



Escola Supercomputador Santos Dumont - 2025

Programação com MPI

Escola de Computação Santos Dumont - 2025

Ementa

- Exemplo de um programa MPI
- Rotinas básicas de gerenciamento.
- Rotinas básicas de comunicação ponto a ponto
- Correspondência entre os tipos MPI e C
- Identificando o tamanho e as informações da mensagem recebida
- Estudo de caso: método do trapézio

Escola de Computação Santos Dumont - 2025

Bibliografia



<https://www.casadocodigo.com.br/products/livro-programacao-paralela>



MPI - Introdução

Pré-requisitos

- Para os exemplos apresentados neste curso, será necessário um sistema operacional do tipo Linux.
- Preferencialmente, você pode utilizar as distribuições Fedora, Ubuntu ou WSL (no Windows).
- Mas os exemplos e comandos podem ser utilizados em outras distribuições desde que os pacotes necessários estejam corretamente instalados.
- Certifique-se de ter um compilador instalado. O MPI geralmente funciona com compiladores como gcc, g++, ou gfortran.

Pré-requisitos

- Instale uma implementação do MPI, as mais comuns são:
 - MPICH: Uma implementação amplamente usada.
 - OpenMPI: Populares em sistemas de alto desempenho.
- Em distribuições Linux do tipo Debian, você pode instalar usando:

```
$ sudo apt install mpich # MPICH
```

```
$ sudo apt install openmpi-bin openmpi-common # OpenMPI
```

- Os exemplos apresentados neste curso estão disponíveis no repositório
<https://github.com/Programacao-Paralela-e-Distribuida/MPI>

Compilando MPI

- De uma maneira simplificada, um programa em MPI pode ser compilado e executado com os seguintes comandos em um terminal de um sistema operacional do tipo Linux utilizando-se o compilador gcc:

```
$ mpicc -o teste mpi_simples.c
```

```
$ mpirun -np 4 ./teste
```

- Onde **np** indica o número de processos com que o programa deve ser executado.
- A opção **-hostfile** define quais máquinas utilizar em um cluster.

Escola de Computação Santos Dumont - 2025

Google Colab

- Os exemplos, notebooks e slides do minicurso estão disponíveis no seguinte endereço:
- <https://github.com/Programacao-Paralela-e-Distribuida/MPI/>

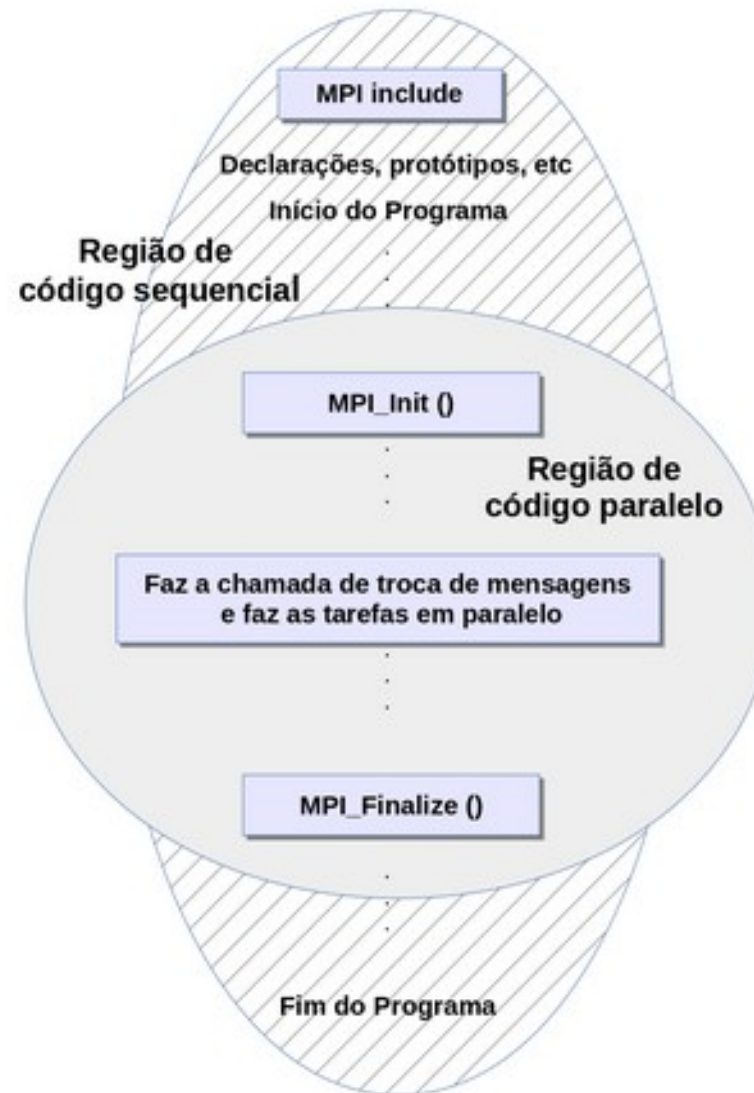
Programando com MPI

- Todo programa em MPI deve conter a seguinte diretiva para o pré-processador:
`#include "mpi.h"`
- Este arquivo, `mpi.h`, contém as definições, macros e funções de protótipos de funções necessários para a compilação de um programa MPI.
- Antes de qualquer outra função MPI ser chamada, a função `MPI_Init()` deve ser chamada pelo menos uma vez.
- Seus argumentos são os ponteiros para os parâmetros do programa principal, `argc` e `argv`.

