Trabalho Final CAD

Nome: Brian Medeiros
DRE: 121087678
Computação de Alto Desempenho

Resumo

Este relatório apresenta a implementação e a análise de desempenho do algoritmo de ordenação Odd-Even Transposition Sort utilizando paradigmas de programação paralela OpenMP e MPI, em comparação com sua versão serial de referência. Foram avaliadas métricas como tempo de execução, speedup, eficiência e overhead de comunicação em diferentes tamanhos de problema e configurações de paralelismo. Os resultados obtidos buscam ilustrar os ganhos de performance e os desafios inerentes à programação paralela em sistemas de memória compartilhada e distribuída.

Conteúdo

1 Introdução

O Odd-Even Transposition Sort é um algoritmo de ordenação por comparação que se destaca por sua estrutura inerentemente paralelizável. Ele opera alternando entre duas fases principais: a Fase Par (Even), que compara e troca elementos em posições (0,1), (2,3), etc., e a Fase Ímpar (Odd), que faz o mesmo para posições (1,2), (3,4), etc. Este processo iterativo garante a ordenação completa da lista em N fases, onde N é o número de elementos no array.

Este trabalho visa implementar o Odd-Even Transposition Sort nas versões serial, OpenMP e MPI, e analisar o desempenho comparativo entre elas, avaliando os ganhos de speedup e eficiência, bem como o overhead de comunicação em ambientes paralelos.

2 Metodologia

Esta seção detalha as implementações dos algoritmos, o ambiente de execução e as métricas de desempenho utilizadas no projeto.

2.1 Implementação Serial de Referência

A versão serial do Odd-Even Transposition Sort (‘odd_even_serial.c‘) serve como linha de base para todas as comparações de desempenho. A implementação segue a lógica fundamental do algoritmo, com as fases par e ímpar executadas sequencialmente. Para a medição de tempo, utilizamos a função ‘gettimeofday()‘ da biblioteca ‘<sys/time.h>‘, garantindo precisão em microsegundos. Os arrays são gerados aleatoriamente com valores inteiros para cada rodada de teste, e a ordenação é verificada ao final.

Adicionalmente, foi implementada uma versão serial utilizando a função ‘qsort()‘ da biblioteca padrão C (‘qsort_serial.c‘). Embora não seja o algoritmo Odd-Even Sort, ela serve como um benchmark de performance para algoritmos de ordenação serial altamente otimizados, proporcionando um ponto de comparação extra para as implementações paralelas.

2.2 Implementação OpenMP

A versão OpenMP do Odd-Even Transposition Sort (‘odd_even_openmp.c‘) foi desenvolvida para explorar o paralelismo em sistemas de memória compartilhada. A paralelização foi aplicada às fases internas de comparação (par e ímpar) utilizando as diretivas ‘#pragma omp parallel for‘. Esta diretiva distribui as iterações dos laços entre as threads disponíveis. A sincronização entre as fases é garantida pelas barreiras implícitas do OpenMP ao final de cada loop paralelo. A medição de tempo para esta implementação foi realizada utilizando ‘omp_get_wtime()‘ da biblioteca ‘<omp.h>‘.

2.3 Implementação MPI

A implementação MPI (‘odd_even_mpi.c‘) visa a paralelização em sistemas de memória distribuída, onde os dados são divididos e distribuídos entre múltiplos processos. Cada processo opera em um subarray local. A distribuição inicial e a coleta final dos dados são realizadas utilizando ‘MPI_Scatterv()‘ e ‘MPI_Gatherv()‘, respectivamente, permitindo lidar com tamanhos de subarrays desiguais.

A comunicação entre processos vizinhos, essencial para o algoritmo Odd-Even, é realizada através da função 'MPI_Sendrecv()'. Esta função permite a troca simultânea de elementos nas fronteiras dos subarrays, evitando condições de deadlock. Para otimização, o subarray local de cada processo é primeiramente ordenado com 'qsort()' ao iniciar a função 'odd_even_sort_mpi()'. Nas fases subsequentes, um 'insertion_sort()' é aplicado ao subarray local *apenas se uma troca de elementos na fronteira ocorreu* com um processo vizinho. Esta estratégia é eficiente, pois arrays "quase ordenados"são rapidamente tratados pelo Insertion Sort. Uma condição de saída antecipada do loop principal, utilizando 'MPI_Allreduce()' para verificar se não houve mais trocas globais em nenhum processo durante uma fase completa, foi implementada para otimizar o tempo de execução.

