Trabalho de Programação Paralela

Algoritmo para Árvore de Redução com MPI

Daniel Souza de Campos

Ciência da Computação – UFMG – 2021/2

1 – Introdução

O MPI é uma especificação de uma API para programação para memória distribuída que permite que processos, cada um com suas próprias memórias, comuniquem entre si por meio de mensagens e realizem trabalho sincronizado. Além disso, é possível agrupar processos em algo chamado de comunicadores e as mensagens enviadas podem ser marcadas com categorias que ajudam a identificar o motivo da mensagem.

Uma das situações mais comuns em aplicações que utilizam o MPI é a distribuição de dados de entrada entre os processos para que cada um deles compute algo. Depois, provavelmente, deseja-se que esse valor seja combinado em um único resultado. Dessa forma, esse problema contém três partes: a distribuição dos dados entre os processos, a computação de um resultado local para cada processo e a agregação desses resultados em um só.

A distribuição dos dados entre os processos pode ser feita de algumas formas diferentes. A primeira delas é a chamada distribuição em blocos. Nesse tipo de distribuição, de acordo com uma entrada de tamanho M, cada um dos n processos recebe $\frac{M}{n}$ dos valores de entrada de forma contínua. Assim, o iésimo processo recebe o iésimo bloco de entrada. Essa abordagem possui problemas caso o custo da computação a ser realizada seja diretamente influenciada pelo dado de entrada, ou seja, se uma certa característica do dado de entrada torna a computação mais custosa ou não.

Pensando nessa questão, se os dados de entrada vierem ordenados, a distribuição em blocos pode fazer com que haja desbalanceamento de carga entre os processos. Dessa forma, um segundo tipo de distribuição é a cíclica. Nesse caso, os processos recebem os dados de entrada de forma intercalada. Por exemplo, com dois processos, o primeiro recebe o primeiro dado, o segundo o segundo dado, o primeiro processo volta a receber o terceiro dado de entrada e o segundo processo recebe o quarto. O próprio MPI já oferece uma especificação de função que realiza a distribuição dos dados: a função MPI_Bcast que realiza, por padrão, a distribuição em blocos dos dados para os processos em um mesmo comunicador.

A computação dos dados pelos processos é dependente do contexto da aplicação. A questão é que um processo irá gerar um resultado local que deverá ser agregado com os resultados dos outros processos. Também existem algumas implementações diferentes para essa parte. A primeira e mais ingênua

seria todos os processos enviarem seus resultados para um mesmo processo escolhido antecipadamente que irá realizar a agregação de todos os resultados. A desvantagem dessa abordagem é que o processo escolhido fica a cargo de toda a computação necessária para a agregação enquanto os outros apenas enviam dados para ele.

Uma segunda abordagem seria a de várias agregações parciais. Dessa forma, são formados pares entre os processos, seguindo alguma lógica, tal que a computação da agregação seja melhor distribuída entre os processos. Também existem várias formas para a escolha dos pares ou grupos de processos e sua agregação. O MPI também oferece a possibilidade de agregação de resultados de diferentes processos via a função MPI_Reduce.

As funções MPI_Bcast e MPI_Reduce são conhecidas como funções de comunicação coletiva, já que elas realizam comunicação com vários processos ao mesmo tempo. O objetivo desse trabalho é implementar todo o processo de distribuição, computação e agregação de resultados em MPI sem fazer uso de funções de comunicação coletiva, mas sim, apenas com as funções base MPI_Send e MPI_Recv usadas para comunicação entre dois processos de forma ponta-a-ponta. A computação realizada é a soma dos dados de entrada.

2 – Ferramentas utilizadas

O programa foi escrito na linguagem C++11, com o *wrapper* de compilador *mpicxx* para a compilação de programas C++ com MPI. Além disso, foi usado o editor de código *Visual Studio Code* e o GitHub para versionamento e repositório do código. O trabalho ficará disponível publicamente em https://github.com/Pendulun/CollectiveComunicationMPI após a data final de entrega do trabalho dia 09/01/2022.