Programando com MPI

- A função `MPI_Init()` permite que o sistema realize as operações de preparação necessárias para que a biblioteca MPI seja utilizada.
- Ao término do programa a função `MPI_Finalize()` deve ser chamada.
- Esta função limpa qualquer pendência deixada pelo MPI, p. ex, recepções pendentes que nunca foram completadas.
- Tipicamente, um programa em MPI deve ter o seguinte leiaute:

Programa MPI



Programa MPI

```
...  
#include "mpi.h"  
...  
main(int argc, char** argv) {  
...  
/* Nenhuma função MPI pode ser chamada antes deste ponto */  
MPI_Init(&argc, &argv);  
...  
MPI_Finalize();  
/* Nenhuma função MPI pode ser chamada depois deste ponto*/  
...  
}
```

Quem sou eu?

- O MPI tem a função `MPI_Comm_Rank()` que retorna o **ranque** (rank) de um processo no seu segundo argumento.
- Sua sintaxe é:

```
int MPI_Comm_rank(MPI_Comm com, int *ranque)
```
- O primeiro argumento é um **comunicador**. Essencialmente um comunicador é uma coleção de processos que podem enviar mensagens entre si.
- Para os programas básicos, o único comunicador necessário é `MPI_COMM_WORLD`, que é pré-definido no MPI e consiste de todos os processos executando quando a execução do programa começa.

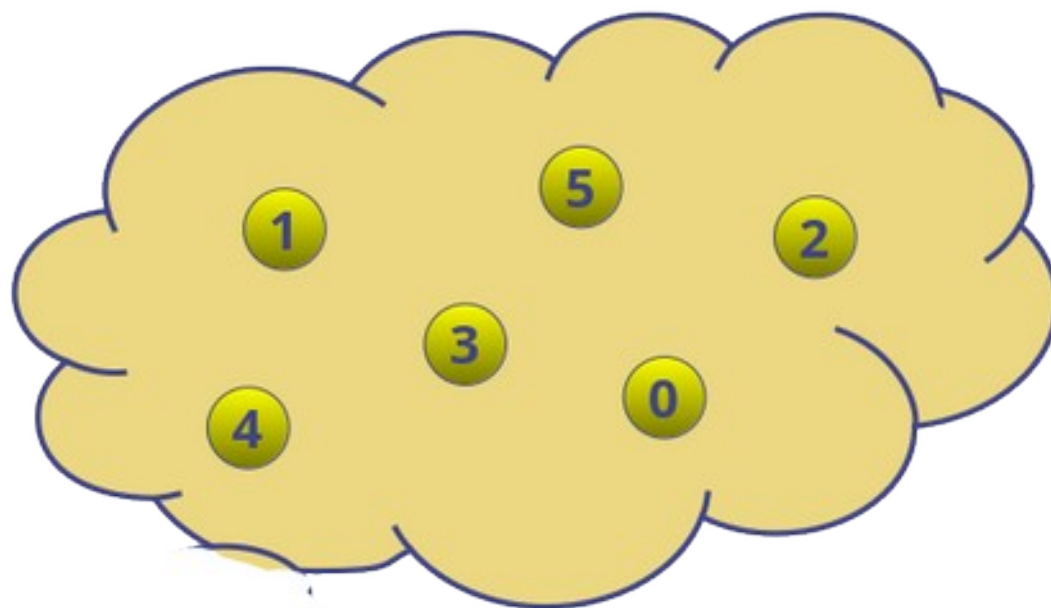
Quantos processos somos?

