Simulação Computacional Paralela Baseada em Autômatos Celulares: Estudo de Caso de Espalhamento de Epidemias

Clarissa Dreischerf Pereira
cdp13@inf.ufpr.br

Universidade Federal do Paraná - UFPR
Departamento de Informática
CI 316 Programação Paralela – Fabiano Silva

Resumo
O delineamento de comportamentos coletivos permitem o conhecimento sobre tendências em uma população, caracterizando sistemas complexos. Esses sistemas podem ser entendidos através de Autômatos Celulares. Esse trabalho trata sobre um estudo de caso de espalhamento de epidemias, caracterizado como um sistema complexo, através da utilização de Autômatos Celulares. Fazendo uso da API OpenMP, visa analisar o aceleramento de execução desses casos, para auxiliar na geração de estudos mais coerentes e eficientes.

Palavras-chave*: Autômatos Celulares. Espalhamento de Epidemias. Paralela.*

Abstract
The outlining of collective behaviors allow knowledge about trends in a population, featuring complex systems. These systems can be understood through Cellular Automata. This work deals with a case study of spreading epidemics, characterized as a complex system, through the use of cellular automata. Making use of the OpenMP API aims to analyze the accelerated execution of these cases, to assist in generating more coherent and efficient studies.

Keywords: *Cellular Automaton. Epidemic Spreading. Parallel.*

Introdução

Os autômatos celulares são sistemas dinâmicos discretos capazes de representar sistemas e fenômenos capturando uma grande variedade de comportamentos. Os ACs auxiliaram e auxiliam o trabalho com sistemas complexos permitindo uma perspectiva mais completa do todo, devido as suas características de relação, sendo capaz de representar interações entre indivíduos com um comportamento coletivo associado. Essas relações são desenvolvidas a partir dos pontos essenciais de um AC: os autômatos celulares são grupos de células (vetores/matrizes) em que cada célula possui um estado; os estados mudam/permanecem em relação a um estado anterior, alterando através das regras locais definidas, as quais dependem dos valores das células vizinhas. [1] [2]

Para melhor esclarecimento, podemos dizer que sistemas complexos são ditos quando se há um grande número de agentes interagindo entre si, aparentemente independentes, e a riqueza dessas interações muitas vezes permite que o sistema como um todo seja auto organizado. [3]

Uma das aplicações dos Autômatos celulares são o espalhamento de epidemias. Esse trabalho utilizou como base regras simples que simulam a propagação de uma doença

genérica entre os indivíduos de uma população por meio de ACs, proposto por Gledson Melotti [3]. O trabalho de Melotti utilizou como principal referência o modelo epidemiológico SIR (suscetível-infectado-recuperado), onde os ACs são tratados principalmente com três estados. Porém, foram feitos alguns acréscimos, como a ideia de deslocamento de indivíduos e vacinação pulsada. O presente trabalho não irá tratar todas as regras propostas pelo artigo de Melotti, mas obterá a sua base em grande parte desse. [3]

1 Modelo para Simulação de Espalhamento de Epidemias

Um dos principais modelos de interesse aos pesquisadores da área é o modelo matemático SIR (Suscetível-Infectado-Recuperado) desenvolvido por Kermack e McKendrick (1927). Este modelo representa a propagação de doenças infecciosas e possibilita o estudo da disseminação de uma doença em uma população. [3]

Tem-se para esses indivíduos as seguintes definições: Suscetíveis são indivíduos que não estão infectados, e podem contrair a doença através de contatos com infectado; Infectados são indivíduos que estão com a doença e podem transmití-la para outros indivíduos; Recuperados são indivíduos que passaram pela doença e não são mais nem suscetíveis nem infectados. Considera-se que a cura confere imunidade. Além disso considera-se que o tamanho da população é constante, ou seja, o número de morte é igual ao número de nascimento.