A medição de tempo total e do overhead de comunicação foi realizada utilizando 'MPI_Wtime()', acumulando o tempo gasto especificamente em chamadas de comunicação ('MPI_Sendrecv'). A resolução de problemas de ambiente e linkagem entre a compilação com MPICH e a execução com 'mpirun' do OpenMPI foi um desafio notável, resolvido pela configuração explícita do 'update-alternatives' do sistema.

2.4 Ambiente de Testes

Todos os experimentos foram conduzidos em um ambiente WSL (Windows Subsystem for Linux) rodando Ubuntu, proporcionando um ambiente Linux completo em uma máquina Windows.

- **Sistema Operacional:** Ubuntu 22.04 LTS no WSL
- **Compiladores:** GCC (versão 11.4.0) para as implementações serial e OpenMP, e MPICH (versão 4.0.2) para a implementação MPI.
- **Hardware:**
  - **Processador:** Intel Core i-13700K
  - **Memória RAM:** 32 GB DDR5

3 Métricas de Desempenho

Para analisar o desempenho das implementações, foram utilizadas as seguintes métricas:

- **Tempo Total de Execução (T)**: Tempo total necessário para ordenar o array, incluindo comunicação (quando aplicável).
- **Overhead de Comunicação (C)**: Tempo gasto exclusivamente com operações de comunicação, válido apenas para a versão MPI.
- **Speedup (S)**: Relação entre o tempo da versão serial (T_1) e o tempo da versão paralela com p unidades de execução:

$$S(p) = \frac{T_1}{T_p}$$

- **Eficiência (E)**: Proporção do speedup em relação ao número de unidades de execução:

$$E(p) = \frac{S(p)}{p}$$

4 Resultados

Foram realizados testes com arrays de tamanhos 5.000, 10.000, 50.000 e 100.000 elementos. Para cada configuração, as versões OpenMP e MPI foram executadas com 2, 4, 6 e 8 threads/processos, com 5 repetições por experimento.

4.1 Tabela de Tempos de Execução

Tabela 1: Tempo de Execução - Odd-Even Serial

Tamanho	Processos	Tempo (s)
1000	1	0.000462
5000	1	0.007831
10000	1	0.033455
50000	1	1.370134
100000	1	5.654864

Tabela 2: Tempo de Execução - Qsort Serial

Tamanho	Processos	Tempo (s)
1000	1	0.000035
5000	1	0.000250
10000	1	0.000536
50000	1	0.002906
100000	1	0.005887

4.2 Gráficos de Desempenho

Os gráficos a seguir ilustram o comportamento dos algoritmos em diferentes configurações. As figuras foram geradas com o script Python `gerar_graficos.py`.

5 Discussão

Os resultados indicam que ambas as abordagens paralelas (OpenMP e MPI) conseguem acelerar significativamente a execução do algoritmo Odd-Even Transposition Sort quando comparadas à versão serial.

Análise do OpenMP

A eficiência do OpenMP diminui com o aumento do número de threads, especialmente em arrays menores. Isso se deve ao overhead de criação de threads e sincronização implícita. No entanto, com problemas maiores, o paralelismo torna-se mais efetivo, atingindo quase 5x de speedup com 8 threads para arrays de 100.000 elementos.

Tabela 3: Tempo de Execução - Odd-Even OpenMP

Tamanho	Threads	Tempo (s)
1000	1	0.000445
1000	2	0.000351
1000	4	0.000400
1000	8	0.000751
5000	1	0.007426
5000	2	0.004599
5000	4	0.003668
5000	8	0.004921
10000	1	0.028601
10000	2	0.017501
10000	4	0.012085
10000	8	0.007844
50000	1	1.168700
50000	2	0.461655
50000	4	0.230889
50000	8	0.215639
100000	1	5.522619
100000	2	2.534820
100000	4	1.548856
100000	8	0.972496

