Estudo de Desempenho de Duas Estratégias Paralelas Aplicadas ao Ajuste de Parâmetros de um Modelo Matemático da Esclerose Múltipla

Gustavo G. Silva[1], **Matheus A. Moreira**[1,2], **Bárbara M. Quintela**[1,2], **Marcelo Lobosco**[1,2]

[1]Departamento de Ciência da Computação/ Instituto de Ciências Exatas e Laboratório de Fisiologia Computacional (FISIOCOMP) – Universidade Federal de Juiz de Fora (UFJF) Juiz de Fora – MG – Brazil

[2]Programa de Pós-graduação em Modelagem Computacional (PPGMC) Universidade Federal de Juiz de Fora (UFJF) Juiz de Fora – MG – Brazil

Corresponding author: barbara.quintela@ufjf.br

Abstract. *Multiple Sclerosis (MS) is a intricate neurological ailment comprehensible via mathematical-computational models. Similar to all models, one must adjust its parameters precisely to represent experimental outcomes. The optimization technique Auto-Adaptive Differential Evolution can serve this purpose. However, these adjustments entail substantial computational costs, which parallel computing can help alleviate. This article introduces a study of two strategies, namely OpenMP and CUDA, utilized to parallelize the parameter fitting of an MS model.*

Resumo. *A Esclerose Múltipla (EM) é uma intrincada doença neurológica que pode ser melhor compreendida por meio de modelos matemáticos-computacionais. Como todo modelo, seus parâmetros precisam ser ajustados para representar adequadamente resultados experimentais. A Evolução Diferencial Auto-Adaptativa é uma técnica de otimização que pode ser empregada para este propósito. Tais ajustes contudo possuem alto custo computacional, que pode ser reduzido com o empregado computação paralela. Este artigo apresenta um estudo de duas estratégias, OpenMP e CUDA, utilizadas para paralelizar o ajuste de parâmetros de um modelo EM.*

1. Introdução

A esclerose múltipla (EM) é uma doença neurológica crônica e autoimune que afeta o sistema nervoso central (SNC). Caracterizada por uma ampla variedade de sintomas e manifestações clínicas, a EM representa um desafio clínico significativo para pacientes, profissionais de saúde e pesquisadores em todo o mundo. Embora sua etiologia exata permaneça incompletamente compreendida, estudos recentes têm avançado consideravelmente em nossa compreensão das bases patogênicas da doença, abrindo novas perspectivas para abordagens terapêuticas mais eficazes [Rodríguez Murúa et al. 2022].

Com intuito de capturar os fenômenos complexos que ocorrem em seu desenvolvimento e progressão, diversos modelos matemáticos foram desenvolvidos ao longo

dos anos. Um dos principais enfoques em modelos matemáticos da EM é a dinâmica da resposta imunológica. O sistema imunológico desempenha um papel crucial no desencadeamento da doença, com ataques autoimunes direcionados à mielina, a proteína que reveste os neurônios. Modelos baseados em equações diferenciais, por exemplo, têm sido aplicados para simular as dinâmicas da EM [Lombardo, M.C., et al 2017, Moise and Friedman 2021].

Nesse contexto, um novo modelo para representar as dinâmicas espaciais e temporais da esclerose múltipla em duas dimensões foi apresentado [de Paula et al. 2023]. A execução desses modelos matemáticos possui um alto custo computacional, em especial em contextos onde são necessárias múltiplas execuções do mesmo modelo, como em análises de sensibilidade [de Paula et al. 2023] e em métodos de computação evolucionista para ajustes de seus parâmetros. Para lidar com esses altos custos computacionais, múltiplas estratégias podem ser empregadas para viabilizar a execução desses modelos em tempo adequado. Dentro desse contexto, uma das melhores abordagens é o emprego de técnicas de programação paralela.

O principal objetivo do presente trabalho é apresentar e analisar os resultados do emprego de duas diferentes estratégias de programação paralela, OpenMP e CUDA, para acelerar a execução do modelo matemático em dois compartimentos que representa a EM, de modo a reduzir os custos associados ao ajuste de seus parâmetros com o método de Evolução Diferencial Auto-Adaptativa.