A máquina para o desenvolvimento é um notebook Dell com 8GB RAM e processador Intel Core i5 de 7ª geração. Seu sistema operacional é Windows 10 e possui o *Windows Subsystem for Linux* instalado.

3 – Compilação e execução

Para a compilação do programa, é necessário o *mpicxx* que é um *wrapper* para um compilador. Na pasta raiz do projeto, existe o arquivo *makefile*. Basta executar *make* que será gerado um arquivo chamado *tp2* também na pasta raiz do projeto.

Para executar o programa, é necessário o *mpiexec*. Existem duas opções para a execução:

1. Executar diretamente via *mpiexec -n <Num. de processos> ./tp2*
2. Executar via *Makefile* com *make run*. Esse está definido para executar com 4 processos.

Os argumentos que o programa espera são, em ordem:

1. Uma flag de tipo de output gerado pelo programa:
    a. *sum*: Mostra apenas a soma final
    b. *time*: Mostra apenas o maior tempo de execução por um processo
    c. *all*: Mostra, primeiramente, a soma final e, depois, o maior tempo de execução.
2. A quantidade de números de entrada M
3. M números de entrada como especificado no argumento anterior

O programa assume que todos os argumentos serão passados corretamente e, portanto, não realiza nenhum tipo de verificação neles.

4 – Distribuição dos dados entre processos

A primeira parte do problema é a distribuição dos dados entre os processos. Pode-se pensar que a computação a ser realizada, o somatório dos números, é influenciado pela magnitude dos números na qual a operação é aplicada. Entretanto, essa influência não deve tornar a operação custosa já que essa computação é bastante simples. Outro fator é que não esperamos que os dados de entrada venham de alguma forma ordenada, decrescente ou crescente, mas sim, de forma aleatória. Assim, foi escolhido realizar o particionamento em blocos dos números para os processos.

Como já dito anteriormente, o particionamento em blocos divide os dados da entrada em blocos sequenciais para cada processo. Isso faz com que o primeiro bloco seja assinalado para o primeiro processo, o segundo bloco para o segundo processo etc... Entretanto, um problema inerente com essa abordagem é quando a quantidade de dados de entrada não é divisível pelo número de processadores, por exemplo, 10 dados de entrada para 4 processos. Intuitivamente, podemos fazer com que 2 processos recebam 2 dados e outros 2 processos recebam 3 dados.

Para tal, precisamos, primeiramente, calcular qual o tamanho mínimo dos blocos que cada processo receberá. Essa conta é simples:

$$n = \left\lfloor \frac{M}{N} \right\rfloor$$

Em que M é o tamanho total da entrada, N é o número de processos e n é o tamanho mínimo dos blocos. Após isso, podemos verificar se algum processo receberá mais números do que outros distribuindo a diferença entre M e $n \times N$:

$$d = M - n \times N$$

Se essa diferença for maior que 0, sabemos que ainda existem números a serem distribuídos. Importante notar que $d < N$, já que, se fosse maior ou igual, o valor de n seria maior do que o atual. Dessa forma, foi escolhido que os d primeiros processos terão o tamanho dos seus blocos incrementados em 1.

Outro problema que pode aparecer é de o tamanho da entrada M ser menor do que o número de processos N. Para isso, podemos fazer com que os M primeiros processos recebam 1 número cada e os $N - M$ processos restantes não recebam nenhum número.

De qualquer forma, teremos um número P de processos que receberam números. Ele será igual a N se $M \geq N$ e será igual a M se $M < N$.

