UNIVERSIDADE FEDERAL DE PELOTAS
INSTITUTO DE FÍSICA E MATEMÁTICA
DEPARTAMENTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

AVALIAÇÃO DA ESTRATÉGIA DE IMPLEMENTAÇÃO CONCORRENTE DO ALGORITMO DE RANDOM WALKER PARA O MODELO DE POTTS CELULAR

Leonardo Lobo da Luz

Pelotas, 2008

Leonardo Lobo da Luz

AVALIAÇÃO DA ESTRATÉGIA DE IMPLEMENTAÇÃO CONCORRENTE DO ALGORITMO DE RANDOM WALKER PARA O MODELO DE POTTS CELULAR

Trabalho acadêmico 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

L979a Luz, Leonardo Lobo da

Avaliação da estratégia de implementação concorrente do algoritmo de Random Walker para o modelo de potts celular / Leonardo Lobo da Luz; orientador Gerson Geraldo H. Cavalheiro. – Pelotas, 2008. – 85f.- 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 multi-core. 3.Multithreading. 4.Modelo de potts celular. 5.OpenMP. I. Cavalheiro, Gerson Geraldo H. II.Título.

CDD: 005.13

Agradecimentos

Quero agradecer, em primeiro lugar e acima de tudo, às duas pessoas no mundo que eu mais amo: meu pai, Gilberto Cardozo da Luz, e minha mãe, Clarice Lobo da Luz. Sem o amor, carinho, dedicação e apoio dessas duas pessoas extraordinárias eu jamais teria conseguido chegar até aqui e conquistado todas as coisas que conquistei até hoje. Muito obrigado por tudo, principalmente por ter me proporcionado o maior de todos os presentes: o dom da vida!

Gostaria de agradecer também a todos os professores com quem tive algum contato durante os meus anos de faculdade. Principalmente, aos professores do curso de Ciência da Computação: Gil Medeiros, Marcello Macarthy, Eliane Diniz, Flávia Azambuja, Colvara e Simone da Costa. Pelo seu comprometimento, dedicação, didática e, principalmente, por me fazerem ver a vida com outros olhos. Aprendi muitas coisas com todos vocês e sou muito grato por isso.

Também gostaria de agradecer a todas as pessoas que, de uma forma ou de outra, tiveram um impacto positivo em minha vida. Principalmente aos meus colegas e amigos de faculdade, Cristian Castañeda e Alexandre Costa, por terem colaborado e participado, de alguma maneira, deste meu trabalho de conclusão de curso. E aos tantos outros colegas e amigos, de faculdade ou não, que eu acabei por conhecer.

E, por último, mas não menos importante, agradeço ao meu orientador Gerson Geraldo H. Cavalheiro. Obrigado pela sua dedicação, amizade e apoio. E principalmente, por ter confiado em mim como aluno. Espero ter atingido as tuas expectativas com este trabalho, professor.

"Toda a nossa ciência, comparada com a realidade, é primitiva e infantil – e, no entanto, é a coisa mais preciosa que temos."

Albert Einstein (1879 – 1955)

Resumo

LUZ, Leonardo Lobo da. **Avaliação da Estratégia de Implementação Concorrente do Algoritmo de Random Walker para o Modelo de Potts Celular**. 2008. 85f. Monografia – Curso de Bacharelado em Ciência da Computação. Universidade Federal de Pelotas.

O modelo de *Potts* Celular é muito utilizado em diversas áreas científicas – tais como a Física, a Biologia, a Química – para evoluir, por meio de simulação, sistemas de agregados celulares, bolhas de sabão e tecido cancerígeno, por exemplo. Através de simulações construídas com base neste modelo, é possível conhecer, estudar e analisar os processos de interação entre partículas de tais sistemas no tempo. Tradicionalmente, o modelo de *Potts* Celular é aplicado em conjunto com a técnica de simulação de Monte Carlo em sistemas computacionais. No entanto, quando implementado desta forma, o desempenho final de execução – tempo de processamento – é baixo, uma vez que a técnica de Monte Carlo mostra-se ineficiente quando aplicada ao modelo de *Potts* Celular. Este fato deve-se não só ao elevado número de cálculos matemáticos realizados durante a simulação, mas também à ineficiente estratégia de seleção de rótulos para evoluir o sistema simulado. Na prática, é comum encontrar implementações paralelas de Monte Carlo para reduzir o impacto de seu alto custo computacional. Como alternativa, apresenta-se a técnica de *Random Walker*, um algoritmo que emprega uma heurística de seleção de rótulos mais eficiente que a aplicada no método de Monte Carlo. Esta estratégia de execução, além de ser mais eficiente, também possui alto potencial de paralelização. Este trabalho apresenta as implementações paralelas dos algoritmos de Monte Carlo e Random Walker para evolução do modelo de Potts Celular e fornece dados comparativos dos resultados de simulação obtidos com a execução destas duas versões da aplicação. Devido ao seu elevado custo computacional, as arquiteturas multi-core serão utilizadas como suporte para a execução das implementações.

Palavras-chave: Multi-core. Modelo de Potts Celular. Random Walker. Monte Carlo.

Abstract

LUZ, Leonardo Lobo da. **Evaluation of Strategy of Parallel Implementation of the Algorithm of Random Walker to the Cellular Potts Model**. 2008. 85f. Monografia – Curso de Bacharelado em Ciência da Computação. Universidade Federal de Pelotas.

The Cellular Potts Model it's too much used in several scientific areas – such as the Physics, the Biology, the Chemistry – to evolve, by means of simulation, systems of cellular aggregations, blisters of soap and cloth carcinogen, for example. From one side to the other simulations constructed based on this model, it is possible to know, study and analyze the suits of interaction among particles of one systems into the time. Traditionally, the Cellular Potts Model is applied in conjunction with the technique of simulation of Monte Carlo in computational systems. However, when implemented from this way, the performance end of execution – time of processing – is down, since that the technique of Monte Carlo it show inefficient when applied to the Cellular Potts Model. This fact owes – if not only the elevated number of mathematicians calculations realized during the simulation, but also by on the inefficient strategy of selection of inscriptions to evolve the system simulator. In practice, it is common meet parallel implementations of Monte Carlo to reduce the impact of your high computational cost. As an alternative, it is presented the technique of Random Walker, an algorithm that employs only one heuristic of selection of inscriptions all-around what the applied into the method of Monte Carlo. This strategy of execution, in addition to be more efficient, also has high potential of parallelism. This paper presents the parallel implementations from the algorithm of Monte Carlo and Random Walker about to evolution of the Cellular Potts Model and provides comparative data from the results of simulation obtained with the execution of these two versions from application. Due to your elevated computational cost, the multi-core arquitetures will be used as a suport for execution for the implementations.

Keywords: Multi-core. Cellular Potts Model. Random Walker. Monte Carlo.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

2D – Bi-dimensional

3D – Tri-dimensional

AMD – *Advanced Micro Devices*

API – *Application Programming Interface*

CAM – *Cell Adhesion Molecule*

DAH – *Differential Adhesion Hypothesis*

MC – Monte Carlo

MIMD – *Multiple Instruction Stream, Single Data Stream*

MISD – *Multiple Instruction Stream, Single Data Stream*

MP – *Message Passing*

NUMA – *Non-uniform Memory Acess*

RISC – *Reduced Instruction Set Computer*

RW – *Random Walker*

SISD – *Single Instruction Stream, Single Data Stream*

SIMD – *Single Instruction Stream, Multiple Data Stream*

SMP – *Symmetric Multiprocessing*

TLP – *Thread Level Parallelism*

UMA – *Uniform Memory Acess*

Sumário

1 Introdução

Desde que surgiu na face da Terra, o ser humano, de uma maneira ou de outra, sempre tentou compreender a complexidade da natureza. Para tanto, o homem fez uso de seu elevado poder de cognição e criou teorias e modelos matemáticos que permitiram explicar de forma razoavelmente satisfatória o que acontece no mundo natural (LOUREIRO, 2006).

No campo da Física, um dos modelos que ajudam a compreender os fenômenos naturais é o modelo de *Potts* Celular. Este modelo procura determinar, através de um processo de simulação, o comportamento de partículas em um sistema natural (físico, biológico ou matemático) e a forma como estas partículas interagem umas com as outras (CERCATO, 2005). Tal modelo é utilizado para simular, por exemplo, a interação entre partículas, estejam estas representando bolhas de sabão, partículas atômicas ou células do corpo. Devido ao fato desses sistemas envolverem uma quantidade muito grande de variáveis, o processo de simulação requer um tempo de processamento elevado.

Alternativas têm sido apresentadas para melhorar o desempenho de execução de simulações baseadas neste modelo. Estas alternativas incluem tanto a modificação do algoritmo básico de simulação quanto a utilização de recursos de processamento paralelo. Nos trabalhos de Gusatto (2004) e Gusatto e colegas (GUSATTO et al, 2005) é apresentado o uso da técnica de *Random Walker* como mecanismo para realizar a evolução da simulação proposta pelo método *Potts* Celular de forma a reduzir o tempo de processamento necessário para simulação de um sistema.

Os trabalhos de Cercato e colegas (CERCATO et al, 2006) e de Alexandre Costa (COSTA, 2007) propõem a adição de técnicas de execução paralela ao algoritmo de *Random Walker* de forma a obter ainda melhores índices de desempenho.

No entanto, os resultados das simulações apresentados pela combinação destas soluções não foram validados em relação aos resultados oferecidos pela aplicação das técnicas convencionais de simulação.

Este trabalho se propõe a realizar uma implementação paralela do modelo de *Potts* Celular e comparar o resultado da simulação com o resultado apresentado por uma implementação seqüencial do modelo convencional. A aplicação paralela foi implementada em C utilizando a interface de programação *multithread* OpenMP (CHANDRA et al, 2001). A comparação entre os resultados foi obtida pela aplicação da Lei de *von Neumann* (HILGENFELDT, 2001).

1.1 Motivação

Em sua dissertação, Cercato (2005) propôs uma versão distribuída do algoritmo de *Random Walker*, tornando possível sua execução em aglomerados de computadores (*clusters of computers*). Esta implementação possibilitou que simulações de sistemas celulares fossem realizadas com um alto grau de realismo e sofisticação com tempo de processamento razoável considerando a complexidade experimentada. O ganho de desempenho obtido com a utilização do algoritmo de *Random Walker* paralelo em relação ao algoritmo tradicional de Monte Carlo foi bastante acentuado conforme os resultados relatados nos trabalhos de Cercato (2005) e Cercato e colegas (CERCATO et al, 2006).

Entretanto, em relação aos resultados de simulação obtidos pela implementação de *Random Walker* paralelo, questiona-se se os mesmos seriam equivalentes aos resultados gerados por simulações utilizando a implementação seqüencial tradicional do método de Monte Carlo.

1.2 Objetivos

O objetivo principal deste trabalho foi o de apresentar resultados quantitativos gerados por simulações concorrentes do modelo de *Potts* Celular que permitam validar ou refutar o algoritmo concorrente de *Random Walker*, através da

verificação de sua equivalência em termos práticos com a implantação tradicional do algoritmo de Monte Carlo.

Tal modelo procurou explorar o potencial das arquiteturas *multi-core* com o auxílio de um ambiente extremamente propício para programação de *threads*: OpenMP. Com isso, obtém-se uma ferramenta de simulação de *Potts* Celular eficiente (para alto desempenho).

1.3 Resultados Alcançados

O trabalho realizado atingiu os resultados esperados, ou seja, foi possível apresentar dados quantitativos de resultados de simulações obtidos pelas versões paralelas dos algoritmos de Monte Carlo e *Random Walker* implementadas neste trabalho. Parte dos resultados apresentados neste texto compõe artigo publicado no ERAD 2008 (LUZ et al, 2008).

1.4 Organização do Trabalho

Este trabalho divide-se em seis capítulos, incluindo o capítulo atual de introdução.

No segundo capítulo é apresentado o modelo de Potts Celular, suas características e as desvantagens do algoritmo tradicional utilizado em conjunto com este modelo. Também é introduzido o algoritmo de *Random Walker*, descrevendo o seu funcionamento e vantagens desse algoritmo em relação ao método de Monte Carlo.

O terceiro capítulo discute a programação paralela em arquiteturas *multi-core*. São apresentadas as principais características deste tipo de arquitetura e da ferramenta de programação paralela *OpenMP*, que será utilizada como suporte a implementação do trabalho.

O quarto capítulo trata exclusivamente da implementação, comparando a versão paralela de ambos algoritmos: Monte Carlo e Random Walker. Este capítulo

também apresenta alguns índices de desempenho obtidos com a implementação paralela com OpenMP.

O quinto capítulo trata do teste de aderência obtido pela aplicação da lei de *von Neumann* e discute a qualidade dos resultados de simulação apresentados pela implementação paralela.

Por fim, no sexto capítulo, são apresentadas as conclusões dos experimentos e algumas sugestões para eventuais trabalhos futuros.

2 Simulação de Sistemas Celulares

Este capítulo apresenta o modelo de Potts Celular, descrevendo os seus objetivos, suas características de execução, desempenho e custo computacional. Também é descrito o funcionamento do algoritmo de Monte Carlo, responsável pela evolução de simulações baseadas neste modelo. Por fim, é introduzido o algoritmo de *Random Walker*, uma alternativa eficiente que visa solucionar os problemas computacionais apresentados pelo modelo de Potts.

2.1 Modelo de Potts

Em mecânica estatística, o modelo de Potts é um modelo computacional estocástico similar ao modelo de *Ising*, sendo utilizado para simular as propriedades magnéticas de materiais e grãos de um policristal (GLAZIER; WEAIRE, 1992).

O nome do modelo foi dado por Renfrey B. Potts em sua tese de doutorado em 1952 e está relacionado com vários outros modelos, incluindo o modelo XY, o modelo de Heisenberg e o modelo N-vetor. Generalizações do modelo de Potts são também utilizadas para simular o crescimento de partículas em metais e a interação entre espumas de sabão. James Glazier e François Graner desenvolveram uma generalização do modelo, conhecida como modelo de *Potts* Celular, que vem sendo utilizada para simular fenômenos estáticos e cinéticos em estruturas celulares (GLAZIER; WEAIRE, 1992).

2.2 Modelo de Potts Celular

O modelo de Potts Celular proposto por Glazier e Graner em 1992 (GRANER; GLAZIER, 1992) representa as propriedades físicas de células biológicas. Tal modelo é utilizado quando se pretende simular a reorganização celular que se dá pela minimização de energia livre de adesão celular, de acordo com a Hipótese da Adesão Diferenciada (DAH – *Differential Adhesion Hypothesis*).

Este modelo tem sido validado na simulação de sistemas em que a reorganização celular de um organismo completo se dá através da Adesão Celular Diferenciada. Também é usualmente chamado de Modelo de *Potts* Estendido com Adesão Diferenciada ou modelo de Glazier e Graner (GRANER; GLAZIER, 1992).

Neste modelo, uma estrutura celular, tal como um agregado de células, é representado em um espaço discreto. Neste caso, uma célula é constituída por um grupo de rótulos espacialmente conexos e de mesmo valor numérico. A Fig. 2.1 ilustra o esquema tridimensional de uma rede cúbica de 10 elementos para cada eixo, totalizando 1000 rótulos (10^3).

Neste segmento da rede existem sete células definidas pelo conjunto conexo de elementos da rede com mesmo rótulo θ, com $1 \leq \theta \leq 7$. As células $\theta = 1$, $\theta = 2$ e $\theta = 6$ são de um tipo celular r e as demais de outro. Se for considerado que a célula $\theta = 5$ possui a mesma forma em todos os 10 segmentos do eixo Z, então esta célula possui volume $V(\theta) = 160$ e superfície $S(\theta) = 200$, que é a superfície dos elementos que compõem a borda da célula e estão em contato com as demais. Este sistema pode conter condições de contorno periódicas, não representadas (CERCATO, 2005).

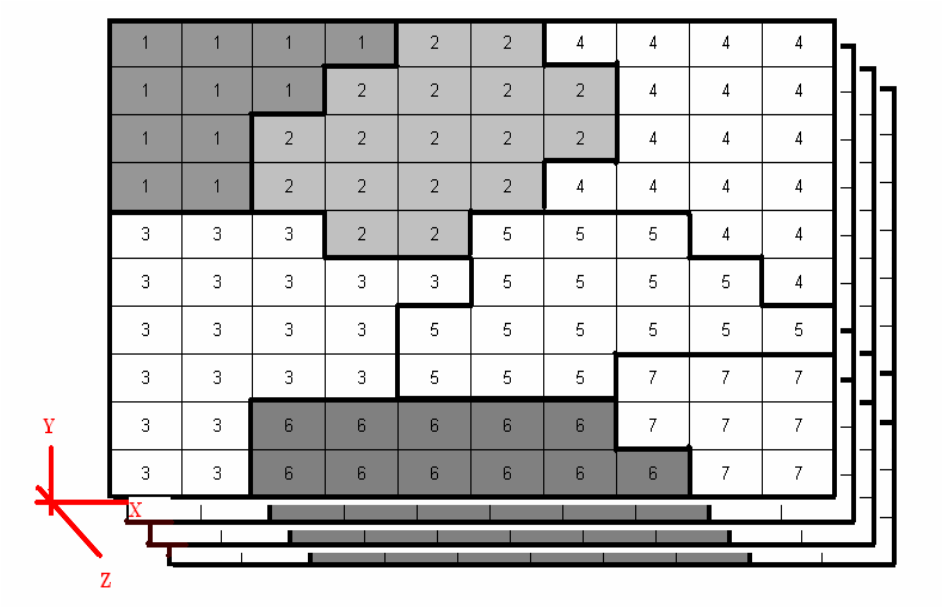

Figura 2.1 – Rede cúbica 10 x 10 x 10. Nesta imagem ilustrativa, o eixo Z, perpendicular à página, exibe apenas 4 dos 10 planos.

Fonte: CERCATO, 2005, p. 20.

