**MPI - Comunicação entre processos**

Vamos verificar agora como que se dá a comunicação entre processos no modelo MPI.

As duas primeiras instruções que veremos são a MPI\_Send e a MPI\_Recv. A primeira, a remetente, tem 6 parâmetros e a segunda, a destinatária, tem 7 parâmetros.

**MPI\_Send** (a, b , c, d, e, f)

**MPI\_Recv** (a, b , c, d, e, f, g)

* a = ponteiro para a estrutura de dados, via de regra um vetor
* b = quantidade de elementos na estrutura
* c = tipo de dado (data type) contido na estrutura
* d = rank do destinatário na MPI\_Send e do remetente na MPI\_Recv
* e = tag identificadora da mensagem
* f = communicator
* g = status da mensagem recebida (somente MPI\_Recv)

Exemplo:

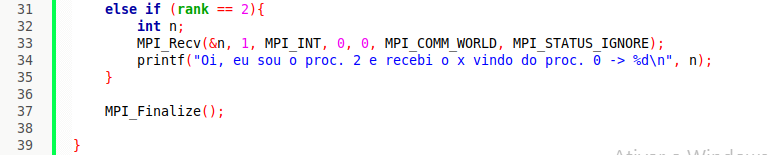
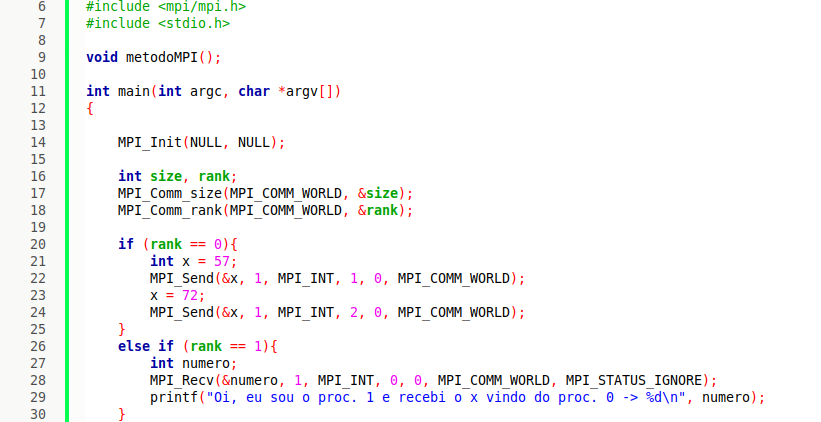


Figura 1

**Envio**

Nesse exemplo (), o processo 0 (linhas de 20 a 25) envia duas mensagens:

Mensagem enviada na linha 22:

**a = &x**: O parâmetro **a** é um ponteiro para a estrutura de dados a ser enviada ao processo destinatário. No caso é uma variável do tipo **int**.

**b = 1**: Quantidade de elementos int na estrutura.

**c = MPI\_INT**: Na api MPI, este é o *data type* identificador do tipo **int**. A tabela de correspondência entre os tipos pode ser vista em <https://mpitutorial.com/tutorials/mpi-send-and-receive/>.

**d = 1**: Identificador do processo destinatário da mensagem, no caso o destinatário é o processo 1.

**e = 0**: *Tag* identificadora da mensagem. Permite ao programador classificar as mensagens em diversos tipos ou diversos IDs.

**f =** communicator.

Mensagem enviada na linha 24:

Nessa mensagem só foi alterado o processo destinatário (agora é o 2) e o conteúdo da variável. Era 57 (linha 21) e agora é 72 (linha 23).

**Recepção**

O código referente à atuação do processo 1 está entre as linhas 26 e 30. É importante observar que a codificação da mensagem de recepção tem que estar casada com a codificação da mensagem de envio, ou seja, o receptor sabe de antemão o que deverá ser enviado para ele (só desconhece os dados residentes na estrutura de dados).

Mensagem de recepção na linha 28:

**a = &numero**: Estrutura enviada pelo remetente (na verdade envia-se o ponteiro). Aqui foi associada à variável local "numero".

**b** e **c** são idênticos aos da mensagem de envio.

**d = 0**: Identificador do processo remetente da mensagem, no caso o remetente é o processo 0.

**e** e **f** são idênticos aos da mensagem de envio.

**g =** Sinalizador do status da mensagem. Pode ser sucesso, erro ou outras condições. No caso, MPI\_STATUS\_IGNORE, está ignorando o status de retorno. Mais informações sobre o status da mensagem vide <https://mpitutorial.com/tutorials/dynamic-receiving-with-mpi-probe-and-mpi-status/>.