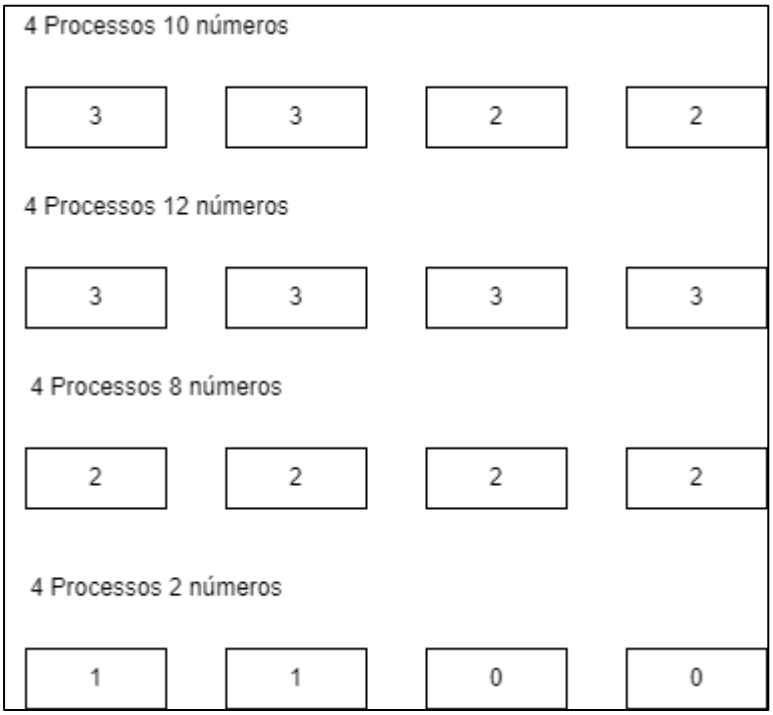


Figura 1: Quantidade de números por bloco

Na implementação dessa fase, o processo 0 é encarregado de ler a entrada do usuário e enviar os dados para ele mesmo ou para os outros processos de acordo com a quantidade esperada por cada processo. Importante dizer que a distribuição é feita enquanto o processo 0 lê as entradas, ou seja, não são lidas todas as entradas primeiro e só depois elas são divididas, mas sim, um dado é lido e ele já é enviado para seu processo correspondente.

Para tal, o cálculo de quantos números cada processo deve receber já foi realizado antes. Esse fato irá influenciar na medida de tempo resultante dos testes realizados posteriormente.

Para guardar os números recebidos, cada processo tem em posse uma pilha.

5 – Soma Cruzada

Essa fase representa a computação interna de um processo em cima dos dados que ele recebeu. Entretanto, de acordo com a especificação do trabalho, um processo não poderia simplesmente somar todos os números que ele recebeu. A cada soma realizada, os números sendo somados deveriam ser originados de processos diferentes. Para tal, a abordagem utilizada foi a criação de pares de processos e esses pares trocariam dados até que ambos ficassem com um único número restante.

A primeira questão dessa parte é a escolha dos pares. Digamos que N processos acabaram recebendo números na fase de distribuição de dados. Dessa forma, começando a contagem do 0, cada processo de index par teve como processo par o processo com index ímpar logo acima.

O problema que surge dessa abordagem é que se N for ímpar, um processo não terá um número par.