- Muitas construções em nossos programas também dependem do número de processos executando o programa.
- O MPI oferece a função `MPI_Comm_size()` para determinar este valor.
- Essa função retorna o número de processos em um comunicador no seu segundo argumento.
- Sua sintaxe é:

```
int MPI_Comm_size(MPI_Comm com, int *num_procs)
```

Ranke

MPI_COMM_WORLD



Funções Básicas

- Abortando um programa:

```
int MPI_Abort(MPI_Comm com, int erro)
```

- Identificando a versão do MPI:

```
int MPI_Get_version(int *versao, int *subversao)
```

- Recuperando o nome do computador:

```
int MPI_Get_processor_name (char *nome, int *comprimento)
```


Funções Básicas

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]) {    /* mpi_funcoes.c */
    int meu_rank, num_procs;
    int versao, subversao, aux, ret;
    char maquina[MPI_MAX_PROCESSOR_NAME];
    /* Inicia o MPI. Em caso de erro aborta o programa */
    ret = MPI_Init(&argc, &argv);
    if (ret != MPI_SUCCESS) {
        printf("Erro ao iniciar o programa MPI. Abortando.\n");
        MPI_Abort(MPI_COMM_WORLD, ret);
    }
    /* Imprime a versão e subversão da biblioteca MPI */
    MPI_Get_version(&versao, &subversao);
    printf("Versão do MPI = %d Subversão = %d \n", versao, subversao);
    /* Obtém o rank e número de processos em execução */
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
    /* Define o nome do computador onde o processo está executando */
    MPI_Get_processor_name(maquina, &aux);
    printf("Número de tarefas = %d Meu rank = %d Executando em %s\n",
           num_procs, meu_rank, maquina);
    /* Finaliza o MPI */
    MPI_Finalize();
    return(0);
}
```

Funções Básicas

```
$ mpicc -o teste mpi_funcoes.c
```

```
$ mpirun -n 2 ./teste
```

Versão do MPI = 3 Subversão = 1

Versão do MPI = 3 Subversão = 1

Número de tarefas = 2 Meu ranque = 1 Executando em Incc-2025

Foram gastos 0.000100 segundos com precisão de 1.000e-09 segundos

Número de tarefas = 2 Meu ranque = 0 Executando em Incc-2025

Foram gastos 0.000135 segundos com precisão de 1.000e-09 segundos

Medindo o tempo de execução

- A função `MPI_Wtime()` retorna (em precisão dupla) o tempo total em segundos decorrido desde um instante determinado no passado.
- Esse instante é dependente de implementação, mas deve sempre o mesmo para uma dada implementação.
- A função `MPI_Wtick()` retorna (em precisão dupla) a resolução em segundos da função `MPI_Wtime()`.
- Um exemplo de uso dessas funções pode ser visto a seguir.

Escola de Computação Santos Dumont - 2025

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[ ]) {    /* mpi_wtime.c */
double tempo_inicial, tempo_final, a;
    tempo_inicial = MPI_Wtime();
    for(long int i = 0; i < 1000000000000; i++) {
        a = (double) i;    /* Realiza o trabalho em paralelo */
    }
    tempo_final = MPI_Wtime();
    printf("Foram gastos %3.6f segundos para calcular a = %3.0f com \
precisão de %3.3e segundos\n", tempo_final-tempo_inicial, a, MPI_Wtick ());
    return(0);
}
```

Medindo o tempo de execução

```
$ mpicc -o teste mpi_wtime.c
```

```
$ mpirun -n 2 ./teste
```

Foram gastos 19.818745 segundos para calcular $a = 9999999999$ com precisão de $1.000e-09$ segundos

Foram gastos 19.900254 segundos para calcular $a = 9999999999$ com precisão de $1.000e-09$ segundos