2.3 Adesão Diferenciada

Processos biológicos que envolvem reprodução celular implicam na reorganização espacial das células, seja em tecidos celulares vivos ou em agregados celulares *in vitro*. Células movem-se intensivamente no desenvolvimento embrionário. Em adultos, elas migram para o local de uma infecção ou ferimento, reorganizando-se na cicatrização de feridas e na reposição de células de tecidos epiteliais em função da descamação da epiderme e mucosas. Da mesma forma, em processos cancerosos, caracterizados pela excessiva reprodução celular, as células tumorais e as células normais do tecido circunvizinho necessitam se reorganizar espacialmente à medida que o tumor cresce (KNEWITZ, 2002).

Dentre os mecanismos que orientam a dinâmica da organização celular, destaca-se o da adesão diferenciada. O mecanismo de adesão diferenciada depende das moléculas de adesão presentes nas membranas das células. Sabe-se que estas moléculas, possivelmente em função das mutações genéticas sofridas pelas células cancerosas, têm sua expressão afetada, ou seja, são produzidas em quantidade ou estrutura diferente da que é produzida por uma célula normal. Isto pode afetar a dinâmica da reorganização celular que se dá através da variação das forças de adesão entre células de tipos diferentes (KNEWITZ, 2002).

2.3.1 Hipótese de Adesão Diferenciada

Baseado em observações de que existe similaridade entre o comportamento de tecidos celulares e o de líquidos imiscíveis (por exemplo, água e óleo), Steinberg propôs, em 1963, a Hipótese de Adesão Diferenciada (*Differential Adhesion Hypothesis* – DAH) (GRANER, 1993).

Trata-se de um modelo termodinâmico segundo o qual a interação entre as células envolve uma energia de adesão superficial que depende do atrito entre as moléculas pertencentes às membranas celulares. A minimização desta energia ou tensão superficial, analogamente ao que ocorre com líquidos, guia a evolução do sistema.

Tensões superficiais representam diferenças de energia por unidade de área numa interface e determinam a configuração com o mínimo de energia global. Em agregados celulares, devido às Moléculas de Adesão Celular (*Cell Adhesion Molecules* – CAM's) (GRANER, 1993), diferentes energias estão associadas às interfaces entre células de tipos iguais e diferentes.

Macroscopicamente, estas energias se manifestam como tensões superficiais entre agregados de células e estas determinam se tipos celulares diferentes se misturam ou se segregam. Tensões superficiais negativas levam à mistura de células de tipos diferentes e, de modo contrário, tensões superficiais positivas conduzem à segregação.

Graner (1993) definiu matematicamente as tensões superficiais entre agregados celulares. As energias de adesão superficial por unidade de área de contato, associadas com essas interfaces, tomam os valores e_{dd}, e_{dl}, e_{ll}, e_{dM} e e_{lM}, onde (l) refere-se a células claras, (d) a escuras e (M) ao meio extracelular. As tensões superficiais são definidas por (GRANER, 1993):

$$\gamma_{dl} = e_{dl} - (e_{dd} + e_{ll}) / 2$$

$$\gamma_{dM} = e_{dM} - e_{dd} / 2$$

$$\gamma_{lM} = e_{lM} - e_{ll} / 2$$

Tais tensões são calculadas a partir da definição das energias de adesão superficial usando as três equações acima. A energia total de configuração do sistema é definida por (GRANER; GLAZIER, 1992):

$$E = \Sigma_{i,j,k} \Sigma_{i',j',k'} \epsilon_{r(\theta), r(\theta')} (1 - \delta_{\theta,\theta'}) + \lambda \Sigma_{\theta} [V (r (\theta)) - v (\theta)]^2$$

onde:

- $r(\theta)$ é o tipo associado com a célula θ;
- $\epsilon_{r(\theta),r(\theta')}$ é a energia de adesão superficial entre elementos do tipo θ e θ';

- λ é um parâmetro associado à compressibilidade celular; e,
- $V(\Gamma(\sigma))$ é o "volume alvo" das células de tipo Γ.

O volume da célula σ é $v(\sigma)$. O primeiro somatório é executado sobre todos os rótulos da rede e o segundo sobre os vizinhos mais próximos de cada rótulo. O trecho a seguir, $(1 - \delta_{\sigma,\sigma'})$, identifica interfaces entre células distintas ($\delta = 1$ se $\sigma = \sigma'$ e $\delta = 0$, caso contrário), fazendo com que sejam contabilizadas apenas as energias de adesão superficial de rótulos pertencentes às bordas (superfície) das células. O terceiro somatório é um termo de energia elástica que serve para regular o volume da célula e sua compressibilidade.

2.4 Método de Monte Carlo

O método de Monte Carlo faz parte da maior e mais importante classe de métodos numéricos para solução de problemas de simulação numérica. Sua aplicação tem a finalidade de determinar propriedades probabilísticas de uma população a partir de uma nova amostragem aleatória dos componentes dessa mesma população (NEWMAN; BARKEMA, 1999). Isto permite apresentar soluções aproximadas para uma grande variedade de problemas físicos e matemáticos por meio de experimentos numéricos que utilizam números aleatórios ou pseudo-aleatórios. Esta técnica pode ser aplicada tanto a problemas de natureza probabilística quanto a problemas determinísticos.

De acordo com (HAMMERSLEY, 1964), o nome "Monte Carlo" surgiu durante o projeto *Manhattan* na Segunda Guerra Mundial. No projeto de construção da bomba atômica, Ulam, von Neumann e Fermi consideraram a possibilidade de utilizar o método, que envolvia a simulação direta de problemas de natureza probabilística relacionados com o coeficiente de difusão do nêutron em certos materiais. Apesar de ter despertado a atenção desses cientistas em 1948, a lógica do método já era conhecida há bastante tempo. Por exemplo, existe um registro de um artigo escrito por Lord Kelvin dezenas de anos antes que já utilizava técnicas de Monte Carlo em uma discussão das equações de *Boltzmann*.

O algoritmo de *Metropolis*, introduzido por Nicolas Metropolis em 1953, é o mais utilizado nas simulações aplicando o método de Monte Carlo. Pode-se considerar o algoritmo de Metropolis um caso especial de amostragem de importância que gera estados com a probabilidade de *Boltzmann*, responsável pela dinâmica da simulação, determinando as taxas de transição entre os níveis de energia em sistemas naturais (KNEWITZ, 2002).

2.4.1 Descrição do Algoritmo de Monte Carlo

Para descrever o algoritmo tradicional de Monte Carlo, é considerado primeiramente, nesta seção, um sistema celular bi-dimensional, tendo como apoio a Fig. 2.2. Depois, será considerado um sistema celular tri-dimensional, representado na Fig. 2.3. Estes sistemas celulares são descritos por uma rede uniforme de rótulos, cada rótulo valorado individualmente.

A evolução do algoritmo de Monte Carlo sobre este sistema celular ocorre da seguinte forma: um rótulo qualquer é sorteado (rótulo escuro no centro da Fig. 2.2). Após, é sorteado um rótulo vizinho candidato à segregação celular dentre os oito vizinhos do rótulo inicialmente sorteado (oito vizinhos em uma rede bi-dimensional, rótulos de cor cinza-escuro na Fig. 2.2). É feita então uma verificação em torno desses oito vizinhos mais próximos e dos outros 12 vizinhos adjacentes (rótulos de cor cinza-claro). Para cada rótulo diferente do escolhido, a energia inicial, que é o valor de energia atual do sistema, é incrementado de um (1).

Este processo é repetido para o rótulo vizinho sorteado como candidato à segregação celular. Desta forma, é computada a energia do sistema caso a segregação celular ocorra.

No passo final, a energia do sistema considerando a ocorrência da segregação celular é subtraída da energia inicial, constituindo a diferença de energia do sistema entre seu estado inicial e seu estado final, caso a segregação celular ocorra. Se esta diferença de energia for igual a zero (0), ou seja, se a energia não foi alterada, deve-se considerar a *probabilidade* de troca, ou seja, a probabilidade de ser aceita a ocorrência da segregação celular, definida para o referido sistema. Caso a computação da probabilidade indique que a segregação celular deva ocorrer, a

troca consiste em substituir o rótulo sorteado com o rótulo vizinho escolhido. Se a diferença de energia for menor do que zero, a probabilidade de troca é de 100% (CERCATO, 2005).

1	1	1	2	2	2	2
1	1	1	2	2	2	2
1	1	1	2	2	2	2
1	1	1	2	2	2	2
1	3	3	3	3	3	3
1	3	3	3	3	3	4
1	3	3	3	3	4	4

Figura 2.2 – Matriz contendo o rótulo escolhido ao centro (em destaque), e seus 20 vizinhos mais próximos em uma representação 2D.

Na evolução de uma simulação pelo método de Monte Carlo, assume-se que um passo de evolução do sistema é equivalente à realização de tantos sorteios quanto forem os elementos sorteáveis em um sistema. Desta forma, a complexidade de cálculo está intimamente ligada ao tamanho do sistema simulado.

Caso o sistema simulado seja representado em uma estrutura tri-dimensional, empregando o mesmo mecanismo de rótulos, as atualizações dos rótulos da matriz envolvem a interação entre o rótulo selecionado e seus 26 vizinhos mais próximos (Fig. 2.3), considerando o volume envolvente.

Estes rótulos vizinhos são os oito rótulos do mesmo plano ao redor do rótulo escolhido, os nove rótulos pertencentes ao segmento de plano imediatamente acima do rótulo escolhido e os nove rótulos pertencentes ao segmento imediatamente abaixo do rótulo escolhido (CERCATO, 2005).

O rótulo vizinho escolhido para simular a troca é sorteado dentre os 26 outros vizinhos e o cálculo de energia segue o mesmo funcionamento descrito anteriormente para uma matriz bi-dimensional.

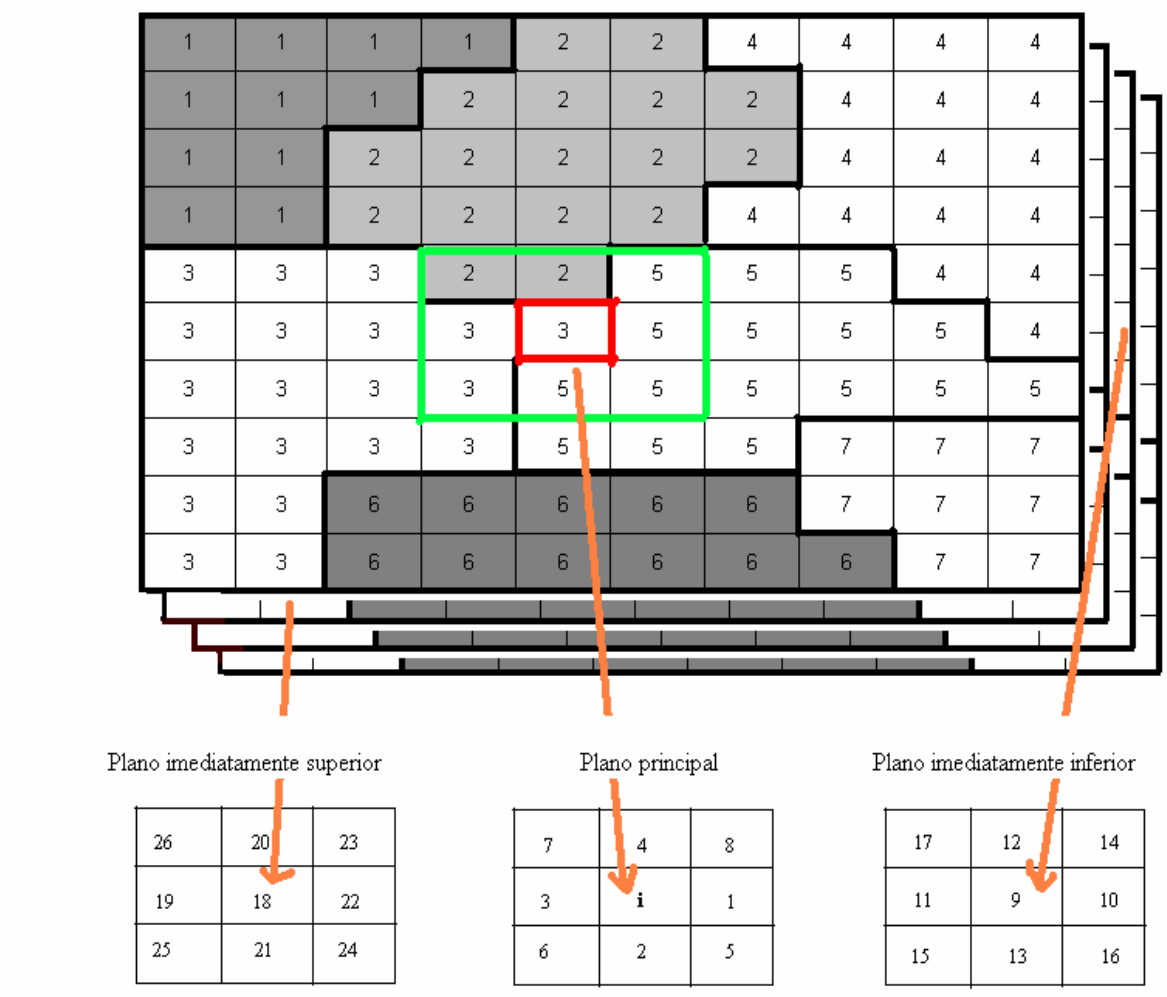

Figura 2.3 – Os 26 vizinhos mais próximos de um rótulo em uma representação 3D.

Fonte: CERCATO, 2005, p. 23.

2.4.2 Algoritmo de Metropolis

O algoritmo de *Metropolis*, também conhecido por algoritmo de *Metropolis-Hastings* foi desenvolvido por Nicolas Metropolis e W. K. Hastings em 1953. Este algoritmo permite determinar valores esperados de propriedades do sistema simulado pela obtenção de uma média sobre uma amostra. O algoritmo foi

concebido de modo a se obter uma amostra que siga a distribuição de *Boltzmann*, sendo o mais utilizado em simulações Monte Carlo (KNEWITZ, 2002).

Para se determinar a probabilidade da ocorrência de uma determinada configuração de um sistema simulado, seria necessário todo o espaço de configurações atingíveis pelo sistema, assim como o conhecimento da probabilidade de ocorrência destas configurações. No caso de sistemas que envolvam variáveis contínuas, seria necessário ainda uma integração da densidade de probabilidade sobre todo o espaço de configurações. Esse procedimento é particularmente custoso quando o número de variáveis apresentado pelo sistema atinge a ordem de centenas (KNEWITZ, 2002).

A eficiência do algoritmo de *Metropolis* está diretamente ligada ao fato de não levar em consideração a probabilidade das configurações em si, mas sim a razão entre elas. A premissa é de que a razão entre as probabilidades de duas dadas configurações pode ser determinada independentemente das outras. Dadas duas configurações *m* e *n* quaisquer, a razão entre a probabilidade da configuração *m*, dada por P_m, e a probabilidade da configuração *n*, dada por P_n, pode ser escrita como:

$$P_m / P_n = (exp (- (U_m / K_b T))) / (exp (- (U_m / K_b T)))$$

$$= exp (- (U_m - U_n) / (K_b T))$$

A partir dessa igualdade, o algoritmo de Metropolis pode ser implementado através do seguinte conjunto de regras:

1) Geração de uma configuração inicial aleatória, ou seja, com valores aleatórios para todos os graus de liberdade do sistema, respeitando as suas restrições. Atribui-se o índice *m* a essa configuração, que é aceita para a amostra;

2) Geração de uma nova configuração-tentativa de índice *n*, resultado de pequenas alterações nas coordenadas da configuração *m*;

3) Se a energia da configuração *n* for menor que a da configuração *m*, inclui-se a configuração *n* na amostra, e atribui-se a ela o índice *m* a partir desse momento. Caso contrário, realizam-se os passos descritos nos itens 4 e 5;

4) Gera-se um número aleatório entre 0 e 1;

5) Se esse número aleatório for menor que P_n/P_m, aceita-se na amostra a configuração *n*, e atribui-se a ela o índice *m*. Caso contrário, o índice *m* permanece designando a configuração original;

6) Repete-se os passos 2 e 3 até que algum critério de parada seja satisfeito. Cada uma dessas repetições é denominada Passo Monte Carlo (PMC).

2.4.3 Desvantagens do Algoritmo de Monte Carlo

O algoritmo que implementa o modelo de *Potts* Celular, em sua forma original, é muito ineficiente computacionalmente, porque o uso do processo Monte Carlo para selecionar aleatoriamente os rótulos dos elementos pertencentes à rede pode levar a muitas seleções que não permitem trocas visando minimizar a energia do sistema, no caso de rótulos internos das células. O número de seleções aleatórias, *n*, do algoritmo é proporcional à dimensionalidade (*d*) do problema investigado $O(n^d)$, determinando que a complexidade de tempo deste problema seja polinomial (CERCATO, 2005).

Ressalta-se, assim, que a maior ineficiência da evolução do processo de *Potts* Celular pelo método convencional de simulação por Monte Carlo se dá na tentativa de efetuar uma troca de rótulo numa região da rede onde todos os rótulos possuem o mesmo valor. Esta tentativa de troca nesta configuração é muito comum, sendo proporcional ao volume médio das células, o que gera um desperdício de tempo em trocas que nunca poderão ocorrer (CERCATO, 2005).

2.5 Random Walker

O algoritmo de *Random Walker* (Caminhante Aleatório) busca apresentar uma solução para os problemas de ineficiência computacional do modelo de *Potts* Celular. Utilizando a dinâmica de *Metropolis*, ele procura reduzir a quantidade de sorteios de rótulos que não pertencem à borda de uma determinada célula, permitindo aumentar o desempenho de uma simulação ao restringir a seleção de

rótulos candidatos à segregação celular aos rótulos pertencentes às bordas de células (CERCATO, 2005).

A aplicação do algoritmo de *Random Walker* para evolução do modelo de *Potts* Celular foi inicialmente proposta por Éder Gusatto (GUSATTO, 2004) para arredondamento de agregados celulares biológicos em 2D. A implementação realizada por Gusatto foi estendida, tornando possível a execução de simulações em 3D (GUSATTO et al, 2005). A implementação deste último trabalho também oferece a capacidade de execução concorrente com objetivo de aumentar o desempenho do modelo de *Potts*, sendo capaz de ser executado em paralelo e, ao mesmo tempo, buscando reduzir o tempo de execução.

