MPI (Message Passage Interface)

MPI (Message Passage Interface)

Programação utilizando passagem de mensagem

MPI é uma biblioteca de funções que permite criar programas paralelos e distribuídos.

Linguagens:

- Standard C, C++, Fortran77, Fortran90, Python e Java.

MPI (Message Passage Interface)

É possível trabalhar com memória distribuída e compartilhada combinando MPI com OPENMP.

É possível executar múltiplos processos MPI em um processador

MPI (Message Passage Interface)

Implementações disponíveis:

- OpenMPI: http://www.open-mpi.org/
- MPICH: http://www.mpich.org/

Hello World

Hello World

Enter e Exit

```
MPI_Init(int *argc,char *argv);
MPI_Finalize(void);
```

Hello World

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

main(int argc, char **argv)
{
   MPI_Init(&argc, &argv);
   printf("Hello world\n");
   MPI_Finalize();
}
```

Compilação e Execução

Para compilar

```
$ mpicc hello.c -o hello
```

Para executar

```
$ mpirun -np 4 hello
Hello world
Hello world
Hello world
Hello world
$
```

Identificando o número do processo

Identificando o número do processo

Processos: são representados por um único "rank"(inteiro) e ranks são numerados $0, 1, 2..., N-1$. (N = total de processos)

Quem eu sou?

```
MPI_Comm_rank(MPI_Comm comm,int *rank);
```

Informa total de processos:

```
MPI_Comm_size(MPI_Comm comm,int *size);
```

Comunicação padrão:

`MPI_COMM_WORLD` contém todos os processos

Hello World

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

main(int argc, char **argv)
{
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &id);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

   printf("I'm process %i out of %i processes\n", id, nprocs);
   MPI_Finalize();
}
```

Tipos de Comunicação

A biblioteca MPI implementa os seguintes tipos de comunicação:

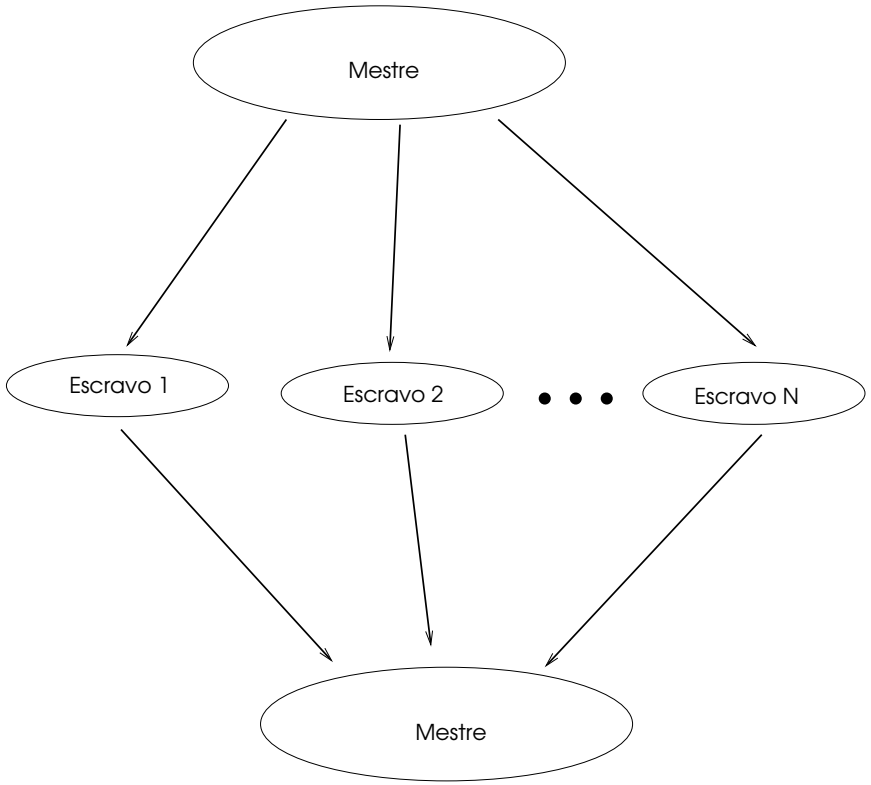
- ▷ Ponto-a-Ponto: send e receive
- ▷ Comunicação coletiva: broadcast, scatter e gather, reduce e scan

