PARALELISMO: ANALISE VIA MÉTODO DE METROPOLIS

ARTHUR KRINDGES[1];
CARLOS ALBERTO VAZ DE MORAIS[2]; FABIO M. ZIMMER[3]

[1] *Universidade Federal de Pelotas – arthur.krindges@ufpel.edu.br*
[2] *Universidade Federal de Pelotas – cavmjunior@ufpel.edu.br*
[3] *Universidade Federal de Mato Grosso do Sul – fabio.zimmer@ufms.br*

1. INTRODUÇÃO

Na atualidade, a busca por respostas mais rápidas e precisas para problemas complexos constitui uma das principais preocupações em diversas áreas do conhecimento. Na física, essa necessidade se mostra particularmente crítica, uma vez que o desenvolvimento de teorias, métodos computacionais e novos materiais demanda simulações cada vez mais sofisticadas e computacionalmente eficientes. Para enfrentar esses desafios, o paralelismo computacional tem se consolidado como ferramenta fundamental, permitindo não apenas a aceleração de cálculos intensivos, mas também a modelagem de sistemas físicos com precisão sem precedentes (ROMERO et al., 2020). O recente avanço da computação paralela em massa, impulsionado em parte pelo desenvolvimento de inteligências artificiais, torna o domínio dessas tecnologias e a capacidade de extrair seu máximo potencial elementos cruciais para o progresso científico (NOORDEN; PERKEL, 2023).

Neste contexto, este trabalho tem como objetivo analisar comparativamente o desempenho de um algoritmo de Metropolis Monte Carlo (MMC) (CHIB; AND, 1995) aplicado ao modelo de Blume-Capel, utilizando três abordagens distintas: o método serial tradicional e as implementações paralelizadas em CPU e GPU. O estudo busca elucidar as nuances específicas de cada método, apresentando uma análise detalhada de suas vantagens, limitações e eficiência computacional, com ênfase nos ganhos de desempenho obtidos por meio das diferentes estratégias de paralelização.

2. METODOLOGIA

Para realizar tal tarefa utilizaremos o modelo Blume-Capel, um modelo do tipo Ising com spin $S = 1$, na presença de um campo de cristal (D) e campo externo (H). O hamiltoniano deste modelo é dado por

$$E = -J \sum_{\langle ij \rangle} S_i S_j + D \sum_i S_i^2 - H \sum_i S_i, \tag{1}$$

onde J é a interação de troca, tomaremos $J < 0$ de forma ao sistema priorizar um estado anti-ferromagnético (AFM). O sub-índice $\langle ij \rangle$ do somatório indica soma sobre primeiros vizinhos. Além disso, é conhecido que neste modelo para uma rede

2D tem as seguintes fases:

$$\text{AFM} = \begin{pmatrix} \uparrow & \downarrow \\ \downarrow & \uparrow \end{pmatrix} \quad , \quad \text{NOM} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \tag{2}$$

$$\text{MFM} = \begin{pmatrix} \uparrow & 0 \\ 0 & \uparrow \end{pmatrix} \quad \text{e} \quad \text{POL} = \begin{pmatrix} \uparrow & \uparrow \\ \uparrow & \uparrow \end{pmatrix}, \tag{3}$$

onde as setas denotam os estado $\uparrow = +1$, $\downarrow = -1$ e 0 (GUERRERO; STARIOLO, 2019).

Para simular este sistema, utilizamos o algoritmo MMC, que parte da suposição de que o sistema se encontra em equilíbrio térmico, e que a probabilidade de ocorrência de cada configuração segue a distribuição de Boltzmann ($P(\{s_i\}) \propto e^{-\beta E(\{s_i\})}$, $\beta^{-1} = k_B T$ onde T é a temperatura), com isso o algoritmo consistem em:

1. Iniciar o sistema em um configuração $\{s_i\}$;
2. Escolher um sítio na rede (s_j) e calcular:
   (a) Determinar a variação da energia ΔE quando $s_j \to s'_j$.
   - Caso $\Delta E < 0$ aceite o novo estado e vai para próxima etapa;
   - Caso $r < e^{-\beta \Delta E}$, sendo $r \in [0, 1)$, aceite o novo estado e vai para próxima etapa;
   - Caso contrário rejeite o novo estado.
3. Repita 2 e retire os valores médios.

