Relatório Parcial 1 - OpenMP

Ícaro Travain Darwich da Rocha[1], **Heitor G. B. Reis**[1], **João Victor Gama Dantas**[1]

[1]Instituto de Ciência e Tecnologia (UNIFESP), São José dos Campos, Brasil

{icaro.travain, h.reis, dantas.joao}@unifesp.br

Resumo. *Este trabalho avalia a paralelização de um algoritmo sequencial utilizando OpenMP para melhorar o desempenho em grandes conjuntos de dados. Foram paralelizadas funções críticas e medidas as execuções com diferentes entradas e números de threads. Os resultados mostram que, para dados pequenos, o speedup é desprezível devido ao overhead, enquanto para grandes entradas o tempo de execução foi reduzido significativamente. A análise dos valores intermediários de SSE confirma que os resultados paralelos e sequenciais são equivalentes, garantindo a correção da paralelização.*

1. Introdução

A crescente demanda por desempenho computacional tem impulsionado o desenvolvimento de técnicas de paralelização, que permitem a execução simultânea de múltiplas tarefas, otimizando o uso de recursos e reduzindo o tempo de processamento. A computação paralela é fundamental para resolver problemas complexos e de grande escala, como simulações científicas, análise de big data e aprendizado de máquina.

OpenMP é um acrônimo para Open Multi-Processing, que é uma API para desenvolvimento de programas em arquiteturas de memória compartilhada. Foi desenvolvido em 1997 por um consórcio liderado pela Intel e pelo Departamento de Energia dos Estados Unidos. O seu principal objetivo é facilitar a paralelização de programas em linguagens como C, C++ e Fortran. [Cornell 2023].

2. Metodologia

O trabalho foi executado em uma máquina com as seguintes configurações:

- Processador: Intel® Core™ i5-1235U
- Número de núcleos: 10
- Total de threads: 12
- Frequência do processador: 3.3 GHz
- Cache: 12 MB
- Memória RAM: 32 GB DDR4 3200 MHz

Os testes foram realizados utilizando o programa em Python *gerador_gray_16.py*. Nele, é possível escolher o número total de pontos gerados, o número de centróides e as faixas que servirão como base para a geração dos dados.

Para automatizar a execução do programa, foi desenvolvido um script que realiza as seguintes etapas:

1. Compila os arquivos em C do código sequencial;
2. Executa o código 10 vezes para três entradas diferentes: 100.000 números, 1.000.000 números e 10.000.000 números;

3. Compila o código paralelizado com OpenMP;
4. Executa o código cinco vezes para cada uma das entradas (pequena, média e grande) utilizando uma thread;
5. Repete o processo para os seguintes números de threads: 1, 2, 4, 6, 8, 10, 12, 16, 18, 20, 28, 36 e 44;
6. Salva os dados no arquivo *resultados.csv*;
7. Executa o código em Python *graficos.py*, o qual adiciona os *speedups* aos resultados e gera os gráficos de *speedup*, tempos de execução e valores intermediários de SSE.

Cabe ressaltar que os programas foram modificados para salvar os valores intermediários de SSE nas versões sequencial e paralelizada com quatro threads.

3. Desenvolvimento

A primeira alteração ocorreu na função responsável pelo cálculo do tempo de execução. Em vez da função `clock()`, que mede o tempo total somado de todas as threads, foi utilizada a função `omp_get_wtime()`, da biblioteca `time.h`, a qual calcula apenas o tempo decorrido (tempo de parede) da execução.

Para paralelizar a função `assignment_step_1d`, foi inserida a diretiva `#pragma omp parallel for`, em conjunto com a cláusula `reduction(sse)`, para realizar a soma dos valores parciais de forma paralela.

Já a função `update_step_1d` precisou ser reescrita para evitar condições de corrida. Ao invés de utilizar um único par de arrays `sum` e `cnt`, foram introduzidos arrays locais `sum_thread` e `cnt_thread`. Cada thread trabalha com seus próprios acumuladores `my_sum` e `my_cnt`, que passam por uma etapa final de redução, na qual os resultados parciais são somados para calcular os centróides.

4. Resultados e Discussão

A primeira análise importante refere-se ao resultado final. Para isso, foi necessário adaptar os programas para imprimir os valores intermediários de SSE. O gráfico 1 ilustra essa convergência. Para os três conjuntos de dados de entrada, o SSE paralelizado com quatro threads sobrepõe-se ao resultado sequencial, indicando fortemente que os resultados parciais são equivalentes.