Escola Supercomputador Santos Dumont - 2025

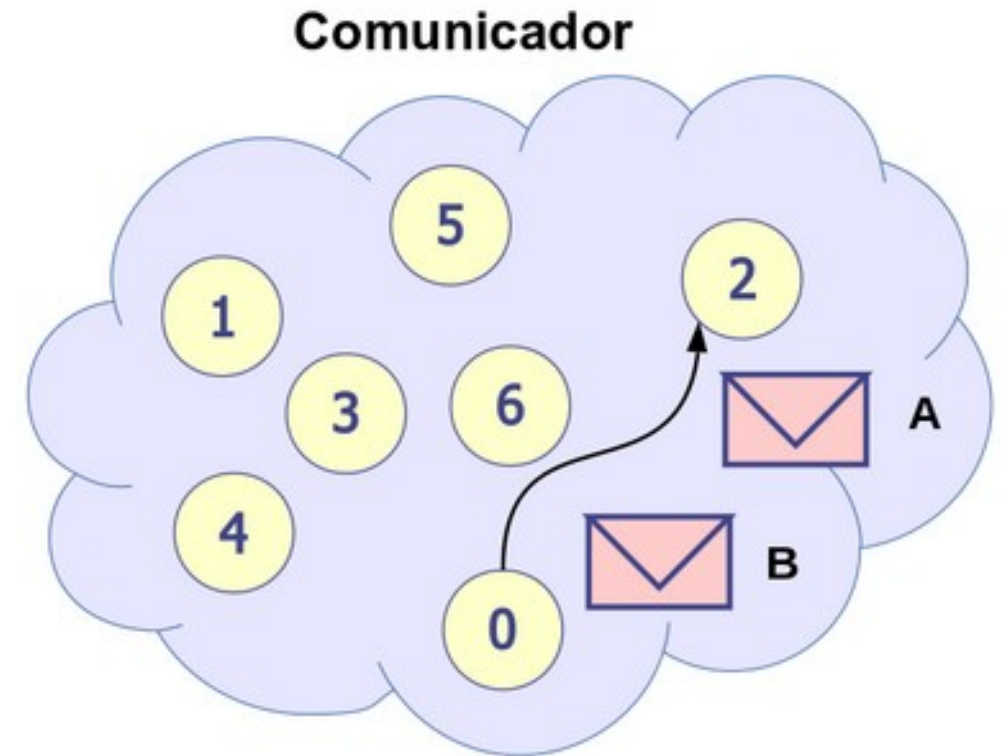
Comunicação Ponto a Ponto

Mensagem MPI

- Mensagem = Dados + Envelope
- Para que a mensagem seja comunicada com sucesso, o sistema deve anexar alguma informação aos dados que o programa de aplicação deseja transmitir.
- Essa informação adicional forma o envelope da mensagem, que no MPI contém a seguinte informação:
 - O **ranque** do processo origem.
 - O **ranque** do processo destino.
 - Uma **etiqueta** especificando o tipo da mensagem.
 - Um **comunicador** definindo o domínio de comunicação.

Ordem das Mensagens

- As mensagens não ultrapassam umas às outras.
- Por exemplo, se o processo com ranque 0 enviar duas mensagens sucessivas A e B, e o processo com ranque 2 chamar duas rotinas de recepção que combinam com qualquer uma das mensagens, a ordem das mensagens é preservada, sendo que A será sempre recebida antes de B.



Comunicação Ponto-a-Ponto

- O mecanismo de troca de mensagens ponto-a-ponto no MPI é realizado pelas funções `MPI_Send()` e `MPI_Recv()`.
- Quando dois processos estão se comunicando utilizando `MPI_Send()` e `MPI_Recv()`, a importância do uso do **comunicador** aumenta quando módulos de um programa são desenvolvidos de forma independente.
- Por exemplo, ao desenvolver uma biblioteca para resolver um sistema de equações lineares, você pode criar um comunicador exclusivo para o solucionador linear.
- Isso evita conflitos entre as mensagens, mesmo que etiquetas idênticas sejam utilizadas em outros módulos.

Comunicação Ponto-a-Ponto

- Vamos utilizar por enquanto o comunicador pré-definido `MPI_COMM_WORLD`, que inclui todos os processos ativos desde o início da execução do programa.
- A rotina `MPI_Send()` envia a mensagem para um determinado processo e a rotina `MPI_Recv()` recebe a mensagem de um processo.
- Ambas são operações **bloqueantes**:
 - Envio bloqueante: aguarda até que a mensagem seja completamente copiada das variáveis do programa do usuário para os buffers de envio, seja do sistema operacional ou da biblioteca MPI.
 - Recepção bloqueante: aguarda até que a mensagem recebida esteja totalmente armazenada nas variáveis do programa do usuário e pronta para uso.

Comunicação Ponto-a-Ponto

- A maioria das funções MPI é do tipo inteiro e retorna um código de erro ao final da chamada.
- Contudo, como a maioria dos programadores em 'C', ao longo deste curso, nós vamos ignorar este código em quase todos exemplos apresentados.
- Apesar disso, em aplicações profissionais, é recomendável implementar o tratamento de erro das funções para identificar problemas durante a execução dos programas.

MPI_Send()

```
int MPI_Send(void* mensagem, int cont, MPI_Datatype tipo_mpi, int destino,  
int etiq, MPI_Comm com)
```

- **mensagem**: endereço inicial da mensagem a ser enviada.
- **cont**: número de elementos da mensagem.
- **tipo_mpi**: tipo de dados da mensagem.
- **destino**: ranque do processo destino da mensagem.
- **etiq**: etiqueta da mensagem.
- **com**: comunicador do contexto da mensagem.

MPI_Send()

- A mensagem a ser enviada está armazenada em uma posição de memória definida pelo ponteiro `*mensagem`.
- A mensagem é um vetor com `cont` elementos do mesmo tipo, especificado pelo argumento `tipo_mpi`.
- Esses dois parâmetros, combinados, permitem que o sistema determine corretamente o tamanho da mensagem a ser enviada.
- Para evitar ambiguidades entre arquiteturas diferentes, o MPI define uma lista de tipos pré-definidos, tais como `MPI_CHAR`, `MPI_INT`, `MPI_FLOAT`, `MPI_BYTE`, `MPI_LONG`, `MPI_UNSIGNED_CHAR`, etc.