Modelo de programação com MPI

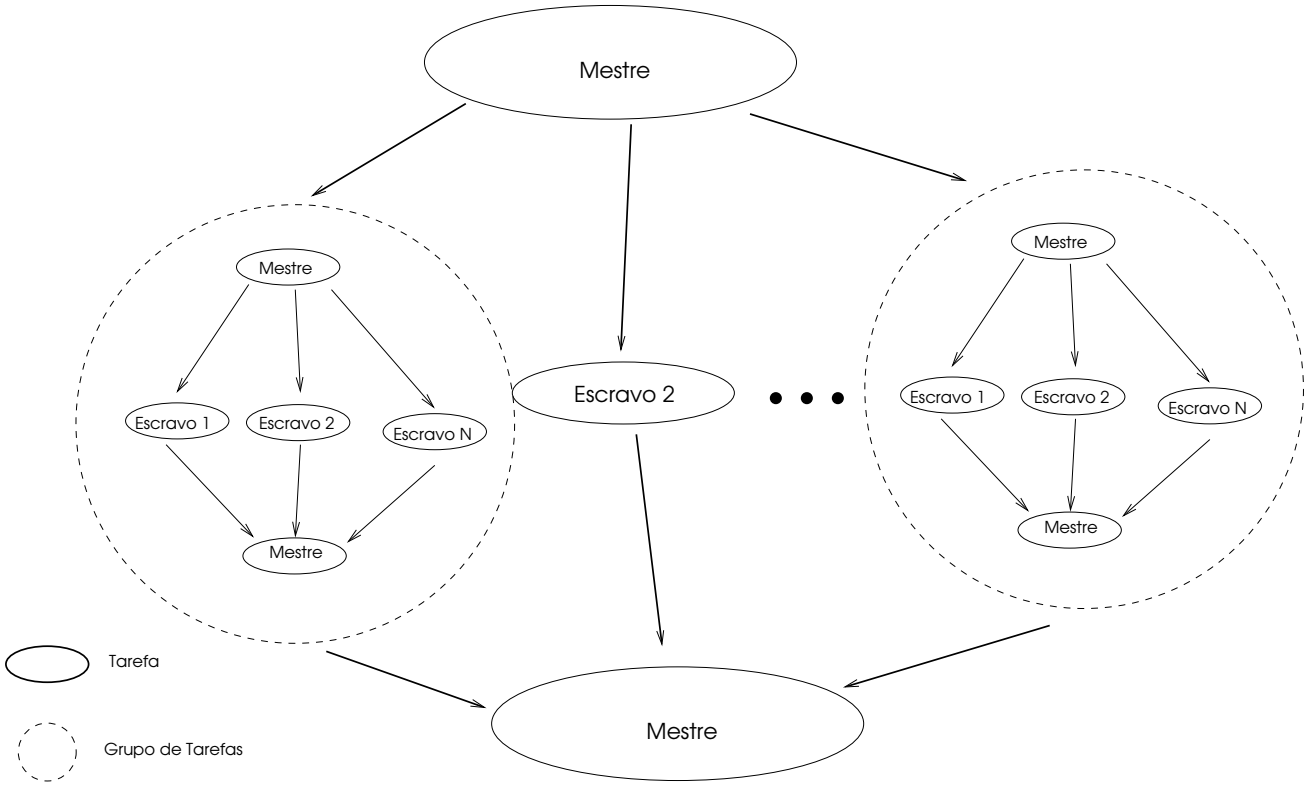
SPMD (Single Program Multiple Data)

Todos executam o mesmo programa, sendo que através de um controle interno ao programa um nó executa a função mestre e os demais são escravos.