**Execução deste programa:**

Como pode ser observado no código da , o processo 0 envia um valor ao processo 1 (57) e outro ao processo 2 (72). Cada um dos processos imprime o valor recebido.

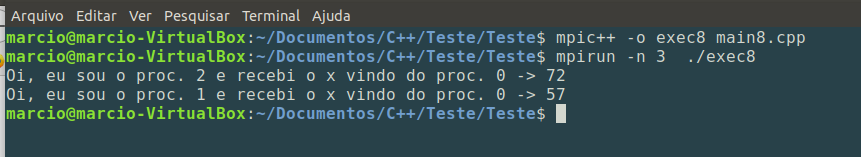


Figura 2

Observe que na chamada para execução foi solicitado o instanciamento de 3 processos. Se não houver especificação, como visto anteriormente, serão instanciados tantos quantos forem os núcleos do computador, mas esses restantes ficarão inoperantes (memória e tempo perdidos).

***Modus Operandi* do processo de troca de mensagens**

A troca de mensagens é intermediada pelo communicator. O remetente não atinge diretamente o destinatário, mas sim o communicator. A mensagem fica sob a tutela do communicator até que o destinatário a solicite.



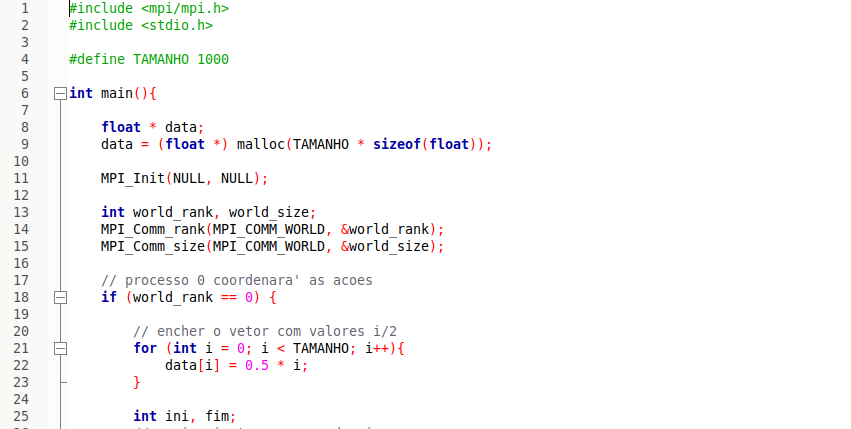
Figura 3

De uma forma simplificada, a troca de mensagens ocorre como ilustrado na . O processo X envia a mensagem para Y, mas, como os processos têm (ou podem ter) operações assíncronas, Y pode não estar preparado para recebê-la neste momento; por outro lado, Y pode solicitar a mensagem antes de estar disponível. O communicator, portanto, faz este "meio-de-campo". E no caso de Y solicitar a mensagem antes de X ter enviado, o processo (Y) fica travado no aguardo.

**Um processo enviar um vetor a outro processo**

Em modelo de colaboração entre processos, caso os processos não forem dividir a tarefa segundo uma regra clara e equitativa, é necessário que um dos processos assuma a posição de mestre, ficando os demais na posição de escravos (modelo *master-slave*).

Vamos supor um vetor em que os processos deverão colaborar para achar a somatória. Caso o vetor fosse dividido de forma equitativa entre os processos, como dito no parágrafo anterior, não haveria a necessidade de um processo coordenador, pois cada um, em função, por exemplo, do seu número de *rank*, poderia calcular sua própria fatia do vetor. Mas, e se esta quantidade não for balanceada e ainda por cima variável em cada execução do algoritmo? Nesse caso alguém precisa centralizar as informações e distribuir aos processos.



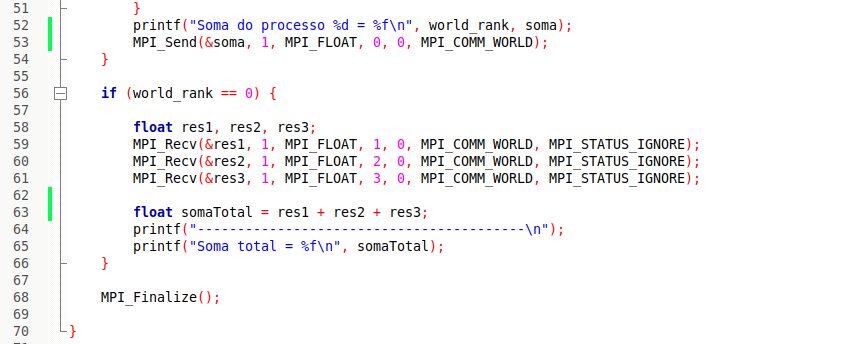


Figura 4

O programa da foi dimensionado para exatos 4 processos.

