Notas dos slides

APRESENTAÇÃO

O presente conjunto de slides pertence à coleção produzida para a disciplina *Introdução ao Processamento Paralelo e Distribuído* ofertada aos cursos de bacharelado em Ciência da Computação e em Engenharia da Computação pelo Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas.

Os slides disponibilizados complementam as videoaulas produzidas e tratam de pontos específicos da disciplina. Embora tenham sido produzidos para ser assistidos de forma independente, a sequência informada reflete o encadeamento dos assuntos no desenvolvimento do conteúdo programático previsto para a disciplina.

computação
Universidade Federal de Pelotas

universidade federal de pelotas
CDTec
centro de desenvolvimento tecnológico

Programação em Ambiente com Memória Distribuída

MPI

> **It is the responsibility of the sender to make sure the receiver understands the message.**
>
> Joseph Batter

Notas da videoaula

DESCRIÇÃO

Nesta videoaula são apresentados os principais conceitos associados às ferramentas de programação RPC e RMI.

OBJETIVOS

Nesta videoaula o aluno conhecerá os fundamentos da programação com MPI, uma ferramenta que propõe recursos para comunicação por troca de mensagens. A partir do conteúdo desenvolvido, o aluno estará habilitado a explorar MPI no desenvolvimento de aplicações mais complexas.

Message Passing Interface

Oferece um modelo de comunicação baseado em **troca de mensagens**.

Padrão de facto (esforços da indústria e da acadêmia).

Especificação desde 1991.

Várias implementações.

É a ferramenta dominante.

Propõe um protocolo para comunicações ponto a ponto e coletivas. Implementada em alto nível, oferece uma interface para acesso a primitivas de comunicação associada a sua semântica operacional.

Foi concebida (e é implementada) para oferecer alto desempenho, escalabilidade e portabilidade.

Troca de Mensagens

A inexistência de um espaço de endereçamento compartilhado força a exploração de uma rede viabilizar a comunicação entre as partes de uma aplicação. Mecanismos de troca de mensagens caracterizam-se por ser uma das bases da implementação de recursos de mais alto nível em ambiente distribuído.

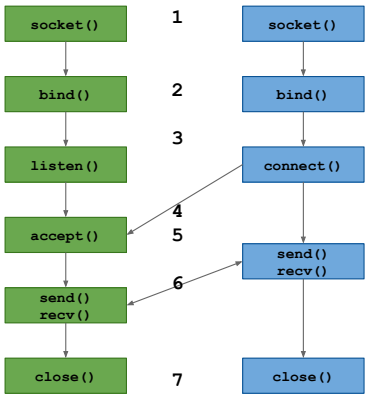
A troca de mensagens é uma técnica para ativar uma execução em um computador (programa) remoto. Isto é, exige envolvimento tanto da origem como do destino da mensagem.

Requisitos mínimos:

- Sender
- Receiver
- Address
- Message
- Protocol
- Synchronization

Onde começar? Sockets

1. socket() cria um comunicador ponto a ponto e retorna o descritor deste comunicador.
2. Um nome pode pode ser associado, bind(), ao descritor para torná-lo reconhecido na rede.
3. listen() habilita o socket a receber requisições de conexões.
4. O cliente solicita conexão com o servidor.
5. O servidor aceita a conexão.
6. Uma vez estabelecida a conexão, a comunicação pode ser feita utilizando send(), recv(), read(), write(), ...
7. A conexão é desfeita com a primitiva close()



Exemplo socket (servidor)

```
... main ... {
    int sockfd, newsockfd, portno, clilen;
    char buffer[256];
    struct sockaddr_in serv_addr, cli_addr;
    int n;
    if (argc < 2) {
        fprintf(stderr,"ERRO: Informe a porta!\n");
        exit(1);
    }
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) error("ERRO: Nao consegui abrir o socket!");
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
            error("ERRO: Nao consegui realizar o binding!");
    listen(sockfd,5);
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0) error("ERRO: Conexão não estabelecida!");
```


MPI facilita!