Para tal, foram tratadas algumas constantes, aplicadas as fórmulas do algoritmo desenvolvido. β é o coeficiente de transmissão que determina a taxa em que novas infecções surgem como consequência do contato entre suscetíveis e infectados, μ é a taxa de novos suscetíveis por unidade de tempo, γ significa a taxa de recuperação dos indivíduos infectados e d é a taxa de indivíduos infectados que morrem por causa da doença.

Através do estudo de doenças infecciosas em diferentes populações, pode-se realizar uma melhor avaliação da importância e eficácia de campanhas de controle. Essas campanhas podem caracterizam a prevenção ou vacinação de uma dada faixa da população, introduzindo assim o conceito de que a vacinação de indivíduos suscetíveis possa auxiliar no controle e possível erradicação de uma certa doença. Dessa forma, acrescenta-se ao modelo SIR a caracterização de vacinação pulsada.

No modelo trabalhado, cada célula pode assumir quatro estados: suscetível, infectado, recuperado ou vacinado. Ele trabalha a vizinhança (*grid*) de Moore, onde são trabalhadas com as oito células ao redor de uma célula central (uma grade quadrada). Não será trabalhado com o deslocamento de indivíduos. Cada célula possuirá três informações: estado, tempo que está vivo (normal) e tempo que está infectado.

Definindo as regras gerais de probabilidade:

- Todos os indivíduos (S, I, R) têm uma probabilidade de morrer que não seja causada pela doença.

- Todos os indivíduos S têm uma probabilidade de serem infectados de acordo com $\beta \times v / vt$. Onde v é a quantidade de vizinhos infectados e vt é o número total de vizinhos.

- Cada indivíduo infectado, I, tem uma probabilidade de tornar-se curada e uma probabilidade de tornar-se morta por causa da doença.

- Para cada indivíduo que morre um suscetível nasce em seu lugar. Portanto, a população permanece constante.

- Um indivídio não pode morrer por causa da doença no próximo instante após adquirir a doença.

- Um indivíduo infectado não pode viver por mais de um tempo delimitado com a doença.

2 Proposta de Implementação Paralela

Como os computadores comerciais estão cada vez mais possuindo processadores com multinúcleos (*multicore*), APIs para programas paralelos estão sendo mais utilizadas. O OpenMP (Open Multi-Processing) é uma API (Application Program Interface) que suporta programação paralela para memória compartilhada em múltiplas plataformas. A utilização do OpenMP tem crescido nos últimos anos, uma vez que as suas funcionalidades facilitam, de forma mais simplificada, o desenvolvimento de aplicações em memória compartilhada. [5]

Para esse trabalho será utilizado o OpenMP para realizar a paralelização do problema. Como a inicialização da matriz se dá por leitura de dados, será utilizada e analisada a paralelização no processamento das células. A sequência do algoritmo se dá por: inicialização da matriz inicial; aplicação das regras sobre todas as células da matriz, revezando as matrizes para preservação dos estados; o programa rodará até a quantidade máximas de iterações estipulado na chamada do programa.

A escolha da arquitetura para realizar os testes do programa foi feita pela quantidade de processadores. No caso, foi optado por uma máquina com 16 processadores iguais (definida abaixo).

3 Estudo Experimental

Foi utilizado para a realização dos testes a servidora do Departamento de Informática da UFPR – cohiba. Essa possui 16 processadores Quad-Core AMD Opteron(tm) Processor 8387, com uma thread cada núcleo, 2799.996MHz, 512 KB *cache size*, com arquitetura x86_64, 64-bit.

Foram realizados testes anteriores para a verificação da melhor paralelização para o algoritmo desenvolvido. Essa se deu principalmente pela alteração dos laços da chamada principal, na main.c, entre linhas e colunas, verificação de paralelização na vacina pulsada e pelo teste de *clause* no laço definido como mais adequado. Essas verificações tiveram como valores fixos em: 8 threads, tamanho de matriz fixo em 200x200, por 200 iterações em 10 execuções de cada. Após análise por média e desvio padrão, foram sendo delimitados o melhor em cada caso, sendo o melhor encontrado utilizado para o restante dos testes.