O gráfico 2 mostra como o aumento da carga de trabalho impacta o *speedup* das execuções.

Um primeiro aspecto a ser destacado é que, mesmo utilizando apenas uma thread, o *speedup* é maior em todos os casos. Isso demonstra que as modificações realizadas no programa, mesmo sem paralelização, já provocaram uma redução significativa no tempo de execução.

Para uma carga de trabalho de 100.000 entradas, percebeu-se que o aumento do número de threads resulta em uma diminuição da eficiência do código. A principal explicação para essa queda é o *overhead* gerado pelo gerenciamento do paralelismo. À medida que a carga de trabalho aumenta, a proporção do *overhead* em relação ao tempo total diminui. Por exemplo, para uma entrada de 10.000.000 de números, o *speedup* chega a aproximadamente 3,5 vezes.

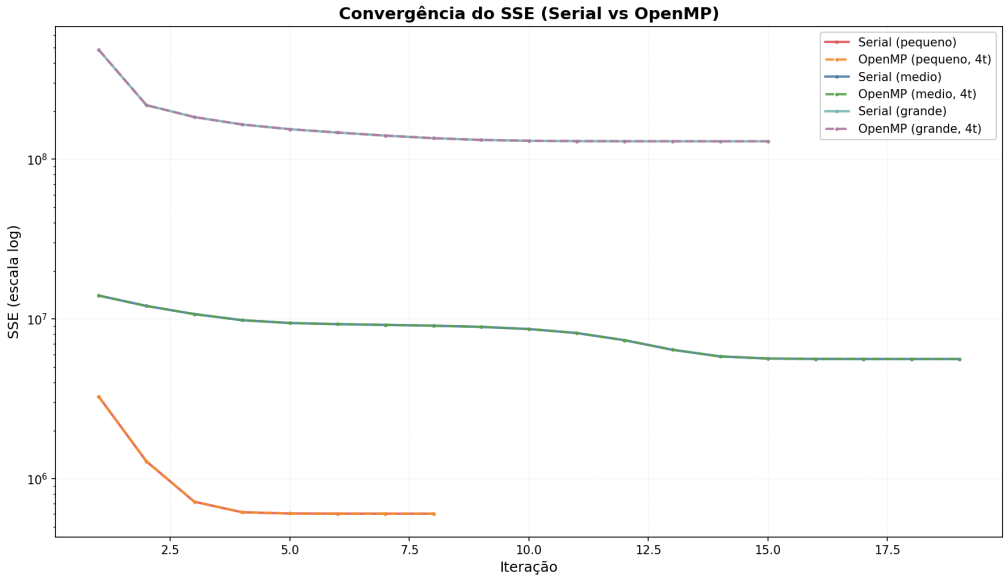

Figura 1. Convergência do SSE

A linha pontilhada no gráfico representa o *speedup* ideal. Como a máquina utilizada possui 12 threads, o limite máximo esperado seria 12 vezes. No entanto, devido ao *overhead* gerado, o *speedup* permanece abaixo desse valor. Observou-se que o *speedup* continuou aumentando até 20 threads, o que indica que uma investigação mais detalhada do hardware é necessária para compreender esse comportamento.