Modelo de programação com MPI



Grupo de tarefas



Para obter um bom desempenho

- Use granularidade grossa
- Minimize o número de mensagens
- Maximize o tamanho de cada mensagem
- Use alguma forma de balanceamento de carga

Primitivas ponto-a-ponto

Tipos de comunicação ponto-a-ponto

De acordo com o modo de transmissão, o padrão MPI pode definir os seguintes tipos de comunicação:

Standard, *Synchronous*, *Ready*, *Buffered*;

Tipos de comunicação ponto-a-ponto

Standard: primitivas de envios bloqueantes e não-bloqueantes;

Bloqueantes: `MPI_Send`, `MPI_Recv`

Não bloqueantes: `MPI_ISend`, `MPI_IRecv`

Tipos de comunicação ponto-a-ponto

Synchronous: O processo de envio fica bloqueado até que o *buffer* da aplicação do processo emissor esteja livre para ser utilizado e que o processo receptor já tenha começado a receber a mensagem.

Tipos de comunicação ponto-a-ponto

Este modo introduz o sincronismo através da confirmação por parte dos processos receptores (*handshake*);

Synchronous: `MPI_Ssend`, `MPI_Recv`

Tipos de comunicação ponto-a-ponto

Ready: O processo que envia os dados manda uma mensagem ao receptor perguntando se o mesmo está pronto para receber a mensagem.

Tipos de comunicação ponto-a-ponto

O emissor só inicia a transmissão da mensagem se o outro processo estiver apto a receber os dados.

Ready: `MPI_Rsend`, `MPI_Recv`

Tipos de comunicação ponto-a-ponto

Buffered: Permite ao programador definir um buffer onde os dados serão copiados e armazenados até serem transmitidos.

Buffered: `MPI_Bsend`, `MPI_Recv`

Primitivas ponto-a-ponto bloqueantes

Enviando Mensagens

```
MPI_Send(void *buf,int count,MPI_Datatype dtype,
         int dest,int tag,MPI_Comm comm);
```

Recebendo Mensagens

```
MPI_Recv(void *buf,int count,MPI_Datatype dtype,
         int source,int tag,MPI_Comm comm,MPI_Status *status);
```

status:

`status.MPI_TAG` e `status.MPI_SOURCE`

`MPI_ANY_TAG` e `MPI_ANY_SOURCE` são *wildcards*.

Tipo de dados

Os principais tipos de dados:

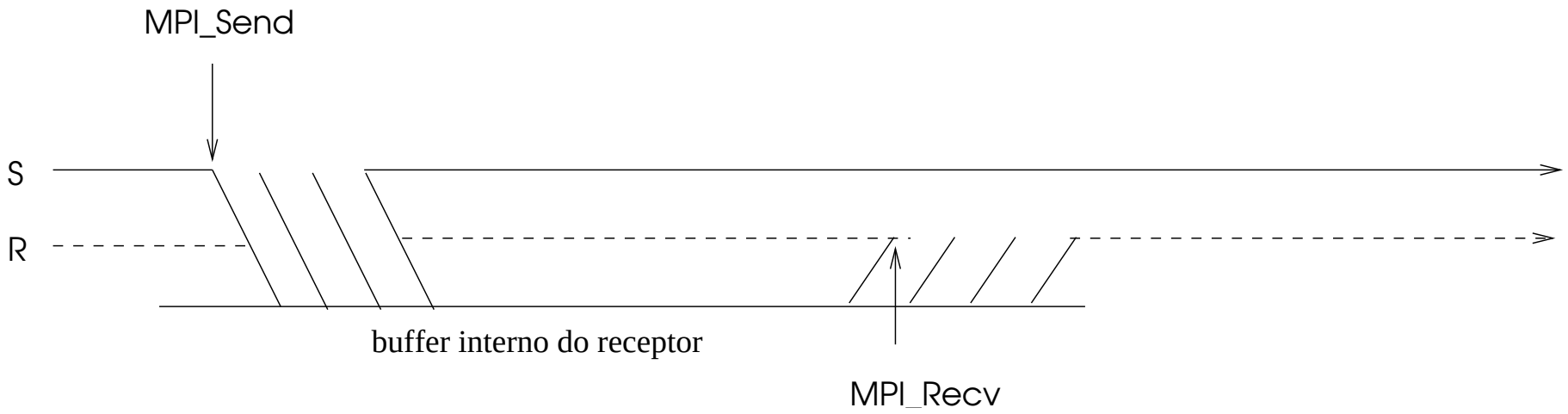
`MPI_CHAR`, `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, entre outros.