2. Métodos

2.1. Modelo Matemático

Com o objetivo de representar as dinâmicas temporais e espaciais da resposta imune que ocorrem durante a EM, foi desenvolvido um modelo com dois compartimentos [de Paula et al. 2023], baseado em Equações Diferencias Ordinárias (EDOs) e Parciais (EDPs), que representam os principais atores associados à doença. Em um compartimento que representa os linfonodos, um conjunto de seis EDOs representa o comportamento temporal das Células Dendríticas Ativadas, Células T CD8, Células T CD4+, Células B, Plasmócitos e Anticorpos. Um segundo compartimento, que representa o tecido cerebral, emprega seis EDPs para representar as dinâmicas espaciais e temporais da Microglia, Oligodendrócitos, Células Dendríticas Convencionais, Células Dendríticas Ativadas, Anticorpos e Células T CD8+. Esse modelo possui um total de 34 parâmetros utilizados para descrever o comportamento da doença. Mais detalhes sobre a doença, o modelo e a sua implementação sequencial estão disponíveis na literatura [de Paula et al. 2023].

2.2. Evolução Diferencial Auto-Adaptativa

A Evolução Diferencial (*Differential Evolution* ou DE) [Storn and Price 1997] é uma estratégia de otimização global frequentemente utilizada para resolver problemas de otimização em diversas áreas, como engenharia, ciência dos dados e aprendizado de máquina. A Evolução Diferencial Auto-Adaptativa (EDAA), do inglês *Self-Adaptive Differential Evolution* (SADE), é uma extensão da DE que visa melhorar a capacidade de busca e adaptação do algoritmo original [Brest et al. 2007]. A ideia principal por trás dessa abordagem é permitir que os parâmetros da DE sejam adaptados dinamicamente

durante o processo de otimização, de modo a se ajustarem às características específicas do problema em questão.

Na EDAA, cada indivíduo da população possui seu próprio conjunto de parâmetros, que podem ser ajustados durante o processo de otimização. A adaptação dos parâmetros é feita por meio da inclusão de genes extras nos indivíduos, que representam os parâmetros de adaptação. A atualização dos parâmetros é realizada de acordo com regras de adaptação definidas. Essas regras podem ser baseadas em estratégias como "*one-fifth success rule*" ou "*self-adaptive control parameters*". No primeiro caso, os parâmetros são aumentados ou diminuídos dependendo do sucesso das soluções em cada iteração. Se uma solução melhorar em relação à geração anterior, os parâmetros são reduzidos. Caso contrário, os parâmetros são aumentados para incentivar maior exploração do espaço de busca. Na segunda abordagem, os parâmetros são atualizados com base em estatísticas coletadas das soluções ao longo das iterações, visando adaptar-se ao comportamento dinâmico do problema.

2.3. Implementação Paralela

Foi verificado que a maior parcela do tempo de execução sequencial é gasta com a resolução através do método das diferenças finitas do sistema de EDPs. Pode-se facilmente paralelizar a porção de código que resolve este sistema, visto que o mesmo conjunto de equações é simultaneamente aplicado nos múltiplos pontos que compõe o domínio simulado. Operações de sincronização, por outro lado, são necessárias a cada passo de tempo, visto que os resultados obtidos em um passo de tempo são utilizados para os cálculos no passo subsequente. As características da aplicação e do hardware disponível levaram a escolha de OpenMP e CUDA para a implementação paralela, que será brevemente descrita nas próximas subseções.

O algoritmo de otimização com uso da EDAA abre a possibilidade de se empregar ainda o paralelismo de tarefas, o que pode ser especialmente interessante na versão de código que usa CUDA para paralelizar a resolução do sistema de EDPs. Neste cenário específico adotou-se uma abordagem de computação heterogênea, que tira proveito tanto da GPU quanto da CPU para executar simultaneamente o modelo e a EDAA. Nesse caso, enquanto a GPU está calculando as EDPs, as outras etapas da EDAA, como a seleção, cruzamento e mutação, podem ser paralelizadas na CPU.