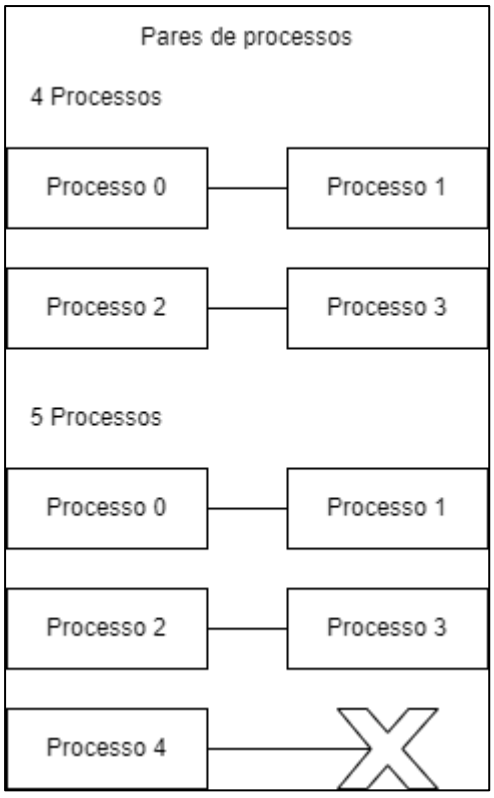


Figura 2: Pares de Processos

Vamos focar, primeiramente, em como ocorrerá a soma cruzada com um número N par. Primeiramente, um processo *i* que faça par com um processo *j* deverá saber quantos números o seu par tem para enviar. Se o processo *i* possui 3 números, ele só deverá enviar 2 números para *j*. O mesmo tem efeito para o processo *j*.

Um número *x* recebido pelo processo i deverá ser somado com o número no topo da sua própria pilha *y* e essa soma *z* também deverá ser inserida na pilha. Perceba que *y* não deverá estar mais na pilha já que a soma *z* já representará esse número com *x*.

Um número *x* enviado deverá ser retirado da pilha do processo de origem.

O pseudocódigo para essa operação é como o se segue:

```
somaCruzada():
Entrada:    O rank do meu processo: meuRank;
            O rank do processo par: parRank;
            A quantidade de números que o processo par irá enviar: qtNums
            A minha pilha de números: minhaPilha
Enquanto (tenho números para enviar ou tenho números para receber):
    Se tenho números para enviar:
        numeroEnviar = minhaPilha.topo()
    Se meuRank é par: //Primeira parte da comunicação
        Se tenho números para receber:
            recebeEEmpilha()
    senão:
        Se tenho números a enviar:
            Envie numeroEnviar para o meu processo par
    Se meuRank é par: //Segunda parte da comunicação
        Se tenho números a enviar:
            Envie numeroEnviar para o meu processo par
    senão:
        Se tenho números para receber:
            recebeEEmpilha
recebeEEmpilha():
    Entrada:    Minha pilha de números: minhaPilha
                Rank do processo de onde esperar: rankProcessoEnvio
numeroASomarCom = minhaPilha.topo()
numeroRecebido = Recebe número do processo rankProcessoEnvio
minhaPilha.empilhar(numeroASomarCom + numeroRecebido)
```

Perceba que, se o processo atual ainda tem um número para enviar, esse número é separado antes de possivelmente receber um número e somar esse número recebido ao número no topo da pilha. Como só será identificado que o processo atual deve enviar um número quando ele contém dois ou mais números na pilha, isso não afetará a operação de topo() na pilha ao receber um número depois.

Além disso, também perceba que o processo par primeiro recebe números e só depois envia enquanto o ímpar é o contrário. Isso é feito para intercalar a comunicação entre os processos e não causar um deadlock onde ambos estão tentando enviar ou receber ao mesmo tempo.