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int worldSize, myRank, aux;
    char computador[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(NULL, NULL);  // Inicialização

    MPI_Comm_size(MPI_COMM_WORLD, &worldSize); // Quantos processos envolvidos?
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);    // Meu identificador

    MPI_Get_processor_name(computador, &aux);

    printf("Estou executando no computador %s, meu rank %d de um total de %d processos\n",
           computador, myRank, worldSize);

    MPI_Finalize(); // Finalização
}
```

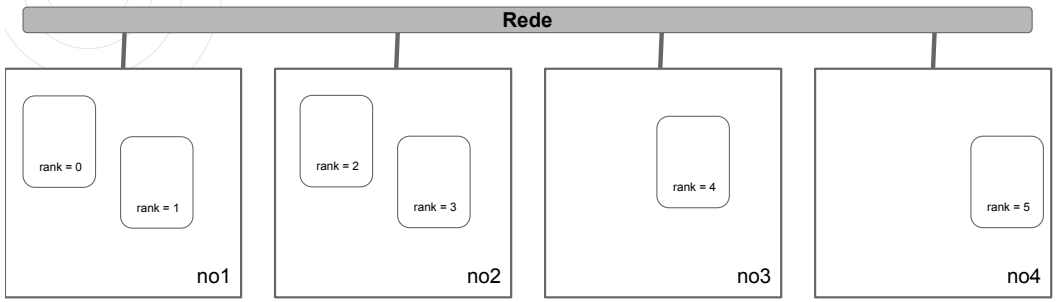

MPI facilita!

```
gersonc@thinkg: ~/fnt/mpi
gersonc@thinkg:~/fnt/mpi$ ls
primeiro.c
gersonc@thinkg:~/fnt/mpi$ mpicc primeiro.c -o primeiro
gersonc@thinkg:~/fnt/mpi$ ls
primeiro  primeiro.c
gersonc@thinkg:~/fnt/mpi$ mpirun primeiro
Estou executando no computador thinkg, meu rank 1 de um total de 2 processos
Estou executando no computador thinkg, meu rank 0 de um total de 2 processos
gersonc@thinkg:~/fnt/mpi$ mpirun --host localhost:4 primeiro
Estou executando no computador thinkg, meu rank 3 de um total de 4 processos
Estou executando no computador thinkg, meu rank 2 de um total de 4 processos
Estou executando no computador thinkg, meu rank 0 de um total de 4 processos
Estou executando no computador thinkg, meu rank 1 de um total de 4 processos
gersonc@thinkg:~/fnt/mpi$
```


Máquina Virtual

```
no1 slots=2
no2 slots=2
no3
no4
no5
```

hostfile



Abstrações de MPI

.

Abstrações:

MPI instancia uma *máquina virtual* sobre uma rede física e oferece abstrações para endereçamento e diversas primitivas para comunicação ponto a ponto e coletiva.

- **Sender** } Nós da máquina virtual: processos ou task (na terminologia MPI)
- **Receiver**
  - A máquina virtual representa um grupo de nós. Cada grupo possui um tamanho (size) e cada nó um identificador (rank)
- **Address** Identificador único do nó no seu grupo rank = 0..size
  - Grupos podem ser criados, tendo o nó um rank diferente para cada grupo ao qual for inserido
- **Messages** São ponto a ponto ou coletivas, havendo primitivas pa empacotamento/desempacotamento de dados.
- **Synchronization** Várias alternativas oferecidas

Abstrações de MPI

MPI oferece uma diversidade de primitivas de comunicação que permitem:

- Otimizar o desempenho (tempo de execução e/ou memória)
  - Tipos de comunicação

- Simplificar a implementação de algoritmos do programa de aplicação
  - Primitivas de comunicação em grupo com transformação da visualização dos dados.

Primeiro Programa

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
 int worldSize, myRank, aux, dest;

 MPI_Init(NULL, NULL);
 MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
 MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

 if( worldSize > 1 )
  printf("Soma dos Ranks: %d\n", emParalelo(myRank,worldSize));

 MPI_Finalize();
}
```