2.3.1. OpenMP

A implementação paralela com OpenMP (OMP) é iniciada com a criação de um time de *threads* para executar as tarefas em paralelo antes do início do laço temporal, empregando para isso a primitiva `#pragma omp parallel`. Dentro deste laço encontram-se a resolução dos sistemas de EDOs e de EDPs. Deve-se ressaltar que o sistema de EDOs, que representa o compartimento do linfonodo, é resolvido de forma sequencial. Os dados gerados pela resolução das EDOs é usado então na computação em paralelo das EDPs. Um segundo laço é empregado para percorrer todos os pontos do domínio espacial. Neste momento utiliza-se uma diretiva do OpenMP (`#pragma omp for`) para dividir o domínio espacial em fatias, atribuindo-as a uma *thread*. Nesse processo, é utilizado o escalonamento estático padrão, pois todos os pontos da malha requerem a mesma quantidade de trabalho para sua computação. Ao final, calculam-se as concentrações médias

das populações de células no tecido, que são então usadas para resolver o sistema de EDOs no próximo passo de tempo. Justifica-se a separação entre as diretivas `parallel` e `for` para reduzir os custos adicionais (*overheads*) decorrentes da criação e destruição das *threads*, que sem esta separação seriam feitas a cada passo de tempo no início do laço espacial. Com a separação este custo adicional é pago apenas uma vez, no início do laço temporal.

2.3.2. CUDA

A implementação da versão paralela com CUDA exige inicialmente a alocação e cópia dos dados para a memória da GPU: os parâmetros do modelo e o tamanho da discretização espacial são gravados na memória de constantes, enquanto os vetores que representam as populações descritas pelas EDPs ficam na memória global. As funções `cudaMemcpy` e `cudaMemcpyToSymbol` são respectivamente usadas para este fim. Novamente o sistema de EDOs é resolvido de forma sequencial pela CPU, sendo posteriormente seus resultados copiados para a GPU. A cada passo de tempo o *kernel* que resolve numericamente o sistema de EDPs do tecido é invocado. Após a execução do *kernel*, aplica-se a redução em algumas das populações do tecido, que serão utilizadas no próximo passo de tempo para a resolução das EDOs. Contudo, para melhorar o desempenho sem impactar na corretude dos resultados numéricos, a implementação emprega diferentes passos de tempo na solução das EDOs e das EDPs: o passo de tempo (Δt) empregado na solução das EDOs é 100 vezes maior do que o utilizado nas EDPs, ou seja, a operação de redução só é realizada a cada 100 passos de tempo para a transmissão de valores gerados pelas EDPs para as EDOs.

3. Resultados e Discussão

3.1. Ambiente Computacional e Versão Sequencial

O ambiente computacional utilizado foi um computador com quatro CPUs AMD EPYC 7713, cada uma com 64 núcleos físicos. O processador possui 32KB de *cache* L1 de instruções, 32KB de *cache* L1 de instruções, 512KB de *cache* L2 unificada, e 32768KB de cache L3 compartilhada entre todos os núcleos, e 528GB de memória principal. O computador possui ainda 2 GPUS NVIDIA A100, sendo que apenas uma foi utilizada durante os experimento executados nesse trabalho. O sistema executa Rocky Linux 8.6 (*kernel* $4.18.0 - 477.15.1$). O compilador utilizado para compilar o código C foi o GCC versão 12.2.0 e o compilador para CUDA foi o NVCC versão 12.2. Nos programas C a *flag* de otimização `-Ofast` foi utilizada, enquanto nos programas CUDA utilizou-se a *flag* `-O3`. O `Valgrind` 3.19 foi utilizado para coletar informações de execução da versão OpenMP do código. Este trabalho utilizou a EDAA implementada na biblioteca Pagmo [Biscani and Izzo 2020]. Essa implementação utiliza OpenMP nativamente para implementar uma versão paralela do código em CPU.

Para execução dos códigos sequenciais e paralelos, foram consideradas as mesmas condições iniciais e constantes utilizadas no artigo que descreve o modelo [de Paula et al. 2023], exceto pelos valores das discretizações temporais e espaciais: adotou-se $\Delta t = 2 \times 10^{-4}$ e $\Delta x = 5 \times 10^{-2}$. Todas as versões foram executadas no

mínimo 10 vezes ou até que o intervalo de confiança dos tempos de execução fosse superior à 95%. O tempo médio de execução sequencial observado foi em média igual à $453,6$ segundos e desvio padrão de $18,41$.

