76	0.23	9.74	0.11
12	11.02	0.36	11.10	0.31	10.70	0.21	10.49	0.08
13	11.87	0.39	11.92	0.35	11.46	0.26	11.23	0.17
14	12.73	0.45	12.75	0.28	12.36	0.15	12.08	0.19
15	13.58	0.44	13.67	0.37	13.10	0.12	12.78	0.22
16	14.45	0.40	14.52	0.30	13.85	0.18	13.53	0.16

NUMA-ICTM: política *cyclic*

NP	Caso 1		Caso 2		Caso 3		Caso 4	
	média	**desvio**	**média**	**desvio**	**média**	**desvio**	**média**	**desvio**
2	1.79	0.10	1.81	0.10	1.65	0.17	1.71	0.09
3	2.65	0.17	2.67	0.17	2.64	0.16	2.65	0.16
4	3.56	0.21	3.42	0.18	3.40	0.04	3.35	0.23
5	4.41	0.29	4.43	0.26	4.37	0.26	4.39	0.23
6	5.28	0.33	5.34	0.31	5.16	0.23	5.20	0.22
7	6.09	0.40	6.16	0.37	5.94	0.25	6.03	0.29
8	6.96	0.47	7.02	0.38	6.67	0.22	6.77	0.23
9	7.79	0.54	7.84	0.50	7.55	0.37	7.47	0.25
10	8.58	0.61	8.62	0.55	8.27	0.42	8.10	0.27
11	9.44	0.68	9.47	0.60	9.03	0.41	8.76	0.31
12	10.26	0.73	10.28	0.62	9.68	0.40	9.48	0.29
13	10.94	1.01	11.00	0.65	10.22	0.34	10.14	0.53
14	11.72	0.84	11.72	0.67	10.84	0.37	10.78	0.55
15	12.50	0.87	12.37	0.69	11.36	0.23	11.18	0.54
16	13.07	1.01	13.09	0.70	11.88	0.47	11.63	0.54

NUMA-ICTM: política *cyclic_block = 10*

NP	Caso 1		Caso 2		Caso 3		Caso 4	
	média	**desvio**	**média**	**desvio**	**média**	**desvio**	**média**	**desvio**
2	1.73	0.06	1.74	0.05	1.74	0.05	1.74	0.05
3	2.65	0.09	2.67	0.08	2.64	0.08	2.59	0.06
4	3.42	0.12	3.46	0.11	3.38	0.11	3.32	0.11
5	4.39	0.15	4.43	0.13	4.35	0.18	4.27	0.03
6	5.25	0.18	5.29	0.17	5.08	0.18	4.98	0.12
7	6.05	0.22	6.10	0.18	5.94	0.14	5.81	0.11
8	6.81	0.24	6.92	0.14	6.71	0.14	6.52	0.07
9	7.62	0.28	7.79	0.22	7.51	0.15	7.25	0.10
10	8.22	0.30	8.54	0.27	8.17	0.26	7.90	0.09
11	9.05	0.34	9.20	0.26	8.95	0.23	8.53	0.11
12	9.80	0.36	10.00	0.31	9.57	0.21	9.06	0.08
13	10.50	0.39	10.63	0.35	10.07	0.26	9.60	0.17
14	11.11	0.45	11.28	0.28	10.71	0.15	10.13	0.19
15	11.93	0.44	11.87	0.37	11.05	0.12	10.47	0.22
16	12.57	0.40	12.65	0.30	11.71	0.18	10.95	0.16