Nas linhas 8 e 9 é instanciado o vetor *float* de 1000 posições. É importante observar que este vetor deve ser instanciado antes do ambiente MPI iniciar (MPI\_Init), caso contrário cada processo criará uma cópia do vetor. O vetor pertence, portanto, ao processo base (0).

Da linha 18 à 40, o processo 0 coordena as ações que os demais processos irão desempenhar.

Nas linhas 21 a 23, o processo 0 atribui a cada posição do vetor a metade do valor do seu índice.

Nas linhas 27 à 36 é definido um intervalo (índices inicial e final) que o processo irá trabalhar no vetor.

Linha 37 - enviado o índice inicial

Linha 38 - enviado o índice final

Linha 39 - enviado o vetor ao processo que vai fazer a soma.

Da linha 42 à 54, os processos receptores entram em ação:

Nas linhas 43, 44 e 45 são recebidos os índices de início e fim do vetor que o processo deve trabalhar (os trechos são chamados de *chunks*).

Nas linhas 46 e 47 é recebido o vetor.

Da linha 48 à 51 é realizada a somatória do trecho (*chunk*) correspondente.

Na linha 53 é enviado o resultado da somatória ao processo base (0).

Da linha 56 à 66, o processo base consolida as somas parciais.

O resultado desse programa é o seguinte:

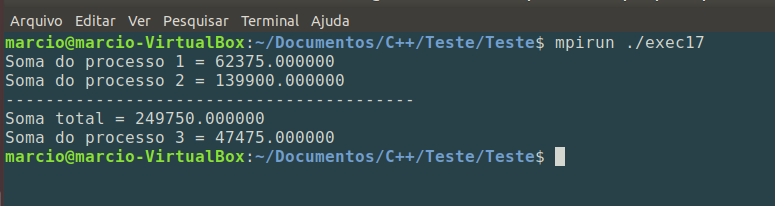


Figura 5

O processo de impressão é extremamente demorado em relação ao processamento em memória, de forma que a temporização (conforme as instruções do programa) não segue o correto sincronismo.

**Broadcast (MPI\_Bcast, MPI\_Reduce)**

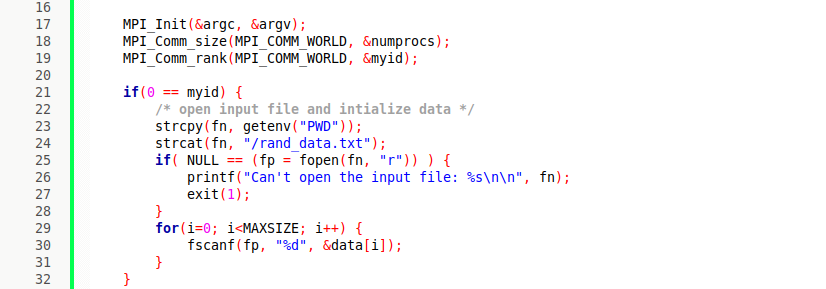
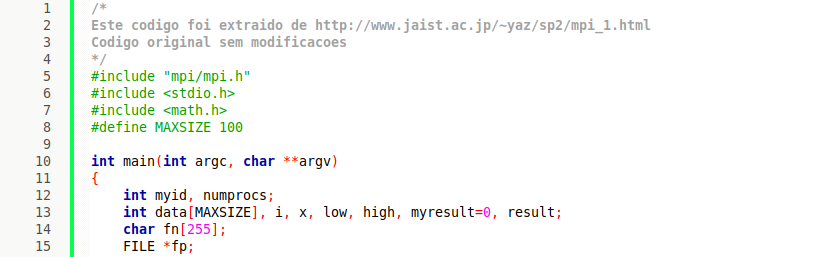
Vimos anteriormente como enviar mensagem de um processo para outro. No programa da , o processo 0 organizou uma divisão do vetor em 3 partes (*chunks*) e enviou uma parte para o processo 1, outra para o 2 e a última para o 3 para que cada um deles somassem os elementos desses trechos e devolvessem os resultados.

No entanto, problemas em que se tem um conjunto de elementos do mesmo tipo e se deseja submetê-los (todos) ao mesmo tipo de operação, pode-se dividir o conjunto equitativamente entre os processos. Outro aspecto a considerar é a preocupação com a quantidade de processos. E se quisermos executar o programa em outro *cluster* de computadores? Teremos que alterar o código? Vamos ver como generalizar a solução.