Primeiro Programa

```
int emParalelo( int myR, int worldSize ) {
 int aux;
 MPI_Status st;
 if( myRank == 0 ) {
  MPI_Send((void*)&myRank,1,MPI_INT,myRank+1, 0, MPI_COMM_WORLD);
  MPI_Recv(&aux,1,MPI_INT,worldSize-1, MPI_ANY_TAG,MPI_COMM_WORLD, &st);
 } else {
  dest = (myRank == worldSize-1)?0:myRank+1;
  MPI_Recv(&aux,1,MPI_INT,myRank-1, MPI_ANY_TAG,MPI_COMM_WORLD, &st);
  aux += myRank;
  MPI_Send(&aux,1,MPI_INT, dest, 0, MPI_COMM_WORLD);
 }
  return aux+myRank; // Será descartado quando myRank != 0
}
```

Vocabulário

Vocabulário

• **Máquina virtual**: a execução de uma aplicação MPI se dá em uma máquina virtual, especialmente contruída para executar uma determinada aplicação.
• **Processo**: um processo é um nó virtual de uma arquitetura MPI. Consiste em uma instância do programa MPI lançado. Um processo Unix torna-se um processo MPI no momento em que executa MPI_Init.
• **Task**: muitas vezes empregado como sinônimo de processo.
• **Síncrono/Assíncrono**: serviços podem ser síncronos ou assíncronos, cabe verificar a propriedade oferecida pela primitiva de interesse.

Vocabulário

• **Bloqueante**: um serviço é dito bloqueante quando o retorno de sua chamada ocorrer somente quando o serviço associado tiver sido completado.
• **Não bloquante:** um serviço é dito não bloqueante quando o retorno de sua chamada ocorrer somente quando o procedimento associado a chamada tiver sido completado, não necessariamente o serviço associado encontra-se completado. O procedimento da chamada tem a função de iniciar a execução do serviço associado.
• **Requisição**: representada por um objeto, uma requisição encontra-se associada a um serviço iniciado através de uma chamada não bloqueante.

Vocabulário

• **Local**: um procedimento é dito local quando sua operação não depender de interação com outros serviços em nós remotos.
• **Não local**: um procedimento é dito não local quando sua operação depender de interação com outros serviços em nós remotos.
• **Coletiva**: um procedimento coletivo implica na interação de todos os processos participantes de um grupo.
• **Pré-definido**: tipo de dado primitivo que o MPI pode manipular.
• **Derivado**: tipo de dado construído a partir de tipos pré-definidos.
• **Portável**: um tipo de dado é dito portável se ele é um tipo pré-definido ou se ele deriva de um tipo pré-definido utilizando construtores de enumeração (vetor, bloco, array, ...)

Mensagens

Base do Send/Recv

```
MPI_Send( <DESCRIÇÃO DO DADO>, <DESTINO>, <TAG>, <COMUNICADOR> );
```

```
MPI_Recv(<DESCRIÇÃO DO DADO>, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
```


Base do Send/Recv

```
MPI_Send( <DESCRIÇÃO DO DADO>, <DESTINO>, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, <DESTINO>, <TAG>, <COMUNICADOR> );
```

```
MPI_Recv(<DESCRIÇÃO DO DADO>, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
```


Base do Send/Recv

```
MPI_Send( <DESCRIÇÃO DO DADO>, <DESTINO>, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, <DESTINO>, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, 3, <TAG>, <COMUNICADOR> );
```

```
MPI_Recv(<DESCRIÇÃO DO DADO>, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, 2, <TAG>, <COMUNICADOR>, <STATUS> );
```

Base do Send/Recv

```
MPI_Send( <DESCRIÇÃO DO DADO>, <DESTINO>, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, <DESTINO>, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, 3, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, 3, 0, <COMUNICADOR> );


MPI_Recv(<DESCRIÇÃO DO DADO>, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, 2, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, 2, MPI_ANY_TAG, <COMUNICADOR>, <STATUS> );
```


Base do Send/Recv

```
MPI_Send( <DESCRIÇÃO DO DADO>, <DESTINO>, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, <DESTINO>, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, 3, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, 3, 0, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, 3, 0, MPI_COMM_WORLD );

MPI_Recv(<DESCRIÇÃO DO DADO>, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, 2, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, 2, MPI_ANY_TAG, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, 2, MPI_ANY_TAG, MPI_COMM_WORLD, <STATUS> );
```