2.5.1 Descrição do Algoritmo de Random Walker

Para aumentar a eficiência da simulação gerada pelo modelo de *Potts*, o algoritmo de *Random Walker* tem como premissa a realização do menor número possível de sorteios com a finalidade de encontrar o primeiro rótulo válido (a primeira borda) e, a partir desse ponto, a evolução do algoritmo é obtida pela técnica de percorrer apenas as bordas das células (caminho indicado pela seta na Fig. 2.4) (CERCATO, 2005).

Existe a possibilidade do percurso alcançar um rótulo que não possui vizinhos diferentes. Neste caso, surge a necessidade de realizar-se um novo sorteio para encontrar uma nova borda válida. Esta implementação elimina em grande parte o desperdício computacional com buscas aleatórias desnecessárias.

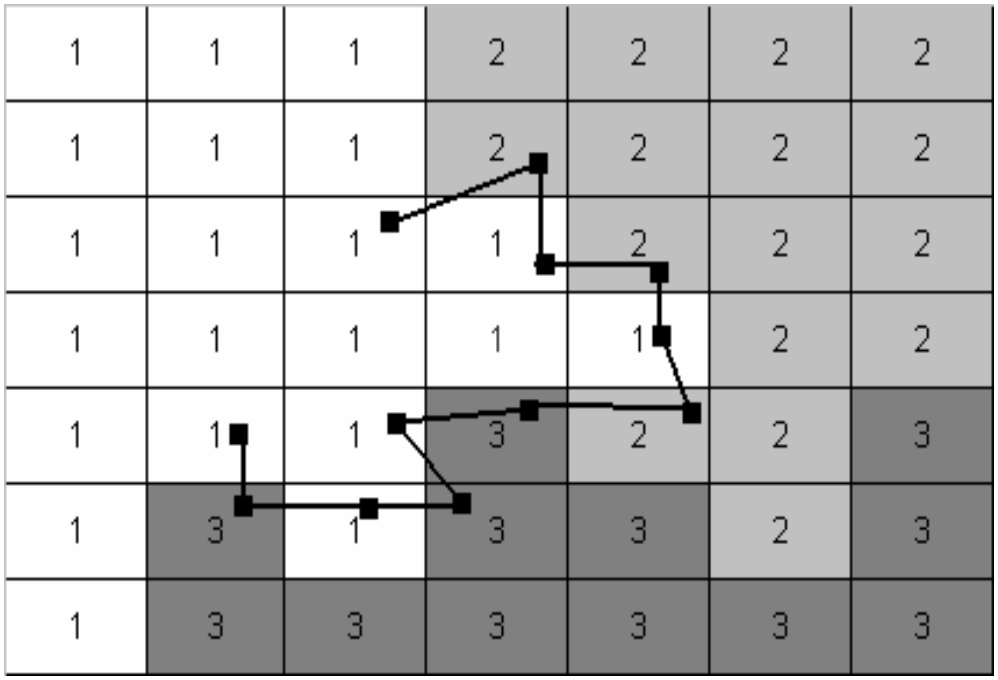

Figura 2.4 – Ilustração do caminho percorrido pela execução do algoritmo através de bordas de diferentes células.

O fato de não eliminar completamente o sorteio de bordas garante que o processo de transição de estados do algoritmo de *Metropolis* possa atingir qualquer estado do sistema a partir de um outro qualquer, se este processo for executado por tempo suficiente. Desta forma, fica garantido o controle de possíveis correlações e comportamentos tendenciosos que um *Random Walker*, preso em um determinado setor do agregado, poderia vir a gerar (CERCATO, 2005).

A Fig. 2.5 apresenta o algoritmo de *Random Walker* em forma de fluxograma. O primeiro passo deste algoritmo consiste em escolher aleatoriamente um rótulo válido para troca, ou seja, um rótulo pertencente à borda de uma célula qualquer. A seguir, é realizado um teste para verificar se existem rótulos vizinhos ao redor do rótulo escolhido que possam realizar uma transição de estado, de acordo com o algoritmo de *Metropolis*. Serão considerados válidos todos os rótulos vizinhos que possuírem valor diferente do escolhido.

Um destes rótulos vizinhos é escolhido aleatoriamente e, de acordo com a probabilidade de transição de estado, o valor do escolhido pode vir a ser substituído pelo valor do rótulo vizinho. Independentemente do resultado desta transição, o vizinho passa a ser o próximo rótulo escolhido, dando continuidade ao algoritmo.

Desta forma, não faz-se necessário realizar um novo sorteio para escolher um novo rótulo.

O tempo é definido como Passo de *Random Walker* (PRW), ou ainda como Passo de Caminhante Aleatório (PCA). Isto significa afirmar que os caminhantes aleatórios são executados uma quantidade de vezes igual à quantidade de rótulos pertencentes às bordas de todas as células (CERCATO, 2005).

Como o número de rótulos em regiões de borda (regiões limítrofes entre uma célula e outra) deve ser menor que o número de rótulos no interior de células, a complexidade de processamento pela técnica de *Random Walker* é inferior a complexidade apresentada pelo método tradicional de evolução por Monte Carlo.

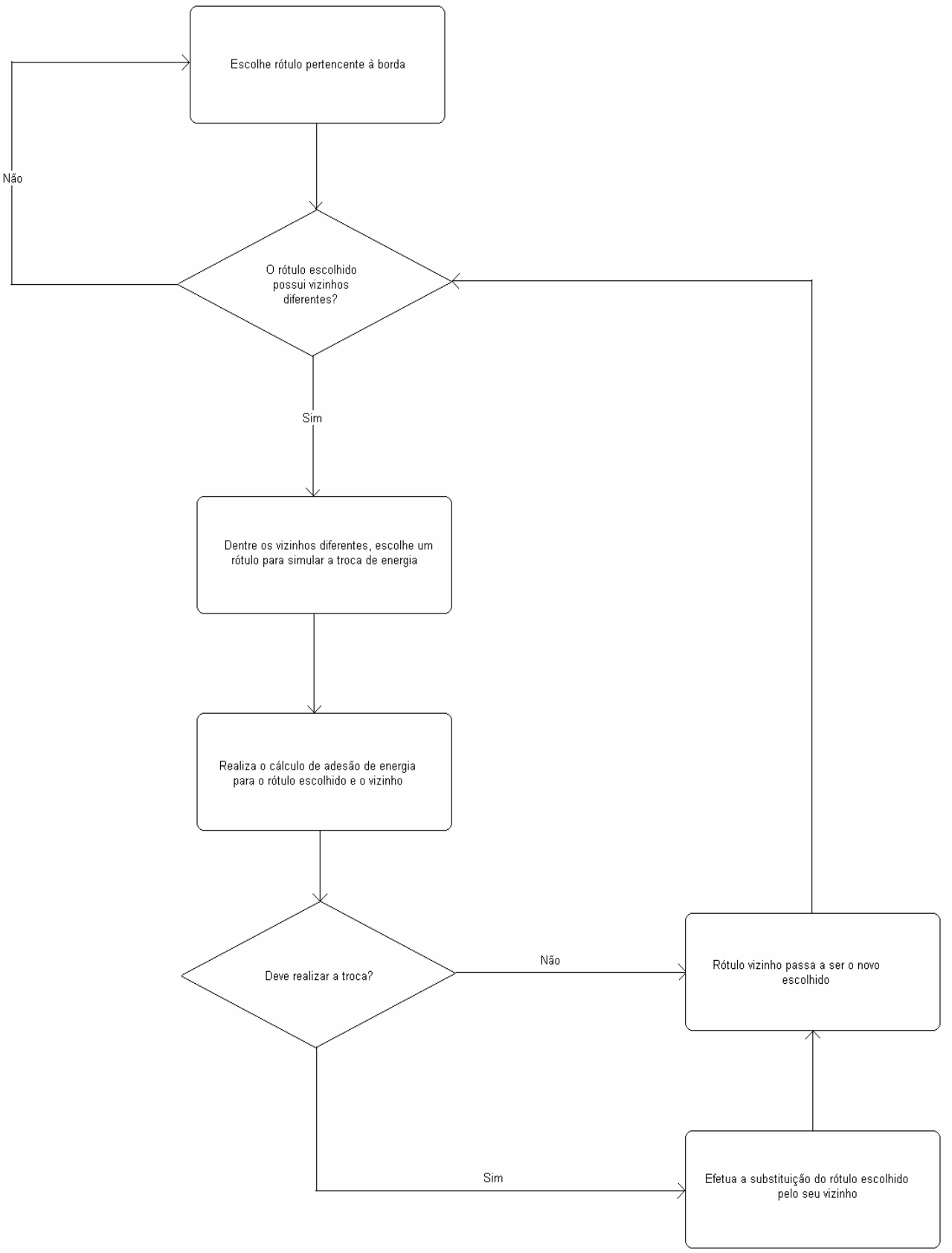

Figura 2.5 – Fluxograma do algoritmo de *Random Walker*.

Fonte: CERCATO, 2005, p. 56.

2.5.2 Implementação Seqüencial do Algoritmo de Random Walker

De acordo com os resultados mostrados no trabalho de Cercato (2005), a implementação seqüencial do algoritmo de Random Walker é cerca de seis (6) vezes mais rápida do que a do algoritmo de Monte Carlo. Esta redução no tempo de processamento reflete a característica deste algoritmo percorrer somente as bordas das células, reduzindo ao mínimo o número de sorteios necessários para encontrar posições de borda válidos para execução de caminhadas. Outro dado retirado do mesmo trabalho indica também que a execução paralela do algoritmo de Random Walker oferece ganho de desempenho frente sua implementação seqüencial.

2.6 Análise de Desempenho das Implementações Seqüenciais

Os algoritmos de evolução de simulações de agregados celulares pelo modelo de *Potts* Celular apresentados neste capítulo foram implementados em versões seqüenciais. A aplicação foco objetiva simular a agregação celular representada por um sistema de bolhas. Nesta aplicação, o sistema inicial é composto por um conjunto constituído por n bolhas e, durante a evolução do sistema, opera o processo de agregação celular, reduzindo o número de bolhas deste mesmo sistema.

Esta seção documenta os resultados de desempenho (tempo de execução, medido em segundos) obtidos pela execução das versões seqüenciais de ambos algoritmos. Estes resultados encontram-se documentados na tab. 2.1, que apresenta os resultados relativos a simulações realizadas em matrizes bi-dimensionais, possuindo três tamanhos diferentes (500 x 500, 1000 x 1000, 5000 x 5000).

O hardware utilizado foi um Intel Pentium Core 2 Duo, com freqüência de clock de 2 GHz e memória RAM de 2 GB. O sistema operacional utilizado foi a versão 7.10 do Ubuntu (baseado em Linux), versão do kernel 2.6.22-14.

Tabela 2.1 – Tempos, em segundo, de execução seqüencial em 2D da evolução de Potts Celular.

	Tamanho da entrada		
Estratégia	500 x 500	1000 x 1000	5000 x 5000
Monte Carlo	5,63	26,39	1221,83
Random Walker	3,56	17,67	683,40

Como pode-se observar nos resultados mostrados na Tabela 2.1 (implementação em matrizes 2D), o algoritmo de Random Walker mostrou-se mais eficiente em termos de desempenho do que o algoritmo de Monte Carlo. Nas três matrizes em que os algoritmos foram implantados, a estratégia de Random Walker é cerca de 2 vezes mais rápida do que a estratégia de Monte Carlo.

2.7 Resumo

Neste capítulo foi apresentado o funcionamento do modelo de *Potts* Celular. Este modelo permite realizar a simulação de diversos tipos de fenômenos celulares, tal como a agregação celular, que ocorre em processos cancerígenos. A implementação deste modelo pelo método de Monte Carlo apresenta alto custo computacional, uma vez que o mesmo desperdiça potencial de processamento ao realizar um grande número de sorteios em posições que não estão sujeitas a trocas de rótulos.

O modelo de *Potts* trabalha com a dinâmica de *Metropolis* em redes de dados de grandes proporções que, de alguma forma, podem ser executadas em mais de um processador simultaneamente. Porém, devem ser levados em consideração os custos necessários para sincronizar os dados utilizados nos cálculos para efetuar a troca de rótulos.

O algoritmo de *Random Walker*, por sua vez, tem por finalidade reduzir o tempo de execução de simulações melhorando o desempenho. Este algoritmo procura manter as características originais das simulações do modelo de *Potts*. O funcionamento se dá selecionando rótulos pertencentes apenas às bordas das

células. Desta forma, consegue-se evoluir um sistema de maneira similar ao algoritmo padrão em uma parcela de tempo de execução reduzida.

A análise de desempenho apresentada documenta os tempos de processamento obtidos para implementações seqüenciais de simulações pelo método de Monte Carlo e *Random Walker*. Os resultados obtidos confirmam que a técnica de *Random Walker* é mais eficiente.

3 Multithreading com OpenMP em Arquiteturas SMP

Este capítulo apresenta uma breve introdução às arquiteturas SMP e processadores *multi-core*, bem como à multiprogramação leve. Este capítulo também apresenta, em termos gerais, POSIX Threads (Pthreads) e OpenMP, duas ferramentas populares utilizadas para explorar o potencial deste tipo de arquitetura.

3.1 Arquiteturas Multiprocessadas

A idéia de usar múltiplos processadores com a finalidade de melhorar o desempenho e/ou a disponibilidade (redundância, tolerância a falhas) não é recente. Flynn (FLYNN, 1972) propôs uma taxonomia simples para categorização das arquiteturas baseando-se no fluxo de instruções e dados observado. Esta taxonomia, ainda atualmente empregada, é composta por quatro categorias:

- SISD – *Single Instruction Stream*, *Single Data Stream*: fluxo único de instruções operando sobre um fluxo único de dados. Nesta categoria estão os uniprocessadores, tais como as máquinas de *von Neumann* tradicionais (microcomputadores pessoais e estações de trabalho).

- SIMD – *Single Instruction Stream*, *Multiple Data Stream*: uma mesma instrução pode operar sobre um conjunto de dados. Nesta categoria estão os computadores que exploram o paralelismo de dados. Exemplos podem ser encontrados em co-processadores multimídia e processadores vetoriais ou matriciais.

- MISD – *Multiple Instruction Stream*, *Single Data Stream*: múltiplos fluxos de instrução operando sobre um único fluxo de dados. Não existem exemplares nesta categoria no mercado. No entanto, alguns autores classificam arquiteturas *pipelined* ou *Dataflow* como MISD.

- MIMD – *Multiple Instruction Stream*, *Multiple Data Stream*: cada processador executa um fluxo de instruções independente que opera

sobre um conjunto de dados particular. Computadores MIMD exploram o paralelismo de *threads* (TLP – *Thread Level Parallelism*).

As arquiteturas MIMD possuem diferentes organizações, as quais podem ser classificadas em três categorias:

- SMP – *Symmetric Multiprocessing*: arquitetura baseada em múltiplos processadores que compartilham uma mesma memória. Também chamada de UMA (*Uniform Memory Access*) ou Multiprocessador ou ainda MIMD Fortemente Acoplada. Nesta categoria, a memória compartilhada é centralizada.

- NUMA – *Non-uniform Memory Access*: arquitetura baseada na existência de memórias dedicadas para cada processador ou grupo de processadores, havendo um mecanismo de interconexão que garante acesso a todos os processadores e todo o espaço de endereçamento.

- Cluster: arquitetura baseada em múltiplos computadores, também chamada de multicomputador ou MIMD Fracamente Acoplada, que visa trocar informações (dados) por meio de troca de mensagens através de uma rede de interconexão (MP – *Message Passing*).

Em relação às arquiteturas paralelas, faz-se necessário destacar que atualmente o paralelismo é explorado em diferentes níveis, em praticamente todas as arquiteturas. Assim, o paralelismo é explorado desde sua granulosidade mais fina, no nível de instrução em uniprocessadores, até a exploração do paralelismo no nível de processos em máquinas MIMD comunicando-se via troca de mensagens.

Também é importante salientar que as formas de exploração de paralelismo podem ser combinadas. Por exemplo, um *cluster* pode ser configurado com base em processadores *multi-core* superescalares. Neste caso, o paralelismo de *threads*, dados e instruções é explorado em diferentes níveis do sistema (CAVALHEIRO; SANTOS, 2007).

3.2 Processadores Multi-Core

Processadores *multi-core* refletem os mais recentes avanços na arquitetura de processadores. São capazes de prover maior capacidade de processamento com um custo/benefício melhor do que processadores *single-core* (um único processador). Cita-se como vantagem dos processadores *multi-core* a possibilidade de aumentar o potencial de desempenho com menor impacto no consumo de energia e geração de calor incorrida pelo mesmo ganho de desempenho em otimizações realizadas com a tecnologia de processadores *single-core*.

Um processador é denominado *multi-core* quando possui dois ou mais núcleos (processadores) idênticos instalados fisicamente no mesmo *chip* (KUMAR; TULLSEN, 2007). Cada *core* (núcleo) corresponde a uma unidade de processamento completa. A arquitetura básica desse tipo de processador segue o modelo SMP (*Symmetric Multi-Processors* – Múltiplos Processadores Simétricos), cujas características básicas são: execução independente de fluxos de instrução distintos para cada processador ativo e compartilhamento de memória (CAVALHEIRO; SANTOS, 2007).

Algumas vantagens de processadores *multi-core* são:

- Maior eficácia do sistema e desempenho aprimorado de programas em computadores executando vários aplicativos simultaneamente;
- Desempenho superior em aplicativos que utilizam processamento de forma intensiva;
- Redução da dissipação térmica quando comparado ao *single-core*;
- Melhora o paralelismo a nível de *threads*;
- Auxilia aplicações que não conseguem se beneficiar de processadores superescalares atuais por não possuírem um bom paralelismo a nível de instruções;
- Melhor localidade de dados;
- Melhor comunicação entre as unidades;

- Economia de espaço e de energia.

Com a popularização dos processadores *multi-core*, a demanda por programação concorrente atinge novos patamares, uma vez que até mesmo usuários domésticos possuem arquiteturas paralelas à sua disposição. Como conseqüência natural, o desenvolvimento de novos aplicativos, desde os programas multimídia até elaboradas aplicações de Computação Científica, devem incorporar técnicas de multiprogramação leve (CAVALHEIRO; SANTOS, 2007).

