Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

Trabalho de formatura supervisionado
Monografia baseada em: Iniciação Científica

Implementação Paralela de Autômatos Celulares para Tecidos Orgânicos

Aluno: Adroaldo L. Moreira Borges
Supervisor: Prof. Dr. Marco Dimas Gubitoso

22 de fevereiro de 2007

Parte I

Parte Técnica

Sumário

Capítulo 1

Introdução

A Simulação computacional surge como uma abordagem para o estudo e resolução de problemas científicos. Basea-se em uso de computação de alto desempenho ou para modelar e simular sistemas complexos.

Hoje em dia, computadores paralelos de alto desempenho estão possibilitando aos pesquisadores a habilidade para implementar inerentemente técnicas paralelas, tais como autômatos celulares, redes neurais, algoritmos genéticos, ect. Isto é, modelos matemáticos para descrição de fenômenos naturais complexos.

Autômatos Celulares (AC) são modelos espaço-tempo discreto que podem ser usados para simular muitos sistemas complexos no universo. Atualmente, as aplicações de ACs, variam muito. Por exemplo, na biologia, têm sido usados para modelar sistemas, desde níveis de atividades celulares até níveis de clusters de celulas e populações de organismos; na quimica, têm sido usados para modelar cinéticas de sistemas moleculares e crescimento de cristal; na física, têm sido usados para estudar sistemas dinâmicos como diversas interações de partículas e aglomeração de galáxias [07]. Ainda informalmente podemos dizer que autômatos celulares são simulações computacionais que tentam emular os comportamentos de diversos sistemas complexos existentes na natureza, sujeitos a conjunto de regras.
No próximo capitulo daremos uma definição formal de Autômato Celular.

1.1 Breves Histórias

Começaremos a nossa caminhada, com breves histórias do primórdio de autômatos celulares e de computação paralela. Ao mesmo tempo que tentaremos levar ao leitor a motivação para o estudo destes.

Contudo não nos aprofundaremos muito nas histórias destes dois assuntos, pois se o fizéssemos, no final desta seção estariamos escrevendo uma monografia de histórias dos assuntos acima mensionados.

1.1.1 Autômatos Celulares

O conceito de Autômatos Celulares (ACs) foi introduzido pelo John Von Newmann por sugestação do Stanislaw Ulam, para tornar mais real o estudo de comportamento de sistemas complexos, isto por volta de 1948. John Horton Conway introduziu o *Game of Life* (a ser estudado no terceiro capítulo), que fez com que houvesse maior atenção ao assunto ACs. As aplicacões acima mensionadas valem como motivação para o assunto Autômatos Celulares .

1.1.2 Computação Paralela

Em 1955 a IBM introduziu IBM 704, a primeira máquina comercial com hardware em pontos-flutuantes. Esta máquina era capaz de executar aproximadamente 5 kFLOPS. Um ano depois a IBM começou **7030** o projeto (conhecido por STRETCH) para produzir supercomputador para *Los Alamos National Laboratory (LANL).*
O objetivo na época era produzir uma máquina 100 vezes mais rápido que qualquer máquina disponível na referida época. De lá para cá o estudo e desenvolvimento de computadores paralelos de alto desempenho tem aumentado. Salienta-se ainda que a IBM viria a entregar a primeira máquina STRETCH em 1959, sendo que a tecnologia nela implementada foi reutilizada em outros projetos. Nos dias hoje, computadores de alto desempenho são usados para processamento de imagens, previsão de tempo, resolução de alguns problemas da classe NP-completo, etc.

Capítulo 2

Conceitos Básicos

Dedicaremos este capítulo aos conceitos básicos ou seja definiçõs matemáticas e computacionais de autômatos celulares e computação paralela.

2.1 Definições

2.1.1 Autômatos Celulares

Várias definições informais de autômato celular foram dadas na introdução. Agora chegou a altura de passarmos as definições mais concisas.

Conceito Matemático

De uma forma matématica podemos dizer que um autômato celular α é um **quadruplo (G, S, N,** Π**)**, onde

- G = {cell(i,j) : $m, n \in \mathbb{N}$ $0 \leq i \leq m-1$, $0 \leq j \leq n-1$ } é uma grade em $m \times n$ com cell(i,j) representando uma célula na i-ésima linha e j-ésima coluna da grade G.

- $S = \{s_1, s_2, ...s_q\}$ é o conjunto de todos os estados possíveis para uma célula qualquer em G. Representaremos o estado da célula cell(i,j) no instante t por $s^t(i, j)$.

- N(i,j) conjunto de vizinhos da cell(i,j) no instante t. A vizinhança da cell(i,j) é composta de estados das celulas com as quais ela está conectada.

- $\Pi(N(i, j))$ é a regra aplicável ao autômato celular que determina o novo estado $s^{t+1}(i, j)$ da célula cell(i,j) no instante t+1.