Novos tipos podem ser criados:

- baseados em tipos existentes
- chamados de tipos de dados derivados

Envio padrão bloqueante

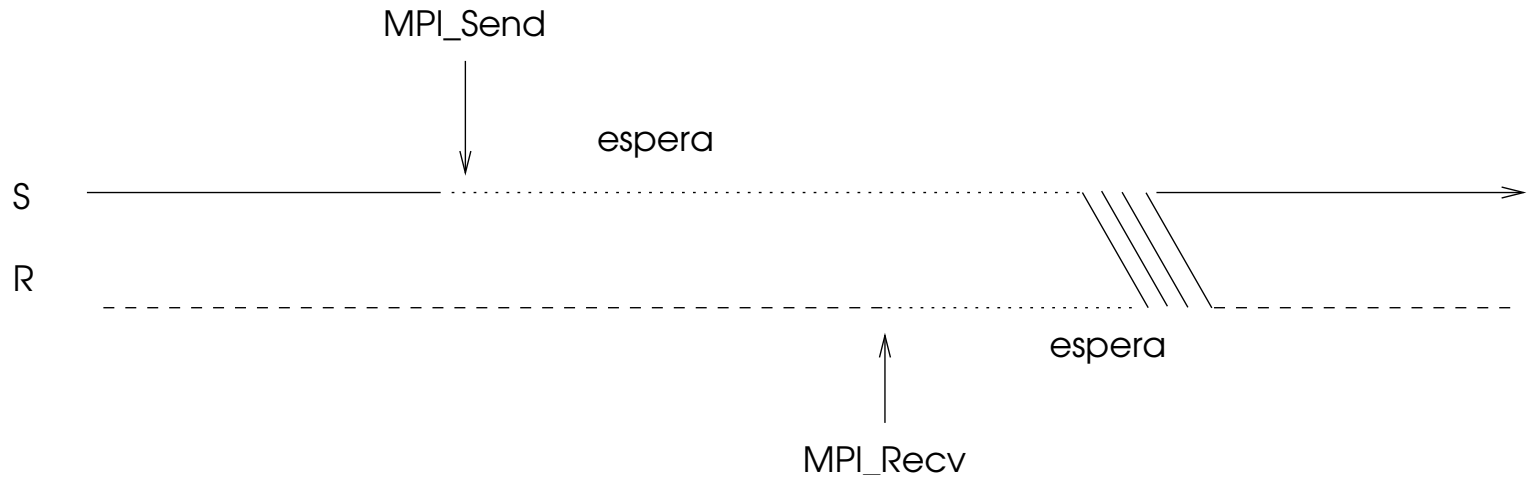
O comportamento do sistema depende do tamanho da mensagem, se é menor ou igual ou maior do que um limite. Este limite é definido pela implementação do sistema e do número de tarefas na aplicação.