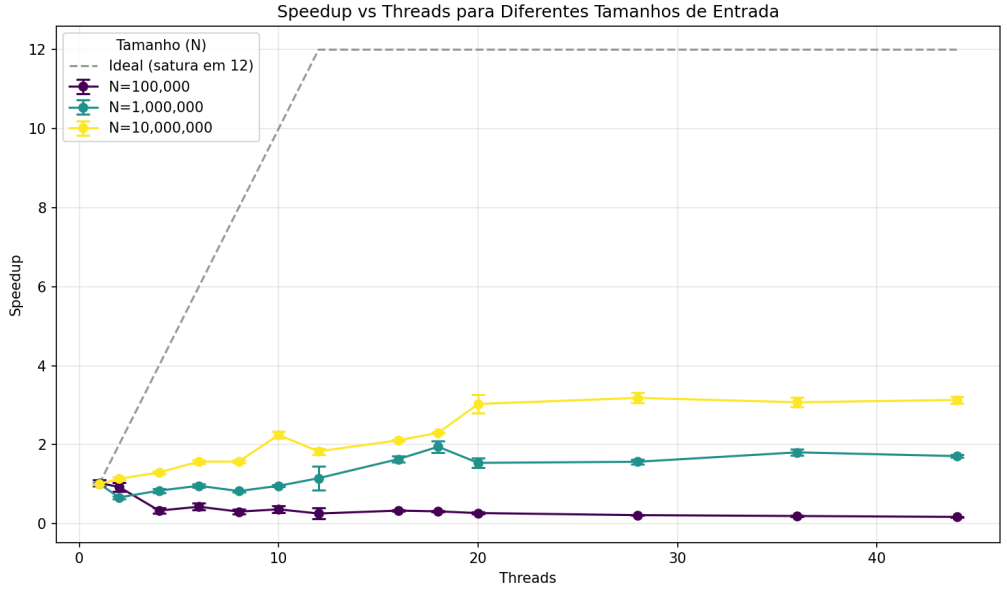

Figura 2. *Speedup* em função do número de threads

Os resultados apresentados no gráfico 3 corroboram as observações anteriores. Para as entradas de 100.000 e 1.000.000 números, os ganhos de tempo com o aumento do número de threads foram pouco significativos. No entanto, para a entrada de 10.000.000

números, observou-se uma melhora expressiva no tempo de execução, que caiu de aproximadamente 1.500 ms para menos de 500 ms com um grande número de threads.

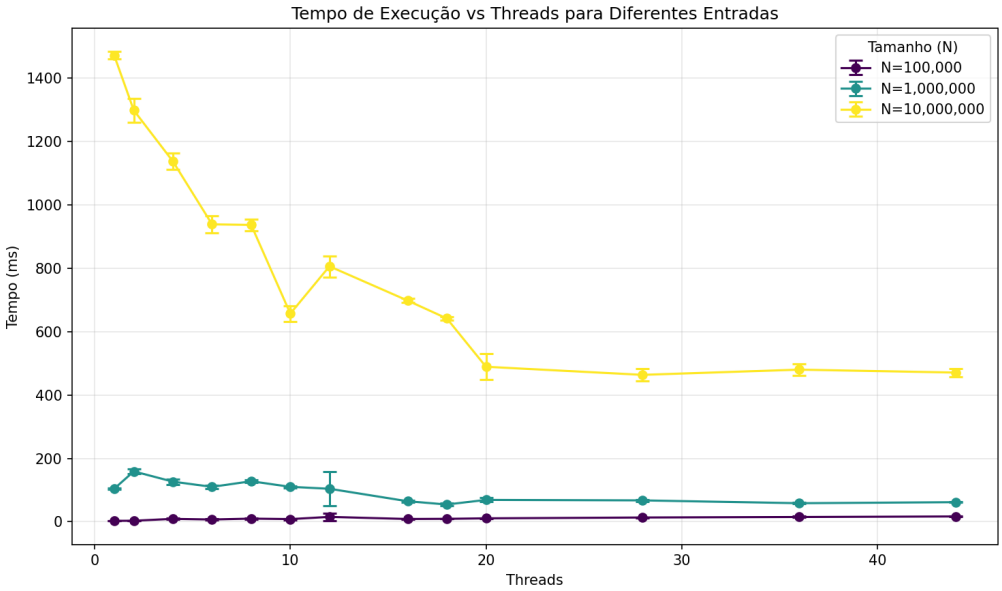

Figura 3. Tempo de execução em função do número de threads

5. Conclusão

O estudo permite concluir que houve uma redução significativa no tempo de execução do código para entradas relativamente grandes. Para entradas pequenas, o *speedup* foi praticamente nulo, e no caso de 100.000 entradas, o desempenho chegou a piorar. Essa queda pode ser explicada pelo *overhead* gerado pelo OpenMP para gerenciar o paralelismo.

O uso do paralelismo pode ser extremamente eficiente para reduzir o tempo de execução de problemas que exigem alto processamento. Entretanto, para problemas de pequena escala, o *overhead* associado pode superar os ganhos, tornando a técnica pouco vantajosa.

Referências

[Cornell 2023] Cornell (2023). Openmp: History and evolution. Acessado em: 18 out. 2025.