3.3 Multiprogramação Leve

A multiprogramação leve oferece recursos de programação com os quais uma aplicação pode ser descrita em termos de um programa composto de diversos fluxos de execução (usualmente chamados *threads*), capazes de executar de forma concorrente. Desta forma, um programa *multithreaded* possui diversos fluxos de execução que podem estar ativos em um determinado instante de tempo e, a exemplo de programas imperativos seqüenciais, cada fluxo possui uma área de endereçamento própria (CAVALHEIRO; SANTOS, 2007).

A diferença é que, além desse espaço de memória privado, as *threads* compartilham acesso a um espaço de endereçamento global. A concorrência, neste caso, expressa não somente a disputa de recursos de processamento, como tempo de processador e espaço de memória, mas também a disputa das *threads* de um programa pelo acesso aos dados armazenados em memória (CAVALHEIRO; SANTOS, 2007).

A relação entre a multiprogramação leve e as arquiteturas SMP é praticamente imediata. Enquanto o modelo da arquitetura física prevê a existência de diversos processadores e uma memória comum, o modelo de programa *multithread* prevê a coexistência de diversos fluxos de execução, que podem usar um espaço de endereçamento compartilhado para comunicar dados entre si. Essa relação é explorada por diversas ferramentas de programação, as quais oferecem serviços para controlar o número de *threads* ativas simultaneamente e para sincronizar as ações destas no acesso aos dados compartilhados. Uma prática bastante comum, embora não única, é explorar essa relação de complementaridade

entre a multiprogramação leve e arquiteturas multiprocessadas, em prol de obter programas eficientes em termos de desempenho (CAVALHEIRO; SANTOS, 2007).

Historicamente, vendedores de *hardware* implementaram suas próprias versões proprietárias de *threads*. Essas implementações eram muito diferentes umas das outras, fazendo com que fosse difícil criar um programa com *threads* portável. Para utilizar todas as vantagens das capacidades das *threads*, era necessária uma interface padronizada. Dentre as ferramentas mais populares para programação *multithread*, cita-se Pthreads e OpenMP. Ambas ferramentas são padronizadas e largamente difundidas entre a comunidade de processamento paralelo. As características principais destas ferramentas são sumarizadas no restante deste capítulo.

3.3.1 Paralelismo de Tarefas: Pthreads

Pthreads é um padrão para bibliotecas padrão POSIX threads (IEEE POSIX 1003.1c, de 1995). Embora disponíveis na maioria dos sistemas operacionais, como no Windows (BUTENHOF, 1997), elas são amplamente populares em sistemas operacionais do tipo UNIX, como Linux e Solaris.

Pthreads oferece mecanismos de sincronização, como semáforos, mutexes e variáveis condicionais, além de mecanismos para o gerenciamento e manutenção das diversas *threads* pertencentes a um sistema. Serão descritos na próxima seção algumas funções responsáveis por estas finalidades. Para ser breve e resumido, não serão mostrados os parâmetros das funções. O leitor interessado pode buscar informações complementares em (CAVALHEIRO, 2004).

3.3.1.1 Gerenciamento de Threads

Abaixo encontram-se citadas e descritas algumas funções relacionadas com a criação e destruição de *threads*.

- **pthread_create( )**: permite criar uma *thread*, partindo do código de uma função existente no corpo do programa;

- **pthread_exit( )**: a chamada a este serviço encerra a execução da *thread* corrente;
- **pthread_join( )**: permite sincronizar a *thread* que chama este serviço com o término de execução de uma *thread* indicada;
- **pthread_attr_init( )**: inicializa os atributos de execução de uma *thread*, manipulados pelo mecanismo de escalonamento provido pelo sistema operacional;
- **pthread_attr_destroy( )**: destrói atributos de uma *thread*;
- **pthread_kill( )**: envia um sinal a uma *thread*.

3.3.1.2 Sincronização

Pthreads oferece inúmeras funções para a sincronização entre *threads*. Estes mecanismos serão descritos a seguir:

- **Mutexes**: uma variável *mutex* (*Mutual Exclusion*) permite implementar a exclusão mútua, ou seja, o acesso a uma região crítica é realizado de forma a impedir os problemas relacionados a sincronização (*Race Conditions*).
  - **pthread_mutex_init( )**: inicializa uma variável *mutex*;
  - **pthread_mutex_destroy( )**: destrói uma variável *mutex*;
  - **pthread_mutex_lock( )**: permite a uma *thread* entrar em uma seção crítica;
  - **pthread_mutex_trylock( )**: adquire um *mutex*, caso o mesmo esteja disponível;
  - **pthread_mutex_unlock( )**: permite a uma *thread* sair de uma seção crítica.
- **Variáveis de condição**: permitem associar uma variável a um evento, que pode ser, por exemplo, um contador alcançando um determinado

valor ou um *flag* sendo ligado/desligado. Em uma aplicação *multithread*, a utilização de variáveis condicionais permite criar um mecanismo de comunicação entre os diversos *threads*, que por sua vez, permitem a execução condicional em torno de eventos (sincronização).

- **pthread_cond_init( )**: inicializa uma variável de condição;
- **pthread_cond_destroy( )**: destrói uma variável de condição;
- **pthread_cond_signal( )**: sinaliza uma condição.
- **pthread_cond_wait( )**: suspende a execução de uma *thread* e aguarda por um sinal, indicando que a condição foi satisfeita.

3.3.2 Paralelismo de Dados: OpenMP

OpenMP (CHANDRA et al, 2001) é uma interface de programação proposta em 1997 que oferece diversos recursos para a criação, desenvolvimento e execução de programas *multithread*. O suporte a esta API encontra-se implementado em C/C++ e *Fortran* sobre diferentes sistemas computacionais, incluindo plataformas *Unix* e *Microsoft Windows*. Além de ser extremamente portável, OpenMP propicia aos programadores uma interface eficiente e flexível para desenvolver aplicações paralelas diversas, desde *desktops* até supercomputadores. É composta por uma série de recursos, tais como: diretivas de compilação, chamadas a funções de biblioteca e variáveis de ambiente.

O grande apelo desta plataforma é devido ao seu alto grau de portabilidade. Para tanto, colabora o fato de diversos compiladores incluírem mecanismos de apoio a esta interface. OpenMP está disponível em diferentes plataformas: 32/64 bits, RISC, Unix, Windows, etc. Somado a este fato, a facilidade de paralelização de código também tem papel crescente na sua popularidade. Os recursos de programação disponibilizados permitem incorporar trechos paralelos a programas seqüenciais de forma incremental, sem alterar o algoritmo da aplicação.

3.3.2.1 Modelo de Programação

OpenMP baseia-se no modelo *fork/join*, que é muito utilizado para criação e sincronização de processos em alguns sistemas operacionais. Há um fluxo de execução principal (*thread* mestre) e, quando necessário, computações concorrentes são disparadas para dividir a carga total de trabalho em uma *seção paralela*. Por fim, ao término de uma seção paralela, é realizado uma operação *join*, retornando o fluxo de execução para o fluxo mestre.

Na nomenclatura OpenMP, o termo *threads* refere-se aos fluxos de execução que suportam a execução do programa. O programa, quando em execução, dispara a criação de trechos de código que podem ser executados como regiões paralelas. As *threads* de suporte à execução são reunidas em um núcleo denominado *thread-pool*. O número de *threads* neste núcleo de execução pode ser alterado dinamicamente (CHANDRA et al, 2001).

Este modelo é indicado para aplicações que devem manipular várias solicitações concorrentes, como, por exemplo, uma aplicação cliente-servidor. Neste tipo de programa, o servidor deve atender a múltiplas solicitações dos clientes. Sendo assim, cada requisição de um cliente pode ser tratada por uma *thread* independente criada pelo servidor. Neste modelo, o número de *threads* é variável, dependendo da carga que a aplicação está sofrendo no momento da execução.

Um recurso interessante de OpenMP é que a sincronização entre as *threads* quase sempre ocorre de maneira implícita. Isso faz com que sua utilização seja algo muito simples. Outro aspecto importante é que os processos de criação e inicialização de regiões paralelas, assim como a divisão de trabalho realizada sobre um arranjo ou vetor, não são visíveis ao programador. Isso garante a transparência quando da utilização pelo usuário (CHANDRA et al, 2001).

3.3.2.2 Variáveis de Ambiente

OpenMP define quatro variáveis de ambiente que controlam a execução de programas escritos com auxílio desta ferramenta. Dentre essas variáveis utilizadas estão: **OMP_NUM_THREADS**, **OMP_DYNAMIC**, **OMP_NESTED** e **OMP_SCHEDULE**.

A variável **OMP_NUM_THREADS** determina o número de *threads default* para executar trechos de código em regiões paralelas.

A variável **OMP_DYNAMIC** permite o ajuste dinâmico do número de *threads* utilizadas em regiões paralelas, de acordo com a carga do sistema. Os valores *default* são dependentes da implementação.

A variável **OMP_NESTED** permite paralelismo aninhado. Por definição, o valor padrão é FALSE (aninhamento desabilitado).

A variável **OMP_SCHEDULE** configura um valor de tipo *string* que controla o escalonamento de um laço de repetição paralelo em tempo de execução. Somente laços que possuam RUNTIME como valor de escalonamento são afetados.

3.3.2.3 Diretivas de Compilação

As diretivas de compilação OpenMP são compostas por **sentinelas**, **diretivas** e **cláusulas**. Uma diretiva é caracterizada como uma linha especial de código-fonte com significado especial apenas para determinados compiladores e se distingue pela existência de uma sentinela no começo da linha. Uma cláusula especifica informação adicional na diretiva de região paralela e são separadas por espaços em branco, se for especificada mais de uma. O formato básico de uma diretiva de compilação encontra-se ilustrada a seguir:

#pragma omp diretiva [cláusula]
corpo

Nesta sintaxe, **#pragma omp** é chamado `sentinela`, sendo dependente da linguagem de programação utilizada. A diretiva OpenMP é identificada em `diretiva`, opcionalmente acompanhada de uma ou mais `cláusulas`. O corpo para as threads é definido em `corpo`, composto por um bloco de instruções ou por um comando de iteração por bloco. Nas instruções do bloco não podem ser incluídas instruções de salto (`goto`) para qualquer posição do programa, mesmo dentro da própria região paralela. As operações *fork* e *join* encontram-se implícitas nesta estrutura: o *fork* é especificado na própria linha onde o pragma se encontra e o

join no final do bloco que especifica o corpo das atividades concorrentes (CAVALHEIRO; SANTOS, 2007).

A diretiva **parallel** é utilizada com o propósito de expressar paralelismo, ocasionando a criação de *threads*, que deverão ser executadas pelas *threads* de serviço do ambiente de execução. Um exemplo do uso dessa diretiva é descrito a seguir, na Fig. 3.1:

```
#pragma omp parallel
{
    printf ( "Ola Mundo!" );
}
```

Figura 3.1 – Exemplo "Olá Mundo" paralelo.

No exemplo acima, o trecho de código dentro da região paralela (a frase "Ola Mundo!") será executado mais de uma vez, dependendo do número de *threads* de serviço expressas pelo programador para executar o programa.

Como o número de *threads* de serviço corresponde ao valor associado à variável de ambiente OMP_NUM_THREADS, o número de *threads* em execução concorrente será dependente deste número. Assim, se esta variável possuir o valor quatro (4), quatro *threads* de serviço executarão, de forma paralela, quatro *threads* do programa de aplicação (CAVALHEIRO; SANTOS, 2007).

Outra diretiva importante a ser citada é **parallel for**. Nela, o intervalo de iteração do *loop* (laço de repetição) é dividido entre as *threads* de forma automática. Por exemplo, dado o seguinte código, especificado na Fig. 3.2:

```
#pragma omp parallel for
for ( int i = 0; i < 400; ++i )
{
    vet [ i ] = i;
}
```

Figura 3.2 – Exemplo de *loop* paralelo.

O corpo do *loop* é executado uma vez para cada valor de `i` entre zero (0) e 399. Porém, isso não seria feito de forma seqüencial: as quatrocentas iterações são distribuídas entre as *threads*. Uma divisão possível seria que a *thread* 0 fosse responsável pelos índices 0 a 99, a *thread* 1 pelos índices 100 a 199 e assim por diante. Entretanto, tal divisão não é fixa, podendo ser alterada. Por exemplo, alterando-se o número de *threads* em tempo de execução, a divisão de índices adequa-se automaticamente.

Faz-se necessário observar algumas regras para construção do *loop* a ser paralelizado. A variável de iteração necessita ser do tipo inteiro com sinal. Da mesma forma, devem ser valores inteiros com sinal o valor de teste da condição de término do *loop* e o valor a ser adicionado ou subtraído da variável de iteração a cada repetição realizada. Estes dois valores podem variar durante toda execução do *loop*. O teste de continuação pode empregar apenas os seguintes operadores relacionais: <, <=, > e >=, incrementando ou decrementando, conforme o caso, a variável de iteração a cada repetição. Outro aspecto importante é que o corpo do *loop* possui apenas uma entrada e uma saída, ou seja, o comando *break* não pode ser utilizado (CAVALHEIRO; SANTOS, 2007).

Embora de aparência e utilização simples, na prática a paralelização de *loops* não é uma tarefa trivial. Um conjunto de regras deve ser seguido para maior eficiência no uso deste recurso. O programador deve evitar situações nas quais a passagem por uma iteração qualquer dependa da execução de uma passagem anterior, como no caso onde a computação da passagem *i* depende dos resultados da computação da passagem *i – 1* (CAVALHEIRO; SANTOS, 2007).

Outra diretiva interessante é **sections**. Ela define blocos de programa independentes que podem ser distribuídos entre as *threads*. Cada *section* é executada apenas por uma *thread*. Trata-se de uma maneira simples de paralelizar tarefas que não tem (ou tem pouca) dependência de dados entre si. Um exemplo é apresentado na Fig. 3.3:

```
#pragma omp sections
{
    #pragma omp section
    {
        incrementa ( x );
    }
    #pragma omp section
    {
        multiplica ( x );
    }
}
armazena ( x, y );
```

Figura 3.3 – Exemplo de utilização da diretiva *sections*.

Nesse trecho de código, as funções `incrementa( )` e `multiplica( )` são executadas em paralelo, enquanto a função `armazena( )` é chamada após todas as seções paralelas terem sido finalizadas. As três funções compartilham a mesma variável x, utilizada como argumento.

Nem tudo pode ser tratado de maneira transparente em OpenMP. Em algumas situações, faz-se necessário o uso de certos mecanismos que permitam garantir uma determinada consistência entre os dados manipulados pelos diversos fluxos de execução.

Deve-se evitar que ocorram *race conditions* (condições de corrida), ou seja, quando o resultado de uma computação varia de acordo com as velocidades relativas dos processos. É necessário prevenir condições de corrida para garantir que o resultado de uma computação seja independente do tempo (CARISSIMI; OLIVEIRA; TOSCANI, 2004).

Entre as principais diretivas de sincronização encontram-se: **barrier**, **critical**, **atomic**, **single**, **master** e **ordered**.

A diretiva **barrier** é a construção mais básica de todas e implica na criação de um ponto de sincronização entre todas as *threads* criadas a partir de uma diretiva *parallel* comum. Este mecanismo garante que, no escopo da execução de cada *thread*, a instrução seguinte à barreira somente será executada após todas as

threads participantes terem satisfeito a condição de sincronização, ou seja, terem atingido a barreira. A seguir, um exemplo do uso desta diretiva:

```
#pragma omp parallel private ( minhaIdent, meuVizinho )
{
    minhaIdent = omp_get_thread_num ( );
    meuVizinho = minhaIdent – 1;
    if ( minhaIdent == 0 )
        meuVizinho = omp_get_num_threads ( ) – 1;
    ...
    a [ minhaIdent ] = a [ minhaIdent ] * 3.5;
    #pragma omp barrier
    {
        b [ minhaIdent ] = a [ meuVizinho ] + c;
        ...
    }
}
```

Figura 3.4 – Exemplo de utilização da diretiva *barrier*.

No exemplo anterior, a barreira expressa pela diretiva **barrier** força a sincronização do vetor `a[]`.

A diretiva **critical** é mais elaborada que a anterior. Seu uso permite controlar a execução de blocos de comandos em um regime de exclusão mútua. Considerando o código apresentado na seqüência, que encontra-se inserido dentro de uma região paralela, o mesmo tem por objetivo manter a variável compartilhada `maior` atualizada com o maior valor encontrado em um vetor.

O trecho de código descrito na Fig. 3.5 exemplifica o funcionamento da diretiva *critical*:

```
int vet [ 100 ], i, maior, Lmaior;
...
// Inicializar vetor
maior = vet [ 0 ];
#pragma omp parallel for private ( Lmaior )
for ( i = 0; i < 100; ++i )
{
      #pragma omp critical
      Lmaior = maior;
      if ( Lmaior < vet [ i ] )
            Lmaior = vet [ i ];
}
#pragma omp critical
if ( Lmaior > maior )
      maior = Lmaior;
printf ( "Maior = %d \n", maior );
```

Figura 3.5 – Exemplo de utilização da diretiva *critical*.

A barreira implementada pela diretiva **critical** tem abrangência global, ou seja, é aplicada a todas as *threads* executando coletivamente. Para evitar contenção desnecessária, é possível nomear seções críticas, identificando claramente o recurso a ser utilizado em exclusão mútua. Assim, no exemplo mostrado acima, poderia ser informado `critical (MAXVAL)` em ambos os pragmas, identificando, na lógica do programa, que o recurso compatilhado é a variável global `maior`.

A diretiva **atomic** é semelhante à diretiva anterior. A diferença é que esta tenta utilizar recursos do *hardware* para aumentar a eficiência.

```
#pragma omp parallel for
for ( int i = 0; i < 1000; ++i )
{
      int resultado_parcial = funcao ( x [ i ], y [ i ] );
      #pragma omp atomic
      {
            soma += resultado_parcial;
      }
}
```

Figura 3.6 – Exemplo de utilização da diretiva *atomic*.

