UNIVERSIDADE FEDERAL DE PELOTAS

Bacharelado em Ciência da Computação

Trabalho de Conclusão de Curso

Implementação eficiente do modelo de Potts Celular em processadores multi-core

Alexandre Gomes da Costa

Pelotas, 2008

ALEXANDRE GOMES DA COSTA

IMPLEMENTAÇÃO EFICIENTE DO MODELO DE POTTS CELULAR EM ARQUITETURAS MULTICORE

Trabalho de conclusão de curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Gerson Geraldo H. Cavalheiro

Pelotas, 2008

Dados de catalogação na fonte:
Ubirajara Buddin Cruz – CRB-10/901
Biblioteca de Ciência & Tecnologia - UFPel

C837i Costa, Alexandre Gomes da
Implementação eficiente do modelo de Potts Celular em processadores multi-core / Alexandre Gomes da Costa ; orientador Gerson Geraldo H. Cavalheiro. – Pelotas, 2008. – 54f. - Monografia (Conclusão de curso). Curso de Bacharelado em Ciência da Computação. Departamento de Informática. Instituto de Física e Matemática. Universidade Federal de Pelotas. Pelotas, 2008.

1.Informática. 2.Arquiteturas milti-core. 3. Multithreading. 4.Modelo de potts celular. 5.OpenMP. 6.Método Monte Carlo. 7.Random Walker. 8.Algoritmo Metropolis. I.Cavalheiro, Gerson Geraldo H. II.Título.

CDD: 005.13

Dedico este trabalho aos meus pais e irmão, Clovis, Lucia e Andrigo, que, através da dedicação, do amor e do exemplo de vida me fizeram chegar até aqui.
Dedico-o, também, a minha namorada Aline e a todos aqueles que tiveram alguma participação positiva na minha vida.

AGRADECIMENTOS

Quero agradecer, em primeiro lugar, a Deus, pelo dom da vida e por todas as bênçãos que Ele me deu durante a minha vida, inclusive o fato de estar concluindo mais esta etapa, e acima de tudo, por Ele nunca ter me abandonado.

Quero agradecer aos meus pais por tudo que eles têm feito por mim, inclusive deixando de lado muitos dos seus sonhos para me dar a chance de estudar. Agradeço ao meu irmão, minha namorada e amigos, por terem me incentivado e apoiado em tudo. Agradeço ao meu colega de aula Leonardo (Xuxu), pois sem a ajuda dele não teria terminado este trabalho. Gostaria também de agradecer a um outro colega Elvio, que me ajudou a realizar os testes de desempenho nos algoritmos, testes estes que não foram poucos. Agradeço também ao pessoal do laboratório de informática, que me deixou usar os computadores para realizar todos os testes. Gostaria de agradecer ao meu orientador Gerson, por ter me ajudado com este trabalho. Peço desculpas se não citei o nome de alguém, mas pode se enquadra como meu amigo.

“Concedei-nos, Senhor, a Serenidade necessária, para aceitar as coisas que não podemos modificar, a Coragem para modificar aquelas que podemos e a Sabedoria para distinguirmos umas das outras.”

São Francisco de Assis

Resumo

COSTA, Alexandre Gomes da. **Implementação Eficiente do Modelo de Potts Celular em Arquiteturas Multi-core.** 2008. 54f. Monografia – Curso de Bacharelado em Ciência da Computação. Universidade Federal de Pelotas

O modelo de Potts Celular é um dos mais utilizado na área de simulação de sistemas celulares. Porém este modelo, em sua forma normal, não oferece bom desempenho de execução. Originalmente, no modelo do Potts Celular, é utilizada a técnica Monte Carlo. Neste trabalho, para obter um índice de desempenho mais elevado, é aplicada a técnica Random Walker no modelo de Potts Celular explorando o potencial de paralelismo de arquiteturas baseadas em processadores multi-core. Esta nova configuração de hardware disponibiliza o processamento paralelo a baixo custo, embora requeira a utilização de recursos de programação especializados. Para o caso de estudo aqui apresentado, implementação de um modelo de Potts Celular, o objetivo é obter índices de desempenho na execução. Para tanto, os cuidados na programação a serem tomados devem contemplar otimização no uso da memória cache e escalonamento por afinidade de processos. A avaliação dos resultados se dará comparando o desempenho obtido pela implementação realizada onde tais questões são tratadas com implementação do mesmo modelo sem o uso destes recursos.
A implementação foi realizada em ambiente Linux, utilizando C/C++ e a biblioteca de Threads POSIX.

Palavras-chave: Random Walker, Modelo Potts Celular, Método Monte Carlo.

Abstract

COSTA, Alexandre Gomes da. **Efficient Implementation of the Potts Model in Architectures Mult-Core.** 2008. 54f. Monografia – Curso de Bacharelado em Ciência da Computação. Universidade Federal de Pelotas

The cellular Potts Model is one of the most used in the area of simulation of cellular systems. But this model, in its normal form, does not offer good performance. Originally, the cellular Potts model, is used the technique Monte Carlo. In this work, to get a higher performance, is applied the Random Walker technique in the cellular Potts model exploring the potential of parallel architectures based on multi-core processors. This new configuration of hardware provides the parallel processing at low cost, but requires the use of specialized programming. The case study presented here, implementation of a cellular Potts model, the goal is to get rates of performance in the implementation. To this end, the care in programming to be taken should include optimizing the use of memory cache and scaling by affinity processes. The assessment results will be comparing the performance achieved by implementing held where such issues are dealt with implementation of the same model without the use of these resources.

The implementation was held in a Linux environment, using C / C + + and the library of POSIX Threads.

Keywords: Random Walker, Cellular Potts model, Monte Carlo method.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

RW	*Random Walker*
RWS	*Random Walker Sequencial*
RWP	*Random Walker Paralelo*
MC	*Monte Carlo*
CdA	*Controle de Afinidade*
UMA	*Uniform Memory Access*
NUMA	*Non Uniforme Memory Access*
SMP	*Symmetric Multiprocessing*

SUMÁRIO

1 INTRODUÇÃO

O modelo de Potts Celular é utilizado na área de simulação como uma ferramenta para auxiliar a avaliação do crescimento de sistemas celulares, tais como evolução de células cancerígenas, desenvolvimento dos membros de aves e desenvolvimento de espumas de sabão (CERCATO, 2005). O custo computacional desse modelo está diretamente relacionado com o tamanho da área a ser simulada, refletindo o tempo a ser despendido na simulação. Em sistemas de grandes dimensões, o tempo de processamento tende a tornar-se muito elevado quando a execução é realizada em arquiteturas seqüenciais.

Alternativas para melhora de desempenho na execução desse modelo de simulação são objetos de estudo em diversos trabalhos, dentre os quais Cercato (2005). Cercato apresenta uma proposta de implementação paralela do algoritmo de Potts Celular sobre aglomerado de computadores, uma arquitetura com memória distribuída. Nesse trabalho, além de explorar técnicas de execução paralela, foi utilizado um algoritmo otimizado para execução do algoritmo de Potts Celular. Como mostram os resultados de desempenho obtidos com esta implementação (CERCATO; MOMBACH; CAVALHEIRO, 2006), foi possível obter bons índices de desempenho conseguindo uma redução significativa no tempo total de execução.

Atualmente, o mercado de processamento paralelo apresenta processadores multi-core como uma nova alternativa ao processamento de alto desempenho. Mais baratos e de utilização mais simples do que os aglomerados de computadores, essas arquiteturas tendem a estar cada vez mais presentes em diversas aplicações onde fatores ligados ao desempenho sejam necessários.

O presente trabalho apresenta uma implementação do modelo de Potts Celular em processadores multi-core, estendendo os trabalhos apresentados em (CERCATO, 2005) e (GUZATTO; MOMBACH; CERCATO; CAVALHEIRO, 2005). Com esta implementação, busca-se explorar de forma efetiva esta nova tecnologia de arquitetura paralela na resolução de uma aplicação real.

Os principais problemas abordados durante o desenvolvimento da implementação estão relacionados à exploração de recursos de programação que habilitem obter o máximo de desempenho dos processadores multi-core. Não

bastando a simples introdução da multiprogramação leve, observou-se ser necessário explorar as características próprias ao hardware multi-core. O enfoque deste trabalho será na aplicação de técnicas de otimização de uso da memória cache pela determinação de afinidade de threads a processadores.

A implementação realizada foi avaliada em termos de desempenho, comparando os resultados de tempo obtidos pela execução de duas versões multithreaded da técnica Random Walker para evolução do modelo de Potts Celular, uma contendo código permitindo otimização de execução e outra sem este código otimizado.

1.1 Objetivos do trabalho

O principal objetivo do presente trabalho é implementar o algoritmo de Random Walker, realizado por Cercato (2005), de forma eficiente com o intuito de obter um melhor desempenho deste algoritmo sobre arquiteturas multi-processadas (multi-core).

Como objetivo secundário buscou-se compreender as estratégias de execução em processadores multi-core.

1.2 Resultados Esperados

Com a crescente popularização das arquiteturas multi-core, espera-se que o trabalho ofereça um implementação eficiente de um mecanismo de simulação de agregados celulares pelo modelo de Potts Celular. Portanto, viabilizar a execução eficiente deste modelo de simulação em arquiteturas multi-core.

1.3 Metodologia

A metodologia usada para desenvolver este trabalho foi relacionar estratégias de execução com o intuito de refinar o algoritmo de Cercato (2005) e Guzatto (2005) para arquiteturas multi-core. A principal estratégia de execução foi determinar afinidade de thread a processador. Por fim, avaliar o desempenho do algoritmo com e sem afinidade.

1.4 Estrutura do texto

O restante desta monografia está organizado como segue.

O Capítulo 2 aprofunda o Método Monte Carlo e o Modelo de Potts Celular e conceitos relacionados. O algoritmo de Metropolis, utilizado em conjunto com a estratégia de Monte Carlo e o algoritmo de Random Walker, concebido como uma alternativa mais eficiente para o modelo de Potts também são apresentados. A Hipótese de Adesão Diferenciada, responsável pela evolução dos cálculos das simulações do modelo de Potts, também é mostrada.

O Capitulo 3 realiza uma breve descrição sobre arquiteturas *multi-core* e multiprogramação leve. Neste mesmo capítulo, discute-se sobre a hierarquia de memória em máquinas *multi-core* e a utilização da memória *cache*.

O Capitulo 4 discute brevemente sobre a ferramenta *Pthreads*, utilizada para implementar os algoritmos, suas características, vantagens e desvantagens. Também é mostrada como a técnica de afinidade de *threads* a processador foi utilizada neste trabalho para aumentar o desempenho da aplicação.

O Capítulo 5 trata de alguns detalhes importantes a respeito do funcionamento dos algoritmos seqüencial e paralelo de *Random Walker*.

O Capítulo 6 apresenta os resultados de desempenho obtidos com a execução das implementações com e sem afinidade.

Por fim, o Capítulo 7 conclui o presente trabalho, discutindo a respeito dos resultados.