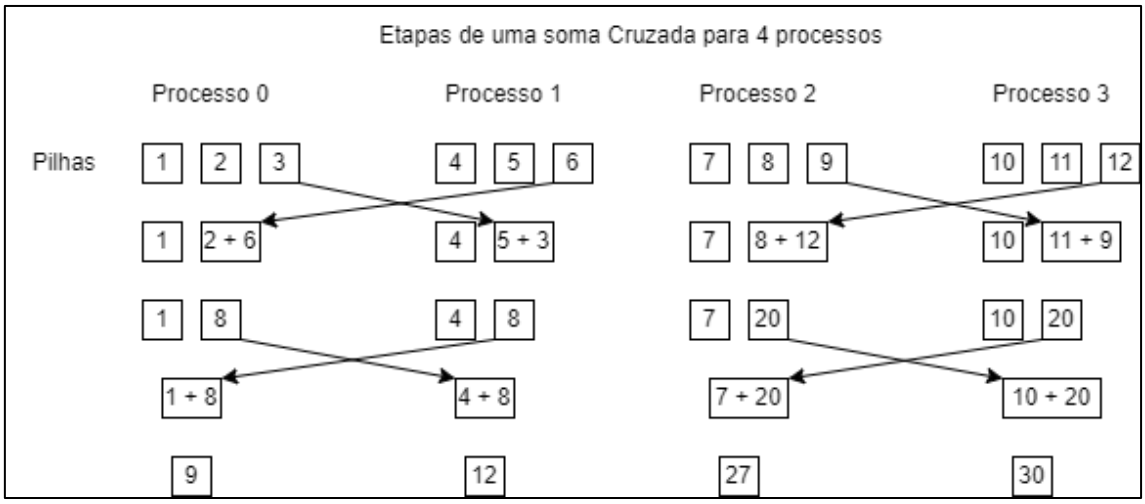


Figura 3: Exemplo de somas cruzadas

No caso em que N é ímpar, um processo sobrará sem um processo par específico. Para tratar esse cenário, digamos que esse processo sobrando possua U números. Desses U números, ele deve ficar com apenas 1, ou seja, $U - 1$ números devem ser somados por outros processos. Dessa forma, a abordagem tomada foi enviar cada um desses $U - 1$ números de forma cíclica para cada um dos outros processos que receberam números.

A Figura 4 mostra o procedimento em uma situação onde existem 5 processos. Após a soma cruzada dos processos pares, se faz necessário mais um procedimento para somar os números do processo sobrando até que sobre apenas um número em sua pilha.

Dessa forma, ainda teremos, no final, 5 processos que contém um número em sua pilha. Outra possibilidade seria enviar todos os números do processo sobrando para os outros processos de forma que sua pilha fique vazia. Esse procedimento também poderia ser aplicado para todos os processos que excedam o maior valor de 2^i tal que $2^i \leq N$ sendo N o número de processos que receberam números na fase de distribuição. Isso poderia ajudar na próxima fase de redução ao ter, com certeza, uma quantidade potência de 2 de processos. Entretanto, isso não foi feito no trabalho e permanecemos com N processos.