A diretiva **single** permite definir um trecho de código que deve ser executado apenas por uma das *threads*. O código a seguir exemplifica o uso desta diretiva considerando duas situações: na primeira, todas as *threads* permanecem bloqueadas aguardando que o usuário seja informado que diferentes *threads* serão criadas para execução de `Trabalho`; na segunda, com auxílio da cláusula *nowait*, o usuário é informado de que a execução de uma destas instâncias foi concluída. A cláusula *nowait* relaxa a condição de sincronização no final do bloco *single*.

```
#pragma omp parallel
{
    #pragma omp single
        printf ( "Iniciando %d instancias de Trabalho. \n",
        omp_get_num_threads ( ) );
    Trabalho ( );
    #pragma omp single nowait
        printf ( "Um trabalho foi terminado. \n" );
}
```

Figura 3.7 – Exemplo de utilização da diretiva *single*.

O padrão não define, para a diretiva **single**, qual *thread* é responsável pela execução do referido trecho de código. É garantido apenas que o código será executado uma única vez, e que todas as *threads* devem permanecer bloqueadas no final do corpo do código da seção **single** aguardando seu término, exceto se for utilizada a cláusula *nowait*. Opcionalmente, a diretiva **master** pode ser utilizada. Esta diretiva garante que o código será executado pela *thread* mestre ou principal de um time.

A diretiva **ordered** especifica que as iterações do *loop* devem ser executadas na mesma ordem em que elas seriam executadas caso fizessem parte de um programa seqüencial.

```
#pragma omp parallel for
for ( int i = 0; I < 1000; ++i )
{
     int s = funcao ( i );
     #pragma omp ordered
     {
          resultado += s;
     }
}
```

Figura 3.8 – Exemplo de utilização da diretiva *ordered*.

No exemplo descrito na Fig. 3.8, o valor da variável `s` vai ser calculado em paralelo para diversos valores de `i`, mas a redução desses valores computados ao valor final, representado pela variável `resultado`, se dará em ordem.

A diretiva **flush** identifica um ponto de sincronização que permite uma visualização consistente da memória por parte das *threads* envolvidas.

Em relação às cláusulas, as mais utilizadas são **private**, **firstprivate**, **lastprivate** e **reduction**. Elas servem como alternativa para manipulação de dados em uma memória compartilhada.

O comportamento-padrão de comunicação entre as *threads* implica que todos os dados locais a *thread* original sejam acessíveis (compartilhados) com outras *threads* criadas. Este comportamento-padrão é alterado através da cláusula **private**, indicando que cópias locais devem ser instanciadas para uso local a cada *thread*. O comportamento-padrão, compartilhamento de instância de dados, é indicado pelo uso da cláusula **shared**.

Em uma situação onde se faça necessário herdar o último valor de uma variável antes da criação de uma *thread*, é utilizada a cláusula **firstprivate**.

No sentido inverso, ou seja, das *threads* criadas para a *thread* original, podem ser aplicadas duas outras cláusulas. A primeira delas é **lastprivate**, que faz com que o valor computado pela última *thread* do programa seja o valor utilizado

pela *thread* original após a sincronização. Uma variável pode ser, ao mesmo tempo, **first** e **lastprivate**.

Uma outra cláusula importante é **reduce**, que por sua vez, é mais elaborada, permitindo a combinação de resultados parciais das *threads* ao final da computação de determinado resultado. Esta combinação pode ser realizada aplicando um dos operadores listados na tab. 3.1. A tabela apresenta também o valor inicial da cópia local do dado em cada *thread*.

Tabela 3.1 – Operadores para redução de retorno de threads em OpenMP.

Operador	**Operação**	**Valor Inicial**
+	Soma	0
-	Subtração	0
*	Multiplicação	1
&	E aritmético	-1
&&	E lógico	1
|	OU aritmético	0
||	OU lógico	0
^	OU exclusivo aritmético	0

Fonte: CAVALHEIRO; SANTOS, 2007, p. 38.

O uso da cláusula **reduce** é exemplificado na seqüência. Neste exemplo, a variável inteira `soma` é inicializada com 100 e são criadas quatro threads, cada uma produzindo o valor 1. No final, o valor acumulado corresponde a 100, ou seja, o valor inicial de `soma` adicionado de uma unidade para cada *thread* executada.

```
int soma = 100;
omp_set_num_threads ( 4 );
#pragma omp parallel reduction ( +: soma )
{
     soma += 1;
}
// Agora soma possui o valor 104
```

Figura 3.9 – Exemplo de utilização da cláusula *reduce*.

Se necessário, pode-se fazer uso da cláusula **if** para tornar uma região paralela condicional. Isto pode ser uma vantagem se não houver trabalho suficiente para tornar o paralelismo interessante.

```
#pragma omp parallel if ( tasks > 1000 )
{
     while ( tasks > 0 )
          donext_task ( );
}
```

Figura 3.10 – Exemplo de utilização da cláusula *if*.

A cláusula **Schedule** permite uma variedade de opções por especificar quais iterações dos laços são executadas por quais threads. Os quatro tipos de escalonamento são: **STATIC**, **DYNAMIC**, **GUIDED** ou **RUNTIME**. Exemplo do uso desta cláusula:

#pragma omp for schedule(DYNAMIC, 4)

No escalonamento **STATIC**, se não houver a especificação do valor ao lado do tipo do escalonamento (como o valor 4, no exemplo acima), o espaço de iteração é dividido em pedaços aproximadamente iguais e cada pedaço é atribuído a cada thread. Porém, se o valor é especificado, o espaço de iteração é dividido em

pedaços, cada um com *n* iterações (no exemplo acima, *n* = 4) e os pedaços são atribuídos ciclicamente a cada thread.

O escalonamento **DYNAMIC** divide o espaço de iteração em pedaços de tamanho *n* (no exemplo acima, *n* = 4), e os atribui para as threads com uma política *first-come-first-served*. Isto quer dizer que se uma thread terminou um pedaço, ela recebe o próximo pedaço na lista. Quando nenhum valor for especificado, é assumido o valor 1 como padrão.

O escalonamento **GUIDED** é similar ao DYNAMIC, mas os pedaços iniciam grandes e tornam-se pequenos de maneira exponencial. O tamanho do próximo pedaço é dado pelo número de iterações restantes dividido pelo número de threads.

O escalonamento **RUNTIME** delega a escolha do escalonamento para a execução, quando é determinado pelo valor da variável de ambiente OMP_SCHEDULE.

3.3.2.4 Serviços de Biblioteca

Além das diretivas implícitas de OpenMP, existe também uma biblioteca com funções que podem ser chamadas de dentro dos programas.

- **void omp_set_num_threads(int *num_threads*)**: configura o número de *threads* requisitadas para as regiões paralelas;
- **int omp_get_num_threads( )**: retorna o número de *threads* em execução em dado momento;
- **int omp_get_max_threads( )**: retorna o valor máximo expresso pela função anterior e geralmente é utilizada para alocar estruturas de dados que possuam um tamanho máximo por *thread* enquanto OMP_DYNAMIC for verdadeiro;
- **int omp_get_thread_num( )**: retorna o número da *thread* (numeradas de 0 até o número de *threads* menos 1);
- **int omp_get_num_procs( )**: retorna o número de processadores disponíveis para o programa;

- **int omp_in_parallel( )**: retorna verdadeiro se a função for chamada dentro de uma região paralela e retorna falso em caso contrário;
- **void omp_set_dynamic(int *expr*)**: habilita (*expr* é verdadeiro) ou desabilita (*expr* é falso) a alocação dinâmica de *threads*;
- **int omp_get_dynamic( )**: retorna verdadeiro ou falso se a alocação dinâmica de *threads* estiver habilitada/desabilitada, respectivamente;
- **void omp_set_nested(int *expr*)**: habilita (*expr* é verdadeiro) ou desabilita (*expr* é falso) paralelismo aninhado;
- **int omp_get_nested( )**: retorna verdadeiro ou falso se o paralelismo aninhado de *threads* estiver habilitado/desabilitado, respectivamente.

3.3.2.5 Vantagens e Desvantagens do Uso de OpenMP

Principais vantagens do uso de OpenMP:

- Simplicidade, pois OpenMP cuida da maioria das coisas para o usuário e a distribuição de tarefas é feita automaticamente pela implementação;
- Flexibilidade, pois paralelizar código já existente é simples, requer poucas modificações e o paralelismo pode ser implementado de forma incremental, já que não requer grandes modificações estruturais no código.

Principais desvantagens do uso de OpenMP:

- Atualmente, só executa de maneira eficiente em arquiteturas de memória compartilhada;
- Requer suporte de um compilador específico (os compiladores que atualmente suportam OpenMP são o GCC, a partir da versão 4.2; o Visual Studio C++ 2005; os compiladores da Intel e o Sun Studio);
- Escalabilidade é limitada pela forma como a memória está instalada no sistema;

- Intensa sobrecarga na paralelização de regiões e laços paralelos;
- Tratamento de erros ainda é um sério problema;
- Falta de alguns tipos de controle mais “finos” sobre as threads e alteração de alguns parâmetros.

3.3.3 Comparação entre OpenMP e Pthreads

Comparado com Pthreads, a ferramenta OpenMP permite uma utilização mais simplificada, devido a apresentar uma curva de aprendizado menos acentuada. Contudo, é possível perceber que com a primeira consegue-se melhores índices de desempenho em casos onde o código da aplicação possa ser bastante otimizado pelo programador.

A flexibilidade de utilização de Pthreads oferece muitos recursos ao programador, cabendo a ele o esforço de utilizá-los a contento. Em contraposição, o paralelismo implícito oferecido por OpenMP restringe o grau de liberdade do programador ao oferecido pelo modelo de programação orientado ao paralelismo de grão fino. Esta menor flexibilidade é a contra-partida requisitada pelo padrão OpenMP para otimizar a execução dos programas.

OpenMP pode ser caracterizada como uma interface de programação voltada para aplicações cuja natureza é melhor expressa pela decomposição de dados. Isto significa que os dados são decompostos, resultando em várias partes que são divididas entre cada uma das regiões paralelas envolvidas no processo de execução da aplicação em questão. Cada região paralela executa o mesmo código, mas sobre um conjunto diferente de dados. Uma vantagem importante desse tipo de método é a possibilidade de divisão do domínio da aplicação, de forma que todas as regiões paralelas envolvidas realizem a mesma quantidade de computação (balanço de carga).

Por outro lado, Pthreads é uma ferramenta de programação voltada para aplicações cuja natureza é expressa através da decomposição de tarefas. Cada fluxo executa uma tarefa diferente. Embora ofereça uma maior flexibilidade de construção de código, este método não mostra-se tão simples de implementar

quanto a decomposição de dados, pois exige uma coordenação maior entre os fluxos de execução envolvidos.

3.4 Resumo

Se comparada com outras arquiteturas multiprocessadas, a arquitetura *multi-core* leva vantagem em alguns aspectos, tais como:

- Economia considerável de espaço e de energia;
- Comunicação mais eficiente entre as diversas unidades (*cores*) presentes no *chip*;
- Desempenho aprimorado para aplicativos *multithreaded*;
- Compatibilidade para mais usuários ou tarefas em aplicativos com muitas transações.

Atualmente, a tecnologia *multi-core* é empregada por diversas empresas fabricantes de hardware e altamente conceituadas no mercado de computadores, como a Intel e a AMD.

Em conclusão, as arquiteturas *multi-core* representam no momento uma alternativa bastante viável em termos de aumento de desempenho e diminuição do consumo de energia. Há de se ressaltar também que, devido a uma gradual e inevitável diminuição do custo desses processadores, pode tornar-se possível a execução paralela de diversas aplicações em *hardware* de baixo custo. Entretanto, para uma total utilização do poder de processamento oferecido pelo *multi-core*, as aplicações devem ser escritas de modo a usar intensivamente o conceito de *threads*. Assim, melhora-se o desempenho de cada aplicação individualmente.

Dentre as diferentes ferramentas de programação *multithread* disponíveis, neste capítulo foram apresentadas características gerais de Pthreads e OpenMP, duas das mais populares neste contexto. A ferramenta OpenMP tem seu foco no paralelismo de dados, o que permite que seja considerada a primeira opção para implementar a aplicação da simulação proposta. Outra característica interessante de

OpenMP é ter sido projetada para promover execução paralela com bons índices de desempenho.

4 Implementação Realizada

Este capítulo apresenta a implementação paralela dos algoritmos de Monte Carlo e Random Walker seguida de uma avaliação do desempenho destes em termos de tempo de processamento. Os trechos de código apresentados referem-se às implementações de ambos algoritmos sobre matrizes bi-dimensionais. As implementações utilizando matrizes tri-dimensionais possuem a mesma estrutura de código, não sendo apresentadas por questões de espaço.

4.1 Diferenças Básicas entre as Implementações Paralelas de Monte Carlo e Random Walker

Apesar de apresentar muitas semelhanças com o algoritmo de Monte Carlo, o algoritmo de Random Walker possui como diferença fundamental a escolha do rótulo para realizar o cálculo de troca de energia.

No algoritmo de Random Walker, um novo sorteio só é realizado quando o rótulo vizinho é igual ao rótulo escolhido. Ou seja, durante a execução, os dois rótulos sorteados (rótulo escolhido e vizinho) possuem o mesmo valor, o que indica que ambos fazem parte da mesma célula. Como não há possibilidade de troca de energia nesta situação, então o cálculo não é realizado e um novo sorteio deve ser feito.

Entretanto, se o rótulo escolhido for diferente de seu vizinho, isto significa que ambos pertencem a extremidades de duas células distintas. Portanto, ao final do processo (cálculo de energia e provável substituição ou não de um rótulo por outro), o rótulo vizinho passará a ser o novo rótulo escolhido, o que descarta a possibilidade de um novo sorteio.

Tal situação é ilustrada no algoritmo apresentado na Fig. 4.1 pelo uso da variável inteira `flagVizinho`, responsável por sinalizar a ocorrência de um sorteio. Se ela é igual ao valor associado à constante simbólica NAO, um novo sorteio

obrigatoriamente é realizado. Porém, se `flagVizinho` é igual ao valor associado à constante simbólica SIM, então o sorteio de um novo rótulo não é realizado.

```
// Sorteio do rotulo a ser escolhido
If ( flagVizinho == NAO )
{
    // Escolhe rotulo pertencente a borda
    RotuloEscolhido [ 0 ] = ( rand ( ) % ( TAM – 4 ) ) + 2;
    RotuloEscolhido [ 1 ] = ( rand ( ) % ( TAM – 4 ) ) + 2;
}
// Armazena o valor do rotulo escolhido
rotEsc = Matriz [ RotuloEscolhido[0] ][ RotuloEscolhido[1] ];
```

Figura 4.1 – Escolha de um rótulo no algoritmo de Random Walker.

O sorteio acontece escolhendo-se aleatoriamente uma posição da matriz (eixos x e y, no caso de uma matriz bi-dimensional e eixos x, y, z, no caso de uma matriz tri-dimensional). Essas posições são posteriormente armazenadas no vetor denominado `RotuloEscolhido[]`. O valor numérico do rótulo escolhido é armazenado na variável `rotuloEscolhido`.

No algoritmo de Monte Carlo, a escolha do rótulo é feita sem o uso da variável `flagVizinho`, pois, independente do rótulo escolhido pertencer a extremidade de uma célula ou não, o sorteio é sempre realizado. A seguir, na Fig. 4.2 é mostrado o trecho de código referente ao processo de escolha de um rótulo no algoritmo de Monte Carlo.

```
// Sorteio do rotulo a ser escolhido
RotuloEscolhido [ 0 ] = ( rand ( ) % ( TAM – 4 ) ) + 2;
RotuloEscolhido [ 1 ] = ( rand ( ) % ( TAM – 4 ) ) + 2;
// Armazena o valor do rotulo escolhido
rotEsc = Matriz [ RotuloEscolhido[0] ][ RotuloEscolhido[1] ];
```

Figura 4.2 – Escolha de um rótulo no algoritmo de Monte Carlo.

No algoritmo Random Walker, para saber se o rótulo escolhido possui vizinhos diferentes, é feito um teste condicional. De acordo com o resultado do teste,

é atribuído um valor à variável `flagVizinho`. Se o rótulo escolhido possui vizinhos diferentes, `flagVizinho` recebe SIM e a execução prossegue; senão, `flagVizinho` recebe NÃO, indicando que um novo sorteio deve ser realizado.

O trecho de código mostrado na Fig. 4.3 pertence ao algoritmo de Random Walker. No algoritmo de Monte Carlo a maneira como é realizado o teste é bastante semelhante. Porém a variável `flagVizinho` não é utilizada.

```
// Teste que verifica se todos os vizinhos são iguais
if (   ( rotEsc != RotulosVizinhos[0] ) ||
       ( rotEsc != RotulosVizinhos[1] ) ||
       ...
       ( rotEsc != RotulosVizinhos[7] ) )
       flagVizinho = SIM;
else
       flagVizinho = NAO;
```

Figura 4.3 – Verificação dos oito rótulos vizinhos ao rótulo escolhido no algoritmo de Random Walker.

O trecho de código na Fig. 4.4 também pertence ao algoritmo de Random Walker e indica que o rótulo escolhido pertence a uma região de borda da matriz. Neste caso, é escolhido o rótulo vizinho e tanto sua posição como seu valor devem ser armazenados para a realização posterior do cálculo de adesão de energia.

```
// Sinaliza que o rotulo escolhido possui vizinhos diferentes
if ( flagVizinho == SIM )
{
       // Sorteia um vizinho dentre os 8 para simular a troca
       linha = rand ( ) %8;
       // Armazena o valor do rotulo escolhido
       vizinhoEscolhido = RotulosVizinhos[ linha ];
                              ...
}
```

Figura 4.4 – Sorteio de um rótulo dentre os oito vizinhos para simular a troca.

Na implementação de Random Walker, se o rótulo escolhido é diferente do rótulo vizinho, então, independentemente do resultado do cálculo de energia, o rótulo vizinho passa a ser o novo escolhido. Isto caracteriza o percurso de caminhada na extremidade de uma célula. Tal situação é representada pelo trecho de código na Fig. 4.5. O valor e a posição do rótulo vizinho são armazenados na variável `rotuloEscolhido` e no vetor `RotuloEscolhido[]`, respectivamente.

```
// Armazena a posicao do novo rotulo escolhido
RotuloEscolhido[ 0 ] = RotEscolhidoVizinho[ 0 ];
RotuloEscolhido[ 1 ] = RotEscolhidoVizinho[ 1 ];
// Armazena o valor do novo rotulo escolhido
rotEscolhido = vizinhoEscolhido;
```

Figura 4.5 – No algoritmo de Random Walker, o rótulo vizinho sorteado passa a ser o novo escolhido, não havendo necessidade de um novo sorteio.