2 SIMULAÇÃO DE AGREGADOS CELULARES

Este capítulo apresenta técnicas utilizadas para evolução de simulações computacionais de agregados celulares. A Seção 2.1 descreve o método Monte Carlo, tradicionalmente empregado para evoluir a simulação de sistemas celulares, juntamente com o Algoritmo de Metropolis, que é um dos mais utilizados para realizar o cálculo de adesão de energia entre células. Na Seção 2.2 é apresentada uma solução alternativa para o método Monte Carlo, o algoritmo Random Walker. Na Seção 2.3 é exposto o modelo de Potts Celular, que é o foco principal desse trabalho. Na subseção 2.3.1 é descrita a Hipótese de Adesão Diferenciada, muito importante para guiar simulações baseadas no modelo de Potts Celular. Por fim, a Seção 2.4 conclui este capítulo.

2.1 Método de Monte Carlo e o Algoritmo de Metropolis

O método de Monte Carlo recebeu esse nome durante a Segunda Guerra Mundial no projeto Manhattan e foi inspirado na aleatoriedade das roletas dos cassinos da cidade de Monte Carlo. Trata-se de um método estatístico utilizado em simulações de diversas áreas científicas como Física, Matemática e Biologia (NEWMAN; BARKEMA, 1999). O método Monte Carlo realiza sorteios aleatórios em um determinado espaço amostral para obter uma aproximação do resultado real. Ele pode ser aplicado tanto em problemas probabilísticos quanto determinísticos. O modelo de *Potts* Celular pode ser evoluído pelo método Monte Carlo junto ao algoritmo de *Metropolis*.

2.1.1 O algoritmo de Metropolis

O algoritmo de *Metropolis* foi desenvolvido por Nicolas Metropolis e W. K. Hastings em 1953. Ele é um dos algoritmos relacionados com o método Monte Carlo mais utilizados no ramo da Física. Tem como objetivo determinar valores esperados de propriedades do sistema simulado, através de uma média sobre uma amostra. Basicamente, o algoritmo de *Metropolis* realiza testes de aceitação em um determinado ponto, gerando um novo estado. Tais testes são realizados de acordo com a temperatura e a diferença de energia do sistema. Quando um estado é aceito,

o sistema é alterado para esse estado. Caso contrário, é mantido o estado original (KNEWITZ, 2002).

2.1.2 Aplicações da técnica de Monte Carlo

Um exemplo de aplicação do método Monte Carlo é o cálculo da superfície de um lago. O modo tradicional seria dividir a superfície do lago em partes menores e calcular a superfície de cada uma delas, o que seria bem trabalhoso. Já pelo método de Monte Carlo, esta tarefa seria dividida em uma série de passos, descritos a seguir:

1. Delimitar toda a superfície do lago dentro de um quadrado de área conhecida.
2. Sortear um número determinado de pontos, como mostrado na Fig. 2.1.
3. Contar o número de pontos que caíram dentro do lago.
4. Calcular o percentual do número de pontos que caíram dentro do lago em relação ao número total de pontos.

O percentual obtido neste último passo corresponde a uma aproximação da superfície real do lago.

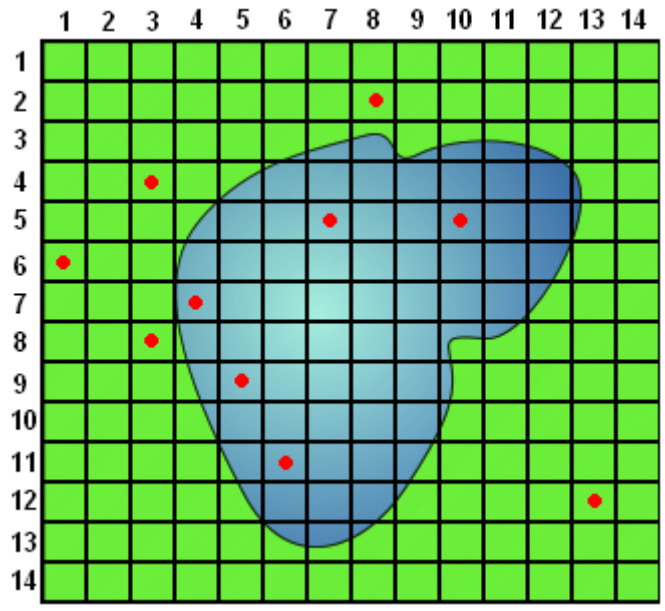

Figura 2.1 – Cálculo da superfície de um lago pelo método de Monte Carlo.

Na simulação de sistemas celulares, o método Monte Carlo permite selecionar, aleatoriamente, pontos (rótulos) de um tecido celular sobre o qual devem ser aplicados cálculos de reações químicas e físicas com os rótulos vizinhos afim de evoluir o sistema analisado. No entanto, observa-se que o sistema celular somente é alterado quando é selecionado um rótulo que situa-se na fronteira entre duas ou mais células distintas. Assim, o método tradicional de Monte Carlo mostra-se ineficiente, uma vez que o sorteio de um rótulo que não esteja na fronteira entre células não ocasionará evolução do sistema simulado (CERCATO, 2005).

O algoritmo de *Random Walker* foi apresentado em (GUZATTO, 2005), como um método alternativo ao Monte Carlo para realizar a simulação deste tipo de sistema.

2.2 Algoritmo de Random Walker

Este algoritmo foi utilizado para realizar o arredondamento de agregados celulares em 2D (GUZATTO; MOMBACH; CERCATO; CAVALHEIRO, 2005). Em (CERCATO, 2005) este algoritmo foi estendido para realizar simulações em 3D. Além disso, Cercato (2005) desenvolveu um algoritmo RW concorrente e conseguiu um aumento de desempenho no modelo de *Potts*, executando esse mesmo algoritmo em aglomerados de computadores (*clusters*).

No caso do modelo de *Potts* Celular, o algoritmo RW executa um número menor iterações do que o algoritmo de Monte Carlo. A estratégia utilizada no algoritmo RW é limitar os sorteios a rótulos que estejam em regiões limítrofes de uma determinada célula. Tais rótulos estão mais propensos a sofrer substituições.

Nos testes que Cercato (2005) realizou, o algoritmo RW seqüencial mostrou ser em média seis vezes mais rápido do que o algoritmo tradicional de Monte Carlo. Também o algoritmo RW paralelo mostrou ser pelo menos 17% mais rápido que o algoritmo RW seqüencial.

Neste trabalho é utilizado o algoritmo RW desenvolvido em Cercato (2005), possibilitando a execução desse algoritmo em processadores *multi-core*.

```
1. rotulo = sorteiaBorda()
2. for( passos = 0 ; passos < limite ; passos++ ){
3.   trocaEnergia(rotulo);
4.   rotulo = sorteiaVizinhoBorda();
5. }
```

Figura 2.2 – Esqueleto básico da função RW.

Na Fig. 2.2 é apresentado o algoritmo básico da função RW, onde observa-se três variáveis e três funções, respectivamente: `rotulo`, `passos`, `limite` e `sorteiaBorda()`, `trocaEnergia()`, `sorteiaVizinhoBorda()`. Na linha 1, a variável `rotulo` recebe um rótulo na borda de uma célula qualquer, provido pela função `sorteiaBorda()`. Esta função retorna o valor numérico de um rótulo posicionado na borda de uma célula qualquer. Na linha 2 é dado início ao laço que garante a evolução do sistema durante um número pré-determinado de passos.

Na linha 3 é efetuado o cálculo de adesão diferenciada sob o rótulo escolhido pela função `sorteiaBorda()`. Este cálculo é realizado pela função `trocaEnergia()`, que avalia o resultado e efetua a substituição do rótulo escolhido pelo seu vizinho. Na linha 4, o rótulo vizinho ao rótulo escolhido na linha 1 passa a ser o novo rótulo escolhido, eliminando a necessidade de um novo sorteio. O valor do rótulo vizinho é armazenado na variável `rotulo`. A função `sorteiaVizinhoBorda()` retorna um novo rótulo da borda próximo ao rótulo tratado para prosseguir a simulação, caso o rótulo escolhido não seja borda é feito uma chamada a função sorteiaBorda. Na linha 5 é finalizado o laço iniciado na linha 2.

2.3 Modelo Potts Celular

O modelo *Potts* Celular foi proposto por James Glazier e Francois Graner (GRANER; GLAZIER, 1992). O modelo de *Potts* Celular vem sendo muito utilizado na simulação de sistemas celulares, sendo um dos mais conceituados dentre os integrantes da comunidade científica. O modelo de *Potts* Celular utiliza a Hipótese de Adesão Diferenciada para calcular a diferença de energia entre as células e a

estratégia Monte Carlo para guiar a evolução do sistema como um todo através do sorteio de rótulos.

Cercato (2005) mostrou em seu trabalho que as implementações tradicionais deste modelo, empregando a técnica de Monte Carlo para evoluir o sistema, são ineficientes e que, em função do custo computacional - processamento e memória -, é inviável a simulação de grandes sistemas celulares em arquiteturas monoprocessadas convencionais. A alternativa por ele apresentada foi utilizar programação paralela em arquitetura com memória distribuída e um algoritmo alternativo, empregando a técnica de Random Walker (GUZATTO, 2005), para realizar a evolução do sistema.

2.3.1 Hipótese de Adesão Diferenciada

A Hipótese de Adesão Diferenciada é um modelo termodinâmico no qual a interação entre as células se dá através de uma energia de ligação superficial que depende das moléculas de ligação nas membranas celulares. A minimização dessa energia é análoga ao que ocorre em líquidos imiscíveis, líquidos que não se misturam (água e óleo), e guia a evolução do sistema. Em agregados celulares, diferentes energias estão associadas às interfaces entre células de diversos tipos. Estas energias se manifestam como tensões superficiais entre agregados de células e determinam se tipos celulares diferentes se misturam ou segregam-se. Quando as tensões superficiais são negativas, células de determinado tipo misturam-se com outras células de tipos diferentes. Por outro lado, tensões superficiais positivas conduzem à segregação das mesmas (GRANER, 1993).

Ressalta-se que, em um sistema celular, o cálculo da variação de energia entre células se dá nas regiões de borda das mesmas. Considerando o modelo computacional, as células são compostas de rótulos. Sendo assim, somente existe possibilidade de troca de energia entre rótulos de células que situam-se em regiões fronteiriças das mesmas. Isto ocorre devido ao fato de rótulos situados na região interna das células possuírem o mesmo potencial de energia. Ou seja, não existe possibilidade de troca entre dois rótulos situados em uma mesma célula.

2.4 Conclusão

Nesse capítulo foi apresentado o modelo de *Potts* Celular, que tem um custo computacional muito elevado. Foi também visto que Cercato apresentou uma alternativa para contornar as questões de desempenho, utilizando técnicas de execução paralela juntamente com o algoritmo de *Random Walker*.

A técnica *Random Walker* é considerada uma alternativa eficiente para realizar a implementação do modelo de *Potts* Celular, pois permite uma considerável redução do custo computacional da simulação restringindo os sorteios de rótulos para áreas limítrofes de células. Este algoritmo é naturalmente paralelizável e reflete a estrutura da arquitetura *multi-core*. Este trabalho visa explorar este aspecto. Neste trabalho, a matriz onde o algoritmo de RW realiza os cálculos foi dividida pela metade no eixo x. Cada uma dessas metades foram armazenadas de forma a explorar a memória *cache* com mais eficiência.

3 ARQUITETURA MULTI-CORE

Neste capítulo são introduzidos alguns conceitos a respeito das arquiteturas *multi-core*. Inicialmente, é apresentado um breve histórico relembrando a arquitetura de Von Neumann, a Lei de Moore e a categorização das arquiteturas paralelas. Em seguida, é descrita a arquitetura *multi-core* propriamente dita, seguindo-se de uma descrição sumária sobre multiprogramação leve.