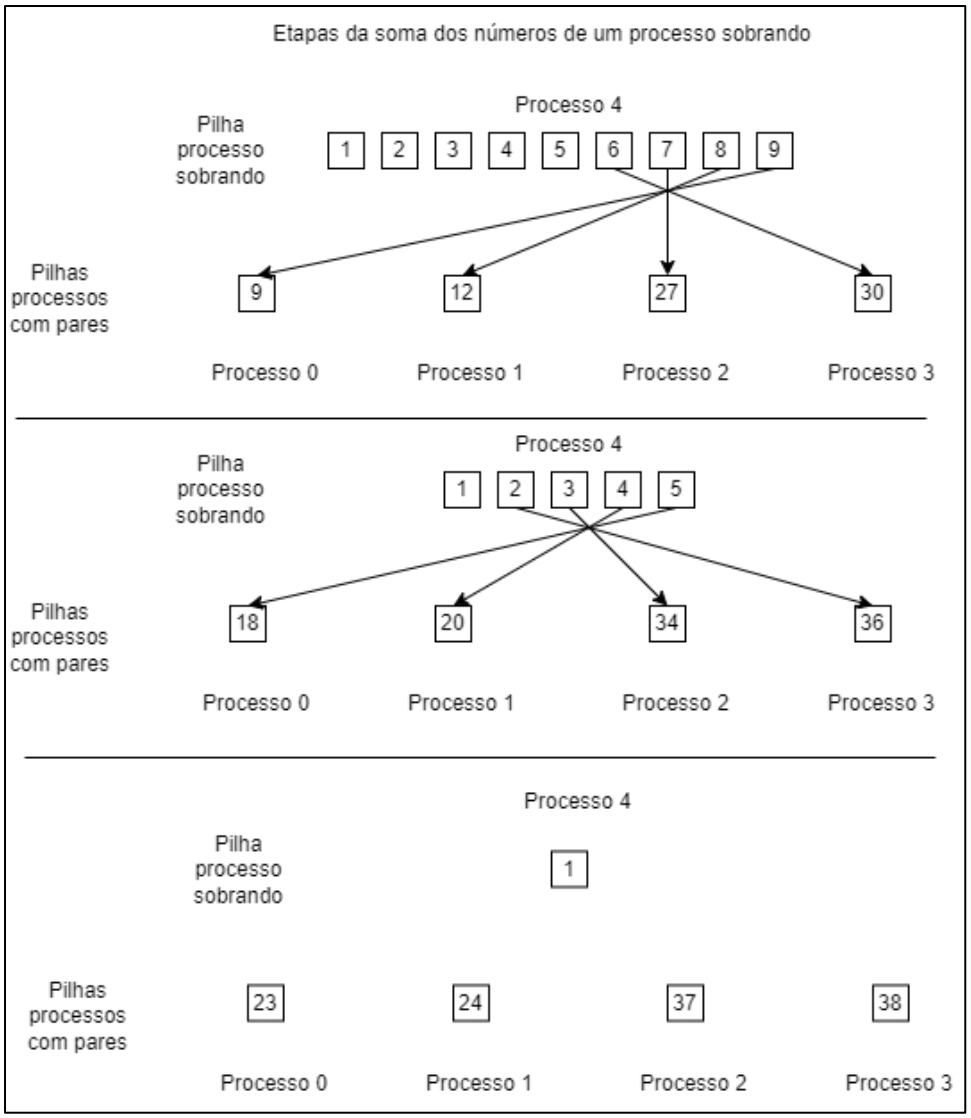


Figura 4: Soma dos números de um processo sobrando

6 – Fase de Redução

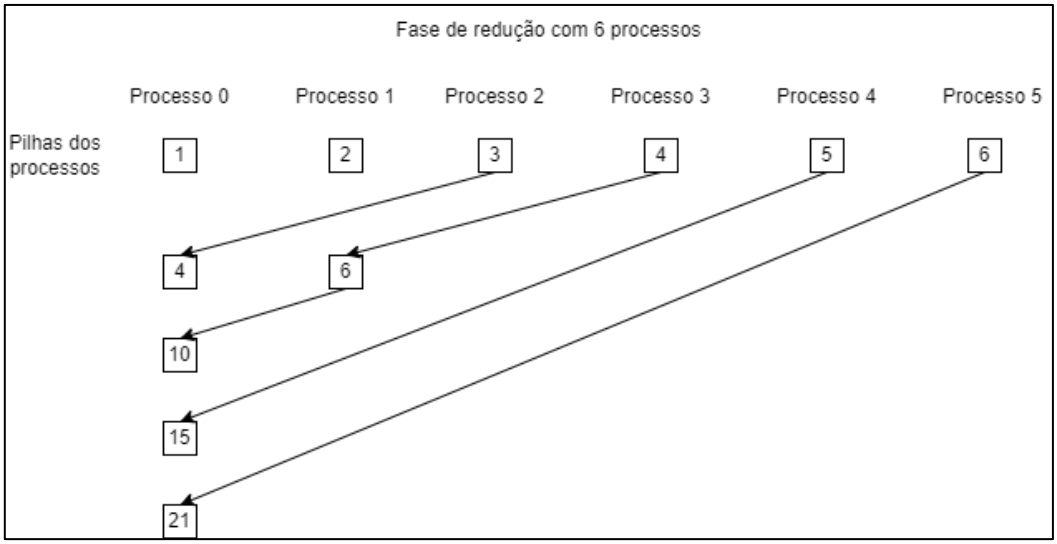
Agora que cada processo que recebeu algum número na fase de distribuição possui apenas um número na pilha, começamos a agregar esses resultados em um só. Um problema a ser resolvido nessa fase é que, caso a quantidade de processos N que receberam números seja diferente de uma potência de dois, mais um procedimento será necessário no final.

