Alunos: Leonardo Bednarczuk Balan de Oliveira e Gabriel Tadioto Oliveira

Trabalho 1 – Modelos de Programação Paralela

Opção escolhida: OP2

Os códigos foram paralelizados utilizando: OpenMP e MPI

1. Metodologia de Paralelização

Bucket_Sort com OpenMP:

A Função *findMax* que encontra o valor máximo no array foi paralelizada utilizando #pragma omp parallel for reduction(max:max). Assim, a busca pelo valor máximo é distribuída entre as threads, sem conflitos no acesso à variável max, utilizando a operação de redução para garantir que o valor máximo final seja o maior entre todas as threads.

Na Função *bucketSort* em si, a inicialização dos tamanhos dos buckets foi paralelizada com #pragma omp parallel for. Cada thread é responsável por inicializar um ou mais buckets, dependendo da quantidade de threads disponíveis, o que acelera o processo de inicialização.

A distribuição dos elementos para os buckets também foi realizada paralelamente com #pragma omp parallel for. Cada thread distribui os elementos para os buckets correspondentes, com um controle de acesso crítico (#pragma omp critical) para garantir que não haja conflitos ao adicionar elementos aos buckets**.**

Na ordenação de cada bucket utilizamos #pragma omp parallel for novamente, permitindo que diferentes threads ordenem os buckets simultaneamente. A ordenação foi feita utilizando o algoritmo Insertion Sort já dado no algoritmo sequencial, que foi executado assim sequencialmente dentro de cada bucket.

Por fim, na etapa de concatenar os buckets ordenados de volta no array original não paralelizamos, pois essa operação depende da ordem dos elementos. Cada thread pode escrever os elementos no array original, mas a sequência das operações precisa ser mantida.

Bucket_Sort com MPI:

Começamos com a inicialização do ambiente utilizando MPI_Init, onde os processos são configurados e a distribuição do trabalho entre eles é feita. rank identifica o processo atual, e size é o número total de processos envolvidos.

No início, o array original é lido e distribuído entre os processos utilizando MPI_Scatter. O processo com rank 0 (mestre) lê o array completo do arquivo e o divide igualmente entre os processos. Cada processo recebe uma parte do array original para processar de forma local. O número de elementos a serem processados é dado pelo número total de elementos pelo número de processos (n / size), onde n é o tamanho do array e size é o número de processos.

Cada processo realiza a ordenação do seu subconjunto de dados localmente utilizando o algoritmo *bucketSort*. A função *bucketSort* permanece semelhante à versão sequencial, com a diferença de que agora é aplicada em pedaços menores de dados. Cada processo aplica a ordenação dos elementos locais de forma independente e sem comunicação com outros processos durante essa fase.

Após a ordenação local, os resultados são enviados de volta para o processo mestre (rank 0) usando MPI_Gather. Cada processo envia seu subconjunto ordenado de volta para o processo mestre, que coleta os dados em um único array.

No processo mestre (rank 0), o array completo, agora composto pelos subconjuntos ordenados, é reorganizado por meio de uma última chamada à função bucketSort. É necessário isso para garantir que todos os elementos do array sejam ordenados globalmente, pois a distribuição de dados e ordenação local nos processos pode ter causado os dados estarem desordenados entre os subconjuntos.

Números_Primos com OpenMP:

No início, o array *primes* é criado com o tamanho MAX e todos os valores são inicializados para 1, assumindo que todos os números são primos. Isso é feito de forma paralela utilizando #pragma omp parallel for, que distribui a carga de trabalho entre as threads. Cada thread é responsável por inicializar uma parte do array.

Em seguida, o algoritmo aplica o *Crivo de Eratóstenes* para identificar os números primos. Quando um número primo é encontrado, a paralelização entra em ação novamente com #pragma omp parallel for para marcar todos os múltiplos desse número como não primos. Cada thread é responsável por um segmento de múltiplos.

Após o crivo, o número de primos encontrados é contado. A contagem é também paralelizada com a cláusula reduction, onde cada thread mantém sua própria cópia da variável count e, ao final, todas as cópias são somadas corretamente. Isso evitou problemas de concorrência.

Números_Primos com MPI:

Começamos com a inicialização do ambiente utilizando MPI_Init, onde os processos são configurados e a distribuição do trabalho entre eles é feita. rank identifica o processo atual, e size é o número total de processos envolvidos.