MPI_Send()

- O argumento **destino** é o ranque do processo para o qual a mensagem será enviada. Não há um “coringa” para o destino; ele deve sempre ser definido de forma única e explícita.
- O **etiq** é um inteiro usado pelo processo de destino para diferenciar mensagens distintas enviadas pelo mesmo processo de origem.
- O parâmetro **com** define, por meio do comunicador, o contexto da comunicação e os processos participantes do grupo. O comunicador padrão é **MPI_COMM_WORLD** e, por enquanto, será o único comunicador que utilizaremos.

MPI_Recv()

`int MPI_Recv(void* mensagem, int cont, MPI_Datatype tipo_mpi, int origem, int etiq, MPI_Comm com, MPI_Status* estado)`

- **mensagem:** endereço inicial onde a mensagem vai ser armazenada.
- **cont:** número máximo de elementos a serem recebidos.
- **tipo_mpi:** tipo de dados esperado na mensagem.
- **origem:** ranque do processo origem.
- **etiq:** etiqueta esperada da mensagem.
- **com:** comunicador do contexto da mensagem.
- **estado:** estrutura auxiliar com informações como o remetente e a etiqueta da mensagem.

MPI_Recv()

- A mensagem recebida vai ser armazenada em uma posição de memória definida pelo ponteiro `*mensagem`.
- A mensagem recebida poderá ter, no máximo, `cont` elementos do do tipo `tipo_mpi`.
- O argumento `origem` é o ranque do processo do qual estamos esperando a mensagem.
- O MPI permite que `origem` seja um coringa (*), neste caso usamos `MPI_ANY_SOURCE` neste parâmetro.
- Reforçamos que não há um “coringa” para o destino da mensagem.

MPI_Recv()

- `etiq` é especificado pelo usuário para distinguir as mensagens de um único processo.
- O MPI garante que inteiros entre 0 e 32767 possam ser usados como etiquetas, mas o valor máximo é dependente de implementação.
- Existe um coringa, `MPI_ANY_TAG`, que a função `MPI_Recv` pode usar como etiqueta.
- O último argumento de `MPI_Recv()`, chamado de `estado`, fornece informações detalhadas sobre os dados efetivamente recebidos.

Comunicação ponto-a-ponto

- Esse argumento corresponde a uma estrutura que inclui os seguintes campos principais:
 - **MPI_SOURCE**: indica o ranque do processo remetente.
 - **MPI_TAG**: identifica a etiqueta da mensagem recebida.
 - **MPI_ERROR**: Informa se a mensagem foi recebida corretamente ou não.
- Por exemplo, se a origem da recepção foi especificada como **MPI_ANY_SOURCE**, o campo **MPI_SOURCE** conterá o ranque do processo que enviou a mensagem.
- Se a etiqueta de recepção foi especificada como **MPI_ANY_TAG**, o campo **MPI_TAG** contém a etiqueta da mensagem recebida.

Comunicação ponto-a-ponto

- Para que a comunicação entre dois processos **A** e **B** no MPI ocorra corretamente, os argumentos usados por `MPI_Send()` no processo **A** devem ser rigorosamente compatíveis com os usados por `MPI_Recv()` no processo **B**.
- Isso ocorre porque o MPI utiliza esses argumentos para estabelecer uma correspondência entre o envio e o recebimento das mensagens.
- Qualquer incompatibilidade pode resultar em erros, mensagens perdidas ou comportamentos inesperados.
- Em primeiro lugar, o mesmo comunicador deve ser usado em ambos processos para que a mensagem seja transferida corretamente.

Comunicação ponto-a-ponto

- No processo **A**, o argumento **destino** em **MPI_Send()** deve corresponder ao ranque do processo **B** (identificador único no comunicador).
- No processo **B**, o argumento **origem** em **MPI_Recv()** deve ser o ranque do processo **A** ou um coringa, como **MPI_ANY_SOURCE**.
- Ambos os processos devem usar a mesma etiqueta para identificar a mensagem.
- A etiqueta permite que um processo receba mensagens específicas, mesmo quando há múltiplas mensagens em trânsito.
- Se o coringa **MPI_ANY_TAG** for usado no processo **B**, qualquer etiqueta será aceita.