4.2 Inicialização Automática de Matrizes

Tanto o algoritmo de Monte Carlo quanto o algoritmo de Random Walker lidam com a manipulação de tamanhos de índice de matrizes muito grandes (100 *x* 100, 1000 *x* 1000 etc, no caso de matrizes bi-dimensionais e 100 *x* 100 *x* 100, 1000 *x* 1000 *x* 1000 etc, no caso de matrizes tri-dimensionais).

A inicialização de uma matriz deste porte de forma manual, via teclado, por exemplo, é impraticável. Desta forma, decidiu-se por inicializar uma matriz de forma automática, independente de seu tamanho e de suas dimensões, gerando uma distribuição de bolhas aleatória.

Por motivos de espaço, o trecho de código referente ao processo de inicialização de matrizes não será mostrado.

4.3 Início da Simulação e Paralelismo de Tarefas

O trecho paralelo que executa a maior parte da computação é apresentado na seqüência. Trata-se de uma diretiva **parallel for** que controla a evolução do

programa em termos de passos a serem executados por cada *thread*. As variáveis responsáveis por armazenar os valores dos rótulos escolhidos, as posições dos rótulos dentro da matriz e dos vizinhos desse mesmo rótulo escolhido são utilizadas dentro desta diretiva, mostradas nas figuras 4.6 e 4.7 (a primeira refere-se ao algoritmo de MC e a segunda refere-se ao algoritmo de RW), de forma privada por cada *thread*. Importante ressaltar que todos os trechos de código mostrados nesta sub-seção são iguais para ambos algoritmos, Monte Carlo e Random Walker.

O número de passos, representado pela constante simbólica `N_PASSOS`, indica a quantidade máxima de iterações (primeiro laço de repetição, mais externo). Dentro de cada passo, é realizado um número *x* de iterações, que varia de acordo com o algoritmo (segundo laço de repetição, mais interno). No caso de Monte Carlo, cada passo executará **TAMxTAM** (sendo **TAM** o tamanho da matriz). A constante simbólica `PMC`, utilizada como condição de parada dentro do segundo laço *for*, representa o valor do Passo de Monte Carlo. A Fig. 4.6 representa o trecho de código onde acontece todo o processo de evolução do sistema no algoritmo de Monte Carlo.

```
// Ponto de partida da evoluçao do sistema
for ( contador1 = 0; contador1 < N_PASSOS; ++contador1 )
{
#pragma omp parallel for private ( RotuloEscolhido,
RotVizinhoEscolhido, RotulosVizinhos, RotVizinhosOito,
rotEscolhido, vizinhoEscolhido, energiaInicial, energiaFinal,
diferencaEnergia, numero, sorteioTroca )
    for ( contador2 = 0; contador2 < PMC; ++contador2 )
    {
                        ...
        // Calculos da simulacao
                        ...
    }
}
```

Figura 4.6 – Definição de variáveis privadas e início da simulação no algoritmo Monte Carlo.

No algoritmo de Random Walker, cada passo executará **N** vezes (sendo **N** o número total de rótulos que encontram-se nas extremidades das células). A constante simbólica `PRW` representa o valor do Passo de Random Walker, sendo utilizada como condição de parada dentro do segundo laço *for*. Uma outra diferença é em relação ao uso da cláusula **firstprivate**, utilizada para fazer com que a variável `flagVizinho` receba seu valor inicial antes de entrar na região paralela. A Fig. 4.7 ilustra o mesmo processo de evolução do sistema, porém utilizando o algoritmo de Random Walker.

```
// Ponto de partida da evoluçao do sistema
for ( contador1 = 0; contador1 < N_PASSOS; ++contador1 )
{
    flagVizinho = NAO;
#pragma omp parallel for private ( RotuloEscolhido,
RotVizinhoEscolhido, RotulosVizinhos, RotVizinhosOito,
rotEscolhido, vizinhoEscolhido, energialnicial, energiaFinal,
diferencaEnergia, numero, sorteioTroca ) firstprivate (
flagVizinho )
    for ( contador2 = 0; contador2 < PRW; ++contador2 )
    {
                        ...
        // Calculos da simulacao
                        ...
    }
}
```

Figura 4.7 – Definição de variáveis privadas e início da simulação no algoritmo Random Walker.

Em ambos algoritmos, antes de entrar dentro do segundo laço de repetição **for**, é definida uma diretiva OpenMP **parallel for**, o que indica que o *loop* que sucede esta linha é paralelizado. A cláusula **private** define um extenso conjunto de variáveis que serão tratadas de forma privada por cada *thread*.

O cálculo de energia é mostrado na Fig. 4.8 e encontra-se dentro do segundo laço **for** mostrados nos trechos de código das figuras 4.6 e 4.7 (referentes aos algoritmos de MC e RW, respectivamente). Inicialmente, o rótulo escolhido é

comparado com todos os seus 20 vizinhos e, para cada rótulo com valor diferente, é incrementada de uma unidade a variável privada `energiaInicial`, responsável por armazenar o valor de energia na configuração atual do sistema.

O laço de repetição responsável pelas sucessivas comparações entre rótulos é executado de forma paralela devido à diretiva **parallel for**. A cláusula **reduction** combina os resultados individuais da variável `energiaInicial`, calculados por cada uma das threads que executaram o laço, somando-os após o término do *loop*.

```
// Energia inicial do sistema
energialnicial = 0;
// Testa o rotulo escolhido com seus 20 vizinhos
#pragma omp parallel for reduction ( +: energialnicial )
for ( numero = 0; numero < 20; ++numero )
{
      // Para todo rotulo vizinho diferente do sorteado
      If ( rotEscolhido != RotulosVizinhos[ numero ] )
            // Adiciona 1 a energia
            energialnicial += 1;
}
```

Figura 4.8 – Realização do cálculo de energia para o rótulo escolhido.

Deve-se observar que uma preocupação a ser tomada quando da implementação de ambos algoritmos, tanto nas implementações seqüenciais como paralelas, é o cálculo de adesão sobre rótulos situados em regiões limítrofes da matriz. Quando um destes rótulos é sorteado, um tratamento especial deve ser realizado, posto que nestes casos é possível que não haja registro de algum dos vizinhos. A situação é ilustrada na Fig. 4.9.

1	1	1	1	2	2	4	4	4	4
1	1	1	2	2	2	2	4	4	4
1	1	2	2	2	2	2	4	4	4
1	1	2	2	2	2	4	4	4	4
3	3	3	2	2	5	5	5	4	4
3	3	3	3	3	5	5	5	5	4
3	3	3	3	5	5	5	5	5	5
3	3	3	3	5	5	5	7	7	7
3	3	6	6	6	6	6	7	7	7
3	3	6	6	6	6	6	6	7	7

Figura 4.9 – Escolha de rótulos em posições limítrofes da matriz.

Em tal situação, a estratégia adotada neste trabalho foi de ignorar os cálculos em rótulos limítrofes. Assim, sendo sorteado um rótulo próximo aos limites da matriz (bi ou tri-dimensional), o sorteio é descartado, sendo sorteado um novo rótulo. Assume-se que o erro nos resultados de simulação não será relevante, tendo em vista as grandes dimensões das matrizes.

Em uma matriz tri-dimensional, a estratégia adotada é a mesma. Porém, deve-se ter cautela em relação aos planos que encontram-se nas extremidades da mesma.

A matriz 3D mostrada na Fig. 4.10 é composta por sete planos. O plano 1, por pertencer a uma extremidade, seria descartado do sorteio. Ou seja, nenhum rótulo pertencente a este plano poderia ser escolhido para o cálculo. Tal situação também aplica-se ao último plano da matriz (plano 7), por fazer parte da extremidade oposta. Em contrapartida, os rótulos pertencentes aos planos restantes (que encontram-se entre os planos 1 e 7) fariam parte do sorteio normalmente.

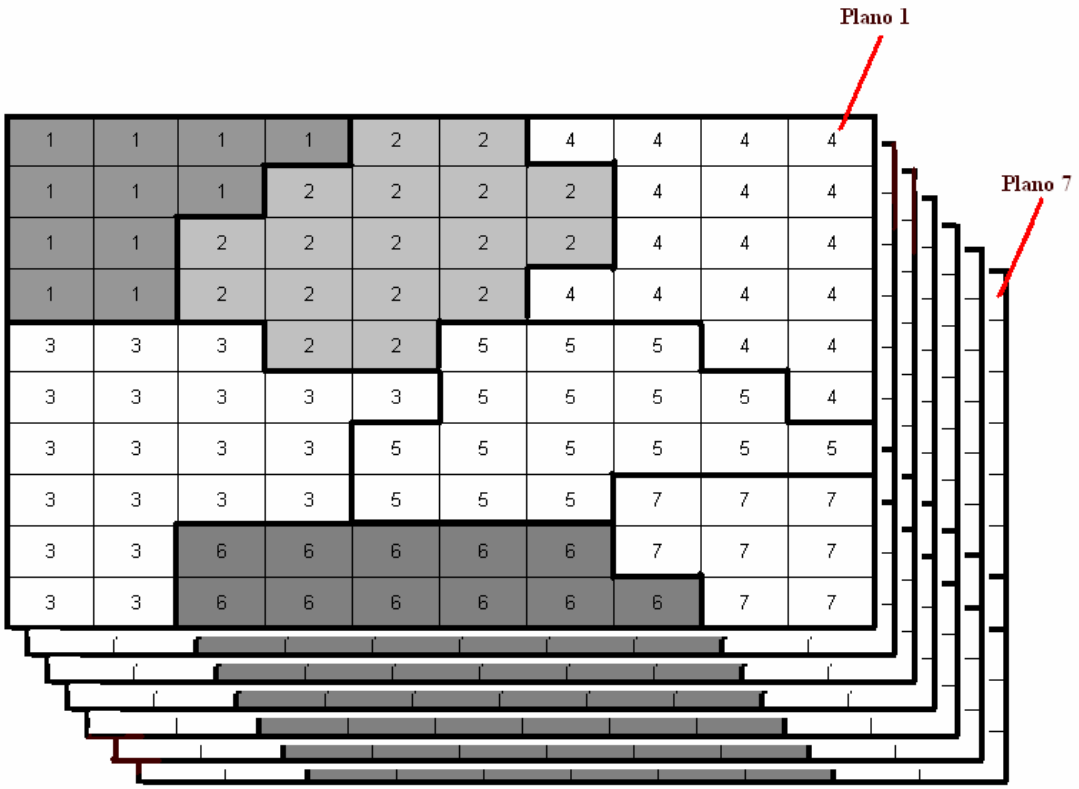

Figura 4.10 – Regiões limítrofes em uma matriz 3D.

O mesmo cálculo de energia realizado sobre o rótulo escolhido é repetido para o rótulo vizinho sorteado. A substituição é simulada e o rótulo escolhido é substituído por seu vizinho. Desta forma, o valor do vizinho é comparado com os 20 rótulos da vizinhança (incluindo ele próprio) e, para cada rótulo diferente, a variável `energiaFinal` é incrementada em uma (1) unidade. Assim como no trecho de código anterior, este laço de repetição é paralelizado e a variável inteira `energiaFinal` tem seus resultados parciais somados ao final do *loop*. A Fig. 4.11 ilustra este processo.

```
// Energia após a possivel troca com o vizinho
energiaFinal = 0;
// Simula a troca entre o rotulo escolhido e o vizinho
#pragma omp parallel for reduction ( +: energiaFinal )
for ( numero = 0; numero < 20; ++numero )
{
     // Para todo rotulo vizinho diferente do sorteado
     If ( vizinhoEscolhido != RotulosVizinhos[ numero ] )
          // Adiciona 1 a energia
          energiaFinal += 1;
}
```

Figura 4.11 – Realização do cálculo de energia para o rótulo vizinho.

Por último, `energiaFinal` é subtraída de `energiaInicial` e o resultado, armazenado em `diferencaEnergia`, determinando o valor da diferença de energia entre as duas situações. Seu valor é decisivo para a evolução do sistema.

```
// Calcula a diferenca na energia entre as duas situacoes
diferencaEnergia = energiaFinal – energiaInicial;
```

Figura 4.12 – Cálculo da diferença de energia entre os rótulos escolhido e vizinho.

Se esta diferença for igual a zero, é considerada a possibilidade de ocorrer a substituição do rótulo escolhido pelo rótulo vizinho. Neste caso, é sorteado um valor 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. Caso a diferença de energia seja menor do que zero, então a substituição ocorre sem qualquer tipo de restrição. Ou seja, a probabilidade de troca é assumida em 100%. A Fig. 4.13 ilustra esta situação.

```
// Variavel utilizada para gerar um numero aleatorio
sorteioTroca = 0;
// Considerando a diferenca de energia
if ( diferencaEnergia == 0 )
      // Sorteio do valor de probabilidade
      sorteioTroca = rand ( ) % 101;
```

Figura 4.13 – Sorteio do valor de probabilidade de troca, caso a diferença de energia seja igual a zero.

A diretiva **critical**, utilizada na parte final do código de ambos algoritmos, garante a exclusão mútua entre *threads* que desejam realizar o processo de substituição ao mesmo tempo. A estratégia adotada minimiza o problema de contenção de execução, permitindo leituras concorrentes e sincronizando apenas quando das operações de escrita. O algoritmo permite que os valores dos rótulos sejam lidos da matriz sem que seja necessário adquirir direito de execução em regime de exclusão mútua.

O cálculo é realizado e neste momento é requisitada exclusão mútua no acesso a matriz. Uma nova leitura dos mesmos rótulos é então realizada, sendo os valores comparados com os da leitura inicial. Caso os valores obtidos nas duas leituras sejam iguais, o cálculo de energia é validado e a matriz é atualizada. Caso os valores sejam diferentes, o cálculo é descartado. Valores diferentes indicam a atuação de um outro *thread* sobre o mesmo conjunto de rótulos. A Fig. 4.14 ilustra a situação de duas *threads* atuando em regiões próximas uma da outra (dois rótulos cinza na Fig. 4.14).

1	1	1	1	2	2	4	4	4	4
1	1	1	2	2	2	2	4	4	4
1	1	2	2	2	2	2	4	4	4
1	1	2	2	2	2	4	4	4	4
3	3	3	2	2	5	5	5	4	4
3	3	3	3	3	5	5	5	5	4
3	3	3	3	5	5	5	5	5	5
3	3	3	3	5	5	5	7	7	7
3	3	6	6	6	6	6	7	7	7
3	3	6	6	6	6	6	6	7	7

Figura 4.14 – Escolha de dois rótulos em regiões próximas.

Importante ressaltar também que é possível que duas *threads* escolham um mesmo rótulo simultaneamente (ao mesmo tempo). Tal fato não constitui um problema grave, tendo em vista que a primeira *thread* que chegar a região crítica do código irá bloquear a outra, impossibilitando que as duas atualizem o mesmo rótulo ao mesmo tempo, o que resultaria em erro. A primeira *thread* que entrar dentro da região crítica realiza a substituição, se necessário, e a segunda irá desconsiderar o cálculo, ao ver que um dos rótulos naquela região da matriz foi alterado.

Na Fig. 4.15 encontra-se o trecho de código que trata da substituição do rótulo escolhido pelo seu vizinho. Após o teste condicional indicar que a substituição deve ocorrer, a execução deve prosseguir de forma não paralela. A diretiva **critical** garante que apenas uma thread por vez execute esse trecho.

Os 20 rótulos vizinhos ao rótulo escolhido, armazenados no vetor `RotulosVizinhos[]`, são comparados um a um com suas posições equivalentes na matriz original. Se algum dos valores presentes na matriz original for diferente do seu correspondente no vetor `RotulosVizinhos[]`, então isso significa que a matriz foi alterada anteriormente por alguma outra *thread* executando próximo aquela região. Se todos os valores forem iguais, então a substituição acontece de

fato, ou seja, a posição da matriz onde encontra-se o rótulo escolhido recebe o valor de seu vizinho. A função da variável `contaNumTroca` é apenas registrar o número total de trocas ocorridas durante toda a simulação.

```
// Se a energia nao mudou, realiza a troca
if ( ( diferencaEnergia < 0 ) || ( sorteioTroca > PROB ) )
{
      // Regiao critica do codigo
      #pragma omp critical
      {
            // Compara os valores armazenados com as
            // posicoes reais da matriz para ver se
            // algum destes valores mudou
            if (  ( RotViz[0] == Matriz[ RotEsc[0] – 1 ][ RotEsc – 1 ] ) &&
                  ( RotViz[0] == Matriz[ RotEsc[0] – 1 ][ RotEsc[1] ] ) &&
                                          ...
                  ( RotViz[19] == Matriz[ RotEsc[0] + 1 ][ RotEsc[1] – 2 ] ) )
            {
                  // Se nenhuma posicao da matriz foi alterada, realiza
                  // de fato a substituicao do rotulo escolhido pelo
                  // seu vizinho
                  Matriz[ RotEsc[0] ][ RotEsc[1] ] = vizinhoEscolhido;
                  contaNumTroca++;
            }
      }
}
```

Figura 4.15 – Utilização da diretiva *critical* para garantir exclusão mútua no momento da substituição de um rótulo.

Em relação ao trecho de código executado dentro da região crítica, é válido ressaltar que poderia ter sido feito de outra forma. Uma alternativa seria dividir a matriz em blocos de tamanhos iguais, de acordo com o número de *threads* disponíveis.

Por exemplo, se houvessem duas *threads*, a matriz seria dividida em duas partes e cada *thread* executaria dentro de uma dessas duas regiões. Se houvessem quatro *threads*, a matriz seria dividida em quatro regiões e cada *thread* executaria dentro de uma dessas quatro regiões, e assim por diante.

Esta estratégia de execução foi descartada, pois espera-se manter a uniformidade de distribuição dos sorteios de rótulos entre todos os *threads*. Como o limite de execução das simulações é determinado em termos de número de passos

de simulação, delimitar zonas de atuação para *threads* implicaria em, eventualmente, observar *threads* com diferentes velocidades de execução, tornando a evolução do sistema irregular.