*Broadcast* (do inglês: irradiar, difundir) é o procedimento de enviar a mesma mensagem a todos os elementos de um conjunto. Ou seja, um dado *array* pode ser enviado a todos os processos e cada um calcular, em função do seu *rank* e a quantidade de processos, qual o intervalo que lhe caberia.

(vide: <https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/>)

Vamos analisar o programa abaixo:



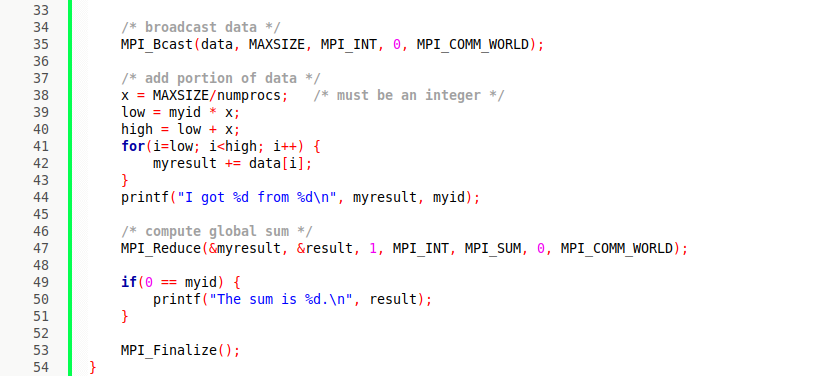


Figura 6

Este código foi extraído do site referenciado na linha 2. Cria um vetor (linha 3) e preenche com dados de um arquivo chamado "rand\_data.txt" (linha 24). Esse arquivo tem números inteiros e pode ser obtido no site.

Da linha 21 à 32 o processo 0 lê o arquivo e preenche o vetor.

Na linha 35 vem a instrução responsável pelo envio do vetor a todos os processos.

MPI\_Bcast(

**data**, (conjunto de dados)

**MAXSIZE**, (quantidade de elementos)

**MPI\_INT**, (*data type*)

**0**, (processo originador da dissiminação)

**MPI\_COMM\_WORLD** (communicator)

)

Os primeiros 2 parâmetros são nomes das variáveis do código.

Observe que a instrução, originada pelo processo 0, não está no trecho exclusivo do processo 0. Todos os processos recebem essa mensagem (inclusive o próprio originador).

Da linha 37 à 44 cada processo vai cuidar da sua parte.

Linha 38 = calcula a quantidade de elementos que irá trabalhar

Linha 39 = calcula a primeira posição do seu *chunk*.

Linha 40 = calcula a última posição do seu *chunk*.

Linhas 41, 42, 43 = faz a soma[[1]](#footnote-1).

Agora como fazemos para juntar todos os resultados e fazer a soma final?

No exemplo anterior, o processo executor devolve o resultado ao processo 0 via MPI\_Send. Neste exemplo utilizado a instrução MPI\_Reduce que faz a consolidação (linha 47).

MPI\_Reduce(

**&myresult**, (variáveis locais - devem ter os mesmos nomes)

**&result**, (variável aonde será depositado o resultado global)

**1**, (quantidade de elementos)

**MPI\_INT**, (data type)

**MPI\_SUM**, (tipo de operação - soma)

**0**, (processo destinatário da mensagem0

**MPI\_COMM\_WORLD** (communicator)

)

(vide <https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>).

A execução desse programa resulta a seguinte saída:

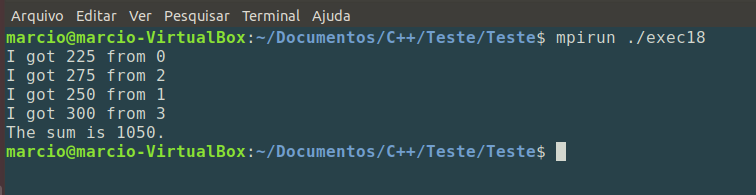


Figura 7

**Exercício para entregar (elaborado por prof. Calebe):**

Construa uma solução em MPI, com apenas MPI\_Bcast e MPI\_Reduce, que procura o maior elemento em um conjunto de dados.

Sua solução deve ter:

* mais que 2 processos em execução
* todos os ranks devem realizar a procura (inclusive o rank 0)
* o tamanho do vetor pode ser proporcional a quantidade de ranks disponíveis
* construa seu conjunto de dados utilizando float
* apenas um rank imprime o resultado final
* Tarefa individual.

*10001001010111100101001001*

1. Observe que todas as variáveis foram declaradas antes de iniciado o ambiente MPI. Quando um processo se utiliza de uma dessas variáveis ele cria uma cópia local, incluindo os ponteiros. O que não é replicado é o objeto para onde o ponteiro aponta (no nosso caso o vetor "data"). [↑](#footnote-ref-1)