3.1 A arquitetura de von Neumann

O modelo de *von Neumann* foi apresentado em 1945 por John von Neumann. A arquitetura de *von Neumann* é um modelo de computação que emprega uma estrutura de memória que armazena tanto as instruções executadas pelos programas quanto os dados por elas manipulados. Algum tempo depois, foi mostrado por John Backus existir um problema nessa arquitetura denominado de "gargalo de *von Neumann*" (*memory gap*). Isso gerou uma preocupação na época, pois a velocidade da memória não acompanhava a velocidade do processador. Porém, essa dificuldade tem sido contornada pela utilização da memória *cache*, muito mais rápida que as memórias convencionais. Ela serve como um repositório de rápido acesso para dados e instruções usadas constantemente pelo processador. Com esse modelo, reforça-se a maneira seqüencial de pensar e escrever programas de computador.

3.2 A Lei de Moore

Gordon Moore, um dos fundadores da Intel, previu empiricamente que com o avanço da tecnologia, o número de transistores em um único processador iria dobrar a cada 24 meses, enquanto os custos permaneceriam constantes. Esta previsão passou a ser chamada de ***Lei de Moore*** e os avanços da indústria comprovam esta previsão até os dias atuais. (BURGER; GOODMAN, 1997).

A previsão propriamente dita não é o problema, mas sim o seu resultado. A quantidade de transistores colocada em um único chip é tão grande que está ficando difícil até mesmo para as ferramentas de CADE (responsável pelo desenho do chip)

controlar tantos componentes. Além disso, o tamanho do transistor está aproximando-se rapidamente de um limite físico.

3.3 Arquiteturas Paralelas

Em 1966, Flynn propôs uma taxonomia para classificar as arquiteturas baseando-se no fluxo de instruções e dados observados (FLYNN, 1972). Ele a destacava em quatro categorias.

- SISD (*Single Instruction Stream, Single Data Stream*): Essa categoria utiliza um único fluxo de instruções operando sobre um único fluxo de dados, essa é a categoria em que se encontra a arquitetura de von Neumann.
- SIMD (*Single Instruction Stream, Multiple Data Stream*): Essa categoria utiliza um único fluxo de instruções operando sobre múltiplos fluxos de dados. Nessa categoria encontram-se as arquiteturas matriciais.
- MISD (*Multiple Instruction Stream, Single Data Stream*): Essa categoria utiliza múltiplos fluxos de instruções e opera sobre um único fluxo de dados. Alguns autores classificam as arquiteturas *pipeline* e *dataflow* como MISD.
- MIMD (*Multiple Instruction Stream, Multiple Data Stream*): Essa categoria utiliza múltiplos fluxos de instruções e opera sobre múltiplos fluxos de dados. É onde se encaixam as arquiteturas paralelas, explorando ao máximo o potencial concorrente dos processos leves ou *threads*.

O foco desse trabalho são as arquiteturas MIMD, mais precisamente em duas subdivisão desta categoria, a arquitetura SMP (*Symmetric Multiprocessing*), representada na Fig. 3.1 e NUMA (*Non-Uniform Memory Access*) mostrada na Fig. 3.2.

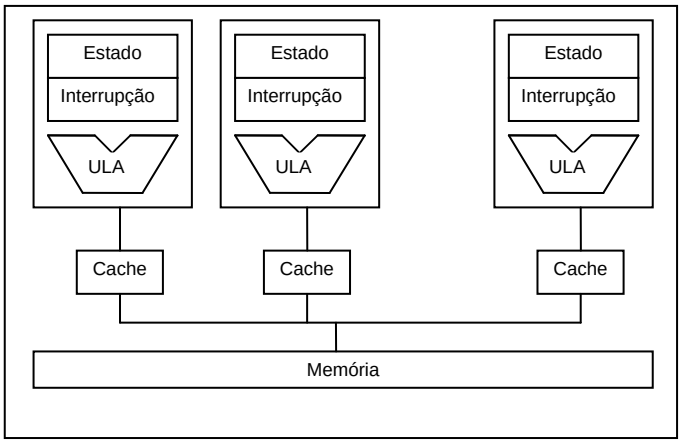

Figura 3.1 – Subdivisão Arquitetura SMP

A arquitetura SMP Fig. 3.1, também chamada de UMA (Uniform Memory Access), é composta por múltiplos processadores que compartilham a mesma memória. Nessa arquitetura todos os processadores têm igual acesso ao barramento e a memória, portanto não há privilégios por parte do Sistema Operacional (SO) a nenhum processador específico. Por essa arquitetura possuir memória compartilhada, o paradigma de programação que melhor tira proveito de suas vantagens é o de multiprogramação leve. Pois, ao mesmo tempo em que a arquitetura SMP prevê a existência de diversos processadores fazendo uso de uma memória em comum, a multiprogramação leve prevê a existência de diversos fluxos de execução utilizando o mesmo espaço de endereçamento para comunicar dados entre si (CAVALHEIRO; SANTOS, 2007). Esta relação é explorada por diversas ferramentas, tais como .Net, OpenMP, Pthreads (CAVALHEIRO; SANTOS, 2007).

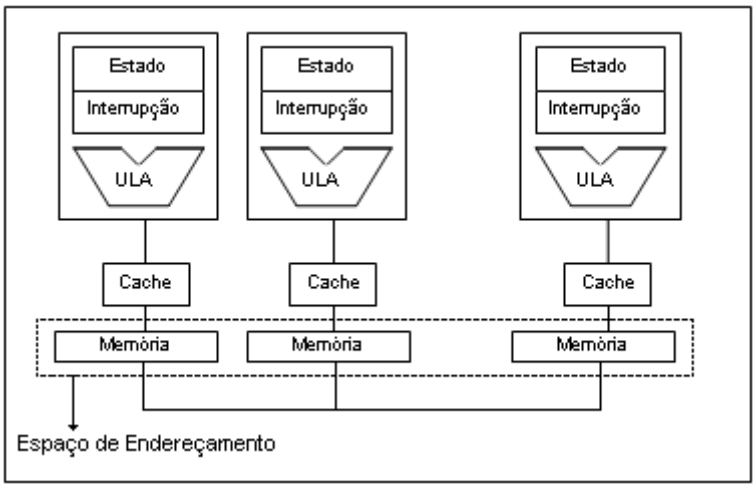

Figura 3.2 – Subdivisão Arquitetura NUMA

Já arquitetura NUMA Fig. 3.2 possui uma memória dedicada a cada processador ou grupos de processadores, geralmente o conjunto processador, cache e memória que é a chamado de módulo. Cada módulo tem acesso a qualquer outro módulo da arquitetura, porém, o tempo de acesso é variável. Os módulos são ligados por crossbar ou switch, isto torna a área de memória compartilhada distribuída.

3.4 Multi-core

Um processador *multi-core* é caracterizado por possuir dois ou mais núcleos de processamento completos dentro de um mesmo *chip*. Estes núcleos correspondem a unidades de processamento, ou seja, consistem em processadores convencionais, recebendo a denominação de núcleos (*cores*). O modelo básico da arquitetura define que os *cores* são tratados de forma independente e uma estrutura de *cache* que pode ter diferentes configurações. As Fig. 3.3 e 3.4 representam dois modelos de arquiteturas *dual-core* com diferentes configurações de *cache*.

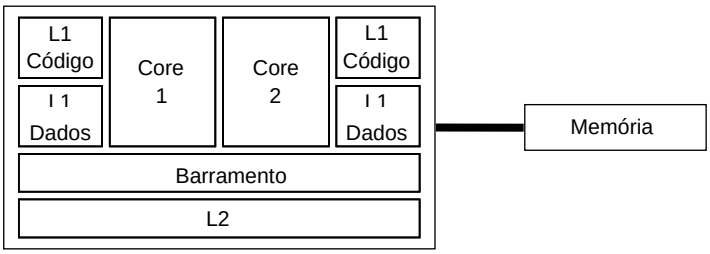

Figura 3.3 - Processador dual-core com nível L2 de cache compartilhado.

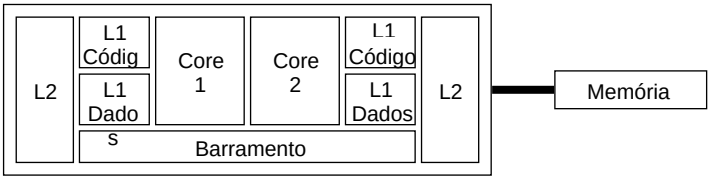

Figura 3.4 - Processador dual-core com nível L2 de cache dedicada por core.

No exemplo da Fig. 3.3, a arquitetura *dual-core* representa o esquema do processador Yonah, da Intel. O Yonah possui uma estrutura de *cache* onde o nível L1 (para dados e para código) é próprio a cada *core*, sendo o nível L2 (dados e código) compartilhado. Já na Fig. 3.4, representando o esquema do processador Dempsey, também da Intel, a hierarquia de *cache* contempla cada *core* com um

nível L2 próprio. Outras estruturas também podem ser concebidas, por exemplo, incluindo compartilhamento ou não de um nível L3 de memória cache.

Embora um dos principais argumentos utilizados para aumentar as vendas de arquiteturas *multi-core* esteja calcada no potencial de ganho de desempenho que estes processadores possuem, deve-se salientar que os mesmos somente podem ser explorados de forma efetiva com o uso da multiprogramação leve. Diversas ferramentas de software foram lançadas no mercado ao longo dos anos para atingir este objetivo, tais como Pthreads, OpenMP, MPI, etc.

3.5 Multiprogramação leve

Atualmente, os sistemas operacionais aceitam aplicações concorrentes, que são escritas utilizando recursos de multiprogramação leve.

No momento do início da execução de um programa, ele torna-se um processo do ponto de vista do sistema operacional. Esse processo é composto por uma área de memória, um registro descritor de processo (Bloco de Controle) e um processo leve ou *thread*. A Fig. 3.5 ilustra a estrutura básica de um processo.

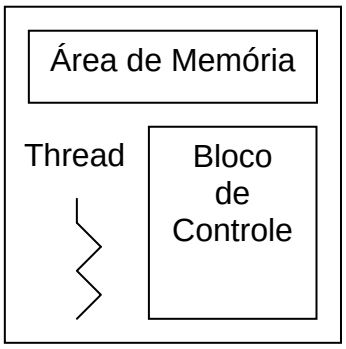

Figura 3.5 – Esquema de processo convencional.

Na área de memória ficam os dados utilizados pelo processo e pelo *thread*. No registro descritor são armazenadas as informações sobre o estado de um processo, a prioridade do mesmo, a lista de arquivos abertos, entre outros dados relevantes que o sistema operacional precisa saber sobre o processo. Um *thread* é um fluxo de execução do processo, ou seja, uma seqüência de instruções a ser executada. Esse thread possui a sua área de memória própria, contendo o registro de invocação de funções e o estado dos registradores.

Já um processo *multithread*, como mostrado na Fig. 3.6, é praticamente igual a um processo comum. Ele possui uma área de memória e um registro descritor. Porém, ao invés de lançar um único *thread*, ele pode lançar duas ou mais *threads*. Em processos *multithreads*, cada *thread* tem uma pilha, onde também ficam armazenados seus registros de invocação de funções e os estados dos registradores.