Mensagem <= limite

Envio padrão bloqueante

Mensagem $>$ limite



Exemplo de mensagens send/recv

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

int main(int argc, char *argv[])
{
   int size, rank;
   int length;
   char name[80];
   int dest = 0;
   int tag = 999;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
    if (rank > 0) {
       MPI_Get_processor_name(name, &length);
       MPI_Send(name, 80, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else {
       MPI_Status status;
       int source;
       for(source = 1; source < size; source++) {
          MPI_Recv(name, 80, MPI_CHAR, source, tag,
                                       MPI_COMM_WORLD, &status);
          printf("msg from %d %d on %s\n", source, size, name);
       }
    }
    MPI_Finalize();
    return 0;
}
```

Exercícios

Exercícios

Escreva um programa paralelo utilizando a biblioteca MPI que some os elementos de um determinado vetor.

1. O processo mestre deve gerar um vetor de n elementos
2. O processo mestre deve enviar pedaços do vetor para os demais processos
3. Cada processo escravo receberá o vetor e calculará a soma parcial
4. Cada processo escravo deverá enviar para o mestre o resultado da soma parcial
5. O processo mestre deverá acumular as somas parciais e mostrar o resultado.

Exercícios (solução)

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

#define N 12

int P;
int tag = 999;

void mestre();
void escravo();
```

```
int main(int argc, char *argv[])
{
   int id;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &id);
   MPI_Comm_size(MPI_COMM_WORLD, &P);
   if (id ==  0) {
      mestre();
   } else {
      escravo();
   }
   MPI_Finalize();
   return 0;
}
```

```
void mestre()
{
   int v[N], i, dest, soma, soma_parcial;
   MPI_Status status;
   for (i = 0; i < N; i++)  v[i] = i;

   for (dest = 1; dest < P; dest++) {
      MPI_Send(v + dest * N/P, N/P, MPI_INT,
                       dest, tag, MPI_COMM_WORLD);
   }

   soma = 0;
   for(i = 0; i < N/P; i++)
       soma += v[i];
```

```
    for (dest = 1; dest < P; dest++){
        MPI_Recv(&soma_parcial, 1, MPI_INT,
                        MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
        soma += soma_parcial;
    }

    printf("soma total: %d\n", soma);
}
```

```
void escravo()
{
   int i, v[N], soma_parcial,id;
   MPI_Status status;
   MPI_Recv(v, N/P, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
   soma_parcial = 0;
   MPI_Comm_rank(MPI_COMM_WORLD, &id);
   for (i = 0; i < N/P; i++){
      soma_parcial += v[i];
   }

   MPI_Send(&soma_parcial, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
```

Exercícios

Escreva um programa paralelo utilizando biblioteca MPI para calcular o maior elemento de um vetor de n elementos.

Escreva um programa paralelo utilizando biblioteca MPI para calcular a multiplicação de uma matriz por um vetor.

Comunicação coletiva

Comunicação coletiva

Comunicação coletiva é um método de comunicação que envolve todos os processos de um dado comunicador.

Comunicação coletiva

MPI fornece uma lista de funções para comunicação coletiva:

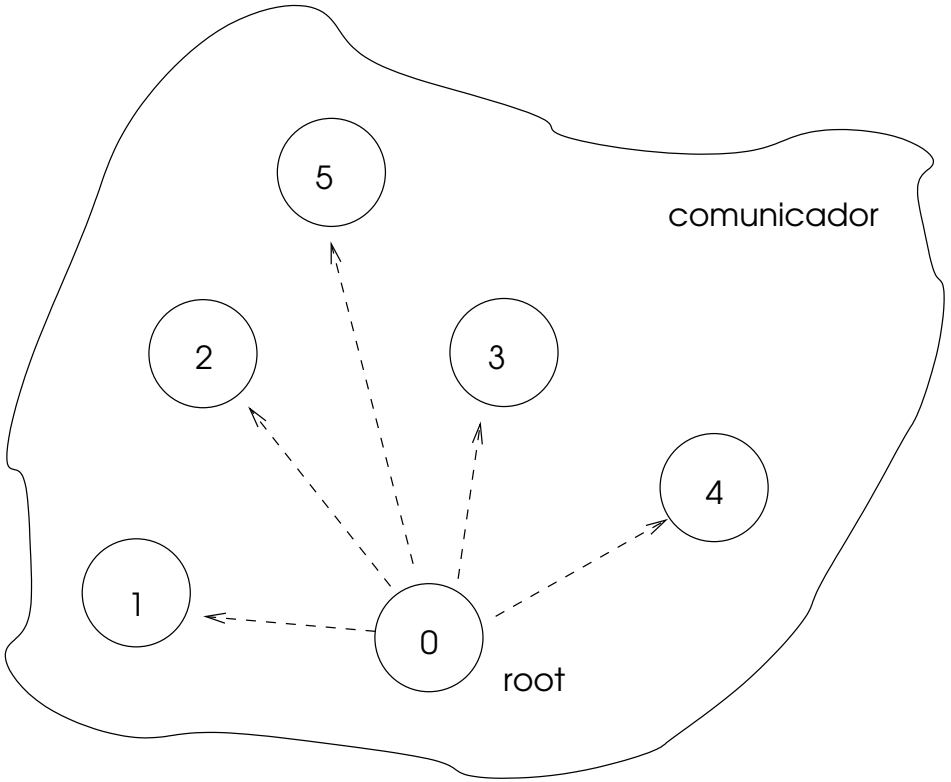
- `MPI_Bcast()`: Broadcast (um para todos)
- `MPI_Scatter()`: Distribuir os dados (um para todos)
- `MPI_Gather()`: Coletar os dados (um para todos)
- `MPI_Reduce()`: Redução (todos para um)
- `MPI_AllReduce()`: Redução (todos para todos)