Vizinhança

A forma de determinar os vizinhos de uma dada célula na grade, depende do modelo de autômato celular adotado. Algumas vizinhanças clássicas são :

- Vizinhança de **Moore** $\mathcal{VM}(i,j) = \{s^t(k,l) : |k-i| = 1 \text{ e } |l-j| = 1\}$
- Vizinhança de **Moore Extendida**
  $\mathcal{VME} = \{s^t(k,l) : 0 < |k-i| \leq 2 \text{ e } 0 < |l-j| \leq 2\}$
- vizinhança de **Von Neumann** $\mathcal{VN} = \{s^t(k,l) : |k-i| + |l-j| = 1\}$

Figura 2.1: Vizinhança de Moore

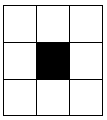

Figura 2.2: Vizinhança de Moore Extendida

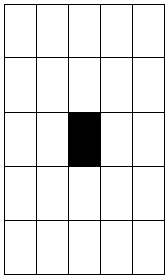

Figura 2.3: Vizinhanças de Von Neumann

Observação
Segundo as definições acima mensionadas, a célula cell(i,j) não pertence a sua própria vizinhança.

Conceito Computacional

Um autômato celular é um programa computacional com as seguintes características :

- O espaço é representado por uma matriz (grade) uniforme. Ou seja a matriz é definida por um tipo de dado específico, exemplo (inteiro, real ou símbolos).
- Cada posição da matriz representa uma célula. Cada uma dessas células contém alguns dados e conecta às outras células seguindo algum padrão.
- As funções são definidas e usadas para alterar o estado de uma posição da matriz no instante t. De acordo com as regras e a vizinhança.
- As funções são (*re*)aplicadas várias vezes, mudando a cada instante t o estado de todas as posições da matriz.
- O tempo t avança discretamente. Ou seja algum modelo discreto descreve o tempo.

  Mais adiante falaremos de alguns autômatos celulares estudados, mas não daremos muita atenção à separação do conceito matemático e o conceito computacional.

2.1.2 Computação Paralela

Não é do nosso interesse aprofundar na abordagem de computação paralela. Mas para que o leitor não fique perdido mais adiante, quando for ler a descrição do algoritmo, sentimos a obrigação de explicar desde já o conceito básico de computação paralela. Ou seja o que vamos abordar neste capítulo são assuntos mais ou menos usados no nosso trabalho.

Arquiteturas Paralelas

A computação paralela não é só a programação paralela. Consiste também em estudo de máquinas paralelas. De um jeito vago podemos dizer que uma máquina paralela é o conjunto de processadores que têm a capacidade de trabalhar de forma conjunta para resolver um determinado problema.

Existem duas classes tradicionais de máquinas paralelas :

- **Máquinas Paralelas com Memórias Compartilhadas** Nesta classe o acesso às unidades de memória é feito através de algum hardware comum. Nesta classe incluem-se todos os computadores com múltiplas unidades processadoras que partilham um espaço de endereços comuns. A arquitectura deste tipo conduz a situações em que um processador, ao aceder à memória comum, pode entrar em conflito (tipo deadlock) com outros processadores. A gravidade destes conflitos depende, no que diz respeito ao hardware, da sofisticação do dispositivo usado para interligar os processadores e as unidades de memória. E no que diz respeito ao software, depende da estratégia usada na implementação de algum aplicativo para o controle de acesso aos recursos compartilhados. Por outro lado, o uso de caches associadas a cada processador permite diminuir o número de conflitos, mas complica o hardware e torna o tempo de acesso aos dados não constante.

- **Máquinas Paralelas com Memórias Distribuidas** Nesta classe, incluem-se as máquinas formadas por várias unidades processadoras, cada uma com a sua memória particular. Denominamos cada uma destas máquinas de nó. Cada uma delas é constituída por processador, memória local e dispositivos de entrada/saída. Neste caso não existe qualquer tipo de memória comum, pelo que a comunicação entre os diferentes nós do multicomputador é feita através de dispositivos dedicados de entrada/saída. Com os progressos realizados nas formas de encaminhamento de dados entre nós, é possível construir máquinas deste tipo com milhares de unidades processadoras. Algumas destas máquinas são denominados por **Beowulf Cluster**.

  Cluster é definido como um sistema onde dois ou mais computadores independentes trabalham de forma conjunta coordenadas por um sistema operacional de natureza paralela, para realizar processamento de tarefas computacionalmente pesados. Para alcançar esses objetivos os processadores dividem as tarefas em partes e comunicam através de troca de mensagens.

  A implementação feita por nós até então, vai de acordo com a estratégia de máquinas desta natureza, ou seja usa a estratégia de máquinas com memórias distribuidas.

Programação Paralela

A tarefa de criar um programa paralelo envolve a identificação das atividades que podem ser executadas em paralelo e escolher o melhor jeito de distribuir essas atividades pelos processadores, bem como os dados que são manipulados. Também é bom planear o acesso aos dados pelos processadores, bem como a sincronição e comunicação entre esses processadores. Sem esquecer as bibliotecas adequadas, por exemplo no nosso caso, estamos usando a MPI.