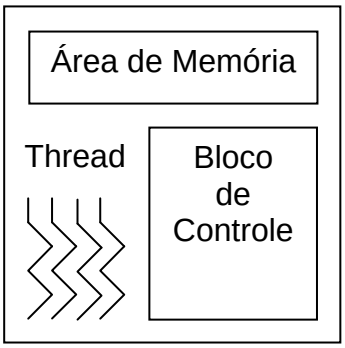

Figura 3.6 – Esquema de um processo multithread.

Cada *thread* de um processo *multithread* é escalonado pelo Sistema Operacional (SO). Portanto, quando um *thread* é lançado não há garantias do tempo que o SO vai levar para escalonar esse mesmo *thread*. Este ciclo de vida de um *thread* inicia-se logo que ela é criada, sendo colocada no estado pronto e, com isso, esperando para ser escalonada posteriormente. O processo básico do ciclo de vida de um *thread* é mostrado na Fig. 3.7.

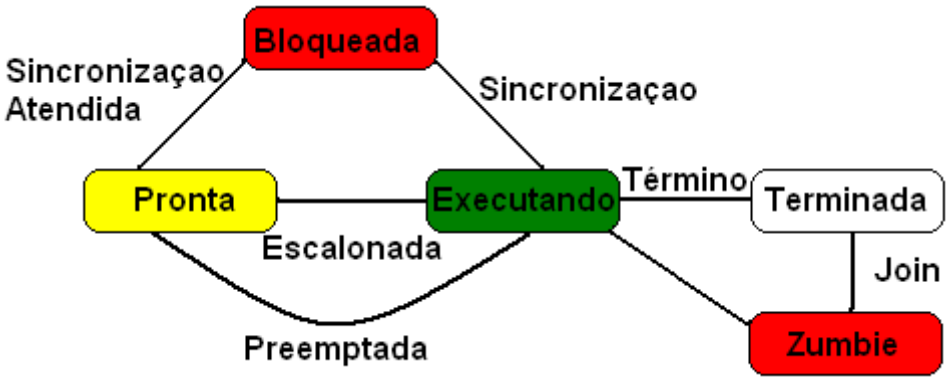

Figura 3.7 – Exemplo do ciclo de vida de um Thread.

Fonte: Cavaleiro, Santos, 2007, p. 17

A multiprogramação leve é o modelo de programação que melhor tira proveito das arquiteturas *multi-core* e SMP devido à memória compartilhada dessa arquitetura. Mas para obter-se um índice satisfatório de desempenho, é necessário adotar técnicas eficientes de programação. Isto pode ser obtido através de uma análise mais detalhada da hierarquia de memória presente nesse tipo de arquitetura.

A idéia básica da hierarquia de memória é antecipar a busca de dados e instruções da memória principal para a memória *cache*. Apenas as instruções e dados acessados constantemente pelo processo ficam armazenadas nessa memória.

Uma estratégia interessante e viável para melhorar o desempenho de uma aplicação é o escalonamento de *threads* por afinidade, considerando a distribuição de carga computacional. Por exemplo, tem-se um (1) processador com dois (2) *cores* e esse processador precisa escalonar quatro (4) *threads*, sendo dois deles orientados a CPU e dois orientados a entrada e saída. Uma boa forma de realizar o escalonamento seria enviar os *threads* orientados a CPU para serem executados em um *core* e enviar os *threads* orientados a entrada e saída para serem executados em outro *core*.

3.6 Conclusão

É possível notar que já vem sendo pesquisado maneiras de obter-se índices mais elevados de desempenho em diversas aplicações, como na implementação de Potts Celular.

As arquiteturas *multi-core* permitem aumento de desempenho. No entanto, deve ser utilizada programação concorrente (*multithread*). Não basta somente explorar o paralelismo de uma aplicação a nível de *software*. Faz-se necessário utilizar de forma eficiente os recursos do *hardware*, explorando a hierarquia de *cache* pela atribuição de afinidade de processador (este aspecto está ligado a localidade de dados). Um exemplo arquitetura que o controle de afinidade possa tirar proveito é em uma arquitetura NUMA, onde o tempo de acesso não é o mesmo.

4 MULTIPROGRAMAÇÃO LEVE COM PTHREADS

Neste capítulo são apresentados conceitos relacionado ao uso de threads, em particular sobre Pthreads, a ferramenta de programação selecionada para implementação deste trabalho. A primeira seção fala sobre o padrão POSIX, de onde foi tirado a sigla Pthread. Na seção 4.2, são exibidas as primitivas providas pela biblioteca Pthreads relacionadas com o processo de criação e destruição de *threads*. Em seguida, a seção 4.3 mostra o uso das variáveis *mutex*, responsáveis pela sincronização dos *threads*. Na seqüência, a seção 4.4 fala sobre o custo do mutex. A seção 4.5 apresenta um dos focos deste trabalho, afinidade de *threads* a processador. A seção 4.6 descreve resumidamente algumas vantagens e desvantagens da utilização da ferramenta Pthreads e, por fim, a seção 4.7 conclui o capítulo.

4.1 Padrão POSIX (Portable Operating System Interface for uniX)

POSIX é um padrão oferecido pela IEEE, que determina um conjunto de interfaces de programação para sistemas Unix, dentre os quais está incluso o padrão para *threads*. POSIX Threads ou Pthreads é disponibilizada pela biblioteca **libpthread.a** em distribuições Linux e encontra-se igualmente disponível em praticamente todos os Sistemas Operacionais baseados em Unix. O padrão é utilizado em conjunto com as linguagens de programação C/C++.

4.2 Criação e término de threads

No instante que um programa começa a executar a função `main()`, uma *thread* é criada. Além dessa *thread* principal, é possível criar várias outras. Porém, não se tem garantia de quando essa *thread* será escalonada, pois quem realiza o escalonamento é o Sistema Operacional. Em *Pthreads*, o serviço que faz essa criação é a primitiva `pthread_create()`. Segue a assinatura para tal primitiva:

```
int pthread_create( pthread_t *tid, pthread_attr_t *atrib, void
                    *(*funcao) (void *), void *args );
```

O valor de retorno dessa função é um número inteiro que determina o sucesso ou o insucesso da criação do *thread*. Quando essa função retorna zero (0), significa que a criação do thread foi bem sucedida.

O ponteiro `tid`, cujo tipo é `pthreade_t,` serve para identificar o *thread* criado e não é possível haver (durante a execução do programa) nenhum outro *thread* com a mesma ID (valor de identificação de uma *thread*).

No momento da criação de um *thread* é possível definir diversos atributos da mesma utilizando o parâmetro `atrib` de tipo `pthread_attr_t`. Quando é especificado o valor NULL nesse parâmetro a biblioteca assume os atributos *default*.

O parâmetro `funcao` recebe a função responsável por executar o trecho paralelo do programa. A assinatura da mesma é dada por:

```
void *thread_funcao( void *args );
```

A saída da função é um ponteiro para uma área de memória que contem os resultados produzidos pela função, esse endereço é do tipo void.

O único parâmetro dessa função é um ponteiro `(*args)` que aponta para uma área de memória onde está armazenado os parâmetros da função. Tais parâmetros, da mesma forma que o valor de retorno, são do tipo *void*.

O quarto argumento da primitiva `pthread_create()` é o endereço de memória onde deve ser recuperado os argumentos passados para a função. Estes argumentos são do tipo *void*.

Para finalizar um *thread*, utiliza-se os serviços `pthread_exit()` ou um simples *return*. Em ambas as situações, o valor informado é o endereço de memória onde se encontra armazenado os valores de retorno do *thread*. A assinatura da função `pthread_exit()` é dada por:

```
void pthread exit ( void *retval );
```

A função `pthread_join` é um modo de sincronização de threads, essa função garante que um thread vai continuar sua execução unicamente quando um determinado thread finalize a execução.

```
void pthread join ( pthread thid, void **ret );
```

O parâmetro thid recebe o numero do thread que vai liberar a função `pthread_join`.

4.3 Mutex

Mutex é uma abreviação para *mutual exclusion* (exclusão mútua). Uma variável *mutex* tem como objetivo implementar sincronização entre threads e proteger dados compartilhados quando da ocorrência de várias threads escreverem dados simultaneamente. Uma variável mutex age como uma “fechadura”, protegendo o acesso a um determinado dado ou recurso compartilhado. O conceito básico de um mutex utilizado em Pthreads é que somente uma thread pode chavear uma variável mutex em um determinado momento. Ainda que diversas threads tentem chavear um mutex, somente uma thread consegue fazer isso com sucesso. Nenhuma outra thread possuirá o mutex até que a thread proprietária o destrave. As threads revezam-se acessando os dados protegidos.

Em Pthreads, variáveis mutex são declaradas pela utilização do tipo `pthread_mutex_t` e devem ser inicializadas antes de serem utilizadas. Existem duas maneiras de inicializar uma variável mutex:

- Estaticamente, quando é declarada. Por exemplo: `pthread_mutex_t meumutex = PTHREAD_MUTEX_INITIALIZER;`
- Dinamicamente, com a função `pthread_mutex_init()`. Este método permite setar os atributos de um objeto mutex, denominado *attr*.

O objeto *attr* é utilizado para estabelecer propriedades para um objeto mutex e deve ser do tipo `pthread_mutexattr_t` quando da sua utilização. Deve ser especificado como NULL inicialmente. As funções `pthread_mutexattr_init()` e `pthread_mutexattr_destroy()` são utilizadas para criar e destruir atributos de objetos *mutex*, respectivamente.

A seguir, encontram-se algumas funções básicas de biblioteca utilizadas para criação e destruição de variáveis do tipo mutex:

```
int pthread_mutex_init( pthread_mutex_t *m, pthread_mutexattr_t *attrib );
int pthread_mutex_destroy( pthread_mutex_t *m );
int pthread_mutex_lock( pthread_mutex_t *m );
int pthread_mutex_unlock( pthread_mutex_t *m );
```

O serviço `pthread_mutex_init()` prepara um mutex para o uso, onde `*m` é a instacia do tipo `pthread_mutex_t` que vai ser inicializada e `pthread_mutexattr_t` serve para definir os atributos para a instancia `*m`. Para definir os atributos default é utilizado o valor NULL. Já o serviço `pthread_mutex_destroy()` serve para inutilizar a instancia `*m`.

O `pthread_mutex_lock()` é um serviço utilizado para adquirir um lock de uma variável mutex especificada. A invocação desse serviço indica que o thread entrou em uma seção crítica. Se o mutex já encontra-se um uso por outra thread, a chamada bloqueia automaticamente a *thread* requisitante até que o *mutex* em questão seja liberado.

O serviço `pthread_mutex_trylock()` tenta “travar” um *mutex*. Entretanto, se o *mutex* já encontra-se em uso, essa função retorna imediatamente um código de erro. Este serviço é útil na prevenção de condições de *deadlock*. Por outro lado, o serviço `pthread_mutex_unlock()` indica que o *thread* chegou ao fim da seção crítica. A função retorna um código de erro se o *mutex* já tiver sido liberado, ou se o mesmo *mutex* está sendo usado por outra *thread*.

4.4 Custo do Mutex

Um problema importante associado com *mutex* é a possibilidade da ocorrência de ***deadlock***. Um programa pode entrar em estado de *deadlock* se duas ou mais threads param ou “trancam” a execução permanentemente. Um exemplo simples de deadlock: a thread 1 obtem um determinado recurso A, a thread 2 obtem um recurso B, a thread 1 deseja ter acesso ao recurso B e a thread 2 deseja obter acesso ao recurso A. Neste caso, as duas threads bloqueiam-se mutuamente, pois cada uma delas espera pela liberação do recurso por parte da outra. Desta forma, as duas threads são impedidas de continuar as execuções de suas tarefas.