Broadcast

```
MPI_Bcast(void *buf, int count, MPI_Datatype dtype,
                                int root, MPI_Comm comm);
```

Na operação broadcast todos os processos especificam o mesmo processo root (argumento root) cujo conteúdo do buffer será enviado. Os demais processos especificam buffers de recepção. Após a execução da chamada todos os buffers contêm os dados do buffer do processo root.

Broadcast



Exemplo de Broadcast

Enviando vetor de tamanho 100 de números inteiros a todos os processos do grupo.

```
MPI_Comm comm;
   int array[100];
   int root=0;
   ...
   MPI_Bcast(array, 100, MPI_INT, root, comm);
```

Distribuição de dados: scatter e gather

Scatter

Scatter é uma operação de comunicação coletiva onde uma tarefa *root* envia um conjunto de dados distintos para cada processo pertencente ao grupo.

Sintaxe:

```
MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
            void *recvbuf, int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm);
```

Scatter

```
MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
            void *recvbuf, int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm);
```

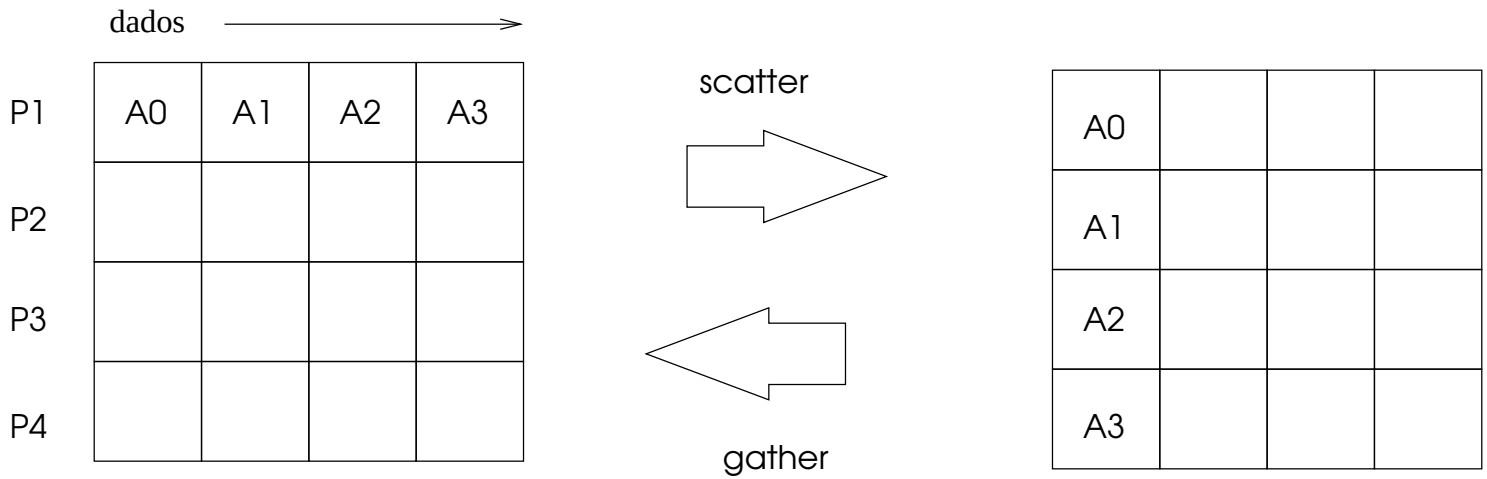
Esta função particiona um vetor referenciado por `sendbuf`, na tarefa `root`, em p segmentos, cada segmento contendo `sendcount` elementos do tipo `sendtype`

