Universidade Federal do Rio Grande do Norte
Departamento de Engenharia da Computação e Automação
DCA3703 - Programação Paralela
Tarefa 17: Estruturas de Dados Não Contíguas em MPI
Aluno: Daniel Bruno Trindade da Silva

1 Introdução

Este relatório tem como objetivo apresentar os conhecimentos adquiridos durante o desenvolvimento da Tarefa 17 da disciplina de Programação Paralela. A atividade teve como objetivo reimplementar o código da tarefa 16 distribuindo as colunas da matriz entre os processos, permitindo que cada um calculasse uma contribuição parcial para todos os elementos do vetor 'y'. ´Por fim, o relatório discute as diferenças de acesso à memória e desempenho desta abordagem em comparação com a distribuição por linhas.

2 Enunciado

Reimplemente a tarefa 16, agora distribuindo as colunas entre os processos. Utilize `MPI_Type_vector` e `MPI_Type_create_resized` para definir um tipo derivado que represente colunas da matriz. Use `MPI_Scatter` com esse tipo para distribuir blocos de colunas, e `MPI_Scatter` ou cópia manual para enviar os segmentos correspondentes de x. Cada processo deve calcular uma contribuição parcial para todos os elementos de y e usar `MPI_Reduce` com `MPI_SUM` para somar os vetores parciais no processo 0. Discuta as diferenças de acesso à memória e desempenho em relação à distribuição por linhas.

3 Desenvolvimento

Fizemos a reimplementação do código original, alterando a estratégia de paralelização para realizar a distribuição da matriz por colunas entre os processos, utilizando tipos derivados do MPI. Essa abordagem visa uma distribuição mais eficiente da matriz e do vetor, aproveitando melhor os recursos de comunicação da biblioteca MPI.

3.1 Criação do Tipo Derivado para Representação das Colunas

A matriz A é armazenada em memória de forma contígua linha a linha (ordem row-major). Como o objetivo é distribuir blocos de colunas entre os processos, foi necessário criar um tipo derivado que representasse de forma lógica esse agrupamento, apesar de sua não contiguidade na memória.

Para isso, utilizamos inicialmente a função `MPI_Type_vector`. Esse construtor cria um tipo derivado com múltiplos blocos, permitindo a seleção de elementos com espaçamento regular:

```
MPI_Type_vector(M, cols_per_proc, N, MPI_UNSIGNED_CHAR, &column_type);
```

Onde:

- **M**: número de blocos, correspondente ao número de linhas da matriz.
- `cols_per_proc`: número de elementos consecutivos a serem enviados por linha, isto é, a quantidade de colunas atribuídas a cada processo.
- **N**: espaçamento entre os inícios de blocos, igual ao número total de colunas da matriz.

- `MPI_UNSIGNED_CHAR`: tipo base, equivalente a `uint8_t`.

Porém, o tipo criado por `MPI_Type_vector` possui um extent (extensão) maior do que o necessário para o envio contínuo entre processos. Por isso, utilizamos a função `MPI_Type_create_resized` para ajustar esse tamanho, garantindo que o `MPI_Scatter` realize a comunicação corretamente:

```
MPI_Type_create_resized(
    column_type, 0,
    cols_per_proc * sizeof(uint8_t),
    &resized_column_type
);
MPI_Type_commit(&resized_column_type);
```

3.2 Distribuição dos Dados com MPI_Scatter

Com o tipo derivado criado, foi possível empregar `MPI_Scatter` para distribuir diretamente os blocos de colunas da matriz entre os processos:

```
MPI_Scatter(
    &A[0][0], 1,
    resized_column_type,
    local_A, M * cols_per_proc,
    MPI_UNSIGNED_CHAR, 0,
    MPI_COMM_WORLD
);
```

- O processo raiz (rank 0) envia uma unidade do tipo derivado para cada processo.
- Cada processo recebe em `local_A` seu bloco de colunas, já linearizado com tamanho M * `cols_per_proc`.