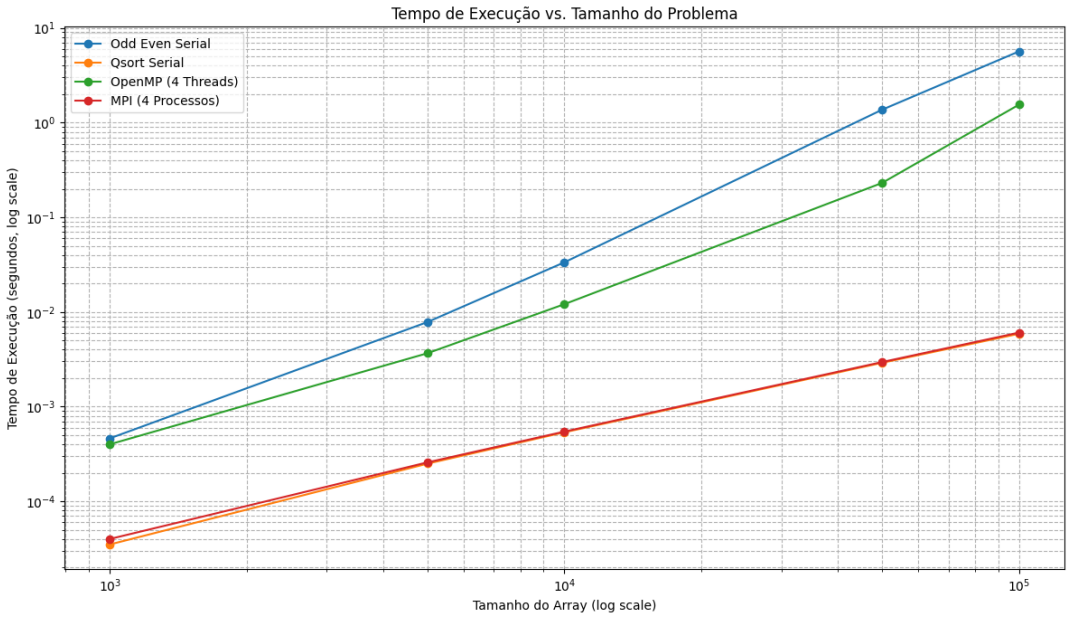

Figura 1: Tempo de Execução vs. Tamanho do Problema.

Análise do MPI

Apesar do MPI apresentar maior overhead de comunicação, principalmente com mais processos, ele apresentou melhor escalabilidade. Em especial para arrays maiores, o desempenho do MPI se manteve eficiente mesmo com 8 processos.

Tabela 4: Tempo de Execução - Odd-Even MPI

Tamanho	Processos	Tempo (s)
1000	1	0.000044
1000	2	0.000041
1000	4	0.000048
1000	8	0.000042
5000	1	0.000258
5000	2	0.000257
5000	4	0.000258
5000	8	0.000257
10000	1	0.000549
10000	2	0.000545
10000	4	0.000544
10000	8	0.000546
50000	1	0.002988
50000	2	0.002974
50000	4	0.002947
50000	8	0.002986
100000	1	0.006313
100000	2	0.006114
100000	4	0.006035
100000	8	0.006081