Exemplo

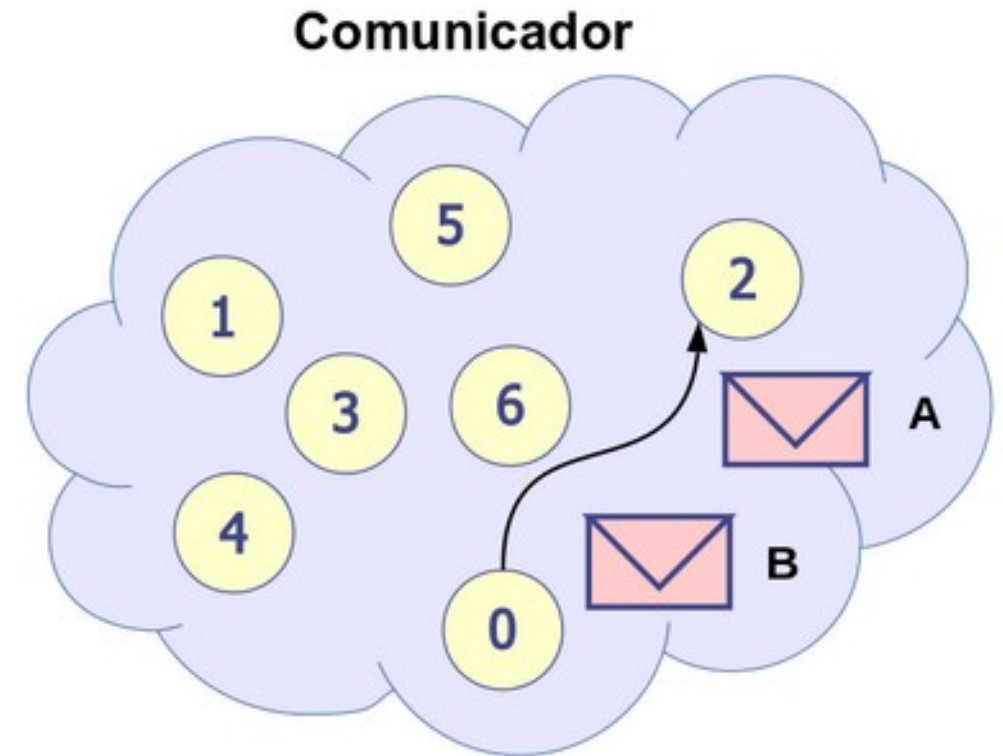
```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char** argv) {
    int meu_ranque, num_procs, origem, destino, etiq=0;
    char mensagem[100];
    MPI_Status estado;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    if (meu_ranque != 0){
        sprintf(msg, "Processo %d está vivo!", meu_ranque);
        destino = 0;
        MPI_Send(mensagem, strlen(mensagem)+1, MPI_CHAR, destino, etiq, MPI_COMM_WORLD);
    }
    else{
        for (origem=1; origem < num_procs; origem++) {
            MPI_Recv(mensagem, 100, MPI_CHAR, origem, etiq, MPI_COMM_WORLD, &estado);
            printf("%s\n", mensagem);
        }
    }
    MPI_Finalize( );
}
```

Mensagem MPI

- Mensagem = Dados + Envelope
- Para que a mensagem seja comunicada com sucesso, o sistema deve anexar alguma informação aos dados que o programa de aplicação deseja transmitir.
- Essa informação adicional forma o envelope da mensagem, que no MPI contém a seguinte informação:
 - O **ranque** do processo origem.
 - O **ranque** do processo destino.
 - Uma **etiqueta** especificando o tipo da mensagem.
 - Um **comunicador** definindo o domínio de comunicação.

Ordem das Mensagens

- As mensagens não ultrapassam umas às outras.
- Por exemplo, se o processo com ranque 0 enviar duas mensagens sucessivas A e B, e o processo com ranque 2 chamar duas rotinas de recepção que combinam com qualquer uma das mensagens, a ordem das mensagens é preservada, sendo que A será sempre recebida antes de B.



Comunicação Ponto-a-Ponto

- O mecanismo de troca de mensagens ponto-a-ponto no MPI é realizado pelas funções `MPI_Send()` e `MPI_Recv()`.
- Quando dois processos estão se comunicando utilizando `MPI_Send()` e `MPI_Recv()`, a importância do uso do **comunicador** aumenta quando módulos de um programa são desenvolvidos de forma independente.
- Por exemplo, ao desenvolver uma biblioteca para resolver um sistema de equações lineares, você pode criar um comunicador exclusivo para o solucionador linear.
- Isso evita conflitos entre as mensagens, mesmo que etiquetas idênticas sejam utilizadas em outros módulos.

Comunicação Ponto-a-Ponto

- Vamos utilizar por enquanto o comunicador pré-definido `MPI_COMM_WORLD`, que inclui todos os processos ativos desde o início da execução do programa.
- A rotina `MPI_Send()` envia a mensagem para um determinado processo e a rotina `MPI_Recv()` recebe a mensagem de um processo.
- Ambas são operações **bloqueantes**:
 - Envio bloqueante: aguarda até que a mensagem seja completamente copiada das variáveis do programa do usuário para os buffers de envio, seja do sistema operacional ou da biblioteca MPI.
 - Recepção bloqueante: aguarda até que a mensagem recebida esteja totalmente armazenada nas variáveis do programa do usuário e pronta para uso.