Base do Send/Recv

```
MPI_Send( <DESCRIÇÃO DO DADO>, <DESTINO>, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, <DESTINO>, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, 3, <TAG>, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, 3, 0, <COMUNICADOR> );
MPI_Send((void*)&myRank, 1, MPI_INT, 3, 0, MPI_COMM_WORLD );

MPI_Recv(<DESCRIÇÃO DO DADO>, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, <ORIGEM>, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, 2, <TAG>, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, 2, MPI_ANY_TAG, <COMUNICADOR>, <STATUS> );
MPI_Recv((void*)&aux,1,MPI_INT, 2, MPI_ANY_TAG, MPI_COMM_WORLD, &st ); // MPI_Status st;
```


Pacote de Dados

As mensagens MPI possuem quatro campos:

- **Identificação da destino**
  - Inteiro
- **Identificação da origem**
  - Inteiro
- **Tag para classificação**
  - Inteiro
- **Dados**
  - Virtualmente ilimitado

Pacote de Dados

```
MPI_Send((void*)&myRank,1,MPI_INT,myRank+1, 0, MPI_COMM_WORLD);
```

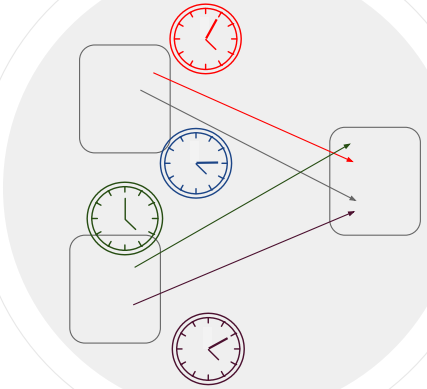

```
MPI_Recv((void*)&aux,1,MPI_INT,myRank-1,MPI_ANY_TAG,MPI_COMM_WORLD, &st);
```


Ordenamento

Inexiste um ordenamento total, posto que não existe um relógio global.

Em um determinado destino, é garantido a entrega ordenada das mensagens provindas de uma determinada origem.

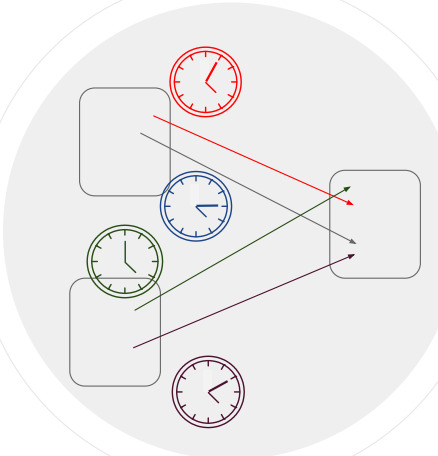
No entanto, não há nenhuma relação de ordem, em um destino, entre mensagens provindas de dois nós distintos.



Ordenamento

Na realização de uma operação receive é possível filtrar as mensagens aguardando tratamento por dois filtros:

- **Origem**: é possível selecionar o recebimento da próxima mensagem de uma origem especificada (ou `MPI_ANY_SOURCE`)
- **Tag**: é possível selecionar o recebimento da próxima mensagem por uma tag especificada (ou `MPI_ANY_TAG`)

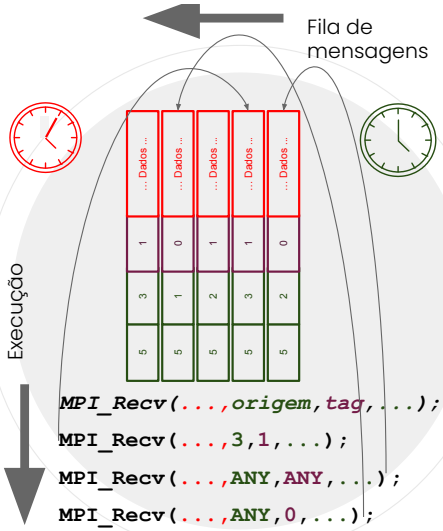


Ordenamento

Na realização de uma operação receive é possível filtrar as mensagens aguardando tratamento por dois filtros:

- **Origem**: é possível selecionar o recebimento da próxima mensagem de uma origem especificada (ou `MPI_ANY_SOURCE`)
- **Tag**: é possível selecionar o recebimento da próxima mensagem por uma tag especificada (ou `MPI_ANY_TAG`)

A fila de mensagens será percorrida para atender a combinação selecionada respeitando a ordem temporal.




```
MPI_Recv(...,origem,tag,...);
MPI_Recv(...,3,1,...);
MPI_Recv(...,ANY,ANY,...);
MPI_Recv(...,ANY,0,...);
```

Manipulação

Ponto a Ponto

Envio de mensagens entre um par de nós

Coletivas

É possível realizar comunicações *1xn*, *nx1* e *nxn*.

Empacotamento

É possível montar mensagens "empacotando" dados.

Modos de comunicação

Padrão, bufferizado, síncrono e pronto

(standard, bufferized, synchronous, ready)

A escolha do tipo de comunicação impacta no desempenho e na estratégia de implementação do algorítmo.

Tipos de dados

Primitivos

Baseados nos tipos de dados nativo da linguagem.

Derivados

É possível descrever um novo tipo de dado, conforme as necessidades do algoritmo.

Empacotados

Um conjunto heterogêneo de dados pode ser encapsulado em uma mesma mensagem.

Tipos de dados primitivos

`MPI_Datatype`

Baseados nos tipos de dados nativo da linguagem.

- `MPI_CHAR`
- `MPI_DOUBLE`
- `MPI_FLOAT`
- `MPI_INT`
- `MPI_LONG`
- `MPI_LONG_DOUBLE`
- `MPI_SHORT`
- `MPI_UNSIGNED_CHAR`
- `MPI_UNSIGNED`
- `MPI_UNSIGNED_LONG`
- `MPI_UNSIGNED_SHORT`
- `MPI_Aint`
- `...`

Tipos de dados primitivos

`MPI_Datatype`

Exemplo sender:

```
int vet[10];
MPI_Send((void*)vet,10,MPI_INT,...);
```

Exemplo receiver:

```
int vet[10];
MPI_Recv((void*)vet,10,MPI_INT,...);
```

Tipos de dados derivados

Definidos conforme necessidade do algoritmo a partir dos tipos primitivos ou derivados já existentes.

Exemplo: uma estrutura (`struct`).

```
int MPI_Type_create_struct(int count, const int blocklengths[],
                           const MPI_Aint displacements[],
                           const MPI_Datatype types[],
                           MPI_Datatype * newtype );
```

Tipos de dados derivados

```
struct Particula {
    char c;
    double d[6];
    char b[7];
};
int main(int argc, char *argv[]) {
      struct Particula vet[1000];
      int myrank;
      MPI_Status st;
      MPI_Datatype Particula_t;
      MPI_Datatype types[3] = { MPI_CHAR, MPI_DOUBLE, MPI_CHAR };
      int blocklen[3] = { 1, 6, 7 };
      MPI_Aint disp[3];

    MPI_Init(&argc, &argv);

    disp[0] = &particle[0].c - &particle[0];
    disp[1] = &particle[0].d - &particle[0];
    disp[2] = &particle[0].b - &particle[0];
    MPI_Type_create_struct(3, blocklen, disp, type, &Particula_t);
    MPI_Type_commit(&Particula_t);

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if( myrank == 0 )
      MPI_Send(particle, 1000, Particula_t, 1, 123, MPI_COMM_WORLD);
    else if (myrank == 1) MPI_Recv(particle, 1000, Particula_t, 0, 123, MPI_COMM_WORLD, &st);
    MPI_Finalize();
    return 0;
}
```


Empacotados

`MPI_Pack` / `MPI_Unpack`

Diferentes tipos de dados podem ser mesclados em uma espaço de memória contíguo que pode ser transmitido em uma mensagem. A ordem de empacotamento deve ser obedecida no desempacotamento.