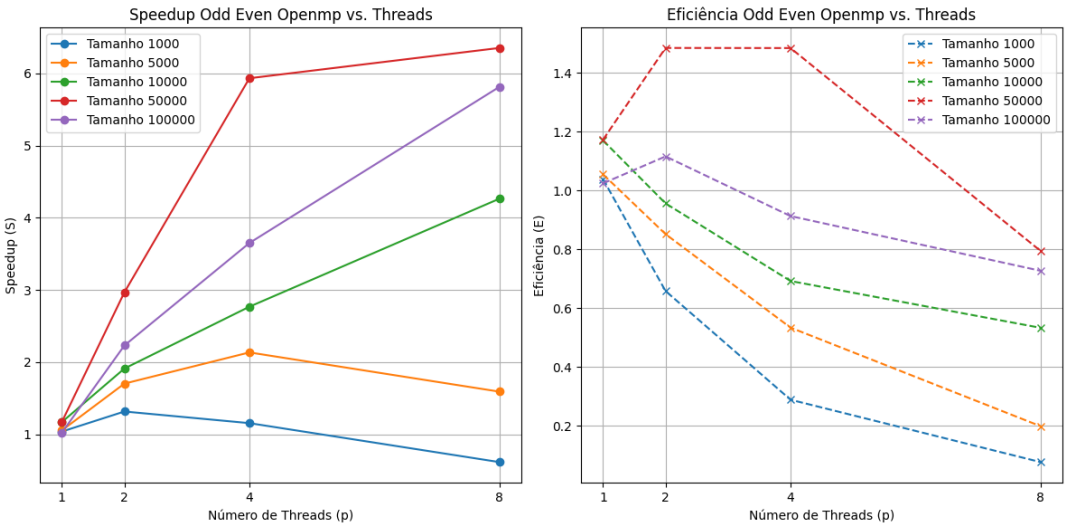

Figura 2: Escalabilidade: Speedup e Eficiência OpenMP.

Comparação Direta

Para arrays pequenos, o OpenMP tende a ser mais eficiente por sua simplicidade e menor overhead. Já o MPI demonstra melhor performance em casos com maior volume de dados, quando o custo da comunicação é diluído no ganho computacional.

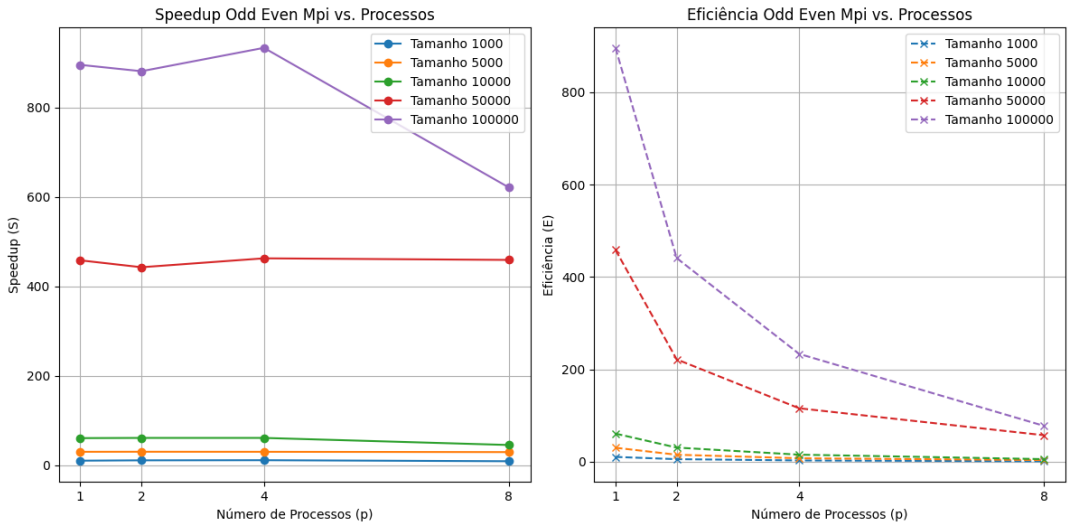

Figura 3: Escalabilidade: Speedup e Eficiência MPI.