Modelos Clássicos de Decomposição A decomposição do problema a resolver consiste em dividir a tarefa em atividades que podem ser desempenhadas em paralela. Esta decomposição deve maximizar o número de atividades, sem perder a idéia de execução em paralelo. Assim o tempo de ociosidade dos processadores deve ser mínima. Usando a programação paralela, queremos que a medida que os processadores vão ficando disponíveis decresça de forma proposicional o tempo que falta para o término da resolução do problema. Não conhecemos uma "fórmula fechada"para este assunto, o que sabemos é que depende do problema a ser resolvido. Ou seja de acordo com o problema podemos optar pelo melhor jeito de fazer decomposição.

- **Decomposição de Domínio** A decomposição de domínio consiste em dividir os dados em aproximadamente partes iguais, e então atribuir estas partes aos diferentes processadores. Cada processador só trabalha no conjunto de dados que lhe foi atribuido, mas de acordo com o problema os processadores podem ter a necessidade de se comunicarem periodicamente para troca de dados. Esta técnica é vantajosa quando precisamos manter um único fluxo de controle. Ela consiste em uma sequência de intruções sequenciais aplicáveis aos dados. Mas o paralelismo de dados pode não ser a melhor opção, ou seja pode não ser a solução mais eficiente. Por exemplo quando o conjunto de dados atribuidos a diferentes processadores requerem processamento em tempos diferentes. Então o desempenho do algoritmo fica dependente da velocidade do processador mais lento. Ou seja alguns processadores podem ficar muito tempo ociosos. Neste caso faz mais sentido falarmos da decomposição funcional ou *paralelismo de tarefas*.

- **Decomposição Funcional** Nesta classe de decomposição o problema é dividido em várias tarefas e então atribuídas aos processadores a medida que estes vão ficando disponíveis. A decomposição de tarefas implementa o paradigma *Master-Slave*, onde as tarefas são alocadas aos processadores trabalhadores através de um processador mestre que também pode executar algumas tarefas. Ainda, após a execução das tarefas os processadores trabalhadores enviam o resultado de volta para o processador mestre.

Master-Slave é interessante no caso em que, por limitação do hardware só um dado processador pode executar algumas operações (por exemplo, I/O) e quando não é possível, à partida, determinar o tempo que demora a execução de uma dada sub-tarefa. Neste caso, se entregarmos inicialmente uma sub-tarefa a cada trabalhador, haverá desequilíbrio na carga dos vários processadores. Assim, pode-se usar um modelo de auto-escalonamento (selfscheduling), no qual são os trabalhadores a tomar a iniciativa de pedir mais trabalho quando terminam a sub-tarefa corrente [Medeiros 10].

Capítulo 3

Alguns Autômatos Celulares Estudados

3.1 *Game of Life*

Em 1970, o **J.H. Conway**[1] inventou aquele que é conhecido como o melhor exemplo de autômato celular, *Game of Life* (GoL). Há muitos anos atrás o **J. Von Neumann**[2] tinha inventado um autômato celular(sistema universal) que se auto-reproduz de 29 estados. Após muitos anos de estudo o Conway conseguiu simplificar esse autômato celular para um AC de dois estados. Este autômato celular de dois estados é o famoso *Game of Life*.

Vários autômatos celulares têm derivado de GoL. Para o estudo de aplicações reais podemos usar este autômato celular como modelo base para simulação de câncer em um tecido orgânico.

.

[1]**Jonh Horton Conway** (1937 -) matemático inglês. Atualmente é Professor na Universidade de Princeton. Apesar de ter colaborado em várias áreas da matématica ficou famoso pela invensão de *Game of Life*

[2]**Jonh Von Neumann** (1903-1957) conhecido pelo seu trabalho em teoria dos jogos, autômatos celulares, etc. Contribuiu em muitas áreas da computação : arquitetura, construção de hardware, programação, análise numérica, computação científica, teoria da computação [Aspray 09].

3.1.1 Descrição Matemática e Computacional de *Game of Life*

Matematicamente podemos dizer que *Game of Life* (GoL) é um autômato celular $\Gamma = (G, S, N, \Pi)$, onde :

- G é uma grade de células em $m \times n$.
- S = {0,1}. 0 representa célula morta e 1 representa célula viva.
- Para todas as células, definimos a vizinhança N(i,j) do mesmo jeito. Ou seja, para todas as células usamos a vizinhança de **Moore** para determinar as suas vizinhas.
- $\Pi(N(i,j))$ atualiza o sistema de acordo com as regras de transição :
  - Uma célula cell(i,j) **nasce**, se tiver exatamente três vizinhas vivas. Caso contrário continua morta.
  - Uma célula cell(i,j) **viva**, continua viva se tiver duas ou três vizinhas vivas. Caso contrário morre.

Sejam $\delta = \sum_{r=-1}^{1} \sum_{p=-1}^{1} s^t(i+p, j+r) - s^t(i,j)$ e a função de transição definida do seguinte jeito:

$$s^{t+1}(i,j) = \begin{cases} 1, & \text{se } s^t(i,j) = 1 \text{ e } \delta = 2 \text{ ou } \delta = 3 \\ 1, & \text{se } s^t(i,j) = 0 \text{ e } \delta = 3 \\ 0, & \text{caso contrário.} \end{cases}$$

Computacionalmente podemos implementar o GoL, em uma matriz retangular de dimensão $m \times n$. Cada posição dessa matriz representa uma célula da grade G. Cada posição da matriz pode assumir valor 0 ou 1, num instante t. Onde 0 representa célula morta e 1 presenta célula viva. Em uma implementação simples podemos fazer com que cada iteração represente um instante t discreto. Além da representação habitual dos estados por um número booleano, também podemos representar uma célula por número de vizinhos vivos que ele contém.