Este problema pode ser evitado assegurando que as *threads* obtenham os recursos em uma ordem pré-definida. *Deadlocks* também podem ocorrer se as *threads* não liberam variáveis *mutexes* corretamente.

Outro problema importante relacionado a um *mutex* são as **“condições de corrida”** (*race conditions*). Elas acontecem quando múltiplas threads compartilham dados e pelo menos uma das threads acessa esses dados sem passar por um mecanismo específico de sincronização. Isto pode ocasionar resultados incorretos que tornam-se difíceis de depurar em uma análise posterior.

Uma outra dificuldade com esse tipo de variável é que a disputa por um *mutex* pode levar a uma **inversão de prioridade**. Uma *thread* com prioridade mais alta pode ter de esperar por uma outra *thread* com prioridade mais baixa se esta última estiver usando um recurso que a primeira *thread* está esperando. Este problema pode ser eliminado ou reduzido limitando o número de *mutexes* compartilhados em um sistema de *threads* que possuem diferentes prioridades.

4.5 Afinidade

Para um melhor uso da memória principal, em arquiteturas multiprocessadas ou *multi-core*, é conveniente associar afinidade de *thread* a processadores. Para o programador poder explorar essa estratégia de forma eficiente, faz-se necessário possuir um dominio do programa em questão, pois a utilização desse recurso retira a autonomia do escalonador.

A Fig. 4.1 ilustra uma arquitetura com oito (8) processadores e duas memórias compartilhadas de 4 gigabytes separadas em dois (2) blocos. Cada bloco possui quatro (4) processadores e uma memória de quatro (4) gigabytes e o tempo de acesso à memória do bloco oposto é diferente. O tempo de acesso é indicado pela linha pontilhada vermelha, que indica os tempos de acesso às duas memórias pelo processador um (1). Quando P1 acessa a memória do seu bloco em tempo x1, ao realizar um acesso à memória do bloco oposto o tempo é x3.

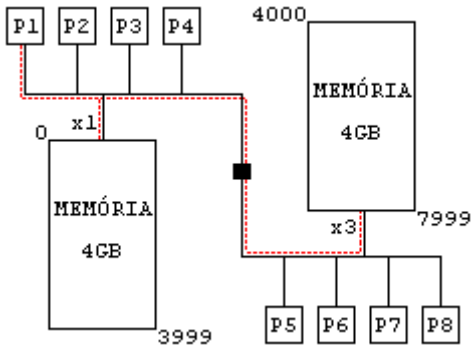

Figura 4.1 – Exemplo de uma arquitetura com 4 processadores compartilhando 2 memórias de 4 gigabytes.

É possível notar que no caso da arquitetura da Fig. 4.1 é conveniente o uso de afinidade de *thread* a processador. Pois ao garantir que um dado que vai ser acessado pelo processador 1 fique na memória, tendo como tempo de acesso x1, ele vai economizar x2 de tempo de acesso em relação ao dado que fica na memória do bloco oposto. Neste trabalho foi utilizada essa técnica de controle da memória *cache*, para que as *threads* tenham um menor tempo de acesso.

Em Pthreads, as chamadas de sistema responsável pelo controle de afinidade são *sched_setaffinity* e *sched_getaffinidade*. A primeira permite associar a afinidade da thread a processador a outra adquire a afinidade já associada. Segue o protótipo de ambas chamadas de sistema.

```
Int sched_setaffinity( pid_t pid, unsigned int tamanho, unsigned long
*mascara );
Int sched_getaffinity( pid_t pid, unsigned int tamanho, unsigned long
*mascara );
```

Em ambiente GNU-Linux, *pid* corresponde ao valor inteiro da identificação de um processo ou *thread*. Para obter o *pid* (*Process Identification*) de um processo pode usar o serviço *getpid*, já para obter o *pid*, ou *tid* (*Thread Identification*), de uma *thread* é preciso usar uma chamada de sistema "*__NR_gettid*", com a primitiva *syscall*. Todas essas, limitam bastante a portabilidade da aplicação. O parâmetro *mascara* é um ponteiro para uma área de memória que contem a mascara de bits identificando os processadores. Cada posição da seqüência de bits da mascara corresponde a um processador. Quando uma posição, na mascara de bits, ta setada

em 1 significa que o processador dessa posição vai ser associado ao thread corrente, no caso do sched_setaffinit, ou para onde é lida a afinidade já lida, no caso do sche_getaffinit. A Fig. 4.2 ilustra o mapa de bits da variável *mascara*.

mascara	p2	p1	p0	
0	0	0	0	Não seta processador
$2^0 = 1$	0	0	1	Seta Processador 0
$2^1 = 2$	0	1	0	Seta Processador 1
3	0	1	1	Seta ambos Processadores
$2^2 = 4$	1	0	0	Seta Processador 2
5	1	0	1	Seta Processadores 0 e 2
6	1	1	0	Seta Processadores 1 e 2
7	1	1	1	Seta os 3 Processadores

Figura 4.2 – Relação do mapa de bits da variável *mascara*.

Note que as posições que definem apenas um processador é potencia de 2. Logo o maior valor é definido por 2^n-1, onde n é o numero de processadores do sistema. Quando ocorre uma falha na operação destes serviços o valor de retorno é -1. Já no caso de sucesso, o sched_setaffinit retorna 0 e o sched_getaffinit retorna o comprimento (em bytes) da mascara retornada.

4.6 Vantagens e desvantagens da utilização de Pthreads

Algumas das vantagens principais de Pthreads são:

- Portabilidade das aplicações escritas utilizando a biblioteca Pthreads para qualquer sistema operacional que suporte o padrão POSIX;
- Voltada para aplicações cuja natureza é expressa através da decomposição de tarefas, onde cada fluxo executa uma tarefa distinta;
- Aperfeiçoamento da performance do programa;
- Redução do *overhead* do sistema;
- Comunicação entre processos distintos ocorre de maneira mais eficiente;
- Aumento de alternativas de execução;
- Maior eficiência na exploração do paralelismo da aplicação.

Algumas das desvantagens principais de Pthreads são:

- Complexidade do programa aumenta devido a utilização de diversos recursos para gerenciamento e sincronização entre as *threads*;
- Ocorrência de *deadlocks*, resultado da complexidade dos programas escritos com a ferramenta;
- Ocorrência de **inversão de prioridade**, quando uma *thread* de prioridade mais alta não executa devido a dependências não resolvidas com outras *threads* de prioridade mais baixa;
- Surgimento de **condições de corrida** entre as *threads* do sistema.

4.7 Conclusão

Para desenvolver este trabalho, foi utilizada a ferramenta de programação *Pthreads*. Este capítulo apresentou apenas uma breve introdução a essa ferramenta de multiprogramação leve, sendo suficiente para o seu entendimento inicial. O uso da variável *mutex* se faz necessário pela presença de uma região critica no programa de Cercato, que será discutida em mais detalhes no próximo capítulo. Porém, levou-se em conta o custo de um *mutex*, visto que a sua invocação transforma-se em uma chamada de sistema, o que acarreta na seqüencialidade das operações definidas dentro dessas regiões críticas. Um dos principais focos deste trabalho é a afinidade de *thread* a processador, que organiza a alocação de memória visando a redução do tempo de acesso.

5 IMPLEMENTAÇÃO REALIZADA

Neste capítulo é apresentado o algoritmo de Random Walker seqüencial e concorrente na forma como foram implementados por Cercato (2006). O algoritmo seqüencial apresentou um desempenho inferior ao algoritmo concorrente. Além disso, é mostrado como foi implementada a afinidade de *threads* a processadores.

5.1 Algoritmo Random Walker Seqüencial

Do algoritmo de Random Walker seqüencial apresentado por (Cercato, 2006), foi considerado o calculo de energia para espuma de sabão e eliminando a parte do código que faz o calculo de energia para sistemas celulares. Segue uma série de passos apresentados no Fluxograma na Fig. 5.1, que compõem o passo de Random Walker:

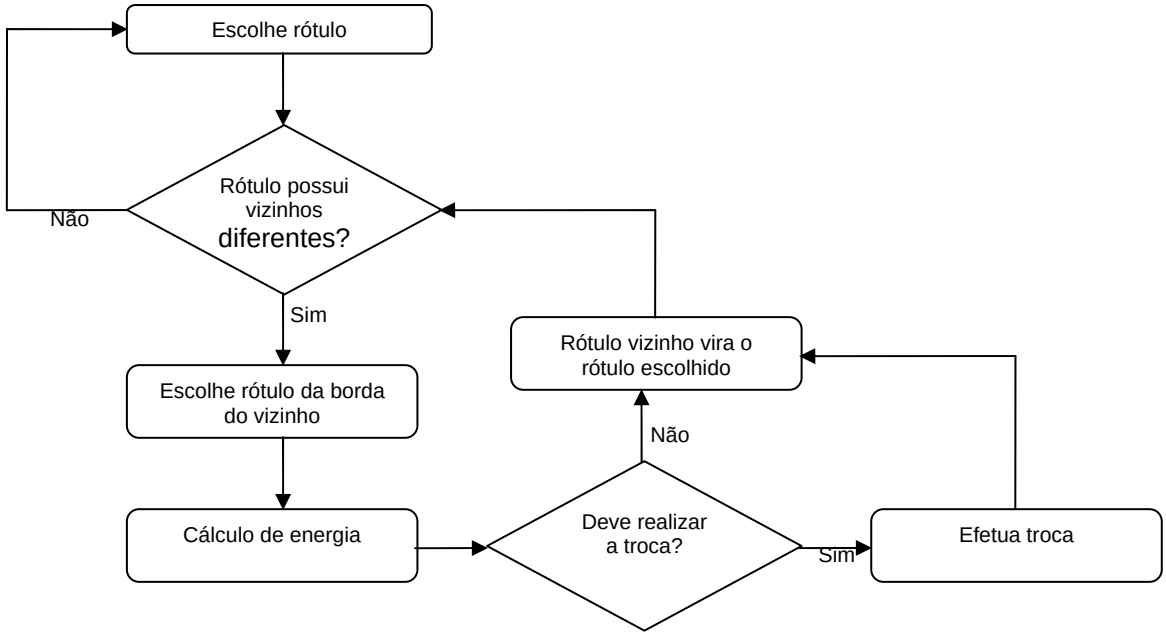

Figura 5.1 – Fluxograma do Random Walker seqüencial.
Fonte: CERCATO, 2005, p.64

Inicialmente, o algoritmo procura por um rótulo cujos vizinhos possuam rótulos numericamente diferentes. Em outras palavras, busca-se um rótulo que situa-

se numa região limítrofe em alguma célula. Assim que encontrar um rótulo com essas características, os 26 rótulos vizinhos a este rótulo escolhido são armazenados em um vetor. Em seguida, os vizinhos válidos ou diferentes ao rótulo escolhido são armazenados em outro vetor à parte. Além disso, utiliza-se um contador para armazenar o número de vizinhos válidos, no caso de não haver vizinhos validos, é realizado um novo sorteio de borda. Caso haja pelo menos um vizinho válido é realizado um sorteio para determinar o vizinho válido escolhido.