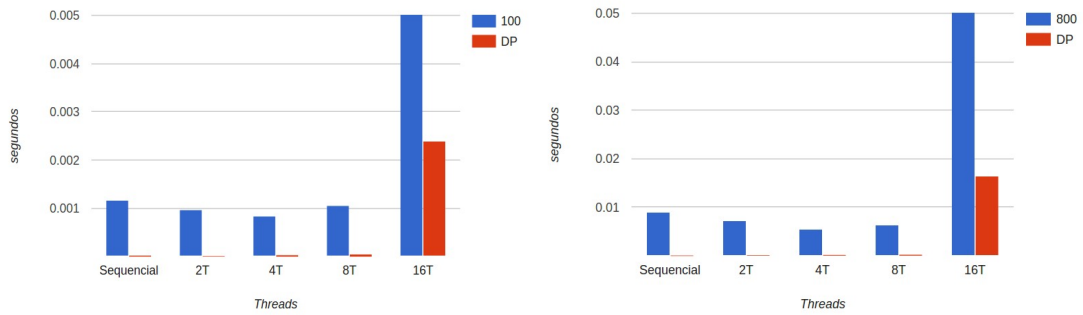
Para os testes escolheu-se quatro matrizes de tamanhos diferentes, fixadas em: 10x10, 100x100, 300x300, 600x600 e 1000x1000. Para cada tamanho de matriz houve variação no número de Threads, entre 2, 4, 8 e 16, e para cada uma dessas houve ainda a variação do número de iterações, de 100, 200, 400 e 800. Para cada resultado foi aplicado o cálculo da média e desvio padrão. Essa forma foi escolhida para que houvesse maior estabilidade numérica, além de avaliar a correlação entre o número de iterações e a convergência dos resultados. Cada experimento foi rodado 10 vezes.

Em relação as constantes definidas para cada possível caso de epidemia, ficaram fixas em: $\beta = 3.5$; $\mu = 0.10$; $\gamma = 0.60$; e $d = 0.30$. Essas são descritas na execução do programa com parâmetro '?' ou 'help' (./cellularAutomaton ? ou ./cellularAutomaton help).

3.1 Resultados

Os dados foram analisados e os gráficos foram trabalhados divididos por tamanho da matriz, apresentando para cada uma a média por quantidade de threads e o seu correspondente desvio padrão. Serão apresentados os gráficos mais significativos de cada matriz.

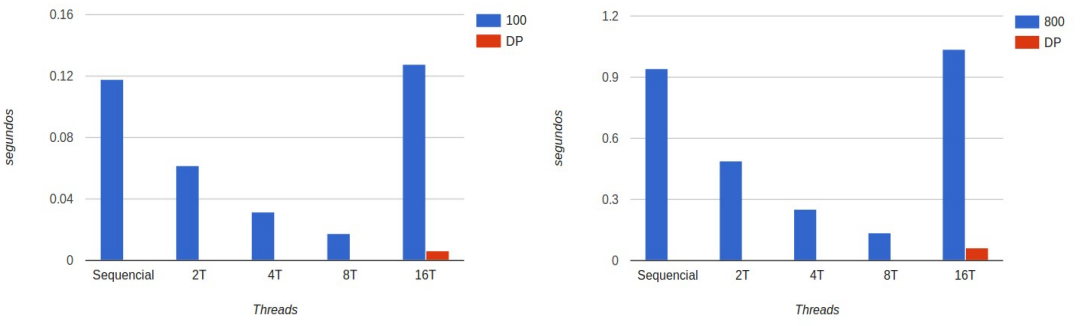
A Figura 1 (a) e (b) apresentam o resultado para a matriz 10x10 de 100 e 800 iterações respectivamente. Sendo que o tamanho do limite mínimo e máximo dos segundos foram alterados para que a visualização dos dados ficasse mais correta. Na Figura 1(a) os segundos para 16 threads obteve media de 0,126255 segundos, com desvio padrão de 0,002388, enquanto a Figura 1(b) obteve 0,998359 de média e 0,016454 de desvio padrão. Houve um pequeno ganho até 4 threads e um aumento extremamente grande com 16 threads. Isso provavelmente ocorre devido ao custo de criação das threads não compensar o tamanho e tempo para essa matriz.