Já a distribuição do vetor x foi realizada em paralelo, utilizando um segundo `MPI_Scatter`:

```
MPI_Scatter(
    x, cols_per_proc,
    MPI_UNSIGNED_CHAR,
    local_x, cols_per_proc,
    MPI_UNSIGNED_CHAR, 0,
    MPI_COMM_WORLD
);
```

3.3 Cálculo da Contribuição Parcial e Redução com `MPI_Reduce`

Após a distribuição, cada processo realizou o cálculo parcial de sua contribuição para o vetor resultado y, realizando a multiplicação das colunas recebidas pelo segmento correspondente de x. O vetor parcial `local_y` calculado por cada processo foi então combinado para formar o resultado final y no processo raiz. Para isso, utilizamos a função `MPI_Reduce` com a operação `MPI_SUM`:

```
MPI_Reduce(local_y, y, M, MPI_UNSIGNED_CHAR, MPI_SUM, 0, MPI_COMM_WORLD);
```

`MPI_SUM` garante que as contribuições de cada processo sejam somadas elemento a elemento. E o resultado final do vetor y é armazenado no processo com `rank 0`.

4 Resultados

Tamanho da Matriz	Abordagem	2 proc	4 proc	8 proc	16 proc	32 proc
2048 x 2048	Por linhas	0.007393	0.012269	0.010754	0.017535	0.032846
	Por colunas	0.008465	0.017355	0.010708	0.015743	0.029602
4096 x 4096	Por linhas	0.024350	0.019467	0.026092	0.018572	0.033169
	Por colunas	0.025716	0.027852	0.028952	0.017797	0.025117
8192 x 8192	Por linhas	0.103332	0.062595	0.058050	0.032905	0.045219
	Por colunas	0.093287	0.071059	0.063350	0.034189	0.038946
16384 x 16384	Por linhas	0.336517	0.201913	0.159433	0.076215	0.075465
	Por colunas	1.379384	0.950391	0.601622	0.317023	0.268764
32768 x 32768	Por linhas	1.314407	0.773659	0.510369	0.249389	0.193285
	Por colunas	1.366954	0.954603	0.613585	0.318262	0.287530

Table 1: Comparação dos tempos de execução (em segundos) entre as abordagens de distribuição por linhas e por colunas