O cálculo de energia para espuma de sabão é definido pela diferença entre a energia final e a energia inicial. A energia inicial do sistema corresponde ao número de vizinhos diferentes do rótulo escolhido. Já a energia final corresponde ao número de vizinhos diferentes do rótulo vizinho escolhido. Caso a diferença de energia seja menor do que zero, então a substituição ocorre sem qualquer tipo de restrição. Neste caso, assume-se a probabilidade de troca em 100%. Caso a energia do sistema seja igual a zero, é considerada a possibilidade de ocorrer a substituição do rótulo escolhido pelo rótulo vizinho sorteado. Neste caso, é sorteado um valor numérico entre 0 e 100 representando um percentual. Se o valor sorteado for maior do que a probabilidade de troca, também determinada no mesmo intervalo, então a substituição ocorre de fato.

Para continuar caminhando na borda da célula o escolhido tem que receber a posição do rótulo vizinho escolhido, garantindo assim que continue caminhando na borda da célula. O numero de rótulos percorrido é equivalente ao numero de rótulos em borda da célula. Esse processo é chamado de passo de Random Walker e define a unidade de tempo do algoritmo.

A inicialização da matriz é efetuada por três funções recursivas, cada uma delas responsável por tratar dos vetores relacionados com as três dimensões da matriz: altura, largura e profundidade. As funções estipulam valores aleatórios para o tamanho do bloco a ser preenchido, de acordo com o volume mínimo definido. Assim que o volume mínimo for atingido, inicia-se o preenchimento dos blocos. Este processo repete-se inúmeras vezes, até que toda matriz tenha sido preenchida.

5.2 Algoritmo Random Walker Concorrente

O algoritmo concorrente de *Random Walker*, implementado por Cercato, explorou o paralelismo natural do problema disparando diversos RW

simultaneamente, onde cada RW disparado é uma *thread*. Essas *threads* compartilham informações de uma mesmo conjunto de dados. Portanto, quando duas ou mais *threads* atuam em regiões próximas umas das outras, pode haver uma sobre-escrita de algum valor quando da substituição de um rótulo.

Para evitar esse problema, que resulta em uma situação de inconsistência para o sistema como um todo (os dados manipulados pelo programa tornam-se desatualizados), faz-se necessário utilizar variáveis *Mutex* de modo a deixar seqüencial o acesso a determinada região critica.

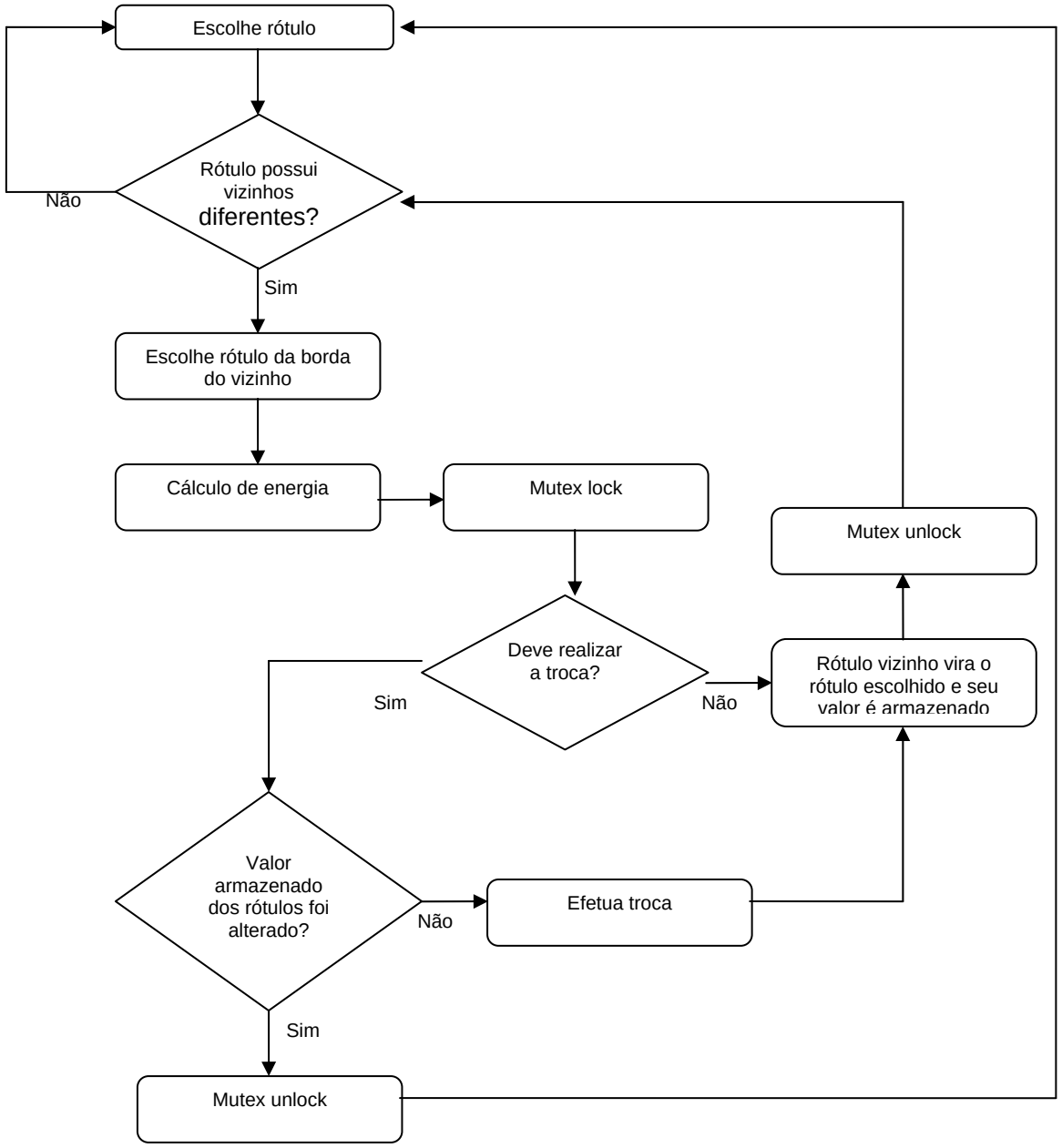

Figura 5.2 – Fluxograma de execução para cada thread executando o algoritmo de Random Walker.

Fonte: CERCATO, 2005, p.71

5.3 Estratégias de execução

Neste trabalho foram utilizadas técnicas de afinidade de *thread* a processador, para tentar extrair um maior potencial de desempenho das arquiteturas *multi-core*.

A aplicação trabalha com uma matriz tri-dimensional, de tamanhos **tamX**, **tamY**, **tamZ**, representados por vetores. Os vetores dos dois primeiros planos, representados por **X** e **Y**, consistem em endereços de memória para os planos imediatamente inferiores: o vetor referente ao plano **X** possui apontadores para os vetores representando diferentes cortes no plano **Y**; o vetor referente ao plano **Y** possui apontadores para cortes no plano **Z**. O plano **Z**, por sua vez, contém os valores correspondentes às posições referenciadas por uma tripla **[x,y,z]**. A Fig. 5.3 ilustra o arranjo dos vetores armazenados na memória, supondo **tamX = tamY = tamZ = 10**.

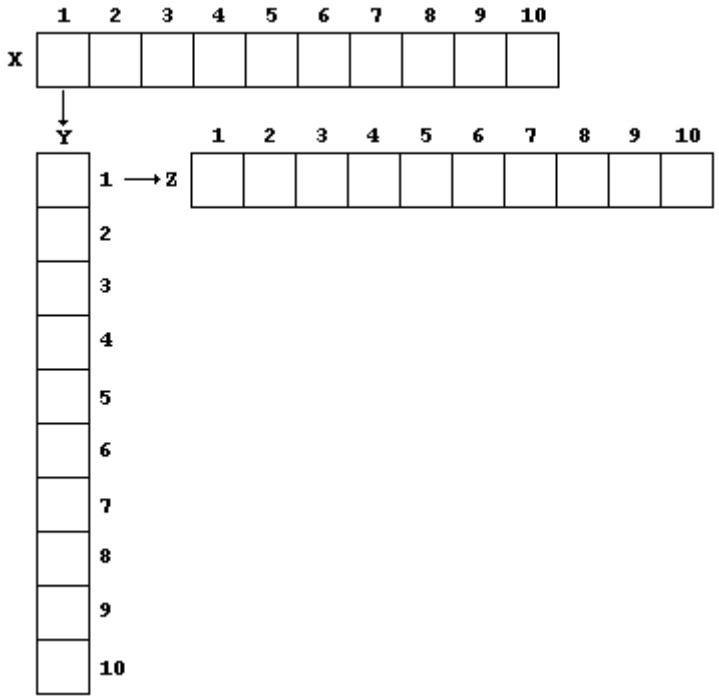

Figura 5.3 – Arranjo da memória alocada para formar a matriz.

Estes vetores são alocados dinamicamente, conforme o tamanho da rede fornecida como entrada. Na linguagem de programação C, a alocação dinâmica de memória reserva um espaço de endereçamento contíguo conforme o tamanho solicitado pela chamada de sistema correspondente. Ressalta-se que, em ambientes GNU-Linux, a área de memória é efetivamente obtida quando for realizado o primeiro acesso à referida área (Vahali, 1995). Portanto, em uma arquitetura multi-processada que possui uma hierarquia de memória entre os processadores, esta propriedade pode ser explorada para obter melhores índices de desempenho na execução de programas.

Assim, a atribuição de afinidade de execução de threads a processadores pode auxiliar na exploração efetiva da memória. Isto pode ser obtido determinando que algumas *threads* específicas executem em processadores próximos aos módulos de memória que contenham os dados por eles manipulados.

A Fig 5.4 ilustra a forma que ficou os vetores depois de ter estendido a matriz.

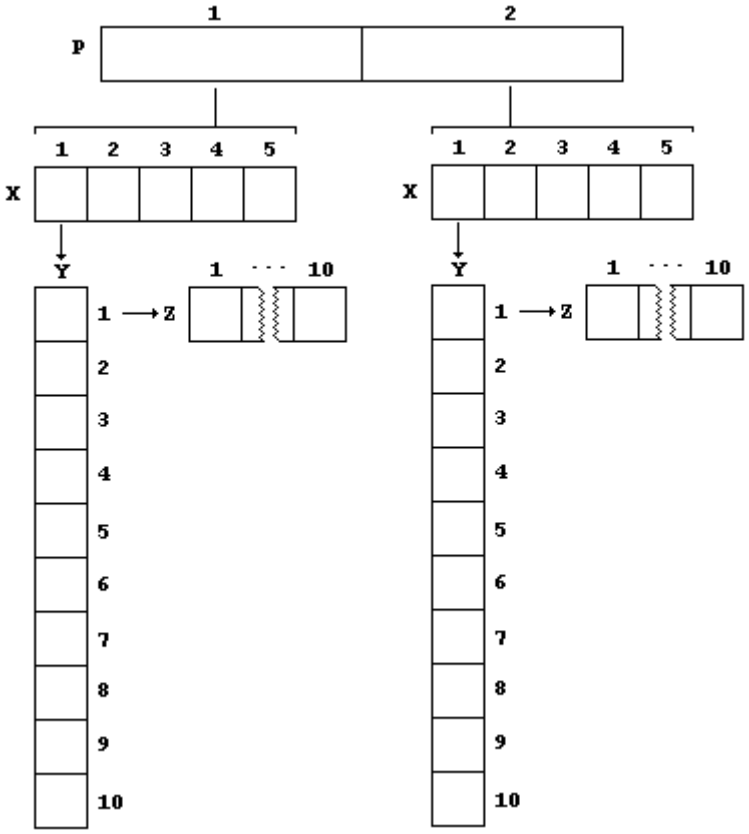

Figura 5.4 – Arranjo da memória com a matriz de quatro dimensões.