(a) **(b)**

Figura 1 – Matriz 10x10 : (a) 100 e (b) 800 iterações

A Figura 2 (a) e (b) apresentam o resultado para a matriz de 100x100 de 100 e 800 iterações respectivamente. Os tamanhos não foram alterados. Pode-se obervar que houve um ganho até 8 threads e para 16 threads ainda não houve ganho, mas sim um aumento. Provavelmente deriva do mesmo motivo relatado no exemplo anterior: o tamanho e tempo para essa matriz ainda não compensa o custo de criação das threads (criadas a cada linha da matriz).



(a) **(b)**

Figura 2 – Matriz 100x100 : (a) 100 e (b) 800 iterações

Para melhor visualizar se está tendo o ganho esperado, a Figura 3 apresenta o *SpeedUp* para a matriz de 100x100. Como pode-se obeservar abaixo o ganho está muito próximo ao máximo esperado, principalmente para 2 e 4 threads, até 16 threads, quando há um declíneo significativo. A média do *SpeedUp* ficou em: 1,917751813 para 2T; 3,652643824 para 4T; 6,750864783 para 8T; e 0,8973715044 para 16T.

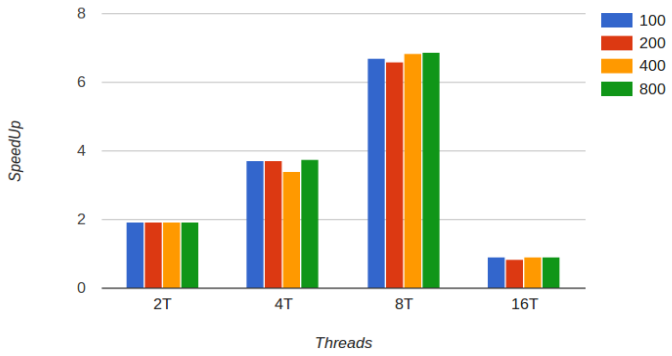
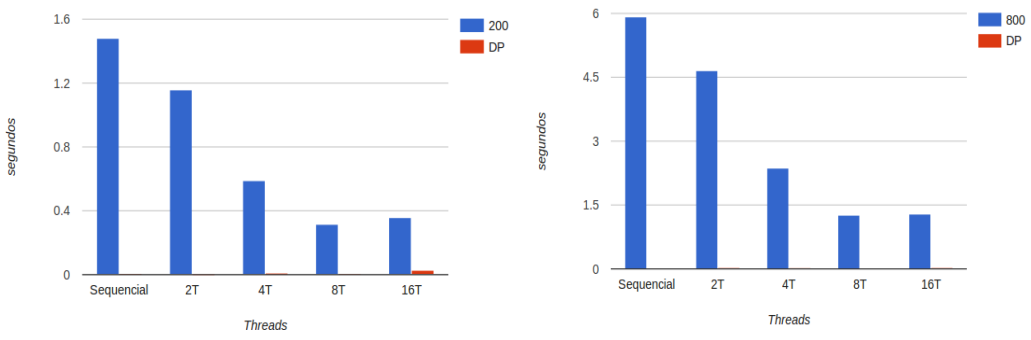


Figura 3 – *SpeedUp* da Matriz 100x100

A Figura 4 (a) e (b) apresentam os resultados para a matriz de 300x300 com 200 e 800 interações respectivamente. Foi optado pelo gráfico de 200 iterações devido ao maior desvio pardrão, para 16 threads, que o gráfico de 100 iterações apresentou, sendo a disponibilizada aqui considerada mais relevante. Pode-se observar que ainda não o tamanho e tempo ainda não compensou completamente para a criação de 16 threads, mas já houve uma diminuição considerável em relação aos testes anteriores, tornando-a ainda promissora para testes futuros.