O array de primos é distribuído entre os processos. O tamanho do bloco de dados que cada processo irá processar é calculado pela divisão do limite MAX pelo número total de processos (chunk_size). Os processos então determinam os seus intervalos de números a serem verificados, definidos pelas variáveis *start* e *end*.

O processo mestre (rank 0) é responsável por inicializar o array *global_primes*, que é usado para marcar os números primos até a raiz quadrada de MAX. O mestre realiza o Crivo de Eratóstenes de forma sequencial para determinar todos os primos até esse limite. Daí esses primos são mandados por broadcast para todos os outros processos usando a função MPI_Bcast.

Após essa distribuição, cada processo calcula os múltiplos dos números primos nos intervalos que lhe foram atribuídos. A comunicação entre os processos não é necessária durante esse estágio, já que cada processo trabalha apenas no seu intervalo de dados.

Depois disso, todos os processos sincronizam com MPI_Barrier, garantindo que todos os cálculos locais sejam concluídos antes de começar a coleta dos resultados. O processo mestre (rank 0) recebe os dados de todos os outros processos utilizando MPI_Recv, coletando os subconjuntos locais ordenados de números primos no array *global all_primes*.

Ao fim, no processo mestre, o array global é percorrido para contar quantos números primos foram encontrados.

N_body com OpenMP:

No início, a função *InitParticles* é responsável por inicializar as partículas com posições e massas aleatórias. A distribuição do trabalho entre as threads é realizada com #pragma omp parallel for padrão, que distribui a carga de trabalho entre as threads.

Em seguida, o cálculo das forças de interação entre as partículas é realizado na função *ComputeForces*. Este cálculo é feito de forma paralela com o #pragma omp parallel for reduction, o que permite que cada thread calcule as forças para um conjunto de partículas e depois combine os resultados, evitando problemas de

concorrência. Também usando #pragma omp critical, onde só uma thread pode acessar por vez, nos cálculos de fx e fy, evitando erros.

Na função *ComputeNewPos*, o cálculo da nova posição das partículas é realizado de maneira semelhante, usando também #pragma omp parallel for para calcular a posição de cada partícula após a aplicação das forças.

Na *RandomSeed* também usamos #pragma omp critical, justamente para evitar erros de múltiplo acesso já no começo.

N_body com MPI:

No início, a função *InitParticles* é responsável por inicializar as partículas com posições e massas aleatórias. A distribuição do trabalho entre os processos é feita, onde o número total de partículas e a quantidade de iterações são enviadas para todos os processos. Em seguida, a função MPI_Bcast é usada para garantir que todos os processos tenham acesso aos dados de partículas e suas velocidades.

A função *ComputeForces* é responsável pelo cálculo das forças de interação entre as partículas. O cálculo é dividido entre os processos, onde cada processo calcula as forças para um intervalo específico de partículas. A utilização de MPI_Allreduce foi necessária para que todos os processos obtenham o valor máximo das forças calculadas, garantindo que o próximo passo de tempo seja adequado para a simulação.

Em seguida, a função *ComputeNewPos* é utilizada para atualizar as posições das partículas, com o trabalho sendo distribuído entre os processos. Utilizamos MPI_Barrier para sincronizar temporalmente todos os processos. Após isso, com o MPI_Allgather, garantimos que todas as partículas, após a atualização de suas posições, estejam também sincronizadas entre os processos.

2. Resultados

Configs padrão da primeira tabela de cada algoritmo:

- **OpenMP:** 12 threads
- **MPI:** 4 processos

2.1 Bucket_Sort:

Tamanho da Entrada X Tempo de Execução em segundos

Tam. da Entrada	Sequencial	OpenMP	MPI
100	0.000009	0.026014	0.000291
1000	0.000285	0.037176	0.000120
10000	0.009668	0.031028	0.004969
100000	0.499214	0.246432	0.433552
500000	12.703975	3.844544	10.387944
1000000	50.218174	15.096886	42.923877
1500000	112.929097	34.207661	97.495221
2000000	201.401950	62.423780	169.040564
3000000	456.351600	144.230315	390.225576

Os resultados da paralelização do Bucket Sort acima apresentaram diferentes desempenhos conforme o tamanho da entrada. Para entradas pequenas, a versão sequencial é mais eficiente devido à baixa sobrecarga. Com o aumento do tamanho dos dados, OpenMP se destacou como a abordagem mais rápida, reduzindo em muitas vezes o tempo de execução. O MPI também acelera o processamento, mas sua eficiência é menor que a do OpenMP devido à comunicação entre processos da implementação. Assim, a paralelização é vantajosa para grandes entradas, com OpenMP sendo a melhor escolha neste contexto.
(OpenMP bem melhor que o MPI aqui para grandes entradas, segundo as nossas implementações).