Comunicação Ponto-a-Ponto

- A maioria das funções MPI é do tipo inteiro e retorna um código de erro ao final da chamada.
- Contudo, como a maioria dos programadores em 'C', ao longo deste curso, nós vamos ignorar este código em quase todos exemplos apresentados.
- Apesar disso, em aplicações profissionais, é recomendável implementar o tratamento de erro das funções para identificar problemas durante a execução dos programas.

MPI_Send()

```
int MPI_Send(void* mensagem, int cont, MPI_Datatype tipo_mpi, int destino,  
int etiq, MPI_Comm com)
```

- **mensagem**: endereço inicial da mensagem a ser enviada.
- **cont**: número de elementos da mensagem.
- **tipo_mpi**: tipo de dados da mensagem.
- **destino**: ranque do processo destino da mensagem.
- **etiq**: etiqueta da mensagem.
- **com**: comunicador do contexto da mensagem.

MPI_Send()

- A mensagem a ser enviada está armazenada em uma posição de memória definida pelo ponteiro `*mensagem`.
- A mensagem é um vetor com `cont` elementos do mesmo tipo, especificado pelo argumento `tipo_mpi`.
- Esses dois parâmetros, combinados, permitem que o sistema determine corretamente o tamanho da mensagem a ser enviada.
- Para evitar ambiguidades entre arquiteturas diferentes, o MPI define uma lista de tipos pré-definidos, tais como MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_BYTE, MPI_LONG, MPI_UNSIGNED_CHAR, etc.

MPI_Send()

- O argumento **destino** é o ranque do processo para o qual a mensagem será enviada. Não há um “coringa” para o destino; ele deve sempre ser definido de forma única e explícita.
- O **etiq** é um inteiro usado pelo processo de destino para diferenciar mensagens distintas enviadas pelo mesmo processo de origem.
- O parâmetro **com** define, por meio do comunicador, o contexto da comunicação e os processos participantes do grupo. O comunicador padrão é **MPI_COMM_WORLD** e, por enquanto, será o único comunicador que utilizaremos.

MPI_Recv()

```
int MPI_Recv(void* mensagem, int cont, MPI_Datatype tipo_mpi, int origem, int etiq,  
MPI_Comm com, MPI_Status* estado)
```

- **mensagem**: endereço inicial onde a mensagem vai ser armazenada.
- **cont**: número máximo de elementos a serem recebidos.
- **tipo_mpi**: tipo de dados esperado na mensagem.
- **origem**: ranque do processo origem.
- **etiq**: etiqueta esperada da mensagem.
- **com**: comunicador do contexto da mensagem.
- **estado**: estrutura auxiliar com informações como o remetente e a etiqueta da mensagem.

MPI_Recv()

- A mensagem recebida vai ser armazenada em uma posição de memória definida pelo ponteiro `*mensagem`.
- A mensagem recebida poderá ter, no máximo, `cont` elementos do do tipo `tipo_mpi`.
- O argumento `origem` é o ranque do processo do qual estamos esperando a mensagem.
- O MPI permite que `origem` seja um coringa (*), neste caso usamos `MPI_ANY_SOURCE` neste parâmetro.
- Reforçamos que não há um “coringa” para o destino da mensagem.

MPI_Recv()

- `etiq` é especificado pelo usuário para distinguir as mensagens de um único processo.
- O MPI garante que inteiros entre 0 e 32767 possam ser usados como etiquetas, mas o valor máximo é dependente de implementação.
- Existe um coringa, `MPI_ANY_TAG`, que a função `MPI_Recv` pode usar como etiqueta.
- O último argumento de `MPI_Recv()`, chamado de `estado`, fornece informações detalhadas sobre os dados efetivamente recebidos.

Comunicação ponto-a-ponto

- Esse argumento corresponde a uma estrutura que inclui os seguintes campos principais:
 - **MPI_SOURCE**: indica o ranque do processo remetente.
 - **MPI_TAG**: identifica a etiqueta da mensagem recebida.
 - **MPI_ERROR**: Informa se a mensagem foi recebida corretamente ou não.
- Por exemplo, se a origem da recepção foi especificada como **MPI_ANY_SOURCE**, o campo **MPI_SOURCE** conterá o ranque do processo que enviou a mensagem.
- Se a etiqueta de recepção foi especificada como **MPI_ANY_TAG**, o campo **MPI_TAG** contém a etiqueta da mensagem recebida.

Comunicação ponto-a-ponto

- Para que a comunicação entre dois processos **A** e **B** no MPI ocorra corretamente, os argumentos usados por `MPI_Send()` no processo **A** devem ser rigorosamente compatíveis com os usados por `MPI_Recv()` no processo **B**.
- Isso ocorre porque o MPI utiliza esses argumentos para estabelecer uma correspondência entre o envio e o recebimento das mensagens.
- Qualquer incompatibilidade pode resultar em erros, mensagens perdidas ou comportamentos inesperados.
- Em primeiro lugar, o mesmo comunicador deve ser usado em ambos processos para que a mensagem seja transferida corretamente.