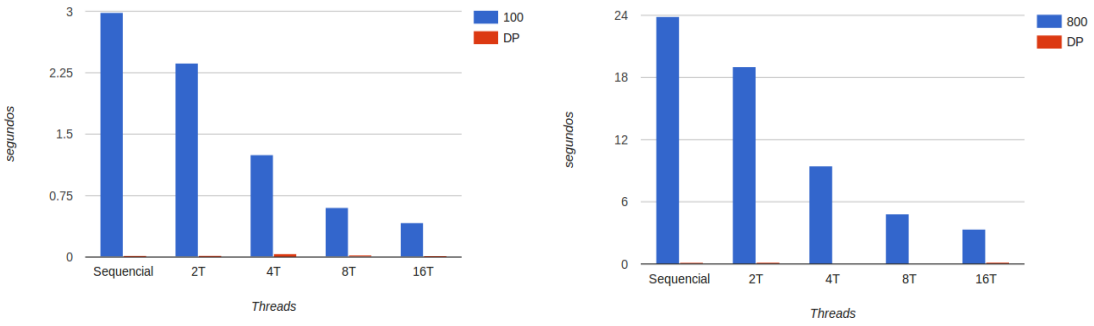
O aumento do número de iterações influenciou, a partir da expansão do tempo de execução do programa, para uma aproximação do tempo para 16 threads e 8 threads.



(a) **(b)**

Figura 4 – Matriz 300x300 : (a) 200 e (b) 800 iterações

A Figura 5 (a) e (b) apresentam os resultados para a matriz de 600x600 para 100 e 800 iterações respectivamente. Esse foi o teste que a utilização de 16 threads começou a apresentar um resultado positivo.



(a) **(b)**

Figura 5 – Matriz 600x600 : (a) 100 e (b) 800 iterações

Apesar de visualmente ter tido um bom resultando, analisando o *SpeedUp* (Figura 6) verificamos que esse está bem abaixo do que poderia ser atingido, mantendo uma faixa muito próxima a metado do esperado. Porém podemos confirmar com esse gráfico a melhora que a utilização de 16 threads finalmente começou a apresentar, comprovando que o tamanho e tempo para as matrizes anteriores não valiam a criação das threads. A média do *SpeedUp* ficou em: 1.257499003 para 2T; 2.419601916 para 4T; 4.929470675 para 8T; e 7.136166991 para 16T.

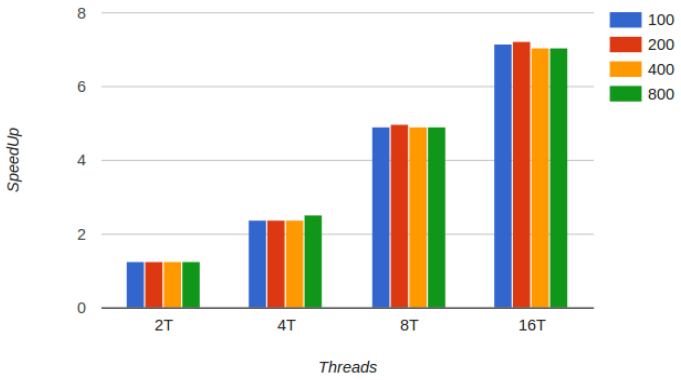


Figura 6 – *SpeedUp* da Matriz 600x600

O resultado da matriz de 1000x1000 obteve uma resolução muito próxima a matriz de 600x600 em questão de *SpeedUp* (Tabela 1).