A alternativa adotada nos dois algoritmos para lidar com o problema da região crítica mostrou-se adequada por diminuir o problema de contenção. Mesmo ocorrendo situações em que o cálculo da troca de energias é descartado, evita-se um grande número de operações desnecessárias de sincronização.

4.4 Análise de Desempenho

Em relação aos testes de desempenho, foi realizado um extenso conjunto de simulações. Os resultados da execução paralela do algoritmo de *Random Walker* foram comparados com os resultados da execução paralela do algoritmo de Monte Carlo. A comparação das execuções paralelas também foi feita com os resultados das execuções seqüenciais de ambos algoritmos.

O *hardware* utilizado na implementação foi um computador Intel Pentium Core 2 Duo 6320, com freqüência de *clock* 2.0 GHz e memória RAM de 2 GB. O sistema operacional utilizado foi GNU-Linux *kernel* 2.6.22-14. A linguagem de programação C e o compilador icc (Intel), versão 10.1.015, para o suporte a sintaxe OpenMP serviram de base para a escrita do código das aplicações.

Na seqüência, encontram-se registrados, nas tabelas 4.1, 4.2 e 4.3, a avaliação de desempenho da execução dos algoritmos MC e RW, tanto seqüenciais quanto paralelos, considerando matrizes bi-dimensionais de diferentes tamanhos (500 *x* 500, 1000 *x* 1000, 5000 *x* 5000). O tempo de execução é considerado em segundos.

Tabela 4.1 – Análise de desempenho dos algoritmos de MC e RW seqüenciais e paralelos, considerando matrizes 2D de tamanho 500 x 500.

Estratégia	Nº de Threads	Tempo de Execução	Tempo de CPU
MC seq	1	5,63	98%
MC par	2	27,78	198%
MC par	4	42,56	197%
MC par	8	64,89	198%
RW seq	1	3,56	99%
RW par	2	19,99	199%
RW par	4	34,46	198%
RW par	8	52,22	197%

Tabela 4.2 – Análise de desempenho dos algoritmos de MC e RW seqüenciais e paralelos, considerando matrizes 2D de tamanho 1000 x 1000.

Estratégia	Nº de Threads	Tempo de Execução	Tempo de CPU
MC seq	1	26,39	97%
MC par	2	102,84	197%
MC par	4	162,49	198%
MC par	8	271,19	199%
RW seq	1	17,67	99%
RW par	2	89,36	198%
RW par	4	143,83	196%
RW par	8	233,98	198%

Tabela 4.3 – Análise de desempenho dos algoritmos de MC e RW seqüenciais e paralelos, considerando matrizes 2D de tamanho 5000 x 5000.

Estratégia	Nº de Threads	Tempo de Execução	Tempo de CPU
MC seq	1	1.221,83	98%
MC par	2	3.186,21	196%
MC par	4	4.656,53	198%
MC par	8	--	197%
RW seq	1	683,40	97%
RW par	2	2.222,24	199%
RW par	4	--	198%
RW par	8	--	198%

Também foi observado em todos os testes que ambas aplicações (MC e RW) caracterizam-se por ocupar grande parte do tempo de processamento disponível na CPU. Nas versões seqüenciais, a taxa de utilização da CPU registrada foi de 100%, enquanto que nas versões paralelas, a utilização esteve muito próxima a 200%, observando o fato da existência de dois processadores. A taxa de utilização de CPU apresentada refere-se a utilização máxima observada com ferramentas de monitoração providas pelo sistema operacional.

Conforme verificado pelos resultados mostrados nas três tabelas mostradas anteriormente, o aumento do número de *threads* utilizados nas versões paralelas acarreta em uma perda de desempenho, de acordo com o tamanho do sistema a ser simulado. Comparando apenas os resultados das versões paralelas entre si, o melhor desempenho foi obtido utilizando duas (2) *threads* de serviço, mais especificamente com o algoritmo paralelo de RW, fato esperado uma vez que a configuração disponível possui dois *cores*. Desta forma, quanto maior o número de *threads* envolvidas no processo de simulação, maior a sobrecarga do sistema como um todo, no caso de processadores com dois núcleos.

Devido a problemas técnicos, não foi possível coletar os resultados de desempenho das execuções dos algoritmos de MC paralelo com oito (8) threads, RW paralelo com quatro (4) threads e RW paralelo com 8 threads, todos implementados em uma matriz 5000 x 5000.

Um detalhe importante a ser comentado é a respeito do desempenho dos algoritmos paralelos em comparação com o desempenho dos algoritmos seqüenciais. Como pode-se observar nos resultados mostrados nas tabelas, tanto a versão paralela de RW quanto a versão paralela de MC apresentaram desempenho inferior às suas respectivas versões seqüenciais.

Tal fato deve-se, provavelmente, ao uso da diretiva *critical* nas aplicações paralelas. Pois, desta forma, uma ou mais threads acabam sendo impedidas de continuar suas execuções devido a uma *thread* que encontra-se em execução dentro da região crítica. Uma forma de solucionar este problema seria otimizar o código dentro da região crítica, afim de diminuir o número de instruções dentro da mesma, fazendo com que o tempo de espera das outras threads fosse reduzido.

Como este trabalho não trata de questões relacionadas a desempenho, sugere-se que este aspecto seja trabalhado em projetos futuros. Em particular, deve ser comparado o desempenho obtido com as implementações originais em Pthreads (CERCATO et al 2006) e as realizadas em OpenMP no presente trabalho. Também devem ser avaliados outros recursos de programação em OpenMP para identificar mecanismos que possam melhorar este desempenho.

5 Teste de Aderência

Este capítulo apresenta uma breve introdução sobre a lei de von Neumann e, em seguida, apresenta a mesma sendo utilizada na comparação dos resultados das simulações para medida das distâncias.

5.1 A Lei de von Neumann

Uma espuma bi-dimensional pode ser descrita como uma camada de bolhas confinadas entre placas de cristal paralelas, cuja distância é menor do que o tamanho de uma bolha. Individualmente, uma bolha assemelha-se a um polígono com extremidades curvadas, sendo separadas de uma outra bolha vizinha através de uma fina película. Também ocorrem trocas de gases entre uma bolha e o meio externo (HILGENFELDT, 2001).

Em 1952, John von Neumann definiu uma fórmula matemática para prever a taxa de crescimento de uma célula ou bolha bi-dimensional (NEUMANN, 1952). A taxa de crescimento ou de encolhimento de uma bolha 2D dentro de uma espuma não depende de seu formato ou tamanho, mas sim do número de extremidades ou lados *n* desta mesma bolha (HILGENFELDT, 2001). Deste modo, a fórmula que define a “lei de *von Neumann*” é:

$$\dot{a} = D_2(n - n0),$$

onde D_2 é denominado coeficiente de difusão e $\dot{a}$ é o valor da área do polígono (bolha). O valor constante que representa um crescimento neutro ou nulo é $n_0 = 6$. Isto significa que bolhas com menos do que seis lados diminuem, bolhas com mais do que seis lados se expandem e bolhas com exatos seis lados permanecem com o seu tamanho inalterado (HILGENFELDT, 2001).

Em três dimensões, a evolução do volume *V* de uma bolha poliedral é governada por:

$$V_F^{-1/3} v_F = D_{eff} G(F) \equiv - D_{eff} V_F^{-1/3} \left\langle \int_{faces} H \, dA \right\rangle_{F'}$$

com outro coeficiente de difusão *Deff*. A taxa de crescimento é proporcional à integral da curvatura *H* de faces, enquanto o crescimento da bolha à custa de outras bolhas vizinhas é governada pelas diferenças de pressão $\Delta p \; \alpha \; H$ (lei de Young-Laplace). Por definição, $H > 0$ em faces convexas, o que favorece a redução da bolha ($V < 0$) (HILGENFELDT, 2001).

Embora seja utilizada por alguns pesquisadores, a fórmula de von Neumann para estruturas 3D ainda não foi totalmente validada. O leitor interessado em saber mais detalhes sobre a lei de von Neumann encontra material disponível em (HILGENFELDT, 2001).

5.2 Resultados das Simulações

Os resultados práticos apresentados dividem-se em duas categorias: variação do número de células (bolhas) presentes no sistema e variação da área média das células como um todo. Abaixo, são mostrados dois tipos de gráficos que representam estas duas situações. Foram realizados experimentos referentes a três algoritmos: MC seqüencial, MC paralelo e RW paralelo, sendo que estes dois últimos foram executados utilizando duas (2) *threads* de serviço na primeira simulação, quatro (4) *threads* na segunda e oito (8) *threads* na terceira simulação.

Todas as simulações apresentadas nos gráficos a seguir evoluíram dentro de um número total de 1100 passos. Foi utilizado o mesmo tamanho de matriz para as três simulações: 2000 x 2000. Importante observar que todas as execuções consideraram a mesma semente para geração da matriz inicial.

Os três primeiros gráficos a seguir mostram a variação do número de células no sistema no decorrer da simulação, de acordo com o número de passos executados. Inicialmente, o número total de células presentes no sistema é de aproximadamente 10800, em todas as três simulações.

Como indicado pelo gráfico da Fig. 5.1, onde as implementações paralelas utilizam duas *threads*, o número de células decresce conforme o número de passos

aumenta ao longo do tempo. No passo 1100 da simulação, as células começam a atingir um ponto de equilíbrio e o sistema torna-se mais estável.

Os algoritmos seqüencial e paralelo de Monte Carlo apresentaram resultados muito semelhantes. Ambos tiveram poucas variações entre os passos executados e, ao final da simulação, apresentaram exatamente o mesmo número final de células. O algoritmo paralelo de Random Walker apresentou resultados um pouco diferentes dos outros dois. O número final de células ao término da simulação foi ligeiramente maior.

Outro detalhe importante é a respeito do número de substituições de rótulos em cada passo de simulação. Nos passos iniciais, este número é bastante elevado, devido ao alto número de células e a proximidade entre as mesmas. Entretanto, essas substituições acabam tornando-se menos freqüentes, conforme o número de células diminui. Tal fato deve-se também ao aumento na área média das células restantes ao longo da simulação, pois as regiões de contato (bordas) entre as mesmas diminui.

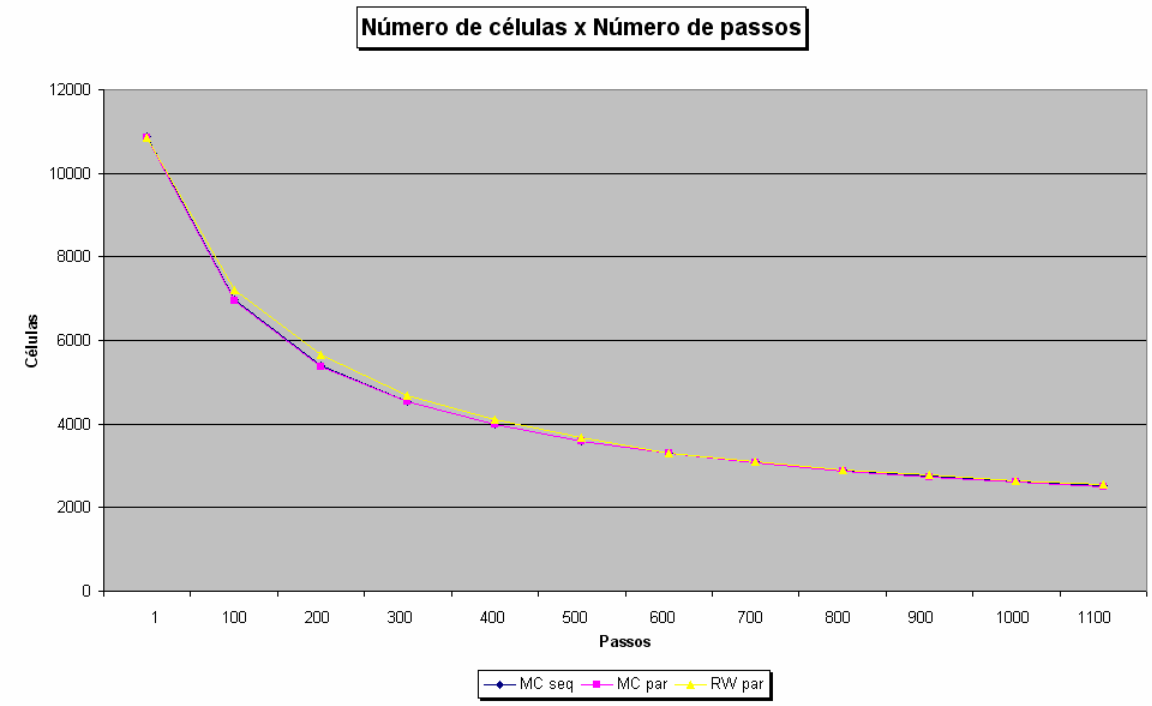

Figura 5.1 – Gráfico mostrando a variação do número de células de acordo com o número de passos simulados, com os algoritmos paralelos de MC e RW utilizando duas (2) *threads* de serviço.

O gráfico ilustrado na Fig. 5.2 mostra a mesma variação do número de células pelo número de passos executados, considerando que os algoritmos paralelos utilizam quatro *threads*. Porém, a diferença dos resultados entre os algoritmos de RW paralelo e MC seqüencial e paralelo mostraram-se ligeiramente maiores. Mesmo assim, ao final do experimento, o número total de células restantes com a execução dos três algoritmos mostrou valores bastante próximos.

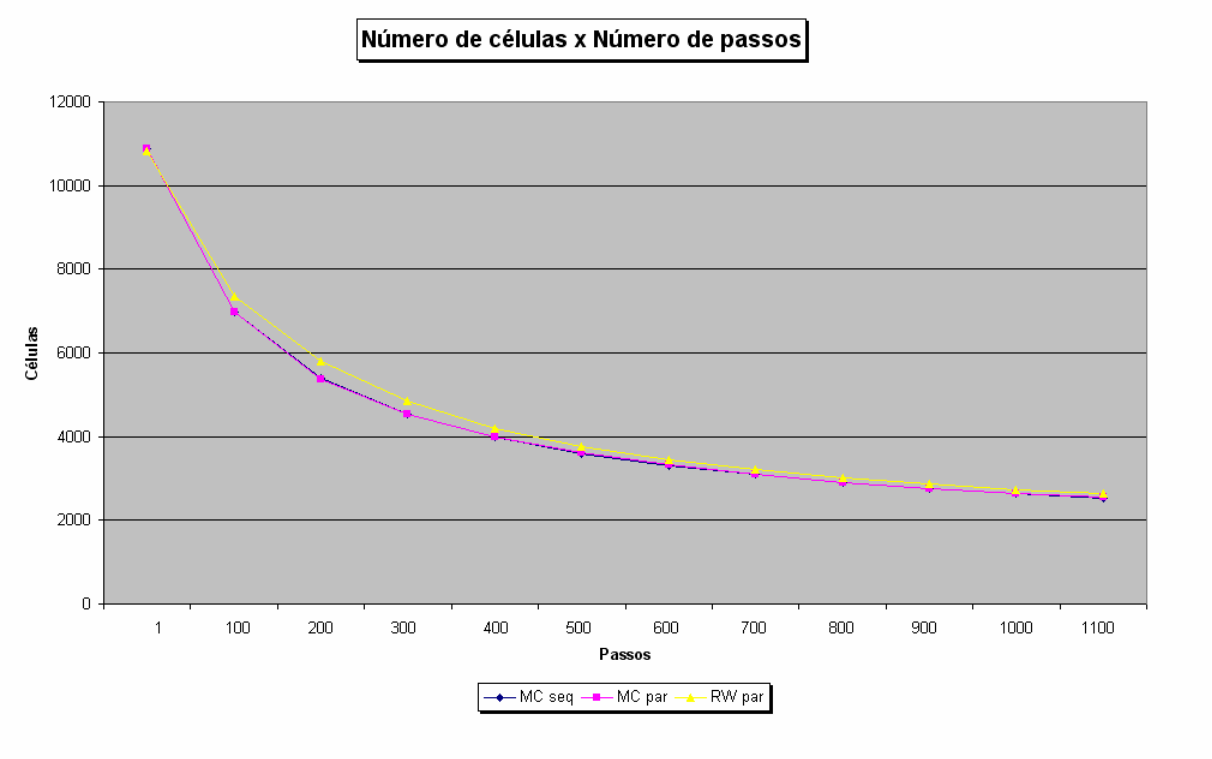

Figura 5.2 – Gráfico mostrando a variação do número de células de acordo com o número de passos simulados, com os algoritmos paralelos de MC e RW utilizando quatro (4) *threads* de serviço.

O gráfico ilustrado na Fig. 5.3 mostra a mesma relação entre o número de células pelo número de passos executados no decorrer do tempo, com os algoritmos de MC e RW paralelos utilizando oito *threads* de serviço. Desta vez, a diferença dos resultados mostrados ao longo da simulação pelo algoritmo de RW paralelo foram mais visíveis. Entretanto, ao final do experimento, os três algoritmos novamente mostraram resultados semelhantes, com uma diferença um pouco mais acentuada do RW paralelo em relação aos outros dois algoritmos.