3.2. Versões Paralelas

Para a execução com OpenMP, foram feitas duas rodadas de execução, com e sem a variável de ambiente `OMP_PROC_BIND` ativada. Esta variável de ambiente, quando ativa, habilita a afinidade de processador durante a execução das *threads* do OpenMP, isto é, as *threads* sempre serão escalonadas para execução nos mesmos *cores*. Além disso, a variável de ambiente `OMP_PLACES` também foi usada para determinar a ordem de alocação dos processadores. Como a máquina possui dois processadores, a ideia é estudar o impacto da alocação na taxa de falhas da *cache* L3, visto que quando não usada o SO pode optar por alocar um grande conjunto de *threads* no mesmo processador, ainda que *cores* estejam disponíveis em outro processador. O desequilíbrio na distribuição das *threads* entre os processadores poderia gerar um impacto negativo na taxa de falhas da *cache* L3 compartilhada. A Tabela 1 apresenta os resultados.

Tabela 1. Comparação de tempos de execução do código OpenMP com e sem afinidade de processador. S/A representa afinidade de processador desligada e ✓ indica afinidade ligada.

Número de Threads	Afinidade	Média (s)	Mediana (s)	Desvio padrão
1	S/A	$488,2$	$470,9$	$45,8$
2	S/A	$281,3$	$257,2$	$48,1$
	✓	$263,2$	$250,1$	$36,8$
4	S/A	$271,3$	$265,2$	$9,3$
	✓	$269,1$	$257,4$	$31,0$
8	S/A	$146,7$	$141,2$	$15,8$
	✓	$145,9$	$141,1$	$14,7$
16	S/A	$82,1$	$79,1$	$9,1$
	✓	$71,4$	$67,6$	$10,0$
32	S/A	$53,8$	$52,0$	$5,9$
	✓	$36,4$	$35,0$	$4,4$
64	S/A	$32,9$	$32,2$	$4,5$
	✓	$21,4$	$20,6$	$2,5$
128	S/A	$28,9$	$26,9$	$4,5$
	✓	$29,0$	$28,5$	$1,3$

Com o objetivo de se verificar o impacto da quantidade de *threads* e da afinidade de processador na hierarquia de memória, foi utilizada a ferramenta `cachegrind` em conjunto com a callgrind annotate (`cg_annotate`), ambas utilitárias do `Valgrind`. A Tabela 2 apresenta os resultados obtidos.

Em relação à implementação CUDA, foram analisadas três versões do código. A primeira versão é uma implementação direta da versão sequencial (CSMC). A segunda versão faz uso da memória de constantes para armazenar as constantes do código (CCMC), que na primeira versão ficavam localizadas na memória global. A terceira versão utiliza *streams* para sobrepor a execução de computação e comunicação entre a CPU e a GPU (CCSP). A Tabela 3 apresenta os resultados obtidos.

Tabela 2. Análise do comportamento da hierarquia de memória para diferentes configurações de execução. Novamente S/A representa afinidade de processador desligada. L1I representa a taxa de falhas da *cache* L1 de instruções; L1D representa a taxa de falhas de *cache* L1 de dados; L2 representa a taxa de falhas na *cache* L2.

Ambiente	16 S/A	16	32 S/A	32	64 S/A	64
L1I	$5,9\%$	$5,3\%$	$3,0\%$	$2,7\%$	$3,1\%$	$2,7\%$
L1D	$99,3\%$	$73,6\%$	$87,2\%$	$80,2\%$	$87,0\%$	$67,5\%$
L2	$78,5\%$	$70,1\%$	$81,3\%$	$77,4\%$	$91,4\%$	$89,3\%$

Tabela 3. Tabela de tempos para execução do código CUDA utilizando-se de memória de constantes e sobreposição de computação e comunicação.