3.2 *Greenburg-Hasting*

O fluxo de atividades elétricas em um sistema biológico tal como a atividade que se dá na contração de tecidos cardíacos é uma das formas de ***excitable media***. O conceito de ëxcitação de neorônio¨é caracterizado pelo estado do neorônio, *excitado* ou *em repouso*. Greenburg-Hasting é usado para modelagem de excitação de neorônio.

Excitable media são sistemas extendidos não-equilibrados com estados uniformes que são linearmente estáveis, porém susceptíveis a perturbações finitas[7]. Também *Hodgepodge Machine* e *Cyclic Space* são *excitable media*. *Excitable media* pode ser usado para modelar sistemas biológicos, físicos e químicos tais como as ondas dinâmicas, transmissão em redes neurais, reações químicas oscilantes, etc.

3.2.1 Descrição Matemática de *Greenburg-Hasting*

Matematicamente dizemos que o *Greenburg-Hasting* é um autômato celular Γ = (G, S, N, Π), onde:

- G é uma grade de células em $m \times n$.
- S = {0, 1, $\cdots$, q-1} conjunto dos possíveis estados das células com q $\in \mathbb{Z}$. Célula cell(i,j) no estado 0 é dita excitada enquanto célula cell(i,j) no estado q-1 é dita relaxada. Os demais estados representam o estado de renascimento da célula cell(i,j).
- Para cada célula determinamos a sua vizinhança usando a vizinhança N(i,j) de **Von Neumann**. E um parâmetro μ valor inicial.
- $\Pi(N(i,j))$ atualizamos o sistema de acordo com as seguintes regras de transição :
  - Uma célula cell(i,j) **relaxada** será **excitada** se tiver n vizinhas **excitadas**. Onde n é o parâmetro.
  - Uma célula cell(i,j) no estado **renascida** vai para o próximo estado **renascida**.
  - Todas as demais células permanecem nos seus estados correntes.

Sejam $\delta = \mathcal{C}\{c(k,l) \in \mathcal{VN}(i,j) : s^t(k,l) = 1\}$ e a função de transição é definida por

$$s^{t+1}(i,j) = \begin{cases} q, & \text{se } \delta \geq \mu \\ s^t(i,j) + 1, & \text{se } 1 \leq s^t(i,j) \leq q-2 \\ s^t(i,j), & \text{caso contrário} \end{cases}$$

Onde $\mathcal{C}$ é a função de cardinalidade que devolve o número de elementos de um conjunto. Uma vez que este autômato celular foi matematicamente definido é fácil de implementar . Basta tomar a grade G por uma matriz de $m \times n$, a cada instante t aplicar a regra sobre uma dada célula de acordo com a função de transição. As células podem assumir todos os valores no intervalo discreto de 0 até $q-1$, onde q $\in \mathbb{Z}$. Para uma implementação simples supomos que o instante t é representado por uma iteração sobre toda a matriz.

3.3 *Hodgepodge Machine*

Este *media excitable* foi primeiro concebido por Alan Turing[1] no seu trabalho sobre o estudo de morfogénesis biológico[7]. Hodgepodge machine modela sistemas químicos. Isto faz dele um *media excitable* diferente de outros *media excitable*, por exemplo greenburg-hasting e espaço cíclico.
A reação química frequentemente modelado por este autômato celular é o **BZR** reação de Belousov-Zhabotinsky.

3.3.1 Descrição Matemática de *Hodgepodge Machine*

Hodgepodge Machine é um autômato celular Γ = (G, S, N, Π), onde :

- G é uma grade de células em $m \times m$.
- S = {0, 1, $\cdots$, k-1} conjunto de possíveis estados das células com k $\in \mathbb{Z}$. Uma célula cell(i,j) no estado 0 é dita **saudável** e uma célula cell(i,j) no estado k-1 é dita **doente**. Uma célula cell(i,j) com estado representado no intervalo 1, $\cdots$, k-2 é dita célula a ser **infectada**.
- As células são atualizadas usando a vizinhança N(i,j) de **Von Neumann**.
- $\Pi(N(i,j))$ atualizamos o sistema usando as seguintes regras de transição :
  - Se a célula cell(i,j) está **doente** então o seu próximo estado é ser **saúdavel**.
  - Se Uma célula cell(i,j) é **saúdavel** será **infectada** e os valores que esta célula pode assumir são inteiros determinados por

$$\lambda = \text{Min}\left\{k-1,\ \text{chão}\left[\frac{\varphi}{k_1}\right] + \text{chão}\left[\frac{d}{k_2}\right]\right\}$$

    Onde k_1 e k_2 são constantes de infecção. Note que a função Min é usada para garantir que o estado da célula não excederá k-1. φ representa o número de células vizinhas da célula em observação infectadas e d representa o número de células vizinhas da célula em observação que estão doentes.
  - Se a célula está **infectada**, estará mais **infectada** e os seus valores serão determinados por

$$\lambda^+ = \text{Min}\left\{k-1,\ \text{chão}\left[\frac{\sigma}{\varphi} + \theta\right]\right\}$$

    Onde θ é a taxa de infecção e o σ é a soma dos valores das células vizinhas da célula em observação.