A abordagem tomada nessa fase do processo também é a criação de pares de processos. Entretanto, apenas um processo irá receber o número do outro processo e somá-lo ao seu. Uma vez que um processo enviou seu único número para outro processo, ele não participará de mais nenhuma fase da redução. A redução foi direcionada para que o processo de rank 0 contenha a soma final de todos os números.

Já que estão sendo formados pares de processos e que um dos processos de um par não voltará a participar das próximas fases, teremos um número mínimo de fases igual a $\lfloor \log_2 N \rfloor$ sendo N a quantidade de processos que contém um número em sua pilha.

Na primeira fase da redução, teremos exatamente $L = 2^{\lfloor \log_2 N \rfloor}$ processos sendo considerados. Para formar os pares, todos os processos cujo rank sejam menores que $\frac{L}{2}$ serão considerados processos de destino de números e os $\frac{L}{2}$ processos restantes enviarão seus números. Ao final, a quantidade de processos a serem levados em consideração na próxima fase será de $L' = \frac{L}{2}$. Essa conta continua até que L^* seja igual a 2, onde o processo 0 receberá do processo 1.

Além disso, caso $N > 2^{\lfloor \log_2 N \rfloor}$, teremos uma quantidade $K = N - 2^{\lfloor \log_2 N \rfloor}$ de processos que sobraram. Nesse caso, teremos uma última fase da redução onde todos esses K processos enviam seus números para o processo 0, terminando a agregação. A figura 5 exemplifica o funcionamento da redução.



Observando a última parte da agregação onde existem K processos sobrando, uma outra abordagem poderia ser feita tal que aconteça uma agregação local entre esses K processos. No final, somaríamos essa agregação local com a agregação dos $N - K$ primeiros processos.

7 – Testes e Discussão dos resultados

Para testar o desempenho da redução, foram geradas listas de números com tamanhos variados para quantidades diferentes de processos. Ao final, foram medidos o tempo médio de execução de 10 iterações para cada caso.

Importante dizer que o tempo medido para cada teste não inclui o tempo de distribuição dos dados já que essa fase, do jeito que foi implementada, é muito dependente da velocidade na qual o usuário fornece os números de entrada.

A Tabela 1 mostra o resultado das medições e o Gráfico 1 permite visualizar melhor os resultados.

Média do Tempo de 10 Execuções (milissegundos)				
Tamanho Entrada	Quantidade Processos			
	1	2	3	4
20	0.002	0.26	0.45	1.18
50	0.004	0.2	1.89	0.29
100	0.007	0.32	1.74	0.47
1000	0.041	0.62	3.69	1.28
2000	0.08	1.16	4.8	3.63
3000	0.119	1.77	6.17	4.62
5000	0.189	2.75	9.45	5.47
10000	0.341	5.84	24.53	10.56