Configuração	Média (s)	Mediana (s)	Desvio padrão
CSMC	$10,7$	$10,7$	$0,10$
CCMC	$11,84$	$11,83$	$0,04$
CCSP	$10,58$	$10,53$	$0,10$

Para verificar como as versões paralelas em CUDA e OpenMP se comportam com o aumento no número de passos de tempo, aumentou-se em 20 vezes a discretização temporal, ou seja, utilizou-se $\Delta t = 4 \times 10^{-5}$. A Tabela 4 apresenta os tempos de execução para CUDA e OpenMP. No caso da versão CUDA, apresentam-se adicionalmente os tempo de cópia dos dados entre memória entre a memória principal e a memória global.

Tabela 4. Tempos de execução dos códigos OpenMP (OMP), com 64 e 128 *threads*, e CUDA. Na tabela HTD representa o tempo de cópia de dados entre a memória principal e a memória global, e DTH representa o tempo de cópia de dados entre a memória global e a memória principal.

Threads	Afinidade	Média	Mediana	Desvio padrão	HTD	DTH
OMP/64	S/A	$371,4$	$344,0$	$64,2$	-	-
	✓	$239,2$	$226,2$	$30,0$	-	-
OMP/128	S/A	$330,9$	$326,3$	$41,9$	-	-
	✓	$314,0$	$306,4$	$18,8$	-	-
CSMC	-	$36,7$	$36,7$	$0,07$	$0,25$	$0,39$
CCMC	-	$53,0$	$53,0$	$0,06$	$0,10$	$0,39$
CCSP	-	$56,3$	$56,3$	$0,14$	$9,93$	$38,4$

Por fim, avaliou-se o impacto do emprego de diferentes discretizações de tempo (Δt) entre as EDOs e as EDPs, bem como a qualidade dos resultados numéricos após o emprego dessa técnica. Foram realizados experimentos considerando o emprego de um mesmo passo de tempo, e diferenças de 50 ou de 100 vezes entre os passos. A Tabela 5 apresenta os resultados para todas as versões CUDA. Apenas as versões CUDA foram avaliadas porque são as únicas que necessitam de operações de cópia de dados, visto que as EDPs são resolvidas na GPU, enquanto as EDOs são resolvidas na CPU, e os resultados das resoluções precisam ser trocados para a correta execução do algoritmo.

Tabela 5. Tempos de execução com o emprego de diferentes saltos de tempo entre a resolução das EDOs e EDPs.

Salto de Tempo	Configuração	Média	Mediana	Desvio padrão
Sem salto	CSMC	$104,42$	$104,55$	$0,64$
	CCMC	$120,81$	$120,8$	$0,38$
	CCCA	$107,74$	$107,66$	$0,20$
50	CSMC	$37,53$	$37,52$	$0,15$
	CCMC	$53,85$	$53,81$	$0,08$
	CCCA	$56,51$	$56,70$	$0,24$
100	CSMC	$36,74$	$36,73$	$0,07$
	CCMC	$53,07$	$53,09$	$0,06$
	CCCA	$56,31$	$56,30$	$0,14$

3.3. EDAA

Considerando que o melhor desempenho foi obtido pela versão CUDA (Tabela 4), esta versão foi escolhida para realizar o ajuste dos parâmetros do modelo. Em particular, ajustou-se o parâmetro α (que interfere na homeostase das células T CD8) e β (que interfere na ativação das células T CD8) a um conjunto de dados experimentais [Song, Zi-Ye et al 2014]. A EDAA obteve ao fim do processo de ajuste $\alpha = 0,00251$ e $\beta = 0,319$. Com relação ao tempo de execução para realizar o ajuste, o tempo médio obtido para 10 execuções foi de $714,4$ segundos, com desvio padrão de $9,7$ segundos.

3.4. Discussão

Comparando-se o tempo de execução sequencial com a versão OpenMP com uma *thread*, observa-se um *overhead* imposto pela implementação paralela de cerca de $7,6\%$. Pode-se também perceber que as acelerações obtidas pela versão OpenMP foram aumentando até a configuração com 64 *threads* (*speedup* de 21 vezes com o emprego de afinidade), e depois começaram a cair. Para a maioria das configurações, observa-se uma leve melhoria de desempenho quando se ativa a afinidade de processador (Tabela 1). Essa melhoria é explicada pela significativa redução na taxa de falhas na *cache*, em especial na L1 de dados, conforme observa-se na Tabela 2. A diferença no emprego da afinidade de processador é acentuada quando uma maior discretização temporal é adotada, como pode ser observado na Tabela 4: ganhos de até $1,5$ vezes são obtidos quando a afinidade é usada na versão com 64 processadores.