Com o algoritmo de Metropolis devidamente explicado, apresentamos agora as três abordagens computacionais distintas para sua implementação e execução:

1. **CPU serial**: implementação tradicional do algoritmo MMC, onde a simulação é executada sequencialmente utilizando apenas um núcleo de processamento;
2. **CPU paralelo**: utiliza de MMC mas todos os núcleos fazem simultaneamente a etapa 2 utilizando distintos s_j;
3. **GPU**: similar ao CPU paralelo mas utiliza de processamento multi-paralelo, com troca de memória entre CPU-GPU.

Rede							
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Sub-Rede A			
1	3	5	7
10	12	14	16
17	19	21	23
26	28	30	32
33	35	37	39
42	44	46	48
49	51	53	55
58	60	62	64

Sub-Rede B			
2	4	6	8
9	11	13	15
18	20	22	24
25	27	29	31
34	36	38	40
41	43	45	47
50	52	54	56
57	59	61	63

Figura 1: A esquerda a rede 8×8, ao centro a sub-rede A e a direita a sub-rede B.

Hardwares (núcleo/*threads*)	**Compiladores**	**Abreviação**
Core I5 7300HQ (4/4)	gfortran (gnu)	I5
Core I5 7300HQ + GTX 1050	nvfortran (Nvidia)	GTX
Ryzen 5 4600g (6/12)	gfortran (gnu)	R5
Ryzen 5 4600g + RX 6600	hipfort (AMD-ROCm/HIP)	RX

Tabela 1: Tabela de *hardwares*, compiladores e abreviações.

O grande impasse está na paralelização da etapa 2 do algoritmo MMC. A principal questão que surge é: será que os sítios escolhidos simultaneamente não interferem entre si, ou seja, será que as atualizações paralelas não afetam mutuamente os valores de ΔE? Para contornar essa questão o sistema é dividido em sub-redes, onde todos os sítios de cada sub-rede não interagem entre si. Para o caso da rede quadrada obtemos duas sub-redes (A e B) conforme a Figura 1. Desta forma, podemos manipular todos os elementos da sub-rede A ao mesmo tempo e posteriormente todos os elementos da sub-rede B.

Para comparação dos métodos os cálculos utilizaram os *hardwares* e compiladores conforme a Tabela 1. O algoritmo de MMC foi aplicado a uma rede 1024×1024 com 20.000 ciclos por sítio e com intervalos de 10 ciclos entre amostragens.

3. RESULTADOS E DISCUSSÃO

Com a metodologia previamente discutida, foi possível obter resultados relacionados à validação do método, à comparação com dados da literatura, e à avaliação e comparação entre diferentes abordagens metodológicas e arquiteturas de *hardware*.