Se dissermos que $\delta = s^t(i, j-1) + s^t(i-1, j) + s^t(i+1, j) + s^t(i, j+1)$ ou $\delta = s^t(i-1, j-1) + s^t(i-1, j+1) + s^t(i+1, j-1) + s^t(i+1, j+1)$ de acordo com a variação da vizinhança de neumann escolhida, a função de transição será definida por

$$s^{t+1}(i,j) = \begin{cases} 0, & \text{se } s^t(i,j) = k-1 \\ \lambda, & \text{se } s^t(i,j) = 0 \\ \lambda^+, & \text{se } s^t(i,j) = p, \text{ onde } p \in \{1, \cdots, k-2\} \end{cases}$$

A implementação deste autômato celular pode ser feita transcrevendo a descrição acima para uma linguagem de programação. Basta representar a grade G por uma matriz quadrada composta de inteiros entre 0 e k-1, onde k é o número de estados no autômato. .

3.4 Espaço cíclico

Espaço cíclico é um *excitable media* como já tinhamos dito. É um autômato celular de transição de fase. Difere do *greenburg-hasting* pelo seguinte fato :
A revitalização das células em *greenburg-hasting* ocorre independentemente do estado da vizinhança, enquanto espaço cíclico permite a revitalização das células somente quando a vizinhança está mais relaxada.

Formalmente *Espaço cíclico* é um autômato celular Γ = (G, S, N, Π), onde :

- G é um ladrilho de células em $m \times m$.
- S = {0, 1, $\cdots$, n-1} conjunto dos estados possíveis, onde n $\in \mathbb{Z}$. Uma célula cell(i,j) no estado 0 é dita **célula a ser excitada**, enquanto uma célula cell(i,j) no estado n-1 é dita **relaxada** e se a célula cell(i,j) estiver nos demais estados, diz-se que a célula se encontra no estado de **revitalização**.
- Para cada célula determinamos a sua vizinhança usando a vizinhança N(i,j) de **Von Neumann**, e um parâmetro θ constante inicial.
- $\Pi(N(i,j))$ atualizamos o sistema com as seguintes regras de transição :
  - Uma célula cell(i,j) no estado n-1 vai para o estado 0, se algum dos seus vizinhos estiver no estado 0.
  - Uma célula cell(i,j) no estado c, $0 \leq c < n-1$ vai para o estado c+1, se tiver algum vizinho no estado c+1.

[1] **Alan Turing**(1912 - 1954) matemático, logicista e criptógrafo. Foi muito dedicado ao estudo da teoria da computabilidade, inteligência artificial e para muitos Turing foi o pai da ciência da computação moderna

– Uma célula cell(i,j) no estado c, $0 \leq c < n-1$ continua no estado c se não tiver algum vizinho no estado c+1.

Sejam $\delta = \mathcal{C}\{c(k,l) \in \mathcal{VN}(i,j) : s^t(k,l) = (s^t(i,j)+1)\}$ e a função de transição é definida por

$$s^{t+1}(i,j) = \begin{cases} s^t(i,j)+1, & \text{se } \delta \geq \theta \\ s^t(i,j), & \text{caso contrário} \end{cases}$$

Onde $\mathcal{C}$ é a função cardinalidade que devolve o número de elementos de um conjunto. Uma vez que este autômato celular foi matematicamente definido é fácil de implementar. Basta tomar a grade G por uma matriz de $m \times m$, a cada instante t aplicar a regra sobre uma dada célula de acordo com a função de transição. As células podem assumir todos os valores no intervalo discreto de 0 até $n-1$, onde n $\in \mathbb{Z}$. Para uma implementação simples supomos que o instante t é representado por uma iteração sobre toda a matriz.

Capítulo 4

Implementações

4.1 Modelo Sequêncial de Game of Life

A simulação sequencial deste autômato celular é bem simples. Basta usarmos a definição matemática dada e assumir também que o tempo seja representado por iteração.

O algoritmo recebe uma matriz $m \times n$ de inteiros 0/1 e o número **itera** de vezes que a simulação será feita. E devolve um novo estado da matriz em geral, e cada célula em particular.

```
Entrada: G de m × n, m, n, itera
Saída: Game of Life em itera iterações
contador ← 0
enquanto contador < itera faça
    para cada i ← 1 até m faça
        para cada j ← 1 até n faça
            Calcula nvizinhos número de vizinhos da célula (i,j) de G ;
            se G(i, j) = 0 então
                se nvizinhos = 3 então
                    G(i, j) ← 1
                fim
                senão
                    G(i, j) ← 0
                fim
            senão
                se nvizinhos = 2 ou nvizinhos = 3 então
                    G(i, j) ← 1
                senão
                    G(i, j) ← 0
                fim
            fim
        fim
    fim
fim
```