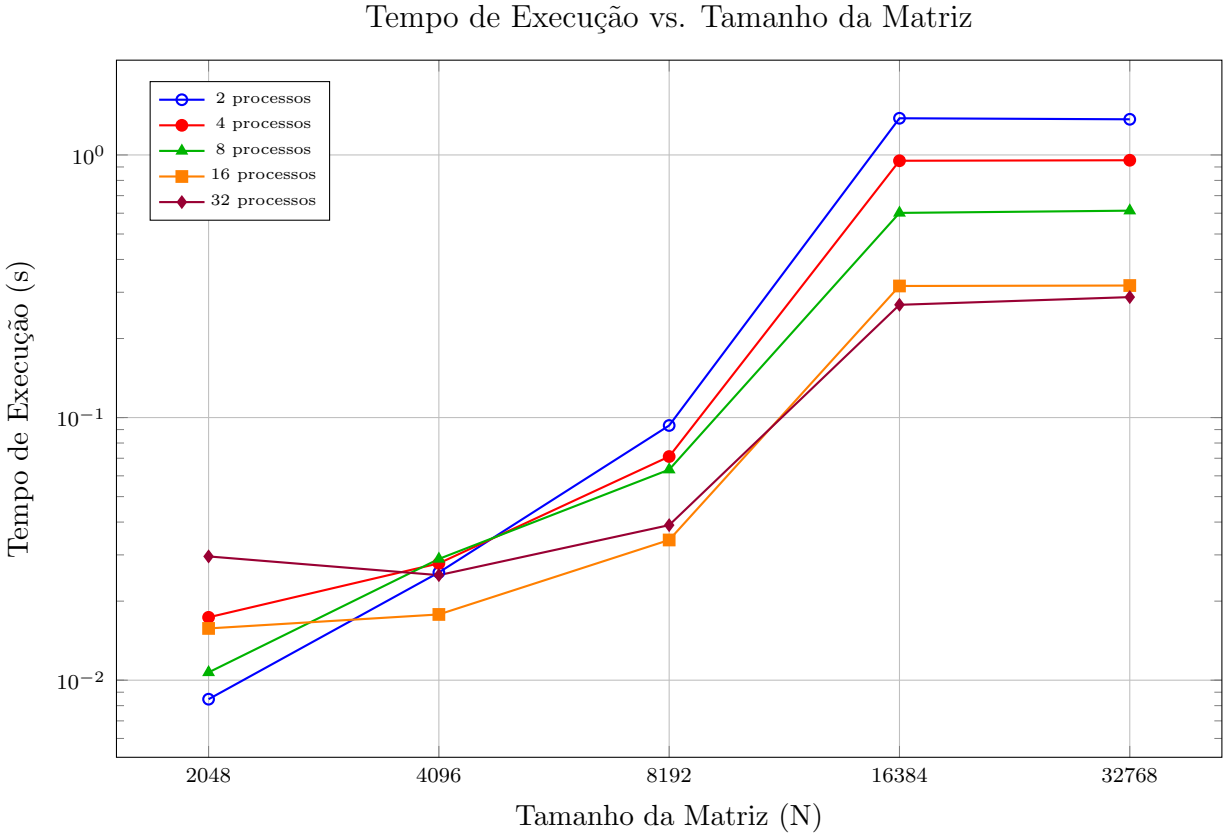

Figure 1: Gráfico de linhas do tempo de execução em função do tamanho da matriz para diferentes números de processos.

5 Analise dos Resultados

5.1 Variação do Tempo com o Número de Processos

Observa-se que, para matrizes de pequeno porte (por exemplo, 2048 x 2048), o tempo de execução não reduz significativamente com o aumento do número de processos. Na verdade, há uma flutuação:

com 2 processos, o tempo foi de aproximadamente 8,4 ms, mas com 4 e 8 processos houve aumento para cerca de 17,3 ms e posterior redução para 10,7 ms, respectivamente.

Esse comportamento se justifica pelo custo de comunicação inerente ao aumento do número de processos, que pode superar os benefícios da paralelização quando o volume de dados por processo é pequeno. Assim, para tamanhos reduzidos de matriz, o overhead de comunicação domina, tornando a execução com mais processos ineficiente.

5.2 Ganhos de Desempenho com Matrizes Maiores

À medida que o tamanho da matriz aumenta, o ganho de desempenho com o aumento no número de processos se torna mais evidente:

- Para a matriz de 8192 x 8192, o tempo caiu de aproximadamente 93 ms com 2 processos para cerca de 34 ms com 16 processos, e manteve-se baixo com 32 processos.
- Para a matriz de 16384 x 16384, a redução foi ainda mais expressiva: de 1,37 s com 2 processos para cerca de 0,27 s com 32 processos, mostrando um ganho de quase 5 vezes.
- O mesmo comportamento foi observado para a maior matriz testada (32768 x 32768), com redução de cerca de 1,36 s para aproximadamente 0,29 s com 32 processos.

Esse comportamento confirma que, para grandes volumes de dados, o custo de comunicação torna-se amortizado pela expressiva redução no tempo de processamento proporcionada pela divisão de trabalho entre múltiplos processos.

5.3 Análise Comparativa entre Distribuição por Colunas e por Linhas

A análise dos resultados revela uma clara dependência entre o tamanho da matriz e a eficiência das duas abordagens de paralelização. Para matrizes pequenas e médias (até 8192x8192), a distribuição por colunas apresenta desempenho competitivo. Isso sugere que, para esses tamanhos, o overhead de comunicação é o fator dominante e a abordagem por colunas consegue um melhor balanceamento da carga computacional. Entretanto, esse cenário se inverte drasticamente quando analisamos matrizes maiores.

Para matrizes grandes (16384x16384 e superiores), a distribuição por linhas demonstra superioridade evidente, sendo 250-370% mais rápida que a distribuição por colunas. Esta diferença é explicada pela localidade de memória: como C armazena matrizes em formato row-major, o acesso sequencial às linhas aproveita melhor o cache do processador, enquanto o acesso às colunas gera cache misses frequentes que degradam significativamente a performance. Adicionalmente, o overhead dos tipos derivados MPI (`MPI_Type_vector`) necessários para a distribuição por colunas se torna proibitivo conforme o tamanho da matriz aumenta, tornando a abordagem por linhas mais robusta e previsível para aplicações de grande escala.

6 Conclusão

A implementação da distribuição por colunas utilizando `MPI_Type_vector` e `MPI_Type_create_resized` foi realizada com sucesso, permitindo a distribuição de dados não-contíguos e agregação via `MPI_Reduce`. A análise comparativa revelou que a eficiência é fortemente dependente do tamanho da matriz. Para matrizes menores a distribuição por colunas foi competitiva, mas para matrizes grandes (16384x16384) tornou-se 250-370% mais lenta devido aos cache misses causados pelo acesso não-sequencial em formato row-major do C. O overhead dos tipos derivados MPI também se intensifica com o tamanho da matriz, tornando a distribuição por linhas mais adequada para aplicações de grande escala.