Tabela 1: Tempo médio de execução para vários testes

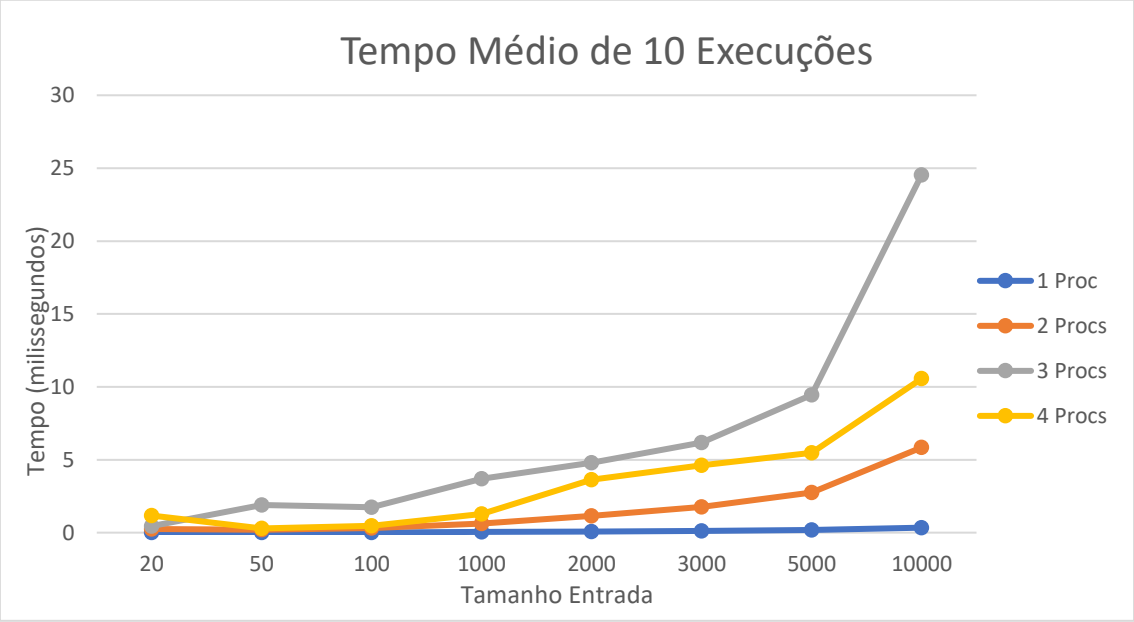


Gráfico 1: Tempo médio de execução para vários testes

Um caso menos comum que pode acontecer é o de apenas um processo possuir números ao definir que apenas um processo deva ser usado na soma. Nesse caso, nenhum tipo de comunicação MPI acontece e a soma dos números se torna sequencial. Dessa forma, de acordo com a tabela acima, podemos ver que a falta de comunicação tornou o programa sequencial bem mais rápido do que os outros.

Pode-se perceber também que, quanto mais processos são utilizados, maior o tempo de execução para todas as entradas. Um caso especial é o caso com 3 processos. O seu tempo de execução foi maior do que ao usar 4

processos. Isso pode estar sendo causado pelo fato de que haverá um processo sobrando na fase de soma cruzada e na fase de redução. Dessa forma, pode-se inferir que usar uma quantidade de processos diferente de uma potência de dois, usando as abordagens do trabalho, causará tempos de execução maior do que outras quantidades de processos próximas mas que sejam potência de dois.

Para justificar o fato do aumento do tempo de execução mesmo com mais processos, deve-se lembrar que, quando um processo se comunica com outro processo, ele envia apenas um número por vez. Dessa forma, quanto mais números ele deve enviar, mais comunicação deve ocorrer. Uma forma de resolver isso seria enviar arrays de números pré-definidos entre os processos para diminuir a quantidade de MPI_Send e MPI_Recv entre os processos.

Isso poderia ser melhor aproveitado na fase de distribuição de números e na fase da soma cruzada. Na fase de distribuição, seria necessário armazenar todos os números que um dado processo receberia para então enviá-los de uma só vez. Na fase da soma cruzada, poderíamos separar todos os números a serem enviados para um processo par e então enviá-los de uma só vez também.

8 – Conclusão

O objetivo do trabalho era implementar um algoritmo de redução com MPI que fizesse uso apenas de comunicação ponta-a-ponta com os métodos MPI_Send e MPI_Recv. Pode-se dizer que esse objetivo foi alcançado visto que foram definidas e implementadas todas as etapas desse processo.

Os resultados alcançados pelos testes mostraram que as abordagens utilizadas pelo trabalho podem ser melhoradas para que, ao utilizar mais processos, o tempo de execução, de fato, diminua, algo que é de se esperar de uma implementação paralelizada corretamente de uma tarefa sequencial.

Para finalizar, com este trabalho foi possível colocar em prática conceitos aprendidos durante as aulas e desenvolver um programa utilizando MPI. Para esse último, foi necessário levar em consideração que os processos não possuíam memória compartilhada, diferentemente de Pthreads ou OpenMP onde pode-se criar e utilizar variáveis globais no código, e potenciais problemas que poderiam ocorrer como deadlock.

9 – Referências

Slides das aulas

An Introduction to Parallel Programing; Pacheco, Peter; Elselvier; 2011