Empacotados

`MPI_Pack` / `MPI_Unpack`

Requisitos: Uma área de dados previamente alocada e um ponteiro para a próxima área de memória a ser considerada:

- No caso de um empacotamento, para um futuro envio, a próxima posição livre na área de dados
- No caso de um desempacotamento, para a próxima posição a ser desempacotada

Empacotados

Exemplo sender:

```
char nome[33], buff[256];
int idade, p = 0;

MPI_Pack(nome,33,MPI_CHAR,256,buff,&p,MPI_COMM_WORLD);
MPI_Pack(&idade,1,MPI_INT,256,buff,&p,MPI_COMM_WORLD);
MPI_Send((void*)buff,p,MPI_PACKED,...);
```

Exemplo receiver:

```
char nome[33], buff[256];
int idade, p = 0;
MPI_Status st;

MPI_Recv((void*)buff,256,MPI_PACKED,..., &st);
MPI_Unpack(buff,256,&p,nome,33,MPI_CHAR,MPI_COMM_WORLD);
MPI_Unpack(buff,256,&p,&idade,1,MPI_INT,MPI_COMM_WORLD);
```


Modos de Comunicação

Modos de comunicação

Standard
O envio é completado assim que a mensagem for enviada.

Bufferizado
O serviço armazena o dado em um buffer próprio, reduzindo a manipulação de dados, sendo completado imediatamente.

Síncrono
O envio é completado na recepção da mensagem, com a recepção de um acknowledgement

Ready
Completada imediatamente, no entanto, se o receiver não estiver pronto, a mensagem é perdida.

Bloqueante
A primitiva não retorna enquanto a operação não for completada.

Não bloqueante
A primitiva posta uma requisição de serviço, retornando imediatamente. Posteriormete a requisição deve ser testada.

Modos de comunicação

I: Imediato
S: Síncrono
B: Bufferizado
R: Ready

Nada impede que os recursos sejam combinados:

Standard:
`MPI_Send/Recv`
`MPI_Isend/Irecv`

Bufferizado:
`MPI_Bsend`
`MPI_Ibsend`

Síncrono:
`MPI_Ssend`
`MPI_Issend`

Ready:
`MPI_Rsend`
`MPI_Irsend`

Comunicação Não Bloqueante

Exemplo sender:

```
MPI_Request req;
MPI_Status st;
int vet[10];
MPI_Isend( vet, 10, MPI_INT, 4, MPI_ANY_TAG, MPI_COMM_WORLD, &req );
 ... // faz outra coisa
 MPI_Wait( &req, &st );
```

Exemplo receiver:

```
MPI_Request req; MPI_Status st;
int vet[10];
MPI_Irecv( vet, 10, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                                        MPI_COMM_WORLD, &st );
... // faz outra coisa
MPI_Wait( &req, &st );
```


Modos de comunicação

- **`MPI_Send`**: Não retorna até que o buffer possa ser reutilizado (foi enviada a mensagem).
- **`MPI_Bsend`**: Retorna imediatamente, o MPI mantém uma cópia do buffer para posteriro envio.
- **`MPI_Ssend`**: Não retorna até que a recepção tenha sido completada.
- **`MPI_Rsend`**: Retorna imediatamente, mas a mensagem é perdida se o receive não estiver postado.
- **`MPI_Isend`**: Retorna após realizado o envio, mas a operação não foi completada.
- **`MPI_Ibsend`**: Envio bufferizado, não bloqueante.
- **`MPI_Issend`**: Realiza um send sincrono, mas não bloqueia.
- **`MPI_Irsend`**: Realiza um send ready, mas não bloqueia.