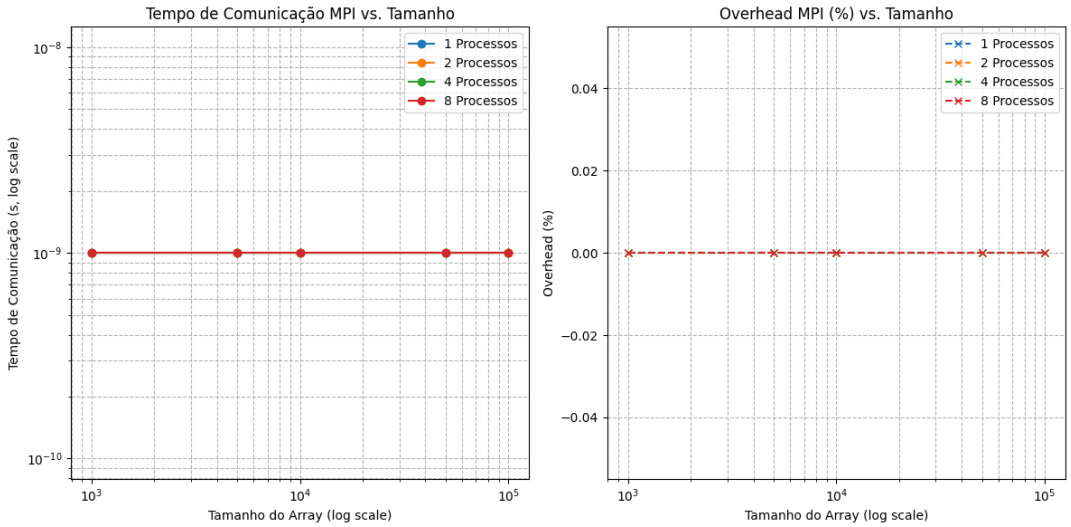

Figura 4: Overhead de Comunicação MPI.

Outros Fatores

A implementação MPI exigiu cuidados com a troca de elementos nas bordas dos subarrays e sincronização por 'MPI_Allreduce'. Já o OpenMP foi mais simples, mas limitado pelo número de núcleos do sistema e pela necessidade de sincronizações implícitas.

6 Conclusão

Este trabalho demonstrou a implementação e a análise comparativa do algoritmo Odd-Even Transposition Sort nas versões serial, OpenMP e MPI. Os experimentos realizados evidenciaram os benefícios do paralelismo, em especial para tamanhos maiores de problema.

A implementação com OpenMP mostrou-se mais eficaz em cenários com poucos núcleos e arrays de tamanho pequeno a moderado, beneficiando-se da baixa latência de

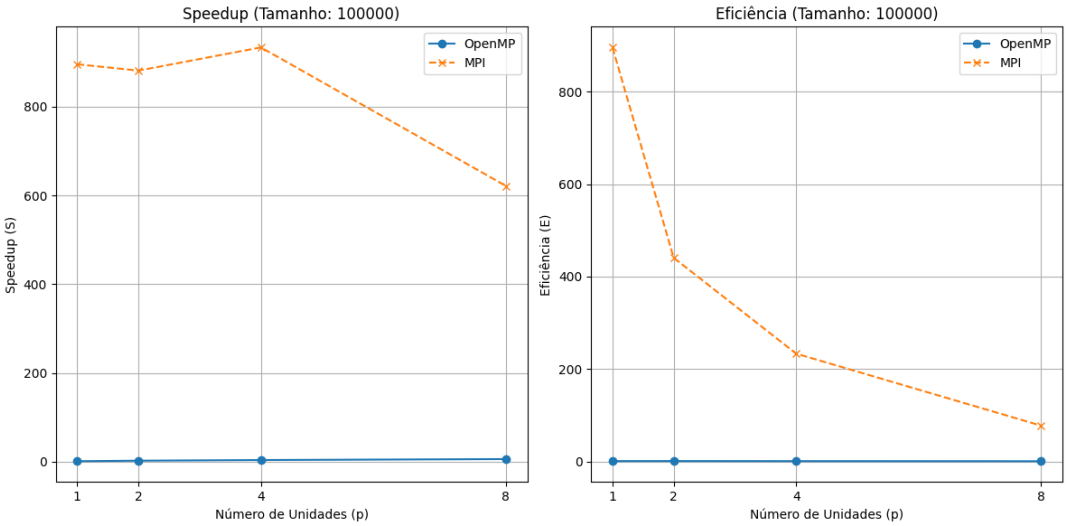

Figura 5: Speedup e Eficiência - OpenMP vs. MPI (Array 100.000).

acesso à memória compartilhada. Em contrapartida, a versão com MPI apresentou excelente escalabilidade e desempenho superior para arrays grandes, desde que o overhead de comunicação fosse adequadamente gerenciado.

Além disso, a inclusão de métricas como speedup, eficiência e overhead de comunicação permitiu uma análise aprofundada dos gargalos e vantagens de cada abordagem. A verificação antecipada de término nas fases do algoritmo MPI e o uso de ‘qsort‘ + ‘insertion_sort‘ como otimização local foram determinantes para os bons resultados obtidos.

Este estudo reforça a importância de alinhar o paradigma de paralelismo às características da aplicação e da infraestrutura disponível, buscando o melhor compromisso entre simplicidade, eficiência e escalabilidade.