A implementação realizada foi estendida para uma matriz de quatro dimensões. Esta dimensão a mais foi definida com o intuito de dividir o eixo X. Com essa divisão é possível controlar em qual memória vai ser alocada a matriz. Houve a necessidade de criar controles em todas as outras funções ao longo do código, para garantir que a matriz fosse interpretada da forma como foi estendida.

5.4 Conclusão

Foi apresentado neste capítulo, de forma breve, ambas as implementações do mesmo algoritmo de *Random Walker* (seqüencial e paralelo), utilizadas originalmente por Cercato em seu trabalho (CERCATO, 2005). Discutiu-se alguns aspectos importantes a respeito da funcionalidade geral de algumas funções pertencentes aos algoritmos, sem no entanto esmiuçar o código-fonte em detalhes. Em seguida, foi explicada uma proposta para aperfeiçoar o algoritmo de *Random Walker* paralelo. Tal proposta baseia-se no controle de afinidade de thread a processadores, que se mostra como uma alternativa bastante eficiente para a alocação de memória em arquiteturas NUMA. Nesse trabalho foi desenvolvido o algoritmo Random Walker paralelo com e sem o controle de afinidade de thread a processadores.

6 RESULTADOS DE DESEMPENHO

Este capítulo apresenta os resultados de desempenho para a implementação paralela realizada neste trabalho do algoritmo Random Walker paralelo. Esta implementação foi estendida para utilizar técnica de controle de afinidade de thread a processadores apresentada anteriormente. Os experimentos realizados comparam o desempenho de execução do Random Walker paralelo com e sem o uso de controle de afinidade. Na seção 6.1 é apresentada as duas configurações utilizadas para realizar os testes. A seção 6.2 apresenta os resultados obtidos com os dois algoritmos, um com controle de afinidade ativado e o outro sem esse controle. Já a seção 6.3 é avaliado os resultados obtido na seção 6.2. Por ultimo a seção 6.4 finaliza o capítulo.

6.1 Ambiente de Experimentação

Os experimentos foram realizados em duas configurações distintas de computadores. A primeira configuração é a de um computador dotado de processador dual-core, a segunda de um computador dotado de oito núcleos de processamento. Estes dois computadores são identificados na seqüência como Desktop e IDBull na seqüência deste capítulo e suas características são apresentadas na Tabela 6.1. Conforme indicado nesta tabela, a máquina IDBull possui uma configuração NUMA (Non-Uniform Memory Access), onde a memória é organizada em blocos de 8 GB a cada grupo de 4 processadores.

Tabela 6.1 – Computadores utilizados na avaliação de desempenho.

Configuração	Desktop	IDBull
Processador	Intel Pentium VI 1.8GHZ	Intel Itanium 1.5 GHZ
Número de processadores	1	8
Cores por processador	2	1
Memória disponível	1 GB	16 GB
Configuração de memória	UMA	NUMA

As duas versões do programa, com e sem uso de controle de afinidade, foram avaliadas considerando a exploração de diferentes níveis de paralelismo do

hardware. Foram coletados dados de desempenho considerando o uso de 1, 2, 4, 6 e 8 threads, cada thread executando o algoritmo de RW. Também foi variado o tamanho do problema, utilizando como entrada da simulação diferentes tamanhos de matrizes: 100, 250 e 500. Houve tentativas de executar matrizes de 750 e 1000, porém não foi apresentado os resultados pois a configuração Desktop não tinha memória suficiente para executar essas matrizes.

Cada resultado apresentado corresponde a uma média de 10 execuções, com desvio padrão máximo observado de 11%. Foram descartadas todas as execuções cujo tempo extrapolou este desvio padrão. Observa-se que maior o número de threads utilizadas, maior o desvio padrão observado no caso, fato justificado devido a maior probabilidade de interferência de eventos externos ao programa na sua execução. O número total de execuções aproveitadas foi 600: 2 versões x 5 opções de paralelismo x 3 tamanhos de entrada x 10 repetições x 2 configurações de hardware.

6.2 Resultados Obtidos

Os resultados coletados encontram-se sumarizados nas tabelas 6.2 e 6.3. Observe-se que, para análise dos resultados, o tempo de execução seqüencial corresponde ao tempo de execução com um (1) thread. Ressalta-se, no entanto, que este tempo de execução seqüencial inclui os sobre-custos de sincronização.

Tabela 6.2 – Tempo médio de execução na configuração IDBull.

Tamanho da Matriz	Número de threads	Tempo Médio (segundos)	
		Sem afinidade	Com afinidade
100 x 100 x 100	1	22,36997	22,498990
	2	12,73943	12,584100
	4	7,219932	7,173314
	6	5,565242	5,799387
	8	4,780586	4,567481
250 x 250 x 250	1	360,969600	361,446304
	2	191,835300	190,826985
	4	107,120900	106,380032

	6	79,666410	78,474802
	8	65,472050	64,545074
500 x 500 x 500	1	2912,988591	2916,778737
	2	1543,154039	1539,631252
	4	855,3689524	849,0826018
	6	630,1254726	623,4535259
	8	521,1554691	512,911467

Tabela 6.3 - Tempo médio de execução na configuração Desktop.

Tamanho da Matriz	Número de threads	Tempo Médio (segundos)	
		Sem afinidade	Com afinidade
100 x 100 x 100	1	8,042759	8,207915
	2	4,888827	4,027674
	4	4,415655	4,031369
	6	4,237902	4,037969
	8	4,229769	4,064162
250 x 250 x 250	1	121,5338823	123,748657
	2	63,9771114	64,9161526
	4	65,4646316	65,6952208
	6	64,5238859	64,3628185
	8	64,4099782	64,3918257
500 x 500 x 500	1	962,3007	971,367848
	2	508,9871	512,450576
	4	503,3513	509,478788
	6	502,9497	506,786976
	8	503,0683	506,402858

6.3 Avaliação de Desempenho

Esta seção faz um apanhado dos desempenhos apresentados nas seções 6.1 e 6.2, colocando-as lado-a-lado e discutindo os resultados obtidos.

O gráfico na Figura 6.1 apresenta curvas comparando o desempenho de execução do programa Random Walker paralelo tendo como entrada uma matriz de tamanho 100^3. As curvas apresentam os tempos de execução medido nas duas arquiteturas considerando as versões com e sem afinidade. Observa-se que o melhor desempenho na configuração Desktop foi atingido com 2 threads tendo sido ativado o controle de afinidade. O aumento do número de threads não refletiu melhora de desempenho, possivelmente dado ao pequeno tamanho da matriz de entrada implicando em um reduzido tempo total de processamento reduzindo o impacto da interferência do controle de afinidade no escalonamento dos threads. Na configuração IDBull, os resultados permitem concluir que, para a referida arquitetura, não houve diferença de desempenho, provavelmente também devido ao pequeno tamanho da entrada do problema.

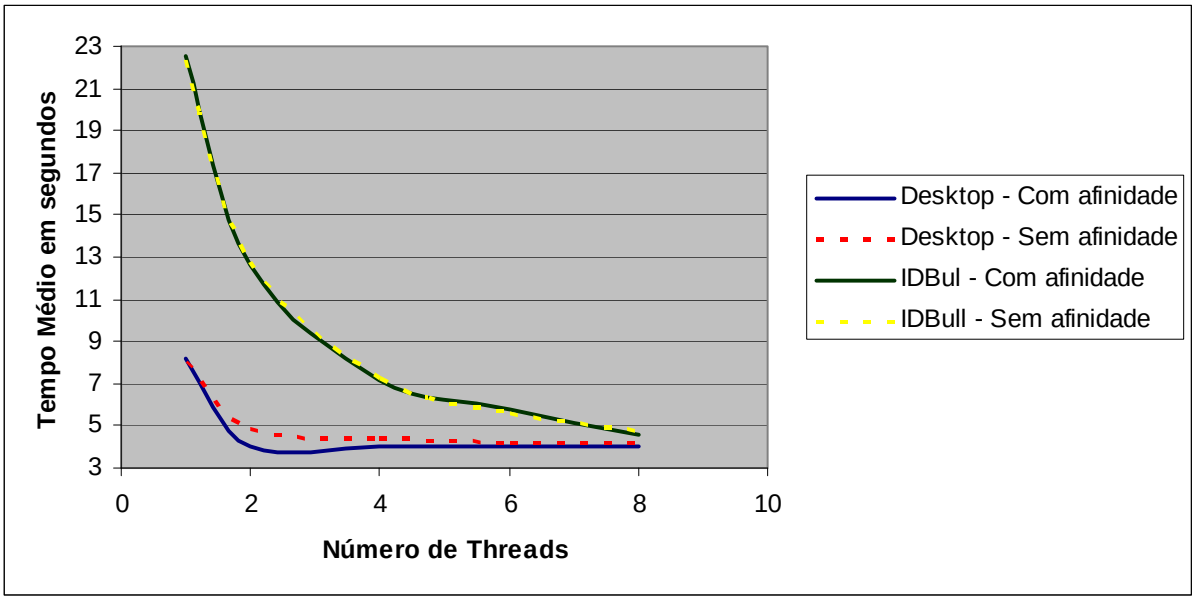

Figura 6.1 – As curvas de tempo com Desktop e IDBull para matriz 100^3

Observa-se na Figura 6.1 que o tempo de processamento requerido para execução do cálculo com controle de afinidade nas duas arquiteturas com um (1) único thread é maior que o tempo necessário para execução sem este controle. Isto deve ser conseqüência do não aproveitamento do paralelismo oferecido pelas arquiteturas, sequencializando a execução do programa em função das decisões de escalonamento do sistema operacional.

A Figura 6.2 apresenta o gráfico dos resultados de desempenho das duas configurações com matriz de 250^3. Neste experimento foi aumentado o tamanho da entrada passando de 100^3 para 250^3, para avaliar se um maior peso computacional reflete-se em uma maior diferença de comportamento das versões da aplicação utilizando ou não o controle de afinidade. Observa-se, neste gráfico, que para este tamanho de entrada, os tempo de execução são praticamente o mesmo em cada uma das configurações, não havendo ganho nem perda de desempenho com o uso do controle de afinidade.