Comunicação Coletiva

Comunicação Coletiva

Envolve todos os nós do grupo, podendo ser:

- ***1 x n***: um sender, vários receivers
- ***n x 1***: vários senders, um receiver
- ***n x n***: todos para todos

Tipos de comunicações coletivas:

- Barreiras
- Troca de dados

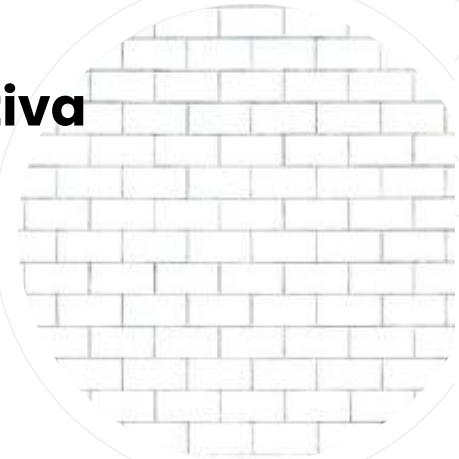
Verificar os modos de comunicação!

Comunicação Coletiva

Barreira

Uma barreira, em MPI, força um ponto de sincronização entre todos os processos envolvidos no grupo: a semântica informa que, a barreira vencida por um processo indica que todos processos atingiram a barreira.

`MPI_Init` e **`MP_Finalize`** são exemplos de barreiras aplicadas ao grupo **`MPI_COMM_WORLD`**

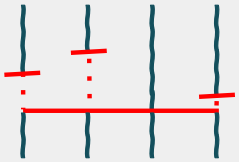


Comunicação Coletiva

Barreira

```
int MPI_Barrier(MPI_Comm comm);
```

A primitiva **`MPI_Barrier`** garante a sincronização de todos os processos do grupo **`comm`**.



Comunicação Coletiva

Troca de dados

Duas categorias:

- Movimentação de dados
  - A comunicação manipula e transforma a representação do dado
    - Broadcast, Gather[v], Scatter[v], Allgather[v], Alltoall[v]
- Computação global
  - A comunicação envolve processamento distribuído do dado comunicado
    - Reduce e Scan

Comunicação Coletiva

Troca de dados

Exemplos:

- Movimentação de dados
  - Broadcast

```
int MPI_Bcast(void* buffer, int count,
              MPI_Datatype datatype, int root,
              MPI_Comm comm)
```

- Computação global
  - Reduce

```
int MPI_Reduce(const void* sbuf, void* rbuf, int count,
               MPI_Datatype stype, MPI_Op op, int root,
               MPI_Comm comm)
```

Comunicação Coletiva

Troca de dados

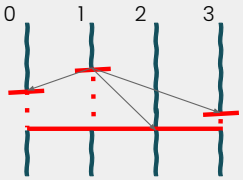
Exemplos:

- Movimentação de dados
  - Broadcast

```
int MPI_Bcast(void* buffer, int count,
              MPI_Datatype datatype, int root,
              MPI_Comm comm)
```

Todos os processos do grupo executam a mesma operação. No processo **root**, `buffer` é o dado enviado. Nos demais processos, `buffer` é o dado recebido.

```
MPI_Bcast( dta, 1, MPI_INT, 1, MPI_COMM_WORLD );
```



Comunicação Coletiva

Troca de dados

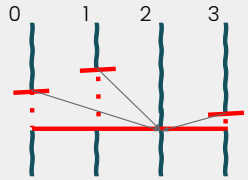
Exemplos:

- Computação global
  - Reduce

```
int MPI_Reduce(const void* sbuf, void* rbuf, int count,
               MPI_Datatype stype, MPI_Op op, int root,
               MPI_Comm comm)
```

Todos os processos do grupo executam a mesma operação. No processo **root**, `rbuf` é o dado resultante da operação. Nos demais processos, `rbuf` não tem significado. Em todos processos, raíz inclusive, `sbuf` representa o dado enviado.

```
MPI_Reduce(src, dst, 1, MPI_INT, MPI_SUM, 2, MPI_COMM_WORLD);
```



Comunicação Coletiva

Troca de dados

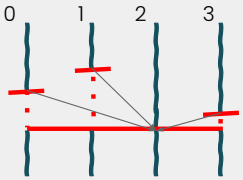
Exemplos:

- Computação global
  - Reduce

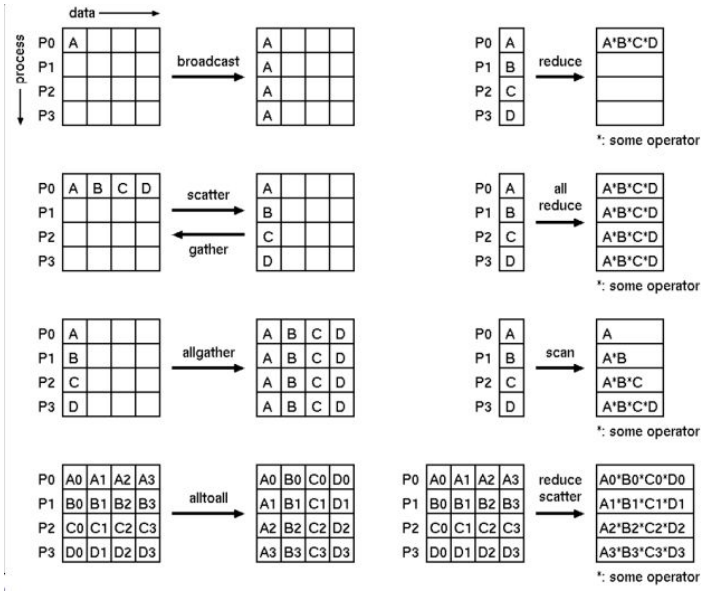
```
int MPI_Reduce(const void* sbuf, void* rbuf, int count,
               MPI_Datatype stype, MPI_Op op, int root,
               MPI_Comm comm)
```

O parâmetro `op` informa qual é a operação de redução a ser realizada.

- `MPI_BAND`
- `MPI_BOR`
- `MPI_BXOR`
- `MPI_LAND`
- `MPI_LOR`
- `MPI_LXOR`
- `MPI_MAX`
- `MPI_MAXLOC`
- `MPI_MIN`
- `MPI_MINLOC`
- `MPI_PROD`
- `MPI_SUM`



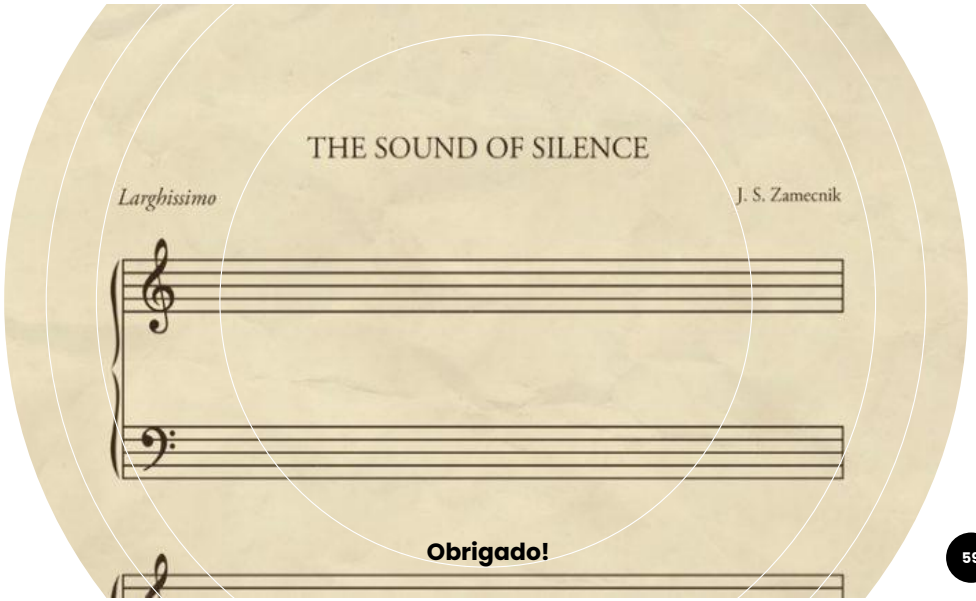
Comunicação Coletiva



Não vimos aqui

MPI é tem muitos recursos!

- Não vimos detalhes de todos os recursos para comunicação coletiva
- Não discutimos detalhes de desempenho dos diferentes modos de comunicação
- Não vimos mecanismos de criação de grupos
- Não vimos criação/remoção dinâmica de processos
- Não vimos E/S paralela
- RMA (mapeamento de memória remota)
- ...



Obrigado!