OpenMP para conjunto de 1 milhão de elementos

Nº de Threads	Tempo de Exec	Speedup	Eficiência
Sequencial (1)	49.384808	-	-
2	49.247956	1	0,5
4	30.474861	1,62	0,41
6	21.567353	2,29	0,38
8	21.623862	2,29	0,28
10	15.543296	3,18	0,32

12	14.357756	3,44	0,29

A análise de escalabilidade (tabela acima) do OpenMP para esse problema mostra que o aumento no número de threads melhora o desempenho, mas com eficiência decrescente. O speedup cresce conforme o número de threads aumenta, atingindo 3,44× com 12 threads, o que indica um bom ganho em relação à versão sequencial. No entanto, a eficiência cai progressivamente, partindo de 50% com 2 threads para cerca de 29% com 12 threads. Isso sugere que a sobrecarga de sincronização e a possível saturação dos recursos limitam a escalabilidade, tornando o ganho marginal à medida que mais threads são adicionadas.
Speedup cresce enquanto a eficiência desce, Escalabilidade mediana.

MPI para conjunto de 1 milhão de elementos

Nº de Processos	Tempo de Exec	Speedup	Eficiência
Sequencial (1)	49.321354	-	-
2	37.766504	1,31	0,655
3	39.431032	1,25	0,42
4	42.760238	1,15	0,29
6	48.405858	1,02	0,17

A escalabilidade do Bucket Sort com MPI para um conjunto de 1 milhão de elementos tem um desempenho pouco eficiente à medida que o número de processos aumenta. O speedup cresce inicialmente, mas de forma modesta, atingindo apenas 1,31× com 2 processos e caindo progressivamente conforme mais processos são adicionados. A eficiência, que começa em 65,5% com 2 processos, despenca para 17% com 6 processos, indicando uma alta sobrecarga de comunicação entre os processos MPI.
Escalabilidade baixa do MPI nesse problema.

2.2 Números_Primos:

Tamanho da Entrada X Tempo de Execução em segundos

Tam. da Entrada	Sequencial	OpenMP	MPI
3000000	0.027091	0.062872	0.033297
30000000	0.390768	0.143744	0.177140
300000000	4.323372	2.135398	2.524195
3000000000	48.644978	27.755251	29.215673

A paralelização aqui apresenta diferentes comportamentos conforme o tamanho da entrada. Para entradas menores, o método sequencial é mais rápido devido à sobrecarga da paralelização, especialmente no OpenMP, que apresenta um tempo de execução superior ao sequencial para 3 milhões de elementos. À medida que o tamanho da entrada cresce, OpenMP se torna mais eficiente, reduzindo o tempo de execução em comparação com a versão sequencial. O MPI também apresenta melhorias, mas seu desempenho é ligeiramente inferior ao OpenMP, provavelmente devido ao custo de comunicação entre processos.
Paralelização vantajosa para grandes entradas.

OpenMP para conjunto de 300000000 números

Nº de Threads	Tempo de Exec	Speedup	Eficiência
Sequencial (1)	4.766395	-	-
2	2.953779	1,62	0,81
4	2.105695	2,26	0,57
6	2.088898	2,28	0,38
8	2.058642	2,32	0,29
10	2.017563	2,36	0,24
12	2.017040	2,36	0,20

A escalabilidade do OpenMP aqui mostra que o aumento do número de threads melhora o desempenho, mas com retornos decrescentes. O speedup cresce até 2,36× com 12 threads, o que indica ganhos moderados em relação à versão sequencial. No entanto, a eficiência diminui significativamente, partindo de 81% com 2 threads para apenas 20% com 12 threads, sugerindo limitações na escalabilidade.