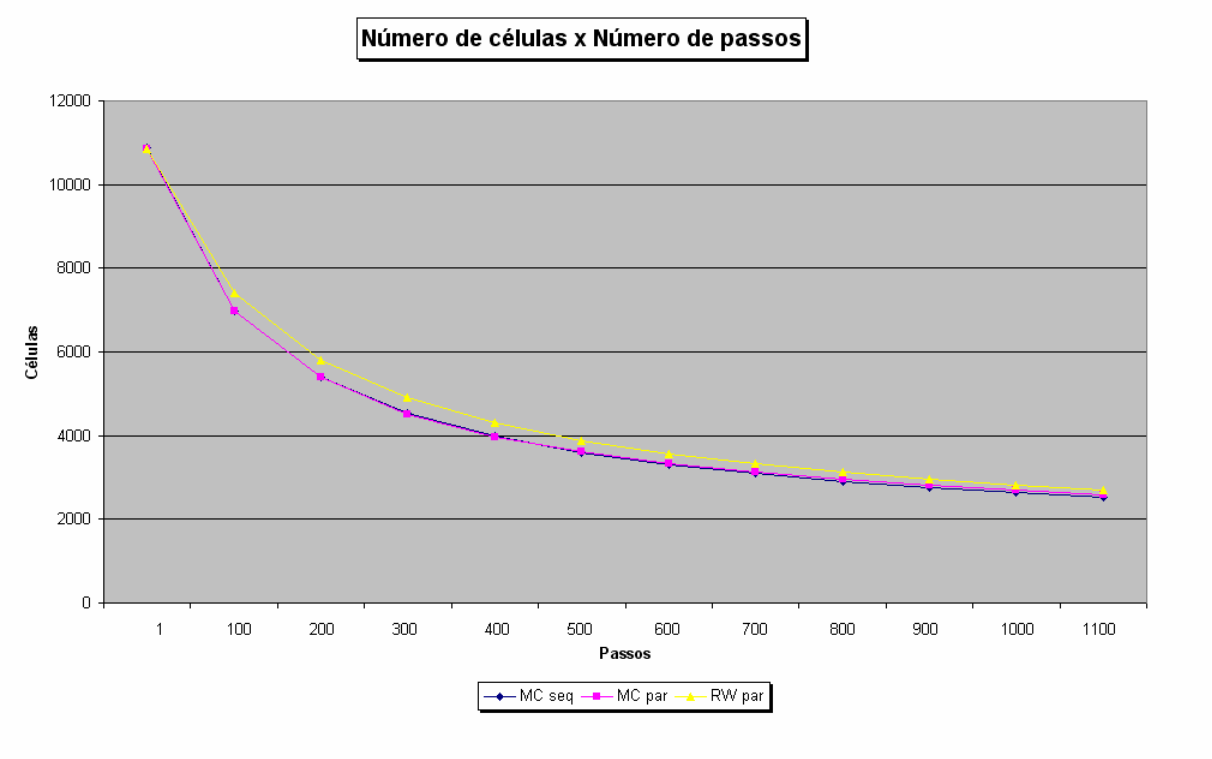

Figura 5.3 – Gráfico mostrando a variação do número de células de acordo com o número de passos simulados, com os algoritmos paralelos de MC e RW utilizando oito (8) *threads* de serviço.

O segundo tipo de resultado, apresentado nos três próximos gráficos, mostra a variação do valor da área média das células presentes no sistema ao longo da simulação, de acordo com o número de passos executados. Inicialmente, o valor da área média é de aproximadamente 368. Ele é calculado através da divisão entre o número total de elementos presentes na matriz pelo número total de células em cada passo da simulação.

Como indicado pelo gráfico da Fig. 5.4, onde os algoritmos paralelos de MC e RW utilizam duas *threads*, a área média das células cresce com o avanço da simulação. A partir de um determinado momento, o sistema atinge uma certa estabilidade e o crescimento das células torna-se mais lento. Com isso, o número de rótulos pertencentes às células (e conseqüentemente, a sua área) também estabiliza-se.

Novamente, as implementações seqüencial e paralela de Monte Carlo apresentaram resultados muito semelhantes, como indica o comportamento das curvas apresentadas. Ao término da simulação dos dois algoritmos, o valor da área média das células foi exatamente o mesmo. O algoritmo paralelo de Random Walker mostrou resultado semelhante aos dois algoritmos de Monte Carlo.

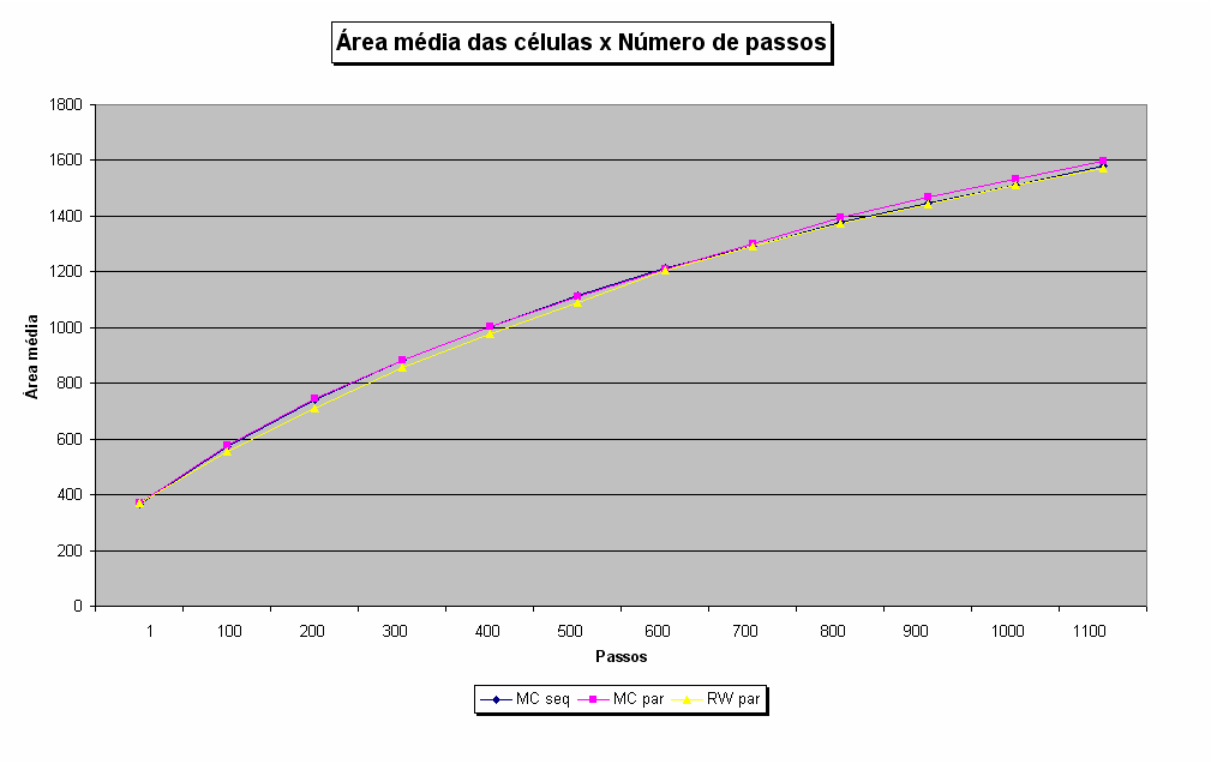

Figura 5.4 – Gráfico mostrando a variação do valor numérico da área média das células de acordo com o número de passos simulados, com os algoritmos paralelos de MC e RW utilizando duas (2) *threads* de serviço.

O gráfico ilustrado na Fig. 5.5 mostra a mesma variação do valor numérico da área média das células pelo número de passos executados, considerando que os algoritmos paralelos utilizam quatro *threads*. Porém, a diferença dos resultados entre os algoritmos de RW paralelo e MC seqüencial e paralelo mostraram-se ligeiramente maiores. Ao final do experimento,os resultados mostrados pelos dois algoritmos de MC e o algoritmo de RW paralelo mostraram valores de área um pouco diferentes.

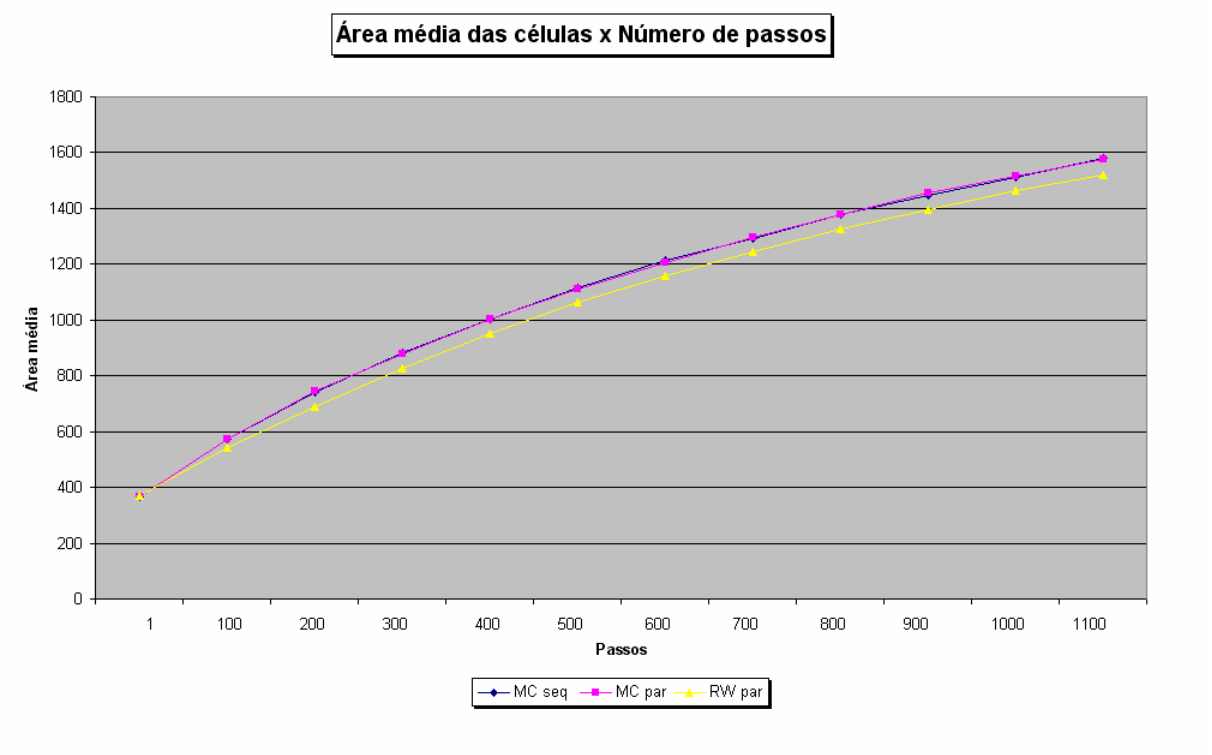

Figura 5.5 – Gráfico mostrando a variação do valor numérico da área média das células de acordo com o número de passos simulados, com os algoritmos paralelos de MC e RW utilizando quatro (4) *threads* de serviço.

O gráfico ilustrado na Fig. 5.6 considera que os algoritmos paralelos utilizam oito *threads*. Desta vez, os três algoritmos mostraram resultados ligeiramente diferentes, onde o RW paralelo mostrou valores de área média menores que os outros dois algoritmos ao final da simulação. Tal fato ocorreu devido ao número de células restantes ao final da simulação gerada pelo algoritmo de RW ter sido maior que o número de células dos outros dois algoritmos.

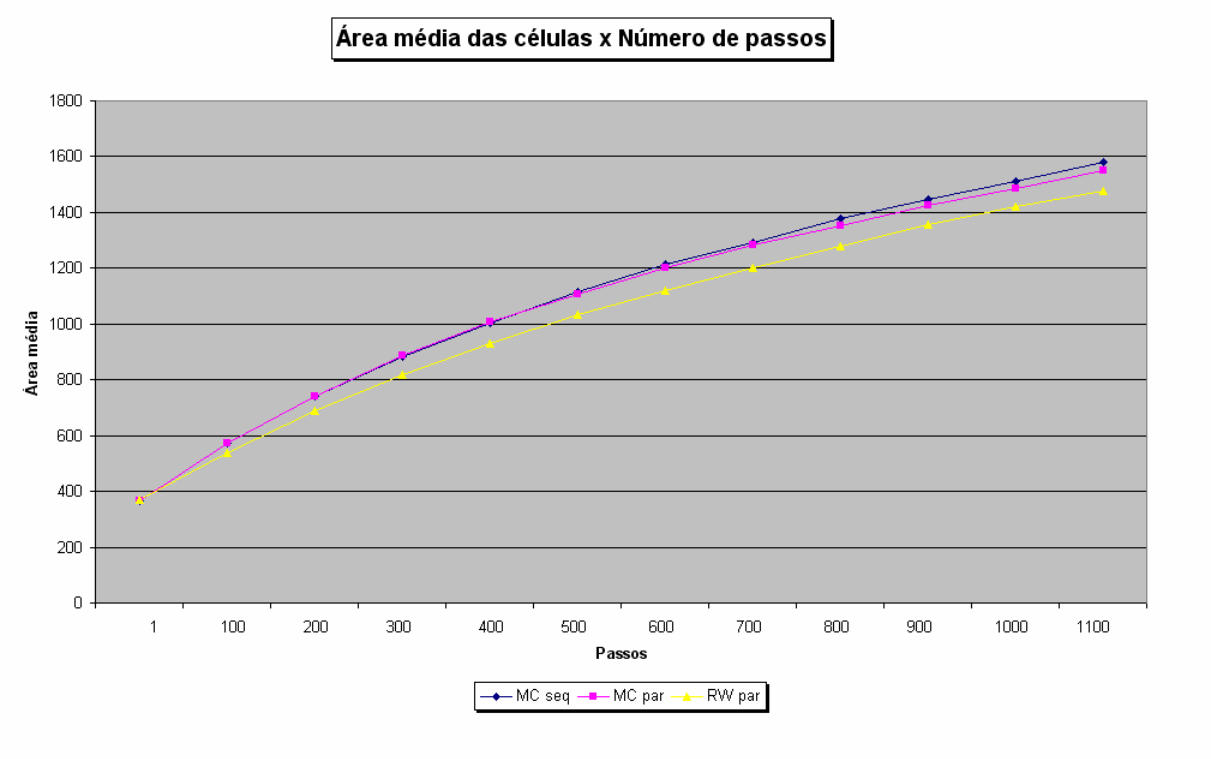

Figura 5.6 – Gráfico mostrando a variação do valor numérico da área média das células de acordo com o número de passos simulados, com os algoritmos paralelos de MC e RW utilizando oito (8) *threads* de serviço.

Estes resultados sugerem que o algoritmo de *Random Walker* não é exato, mas fornece uma aproximação muito boa dos resultados de simulações em relação ao algoritmo de Monte Carlo. Sendo assim, o primeiro torna-se uma ótima estratégia para obtenção de resultados de forma eficiente, sem consumo adicional de memória.

Como continuação a este trabalho, sugere-se uma avaliação mais criteriosa dos resultados apresentados para determinação da viabilidade de utilização da técnica de *Random Walker* paralelo como opção ao Monte Carlo.

6 Conclusão e Trabalhos Futuros

Este trabalho contribuiu por apresentar duas implementações paralelas, com OpenMP, dos algoritmos de Monte Carlo e Random Walker e um comparativo dos resultados das simulações apresentadas pela execução destes dois algoritmos.

As sugestões de trabalhos futuros também foram identificadas no final dos capítulos 4 e 5, sendo ressaltadas também no presente capítulo.

Como dito anteriormente, este trabalho não trata de questões relacionadas ao desempenho das aplicações. Sugere-se que este aspecto seja trabalhado em projetos futuros. Devem ser avaliados outros recursos de programação em OpenMP para identificar mecanismos que possam melhorar este desempenho, assim como alternativas para otimizar alguns trechos de código visando melhorar a performance dos algoritmos paralelos.

E por último, mas não menos importante, necessita-se fazer uma análise criteriosa dos resultados quantitativos apresentados neste trabalho afim de validar futuramente o algoritmo de *Random Walker*. Tal avaliação deve ser feita, preferencialmente, por profissionais ligados às áreas de Bio-Informática, Física e Biologia, devido ao fato da validação deste mesmo algoritmo não fazer parte do escopo da área de Ciência da Computação.

Referências

BUTENHOF, David R. **Programming with POSIX Threads**. Addison-Wesley Professional Computing Series, 1997.

CARISSIMI, Alexandre da Silva; OLIVEIRA, Rômulo Silva de; TOSCANI, Simão Sirineo. **Sistemas Operacionais**. Série livros didáticos, número 11, 3ª edição. Porto Alegre: Instituto de Informática da UFRGS: Editora Sagra Luzzatto, 2004.

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.

CAVALHEIRO, Gerson. Geraldo. H. Princípios da programação concorrente. In: Adenauer Correia Yamin; Jorge Luis Victória Barbosa. (Org.). ERAD 2004 – 4ª Escola Regional de Alto Desempenho. 1ª ed. Porto Alegre: SBC, 2004, v. 1, p. 3 – 39.

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

CERCATO, F. P., MOMBACH, J. C. M., CAVALHEIRO, G. G. H. **High Performance simulations of the cellular 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.

CHANDRA, R., DAGUM, L., KOHR, D., MAYDAN, D., McDONALD, J., and MENON, R. **Parallel Programming in OpenMP**. Morgan Kaufmann, San Francisco, 2001.

COSTA, Alexandre Gomes da. **Implementação eficiente do modelo de Potts celular em processadores multi-core**. Pelotas: UFPEL, 2007. (Não publicado)

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

GLAZIER, J. A.; WEAIRE, D. **The kinetics of cellular patterns**. *Journal of Physics*, v. 4, n.1, p. 1867 – 1894, 1992.

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.

GUSATTO, Éder. **Uma nova abordagem na implementação do modelo de Potts celular buscando eficiência e explorando paralelismo**. Monografia (Graduação). São Leopoldo: UNISINOS. 2004.

GUSATTO, 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.

HAMMERSLEY, J. M. and HANDSCOMB, D. C. **Monte Carlo Methods**. Methuen & Co., Londres, 1964.

HILGENFELDT, Sascha, Kraynik, Andrew M., Koehler, Stephan A., Stone, Howard A. **An Accurate von Neumann's Law for Three-Dimensional Foams**. *Physical Review Letters*, V. 86(12). 2001.

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

KUMAR, R; TULLSEN, D. M. The Architecture of efficient multi-core processors: a holistic approach. In: **Advances in Computers**, V. 69. ZELKOWITZ, M. V., ed. Amsterdam: Elsevier. 2007.

LOUREIRO, Marcos Paulo de Oliveira. **Propagação de Danos no Modelo de Potts**. Minas Gerais: Universidade Federal de Viçosa, 2006.

LUZ, Leonardo Lobo, CAVALHEIRO, Gerson Geraldo H., CASTAÑEDA, Cristian Fernando F., GUIDO, Vitor Härter. **Análise de Desempenho do Algoritmo Paralelo de Monte Carlo Utilizando a Ferramenta OpenMP**. In: **VIII Escola Regional de Alto Desempenho 2008**, Santa Cruz do Sul, SBC, 2008.

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

NEUMANN, J. von, in **Metal Interfaces**. *American Society for Metals*, Cleveland, p. 108, 1952.