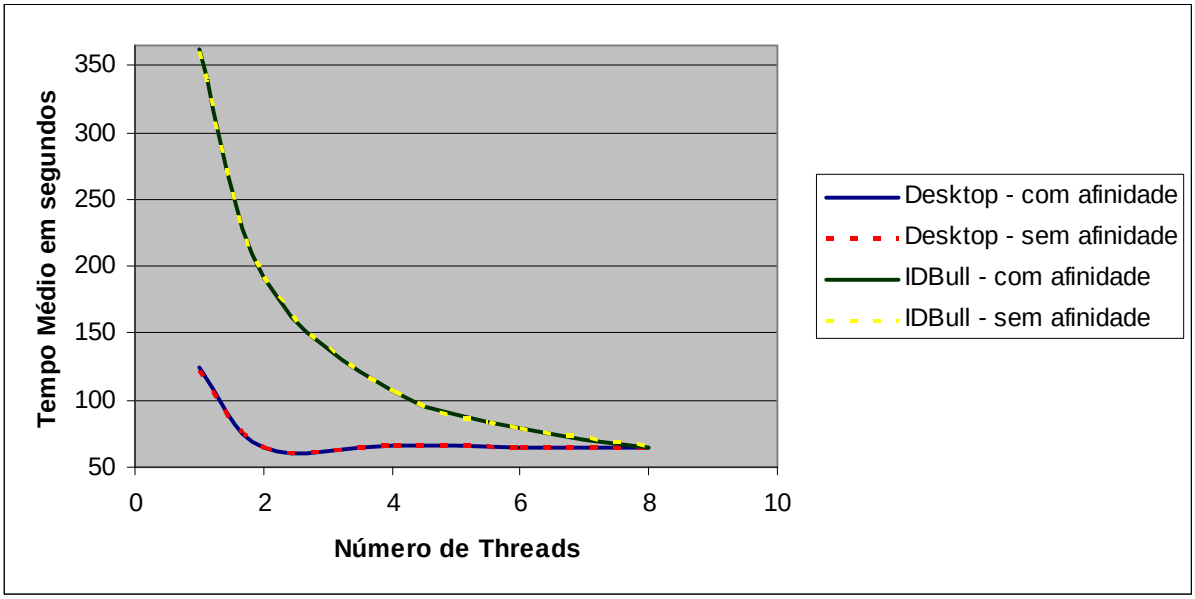

Figura 6.2 – As curvas de tempo com Desktop e IDBull para matriz 250^3

O gráfico na Figura 6.3 apresenta os tempos coletados com a maior entrada considerada no conjunto de experimentos realizados: matriz 500^3. Nesta situação, o uso do controle de afinidade permitiu ganho de desempenho na configuração IDBull, mas teve influência negativa no tempo de processamento na configuração Desktop. Esta situação pode ser explicada pela capacidade da implementação explorar de forma mais efetiva o paralelismo disponibilizado pela configuração IDBull e extrapolar a capacidade de processamento oferecida pela configuração Desktop.

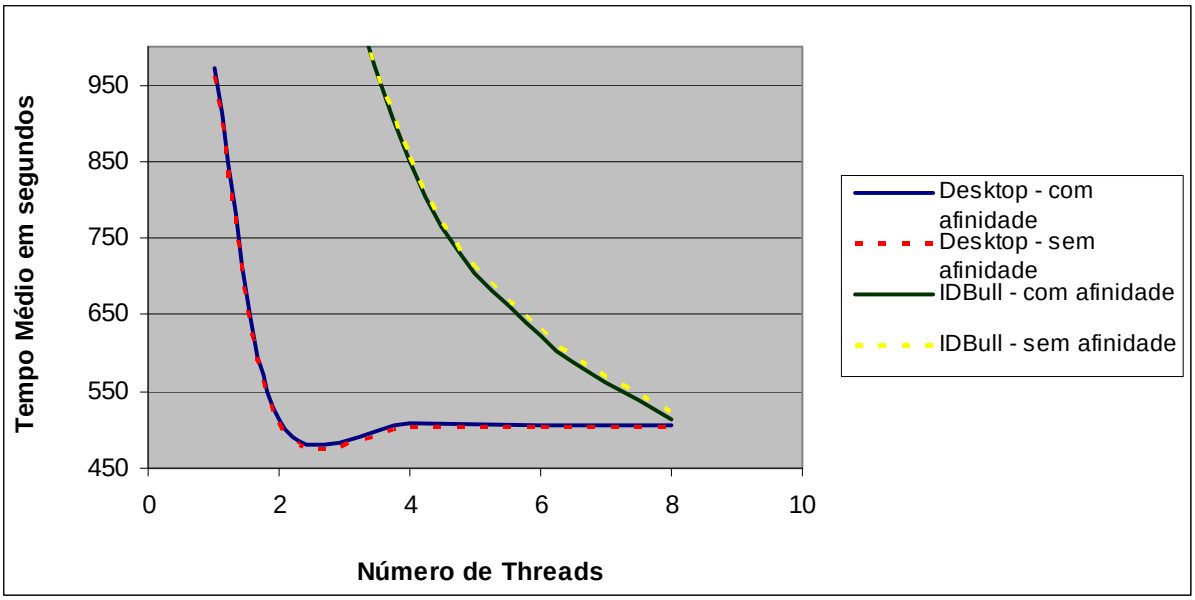

Figura 6.3 – As curvas de tempo com Desktop e IDBull para matriz 500^3

O melhor resultado, em relação aos algoritmos com e sem afinidade, foi na configuração IDBull. Onde o algoritmo com controle de afinidade de thread a processadores tem um ganho de desempenho de 4,46% em relação ao algoritmo sem afinidade.

6.4 Discussão Final

Desde a versão 2.5 do kernel do Linux, o escalonador do sistema operacional incorpora mecanismo que buscam otimizar a relação de afinidade de threads com processadores. O experimento conduzido neste trabalho avaliou o impacto do uso na camada aplicativa de técnicas de manipulação do controle de afinidade. Os resultados mostraram que existe possibilidade de ganho de desempenho, sendo este ganho observado em alguns experimentos. No entanto, situações onde a aplicação do controle da afinidade ofereceu resultados inferiores ao não uso desta técnica indica que outros fatores atuam no comportamento da execução.

Em (Foong, 2008) encontra-se documentado outro experimento para avaliação do impacto do uso do controle de afinidade. O contexto desse trabalho relaciona o uso do controle de afinidade em uma aplicação de rede (protocolo TCP). Embora de natureza diversa, os resultados que estes autores encontraram possuem

semelhanças com os obtidos no presente trabalho, uma vez que o ganho de desempenho por eles registrados não apresenta uma significativa diferença no tempo de processamento.

Foong (2008) e seus colegas concluem afirmando que aplicar unicamente a técnica de controle de afinidade não garante melhora de desempenho de execução.

7 CONCLUSÃO

A principal contribuição deste trabalho foi a avaliação do impacto do uso de técnicas de controle de afinidade de threads a processadores. Foi realizado um experimento onde uma aplicação paralela foi implementada em duas versões, com e sem o uso de técnica de controle de afinidade, sendo os tempos de execução destas duas versões avaliado em duas arquiteturas paralelas distintas.

Este trabalho apresentou a implementação paralela com threads do modelo Potts Celular estendido com o algoritmo de Random Walker. Esta implementação mostrou um melhor desempenho que a tradicional implementação utilizando o algoritmo de Monte Carlo (Cercato, 2005). Para a implementação deste trabalho utilizado o algoritmo de Random Walker Paralelo desenvolvido por Cercato (2005). Esse algoritmo foi estendido para receber a estratégia de programação, que faz o controle de afinidade de threads a processadores.

Os testes foram realizados em duas máquinas com configurações distintas, onde a máquina Desktop possuía uma configuração de memória UMA (Uniform Memory Access) e a outra (IDBull) com uma configuração de memória NUMA (Non-Uniform Memory Access). O esperado nos testes era que fosse obtido um melhor desempenho na segunda configuração, pois o controle de afinidade consiste em garantir que o processador busque a informação, que ele precisa, no local mais próximo a ele. Porem em alguns casos, a configuração Desktop obteve um ganho em desempenho com a implementação com controle de afinidade. Esse ganho de desempenho foi visto em testes onde a quantidade de memória usada era baixa, ou seja, onde o custo da troca de contexto não era grande. Por isso, a quantidade de memória da máquina esta diretamente ligada ao desempenho da simulação, assim como o volume da simulação esta diretamente ligada à quantidade de memória.

A principal conclusão obtida nos experimentos realizados é que o uso de controle de afinidade pode resultar em um ganho de desempenho em termos de tempo de execução. No entanto, a portabilidade deste recurso é restrita, tendo em vista as diferenças de desempenho obtidos nas duas configurações utilizadas. Outra dificuldade observada no uso desta técnica deve-se ao fato que ela reflete uma decisão de escalonamento estática, ou seja, definida no código do programa. Para aplicação utilizada como estudo de caso neste trabalho, este aspecto não teve

influência negativa, pois a entrada do problema, a matriz de simulação, é conhecida previamente. Outras aplicações onde a geração de carga computacional é dinâmica deve oferecer outros complicadores para determinação de afinidade.

A seqüência natural do presente trabalho é avaliar técnicas de escalonamento de aplicações estáticas em arquiteturas NUMA, onde existe a possibilidade de observar maior impacto na exploração da hierarquia de memória. Em relação a presente implementação, outras técnicas de otimização de código podem também ser aplicadas, melhorando o desempenho de execução deste modelo de simulação. Dentre novas otimizações, podem ser citados a utilização de algoritmos não bloqueantes (livres de sincronização) e utilização de mecanismos de aproveitamento de área de memória cache pela compactação de dados.

Referências

BURGER, D., GOODMAN, J. R. **Billion-transistor architectures – guest editor's introduction**. *IEEE Computer*, 30(9):46-49, 1997.

CAVALHEIRO, Gerson Geraldo H.; SANTOS, Rafael R. Multiprogramação leve em arquiteturas multi-core. In: Atualizações em Informática. Cap 7. Rio de Janeiro: PUC-Rio. 2007.

CERCATO, F. P., MOMBACH, J. C. M., CAVALHEIRO, G. G. H. **High Perfomance simulations of the clellular Potts model.** In: HPCS 2006 - International Symposium on High Performance Computing Systems and Applications, 2006, Saint Johns. XX International Symposium on High Performance Computing Systems and Applications. Los Alamitos: IEEE Computer Society, 2006.

CERCATO, Fernando Piccine. **Um Algoritmo de Alto Desempenho para Evoluir o Modelo de Potts Celular.** São Leopoldo: UNISINOS, 2005.

FLYNN, M. J. **Some computer organizations and their effectiveness**. *IEEE Transactions on Computers*, Vol. C-21, pp. 948, 1972.

FOONG, Annie; Fung, Jason; NEWELL, Don. Improved Linux* SMP Scaling: User-directed Processor Affinity. http://softwarecommunity.intel.com/articles/eng/1781.htm (acesso em 5 de agosto de 2008). 2008.

GRANER, F. **Can suface adhesion drive cell rearrangement? Part i: Biological cell-sorting**. *Journal of Theoretical Biology*, v. 164, p. 455-476, 1993.

GRANER, F.; GLAZIER, J. A. **Simulation of biological cell sorting using a twodimensional extended potts model**. *Physical Review Letters*, v. 1, n. 69, p. 2013-2016, 1992.

GUZATTO, É., MOMBACH, J. C. M., CERCATO, F. P., CAVALHEIRO, G. G. H. **An efficient parallel algorithm to evolve simulations of the cellular Potts model.** Parallel processing letters. v.1, p.199 - 208, 2005.

KNEWITZ, Marcos André. **Um Modelo para Investigação do Crescimento e da Morfologia de Tumores**. Dissertação de mestrado. São Leopoldo: UNISINOS, 2002.

NEWMAN, M. E. J.; BARKEMA, G. T. **Monte Carlo Methods in Statistical Physics.** 1. ed. [S.l.]: Oxford University Press, 1999.

VAHALI, Uresh; Sulus, Peter H. **UNIX Internals: The New Frontiers.** 1.ed. New Jersey: Prentice Hall, 1995. 601p.