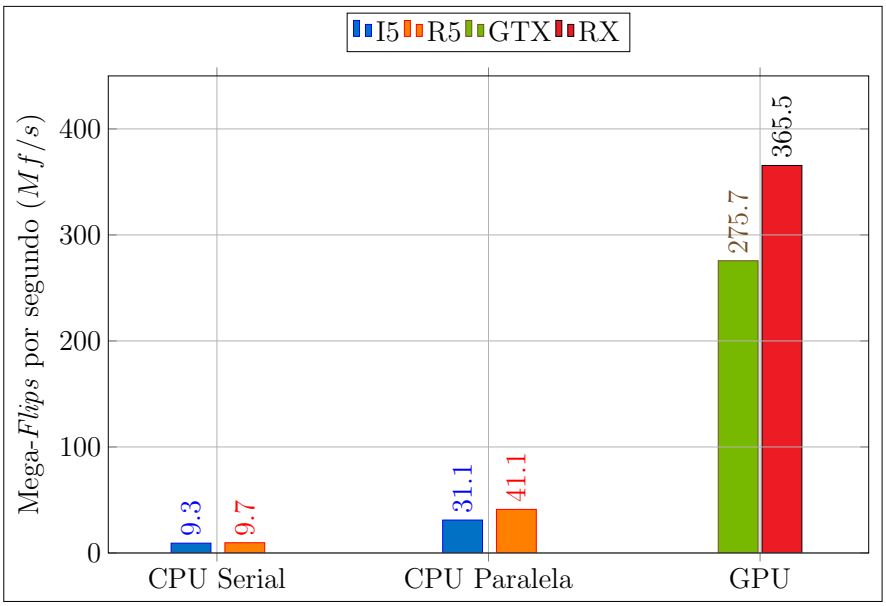

Figura 2: Número de Mega-*flips* por segundo em função da metodologia e *hardware*.

O principal resultado que obtivemos pode ser visto na Figura 2, onde temos o número de Mega-*flips* por segundo (Mf/s) para cada metodologia e *hardware*. Começando com os resultado para CPU Serial, podemos observar que o R5 foi pouco

melhor em comparação com I5, em função da arquitetura ser mais nova e o *clock* ser mais alto. Essa diferença entre I5 e R5 aumenta mais quando olhando para o método CPU Paralela, pois o R5 apresenta mais *threads* que o I5, o que favorece um desempenho significativamente superior nessa abordagem. Entretanto, ao analisarmos os resultados obtidos com GPU, observamos um aumento significativo no número de Mf/s: tanto a GTX quanto a RX apresentaram desempenho aproximadamente 9 vezes superior se comparado com CPU Paralela. Essa discrepância se deve ao número exorbitante de *threads* disponíveis nas GPUs, que permite o processamento simultâneo de uma quantidade de dados muito maior do que nas CPUs. O maior desempenho da RX em relação à GTX se deve à sua arquitetura mais recente - Navi (2021), em comparação com a arquitetura Pascal (2016) da GTX - além de contar com maior poder de processamento bruto.

4. CONCLUSÕES

Neste trabalho, apresentamos o modelo de Blume-Capel em uma rede bidimensional (2D), aplicado ao algoritmo de Metropolis Monte Carlo (MMC). Descrevemos a metodologia por trás do MMC e exploramos diferentes possibilidades de implementação: CPU serial, CPU paralela e GPU. Detalhamos a estratégia de decomposição da rede necessária para a paralelização do algoritmo e realizamos uma comparação entre as abordagens em diferentes tipos de *hardware*. Observamos que a implementação utilizando GPU superou o desempenho das versões em CPU.

Apesar dos benefícios, a implementação de métodos de paralelismo pode apresentar alguns desafios. O primeiro é identificar corretamente as partes do código que podem ser paralelizadas sem causar conflitos. O segundo diz respeito ao *hardware*: cada GPU exigiu um compilador específico (como `nvfortran` para NVIDIA e `hipfort` para AMD), dificultando a portabilidade e o uso de diferentes plataformas. O terceiro obstáculo é a curva de aprendizado: embora os programas tenham sido escritos em *Fortran*, o paralelismo em CPU utilizou a API *OpenMP*, enquanto nas GPUs foram empregadas *CUDA* (GTX) e *HIP-Fortran* (RX), esta última exigindo também o uso de *C++* para sub-rotinas. Ainda assim, o esforço de paralelização resultou em ganhos de desempenho significativos, o que pode ser essencial para cientistas e pesquisadores — como no caso do desenvolvimento de inteligência artificial.

REFERÊNCIAS

CHIB, S.; AND, E. G. Understanding the metropolis-hastings algorithm. **The American Statistician**, v. 49, n. 4, p. 327–335, 1995.

GUERRERO, A. I.; STARIOLO, D. A. The blume-capel model in a square lattice with $J_x = -J_y$ interactions in an external field. **Physica A: Statistical Mechanics and its Applications**, v. 532, p. 121839, 2019.

NOORDEN, R. V.; PERKEL, J. Ai and science: what 1,600 researchers think. **Nature**, v. 621, p. 672–675, 2023.

ROMERO, J. et al. High performance implementations of the 2D Ising model on GPUs. **Computer Physics Communications**, v. 256, p. 107473, 2020.