Algoritmo 1: Algoritmo Sequencial de *Game of Life*

O algoritmo (alg. 2) informa o número de vizinhos vivos para uma determinada célula da matriz. Implementa a política de Moore para esta determinação de vizinhos. Para tal, recebe uma matriz e a célula cell(i,j) em análise e devolve a solução em **nvizinhos**.

```
Entrada: G de m × n, m, n, i, j
Saída: Um inteiro que represente o número de vizinhas vivas da célula (i,j)
nvizinhos ← 0
para cada k ← i − 1 até i + 1 faça
    para cada l ← j − 1 até j + 1 faça
        se k ≠ l e G(k, l) = 1 então
            nvizinhos ← nvizinhos + 1
        fim
    fim
fim
retorna nvizinhos
```

Algoritmo 2: Cálculo de vizinhos usando vizinhança de **Moore**

4.2 Modelo Paralelo de *Game of Life*

sejam idproc identificador de processador, m número de linhas, n número de colunas, itera número de vezes que simulamos *Game of Life*, numproc número total de processadores alocados e G a matriz. Como usamos a decomposição de domínio, todos os processadores devem receber a mesma quantidade de tarefa, a partida. Para tal implementamos a técnica master-slave.

Entrada: G de $m \times n$, m, n, itera, p, numproc
Saída: *Simulação Paralela de Game of Life*
contador $\leftarrow 0$
se *idproc = 0* **então**
    Lê dados do usuário ;
    se *m mod numproc = 0* **então**
        Distribui m, n, e itera para todos os processadores
        Atribui $\frac{m}{numproc}$ *linhas de G a cada processador idproc*
    senão
        Sai do programa
    fim
fim
Todos os processadores recebem a parte da matriz correspondente.
processador idproc **executa** *o algoritmo sequencial com* $\frac{m}{numproc}$ *linhas, n colunas e itera vezes*
processadores idproc envia a simulação feita para o processador mestre 0
se *p = 0* **então**
    Imprime o resultado
fim

Algoritmo 3: Algoritmo Principal da versão Paralela de *Game of Life*

O processador mestre envia a cada processador um bloco de linhas da matriz. Cada processador recebe o bloco de linhas enviado pelo processador mestre e simula sequencialmente esta parte da matriz original em conjunto com os processadores vizinhos. Pois cada processador deve conhecer as bordas dos seus vizinhos (superior e inferior caso tenha ambos) que interagirão com as suas próprias bordas a fim de obter uma nova configuração. Realça-se o fato de que os processadores 0 e numproc-1 enviam somente última e primeira linha respectivamente, assim como recebem somente primeira e última dos seus vizinhos respectivamente. Para saber como são feitos os cálculos para que o processador idproc receba/envie a sua parte do/ao mestre veja o código.

4.3 Experimentos

Todos os experimentos que aqui apresentamos foram realizados em um computador com processador Intel(R) Pentium (R) 4 CPU 2.66 GHz, com 512 MB de memória RAM e tem linux(Debian) como sistema operacional (em uma das partições).
A tabela 4.1 apresenta os resultados das simulações sequenciais feitas. Fixamos a variável < número de iteração > em 10.000 e 100.000.

Tabela 4.1: Tabela de Simulações Sequenciais

$m \times n$	10.000	100.000
10×5	0,02 s	0,24 s
25×60	0,62 s	6,12 s
100×100	4,03 s	39,68 s
432×101	17,76 s	174,52 s

Tabela 4.2: Tabela de Simulação Paralela - 10.000 Iterações

	Número de Processadores							
$m \times n$	2	3	4	5	6	8	9	10
10×5	0,12 s			0,06 s				
25×60				0,54 s				
100×100	5,35 s		3,18 s	2,50 s				1,78 s
432×101		16,78 s			10,28 s	8,23 s	8,01 s	

Tabela 4.3: Tabela de Simulação Paralela - 100.000 Iterações

	Número de Processadores							
$m \times n$	2	3	4	5	6	8	9	10
10×5	1,07 s			0,56 s				
25×60				5,50 s				
100×100	49,80 s		31,34 s	24,56 s				16,65 s
432×101		165,04 s			101,22 s	81,74 s	82,88 s	

Observação
No experimento realizado, pegamos o tempo de processamento. Assim, no teste da simulação paralela marcamos somente o tempo no processador **"mestre"** zero uma vez que implementamos a estratégia de master-slave.

4.3.1 Análise de Experimentos

Após a execução de testes, chegamos a seguinte conclusão:
Se diminuirmos o tamanho da matriz teremos que aumentar o número de processadores para simulação de aplicação paralela com a finalidade desta ser mais rápida que a simulação sequencial. Mas repare que o aumento de processadores faz sentido somente quando simulamos uma aplicação do tipo câncer em um tecido orgânico, pois na simulação da aplicação tipo *Game of Life* não faz sentido. A menos desta situação, sempre que tivermos uma matriz de dimensão grande faz sentido dar preferência a implementação paralela. A escolha de número de processadores para simulação paralela foi feita aleatoriamente, mas com limitação entre um e dez.