Mesmo trazendo ganhos a Escalabilidade é limitada.

MPI para conjunto de 300000000 números

Nº de Processos	Tempo de Exec	Speedup	Eficiência
Sequencial (1)	4.763595	-	-
2	3.208710	1,48	0,74
3	2.675149	1,78	0,59
4	2.506813	1,90	0,48
6	2.477196	1,92	0,32

A escalabilidade com MPI mostra uma melhora inicial no desempenho, mas com eficiência decrescente conforme o número de processos aumenta. O speedup cresce até 1,92× com 6 processos, indicando que a paralelização traz benefícios, mas com retornos reduzidos. A eficiência começa em 74% com 2 processos, mas cai para apenas 32% com 6 processos, sugerindo um aumento da sobrecarga de comunicação e sincronização entre os processos.
Mesmo também trazendo ganhos a Escalabilidade é limitada.

2.3 N_body:

Tamanho da Entrada (npart e cnt) X Tempo de Execução em segundos

Tam. da Entrada	Sequencial	OpenMP	MPI
4 10	0.00001	0.017466	0.000329
100 20	0.00196	0.017253	0.000555
500 30	0.03714	0.026527	0.009766
1000 30	0.12761	0.054217	0.037582
5000 40	4.22248	0.989194	1.201544
8000	13.31481	3.064515	3.998970

50			
10000 50	20.78752	4.877163	6.250676
15000 60	55.74029	13.235040	16.795461

A paralelização do N-body é eficiente para grandes volumes de dados, reduzindo significativamente o tempo de execução. No entanto, para pequenos conjuntos de partículas, a sobrecarga da paralelização torna a execução mais lenta do que a versão sequencial. Bem interessante os resultados aqui, a paralelização teve bons ganhos. Mais uma vez o OpenMP se saindo melhor que o MPI.

OpenMP para entrada de 15000 e 60

Nº de Threads	Tempo de Exec	Speedup	Eficiência
Sequencial (1)	54.580960	-	-
2	28.905279	1,89	0,95
4	18.968356	2,88	0,72
6	18.583402	2,94	0,49
8	17.286758	3,16	0,40
10	15.265770	3,57	0,36
12	13.817803	3,95	0,33

A escalabilidade do OpenMP mostra bons ganhos de desempenho com o aumento do número de threads, mas com eficiência decrescente. O speedup atinge 3,95× com 12 threads, indicando uma melhoria muito boa em relação à versão sequencial. No entanto, a eficiência começa alta (95% com 2 threads) e cai para 33% com 12 threads, devido à sobrecarga de sincronização e possível desbalanceamento de carga. Isso sugere que, embora a paralelização seja vantajosa, há limitações na escalabilidade com o aumento do número de threads, já que os ganhos se tornam marginais em relação ao custo computacional adicional.
Mesmo assim, gostamos dos resultados aqui.

MPI para entrada de 15000 e 60

Nº de Processos	Tempo de Exec	Speedup	Eficiência
Sequencial (1)	54.428565	-	-
2	28.533919	1,91	0,96
3	20.620268	2,64	0,88
4	16.516605	3,29	0,82
6	20.023609	2,72	0,45

A escalabilidade do MPI mostra também ótimos ganhos de desempenho inicialmente, com speedup de 3,29× com 4 processos, e alta eficiência (96% com 2 processos). No entanto, a eficiência diminui à medida que o número de processos aumenta, caindo para 45% com 6 processos, devido à sobrecarga de comunicação e sincronização. Isso indica que, embora a paralelização seja vantajosa até um certo ponto, a utilização de mais processos não gera ganhos proporcionais e a sobrecarga de comunicação se torna limitante.

3. Conclusão

Concluindo, observamos que, embora a escalabilidade tenha se mostrado limitada em relação à eficiência nas soluções paralelas, na maioria dos casos houve um ganho considerável em termos de tempo, com os algoritmos sendo significativamente mais rápidos que a versão sequencial. Também, vimos que o OpenMP se destacou como o método de paralelização mais eficaz, muito por conta da facilidade maior de usarmos ele em relação ao MPI que pra nós é mais complicado. Dessa forma, foi mais fácil desenvolver uma boa solução paralela utilizando o OpenMP e deu certo.