Em relação às versões CUDA, observa-se inicialmente um grande ganho de desempenho quando comparada a versão sequencial (43 vezes) e ao melhor tempo da versão OpenMP (cerca de duas vezes). O emprego de memória de constantes e da sobreposição de computação e comunicação não tiveram grande impacto na configuração base de execução, e surpreendentemente se saíram piores quando uma maior discretização temporal foi adotada. A maior melhoria foi obtida quando passos de tempo distintos foram adotados durante a resolução das EDOs e EDPs, sendo a melhor melhoria observada entre as versões sem salto e com salto de 50 passos de tempo. Melhorias marginais foram observadas entre as versões que adotaram 50 e 100 saltos de tempo, O *profiling* da execução, feito com a ferramenta `nsight`, indicou que essa técnica reduziu em 10 vezes o tempo médio de cópia entre CPU e GPU. Também constatou-se que essa estratégia não causou

grande impacto na precisão numérica dos resultados, causando um impacto de menos de 3% nos resultados avaliados.

4. Conclusão e Trabalhos Futuros

Este trabalho apresentou duas estratégias, OpenMP e CUDA, para implementação paralela de um modelo matemático para representar a evolução temporal e espacial da esclerose múltipla. O desempenho de cada estratégia implementada foi avaliada, considerando diferentes versões do código e configurações de execução. Os resultados indicaram que a melhor estratégia foi a implementação em CUDA, que obteve ganhos de desempenho de até 43 vezes. Esses resultados levaram a escolha de CUDA para ser utilizada no ajuste dos parâmetros do modelo com Evolução Diferencial Auto-Adaptativa. Durante o ajuste de parâmetros, o modelo a ser ajustado é executado em CUDA, enquanto a evolução diferencial é paralelizada com OpenMP.

Apesar dos bons resultados obtidos, ainda há espaço para maiores reduções no tempo de computação. Este trabalho apresentou a implementação de duas técnicas que visavam reduzir o *overhead* na transferência de dados e os custos de acesso à memória pela GPU. Entretanto, outras técnicas podem ser avaliadas assim como uso de múltiplas GPUs também será objeto de implementações futuras.

Referências

Biscani, F. and Izzo, D. (2020). A parallel global multiobjective framework for optimization: pagmo. *Journal of Open Source Software*, 5(53):2338.

Brest, J., Greiner, S., Bošković, B., Mernik, M., and Zumer, V. (2007). Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *Evolutionary Computation, IEEE Transactions on*, 10:646 – 657.

de Paula, M. A. M., de Melo Quintela, B., and Lobosco, M. (2023). On the use of a coupled mathematical model for understanding the dynamics of multiple sclerosis. *Journal of Computational and Applied Mathematics*, 428:115163.

de Paula, M. A. M., Silva, G. G., Lobosco, M., and Quintela, B. M. (2023). Sensitivity analysis of a two-compartmental differential equation mathematical model of ms using parallel programming. In *Computational Science – ICCS 2023: 23rd International Conference, Proceedings, Part II*, page 714–721, Berlin, Heidelberg. Springer-Verlag.

Lombardo, M.C., et al (2017). Demyelination patterns in a mathematical model of multiple sclerosis. *J. Math. Biol.*, 75(2):373–417.

Moise, N. and Friedman, A. (2021). A mathematical model of the multiple sclerosis plaque. *J. Theor. Biol.*, 512:110532.

Rodríguez Murúa, S., Farez, M. F., and Quintana, F. J. (2022). The Immune Response in Multiple Sclerosis. *Annu Rev Pathol*, 17(1):121–139.

Song, Zi-Ye et al (2014). Peripheral blood T cell dynamics predict relapse in multiple sclerosis patients on fingolimod. *PLoS One*, 10(4):e0124923.

Storn, R. and Price, K. (1997). Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359.