Comunicação ponto-a-ponto

- No processo **A**, o argumento **destino** em **MPI_Send()** deve corresponder ao ranque do processo **B** (identificador único no comunicador).
- No processo **B**, o argumento **origem** em **MPI_Recv()** deve ser o ranque do processo **A** ou um coringa, como **MPI_ANY_SOURCE**.
- Ambos os processos devem usar a mesma etiqueta para identificar a mensagem.
- A etiqueta permite que um processo receba mensagens específicas, mesmo quando há múltiplas mensagens em trânsito.
- Se o coringa **MPI_ANY_TAG** for usado no processo **B**, qualquer etiqueta será aceita.

Exemplo

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char** argv) {
    int meu_ranque, num_procs, origem, destino, etiq=0;
    char mensagem[100];
    MPI_Status estado;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    if (meu_ranque != 0){
        sprintf(msg, "Processo %d está vivo!", meu_ranque);
        destino = 0;
        MPI_Send(mensagem, strlen(mensagem)+1, MPI_CHAR, destino, etiq, MPI_COMM_WORLD);
    }
    else{
        for (origem=1; origem < num_procs; origem++) {
            MPI_Recv(mensagem, 100, MPI_CHAR, origem, etiq, MPI_COMM_WORLD, &estado);
            printf("%s\n", mensagem);
        }
    }
    MPI_Finalize( );
}
```



Escola Supercomputador Santos Dumont - 2025

Correspondência tipos MPI e C

Correspondência entre tipos MPI e C

Tipo de dado MPI	Tipo de dados C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED CHAR	unsigned char
MPI_UNSIGNED SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG DOUBLE	long double
MPI_BYTE, MPI_PACKED	

Correspondência entre tipos MPI e C

- Os dois últimos tipos, `MPI_BYTE` e `MPI_PACKED`, não possuem correspondência direta com os tipos padronizados em C.
- O tipo `MPI_BYTE` é útil quando não se deseja realizar conversões entre tipos de dados diferentes.
- Já o tipo `MPI_PACKED` é empregado no envio de mensagens empacotadas.
- É importante observar que a quantidade de espaço alocado no *buffer* de recepção não precisa ser idêntica ao tamanho da mensagem recebida.
- O MPI permite que uma mensagem seja recebida enquanto houver espaço suficiente disponível no *buffer*.

Comunicação ponto-a-ponto

- A informação sobre a recepção de mensagem com o uso de um coringa é retornada pela função `MPI_Recv` em uma estrutura do tipo “`MPI_Status`”.
- Essa estrutura tem diversos usos, por exemplo, para saber o total de elementos recebidos utilize a rotina:

```
int MPI_Get_count(MPI_Status *status,  
MPI_Datatype datatype, int *count)
```

Informação	C
remetente	status.MPI_SOURCE
etiqueta	status.MPI_TAG
erro	status.MPI_ERROR

Exemplo

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define MAX 100
int main(int argc, char *argv[]) { /* mpi_status.c */
    int meu_ranke, total_num, etiq = 0;
    int origem = 0, destino = 1, numeros[MAX];
    MPI_Status estado;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranke);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranke);
    if (meu_ranke == origem) {
        /* Escolhe uma quantidade aleatória de inteiros para enviar para o processo 1 */
        srand(MPI_Wtime());
        total_num = (rand() / (float)RAND_MAX) * MAX;
        /* Envia a quantidade de inteiros para o processo 1 */
        MPI_Send(numeros, total_num, MPI_INT, destino, etiq, MPI_COMM_WORLD);
        printf("Processo %d enviou %d números para 1\n", origem, total_num);
    }
}
```

Exemplo

```
else {  
    /* Recebe no máximo MAX números do processo 0 */  
    MPI_Recv(numeros, MAX, MPI_INT, origem, etiq, MPI_COMM_WORLD, &estado);  
    /* Quando chega a mensagem, verifica o status para saber quantos números foram recebidos */  
    MPI_Get_count(&estado, MPI_INT, &total_num);  
    /* Imprime a quantidade de números e a informação adicional que está no manipulador "estado" */  
    printf("Processo %d recebeu %d números. Origem da mensagem = %d, etiqueta = %d\n", destino, \  
        total_num, estado.MPI_SOURCE, estado.MPI_TAG);  
}  
MPI_Finalize();  
return(0);  
}
```

Escola de Computação Santos Dumont - 2025

Exemplo

```
$ mpirun -np 2 ./teste
```

Processo 0 enviou 93 números para 1

Processo 1 recebeu 93 números. Origem da mensagem = 0, etiqueta = 0