		Sequencial	**2T**	**4T**	**8T**	**16T**
	100	8.234784889	6.622371173	3.334087062	1.674599004	1.193719554
DP		0.02741801067	0.03276661915	0.01866662561	0.01617313432	0.01204773506
	200	16.45258188	13.24491544	6.646593714	3.345320845	2.37352736
DP		0.06465489198	0.04594504379	0.02998786862	0.01554405524	0.01920958957
	400	33.01374531	26.5522491	13.32829323	6.702990842	4.769429731
DP		0.09615518462	0.1105880287	0.0731128904	0.0293339265	0.02362458452
	800	65.89220269	53.06596968	26.67822514	13.42140465	9.542526627
DP		0.2342655865	0.1811876639	0.1250625329	0.05464296229	0.06173931258
SpeedUp			**1.242678643**	**2.473017908**	**4.917566755**	**6.914296218**

Tabela 1 – Apresentação dos testes para Matriz 1000x1000 e o *SpeedUp*

3.2 Análise dos Resultados

Foi observado que nas matrizes de tamanho até 300x300 não houve uma boa utilização e melhora por parte do aumento do número de threads. Como foi considerado, o aparente motivo para o programa não melhorar com um maior número de threads para essas matrizes pequenas é que o tamanho da matriz, bem como o seu tempo de execução, não compensaram o tempo gasto para criação das threads. Com isso, podemos verificar que conforme a matriz aumenta, acresce também o número de threads que essa possui um melhor resultado. Por exemplo, enquanto a resolução excelente para a Matrix 10x10 foi com 4 threads, a Matriz 100x100 obteve o melhor pico com 8 threads.

Outro ponto averiguado é a redução do *SpeedUp* a partir de um certo tamanho da Matriz. Isso pode ser derivado do acréscimo de falha de acesso à memória cache, devido ao aumento da matriz e volume de dados maior, prejudicando diretamente o desempenho. Dessa forma, o paralelismo diminui seu melhoramento.

Considerações Finais

Esse trabalho visava realizar a análise dos ganhos do uso de paralelismo, explorando diferentes testes, em um estudo de caso de espalhamento de epidemias. Para isso utilizou-se a API OpenMP para trabalhar paralelismo em memória compartilhada. Observou-se um ganho no desempenho através do paralelismo, obtendo-se um *SpeedUp* quase ótimo até um tamanho máximo da Matriz. A partir de um ponto, verificou-se que o *SpeedUp* decai, porém ainda mantém um melhoramento razoável em relação ao sequencial. Com isso foi possível analisar a melhor utilização e forma de paralelismo, mostrando possíveis futuras aplicações para o campo de espalhamento de epidemias.

Referências

[1] SILVA, A.R,MARTINS, C.A.S.P, JUNIOR, M.M.G. **Simulação Computacional Paralela Baseada em Autômatos Celulares: Estudo de Caso em Simulação da Dinâmica de Nuvens**. Anais do EATI. Pontifícia Universidade Católica de Minas Gerais (PUC Minas). Disponível em: <http://www.eati.info/eati/2013/assets/anais/artigo163.pdf>. Acesso em: 02 mai. 2015.

[2] THAULOW, L.V. **A Study and Comparision of First and Second Order Cellular Automata with Examples.** NTNU, June 2010. Disponível em: <http://www.diva-portal.org/smash/get/diva2:351877/FULLTEXT01.pdf>. Acesso em: 03. mai. 2015.

[3] MELOTTI, G. **Aplicação de Autômatos Celulares em Sistemas Complexos: Um Estudo de Caso de Espalhamento de Epidemias.** Universidade Federal de Minas Gerais, Belo Horizonte, 12 de fevereiro de 2009. Disponível em: <http://www.ppgee.ufmg.br/defesas/335M.PDF>. Acesso em: 03. mai. 2015.

[4] CASTRO, A.P, LIMA, D.A. **Autômatos celulares aplicados à modelagem de dinâmica populacional em situação de risco.** Instituto Federal do Triângulo Mineiro. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/wcama/2013/009.pdf>. Acesso em: 05. jun. 2015.

[5] CHAPMAN, B. JOST, G. PAS, R.V.D. **Using OpenMP – Portable shared memory parallel programming.** Scientific and engineering computation series, 2008, Massachusetts Institute of Technology. Disponível em: <http://lib.mdp.ac.id/ebook/Karya%20Umum/Portable_Shared_Memory_Parallel_Programming.pdf>. Acesso em: 07. mai. 2015.