4.4 Ferramentas Usadas

Na implementação da versão sequencial do *Game of Life* usamos a linguagem de programação C-ansi e compilamos com GCC. Para a interface gráfica usamos **X11-Xlib**.

Na implementação paralela de *Game of Life* usamos **LAM/MPI** para comunicação entre os processadores, ou seja, para recepção e envio de dados. Uma vez que este problema não é computacionalmente pesado, não há necessidade de rodar o programa em um computador de natureza paralela, por exemplo um *cluster*. Então usamos os recursos de **LAM/MPI** para criar processadores virtuais em uma máquina de um processador. Ainda para esta versão do problema implementado, usamos a linguagem de programação **c++** e compilamos com a **mpicc** que é um compilador de código c++ para aplicativos paralelos.

4.5 Trabalhos Futuros

O modelo implementado por nós, pode ser extendido para uso em uma aplicação real de simulação paralela de um câncer. Numa futura abordagem, o trabalho pode ser desenvolvido no sentido de implementar o artigo de [Ruanxiaogang 8] usando a programação paralela e algum algoritmo de auto-escalonamento caso for necessário.

4.5.1 Uma idéia de algoritmo de auto-escalonamento

Na simulação paralela de um câncer é natural existir desiquilibrio de carga nos processadores. Ou seja, algum processador pode levar bastante tempo para terminar a sua tarefa em relação aos demais. Isto pode ocorrer se não houver igualdade de distribuição das células que proliferam o câncer em todas as regiões do tecido orgânico. Se assim for, precisaremos de algum algoritmo de auto-escalonamento. Passo a descrever uma idéia de um algoritmo de auto-escalonamento em baixo.

Entrada: intervaloDeColeta
Saída: Nada
se ***tempo*** $\neq 0$ **e** ***tempo*** *mod* ***intervaloDeColeta*** $= 0$ **então**
    processador ***idproc*** *pára*;
    faz o cômputo de ***diferencaDeTempo*** *e* ***numeroDeLinha*** *e envie para o processador 0*;
    se ***idproc*** $= 0$ **então**
        crie uma lista local que receberá todas as informações enviadas pelos processadores;
        listaLocal[*processador*] $\leftarrow$
        (**numeroDeLinha**, **diferencaDeTempo**) ***algoritmoDeDistribuicaoDeCargaPorProcessador****(listaLocal)*;
    fim
fim

Algoritmo 4: Algoritmo de auto-escalonamento

No algoritmo descrito encima o usuário passa o **intervaloDeColeta** que será o momento de parada obrigatória para todos os processadores. Após a parada todos os processadores mandam para o processador mestre (zero), a diferença **diferencaDeTempo** entre instante final e instante inicial de processamento de cada processador e número de linha **numeroDeLinha** da matriz que cada processador contém até então. O processador mestre guarda os dados recebidos em uma lista **listaLocal** na qual a posição **i** armazena os dados do processador **i** . Este ciclo se repete até ao fim da simulação com a particularidade de ser a cada intervaloDeTempo. Também usamos a denominação **idproc** para identificar um processador e a variável **tempo** é o número de iteração até o instante. O algoritmo algoritmoDeDistribuicaoDeCargaPorProcessador(listaLocal) recebe os dados dos processadores e executa um cálculo justo para distribuição de carga entre os processadores.

Referências Bibliográficas

[1] Peter, Pacheco (1996), *Parallel Programming with MPI*. San Francisco California: Kaufmann Publisher.

[2] E.F.Codd, *Cellular Automata*, press academy, New York (1968)

[3] Delfim F.M.Torres *Autómatos Celulares*, www.mat.ua.pt/delfim/artigos/celular.pdf

[4] Conway J. H.e R. K. Guy, *Mathematical Magus*, two year college Math. J. 13(5)(1982) 290-299.

[5] W.Gropp, R.Lusk, R.Ross e R.Thakur *Advanced MPI I/O and one sided communication*

[6] Tommaso, Toffoli and Norman, Margolus (edition age) *Cellular Automata Machines*. MIT Press Series in Scientific Computation, Dennis Gannon editor.

[7] R.P. Saldaña, W.C.Tabares, W.E.S.Yu, *Parallel Implementations of Celullar Automata Algorithms on the AGILA High Performance Computing System*, Ateneo de Manila University, Loyola Heights, Quenzon City.

[8] H. Ruanxiaogang, *A simple Cellular Automaton Model for Tumor-immunity System*, School of Electronic Information and Control Engineering, Beijing Polytechnic University.

[9] Aspray, William (1989), *John von Neumann's Contributions to Computing and Computer Science*. IEEE Annals of the History of Computing, vol. 11, no. 3, pp. 189-195.

[10] Medeiros, Paulo D. (1998). Introdução ao Processamento Paralelo

Parte II

Parte Subjetiva

4.6 Desafios e frustrações

4.6.1 Frustrações

Suponho que quem tem ambição de aprender muito acaba por ter pelo menos alguma frustração. Ou melhor fica sempre aquela sensação de que algo poderia ter ficado melhor.

- A minha iniciação científica é diretamente dependente de computação paralela. No entanto não tive a sorte de fazer MAC0431, ou seja Introdução à Computação Paralela e Distribuida. Porém gostei de ter pesquisado e continuo pesquisando boa parte dos conceitos e ferramentas com as quais trabalho. Mas a velocidade com que o trabalho anda seria maior se tivesse feito a matéria acima mensionada. Não fiz porque durante este ano de 2006 não foi oferecida.

- Não consegui fazer tudo que desejava antes da entrega de monografia.

4.6.2 Desafios

Alguns dos meus desafios são :

- Um desafio simples mas de bom agrado é a recuperação de **BEOWULF/IME**. Este é um cluster (também conhecido por ***tiramisu***) do Instituto da Matemática e Estatística usado normalmente pelo grupo da bioinformática e processamento de imagens.

- Aprofundar mais o meu conhecimento em ciência da computação no sentido de tentar mestrado e se possível ir mais além.

- Bem, o meu país proclamou a independência em 1973. Há poucos anos (três ou quatro) atrás apareceu a primeira Universidade **Universidade Amilcar Cabral**[1]. Esta se encontra ainda na fase experimental. Moralmente tenho compromissos e obrigações para com o meu povo. Para tal terei que levar a cultura imeana para o ensino de computação acolá. Pretendo atuar na área de indústria de software, assim como na vida académica, se possível. Mas salienta-se que estas atividades serão independentes de qualquer entidade política e governamental.

- E por fim espero que um dia haja alunos brasileiros interessados em fazer convênio com a(s) universidade(s) da Guiné-Bissau[2].

[1] Amilcar Cabral (1924-1973) fundador da nacionalidade guineense e caboverdiana

[2] República da Guiné-Bissau situada na costa ocidental da África, com uma superfície de 36.125km^2 aproximadamente e com cerca de 1 milhão e 400 mil habitantes

4.7 Disciplinas cursadas no BCC mais relevantes para o trabalho

Acho que todas as matérias cursadas no curso direta ou indiretamente influenciaram o meu trabalho, mas claro existem aquelas que foram mais relevantes e menos relevantes. Vou citando os mais relevantes :

- **MAC0110 Introdução à Computação**: para mim foi o começo de tudo. Nesta matéria tive o primeiro contato com a programação usando um computador. Antes que isso tinha lido algumas coisas de *pascal*, mas sem entender a lógica. Aliás, mal sabia usar um computador.
- **MAC0122 Princípios de Desenvolvimento de Algoritmos**: a partir desta matéria comecei a pensar como projetar, como implementar, que ferramenta usar para resolver os meus problemas, etc. No projeto que venho fazendo o uso de estruturas de dados e melhores algoritmos têm peso relevantes.
- **MAC021X Laboratórios de Programação**: nessas matérias comecei uma fase de programação para problemas de tamanhos razoáveis. Isso fez-me preparar o trabalho de um jeito organizado.
- **MAC0239 Métodos Formais em Programação**: muito do que tenho feito em computação tem a influência desta matéria. Digamos é a lógica e também a técnica de demonstração de corretude de programas.
- **MAC0300 Métodos Numéricos da Álgebra Linear**: a matéria trata vários assuntos estudados em computação paralela. Talvez estudarei esses assuntos no futuro. Só por isso essa matéria é um dos motivos para o trabalho que venho fazendo.
- **MAC0316 Conceitos Fundamentais de Linguagens de Programação**: essa matéria é de muita importância para ter alguma visão de funcionamento de compiladores e linguagens de programação. Uma vez que tive que estudar como os processadores se comunicam, essa matéria deu alguma coragem para enfrentar o assunto.
- **MAC0338 Análise de Algoritmos**: estudo de melhores algoritmos, a sua complexidade e comparação de algoritmos.
- **MAC0412 Organização de Computadores**: esta matéria foi a principal motivação para estudar a programação paralela. Pois nesta matéria estudamos arquiteturas dos computadores, microprogramação, etc.

- **MAC0414 Linguagens Formais e Autômatos**: deu-me a noção de autômatos, das linguagens de programação, das sintaxes e das semânticas.
- **MAC0438 Programação Concorrente**: noções teóricos e práticos de programação *multithreads*.

4.8 Interação com o orientador(supervisor)

Quando fui falar com o Professor Gubi, sobre a iniciação científica e trabalho de formatura supervisionado, ele de imediato respondeu :
" - me procura na semana que vem, me cobra". Isso deixou-me tranquilo e com confiança de que ia fazer algo de bom. Uma vez que não tinha noção de programação paralela ele sempre me ajudou a entender a semântica e a sintaxe dos exemplos vistos e sempre me incentivou. Tinhamos reuniões semanalmente para explicá-lo como anda o trabalho. Por tudo isso, posso dizer que a interação tem sido ótima.

4.9 Agradecimentos

Este trabalho com certeza é o maior feito acadêmico executado por mim. Por isso não posso deixar de agradecer a Deus em primeiro lugar, à minha família pelos apoios em todos os sentidos, ao meu orientador, ao meu tutor pelos conselhos, à todos os meus professores durante o curso, aos meus amigos e colegas do bcc e não só, e por fim à todos os meus amigos espalhados pelo mundo que me têm apoiado.