Gather

```
MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
           void *recvbuf, int recvcount, MPI_Datatype recvtype,
           int root, MPI_Comm comm);
```

a operação gather é o reverso da operação scatter (dados de buffers de todos processos para processo root)

Scatter e Gather

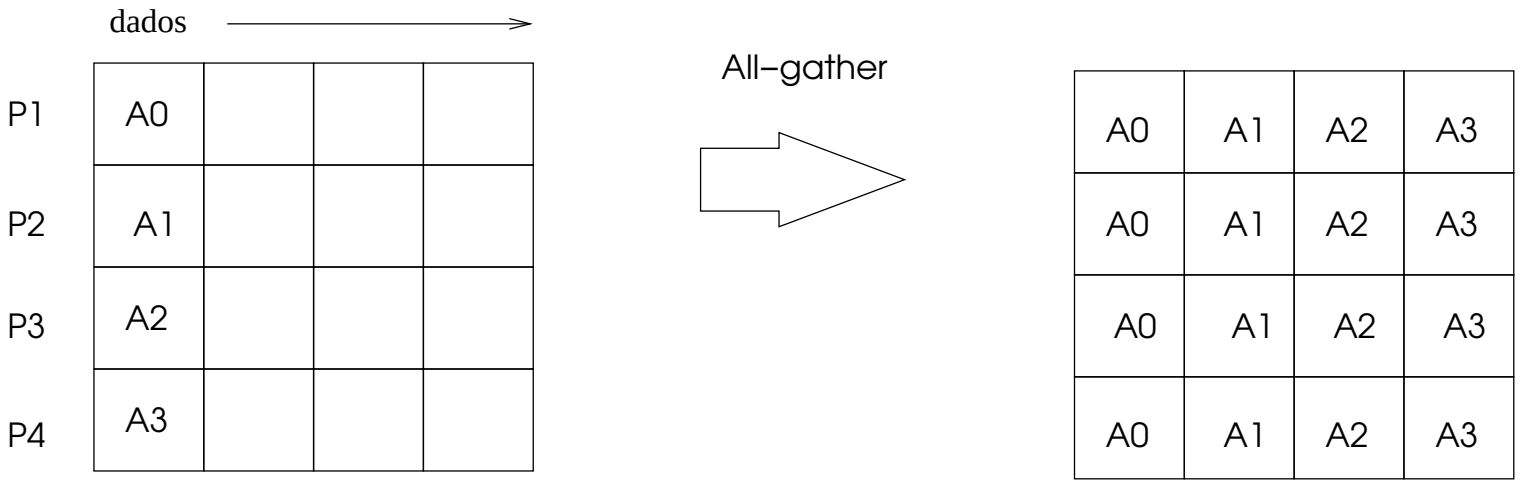


All-Gather

```
MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
              void *recvbuf, int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm);
```

A operação all-gather é semelhante à operação gather. A única diferença é que na operação all-gather todos os processos coletam os dados de cada processo da aplicação, enquanto que na operação gather só o `root` coleta os dados.

All-gather

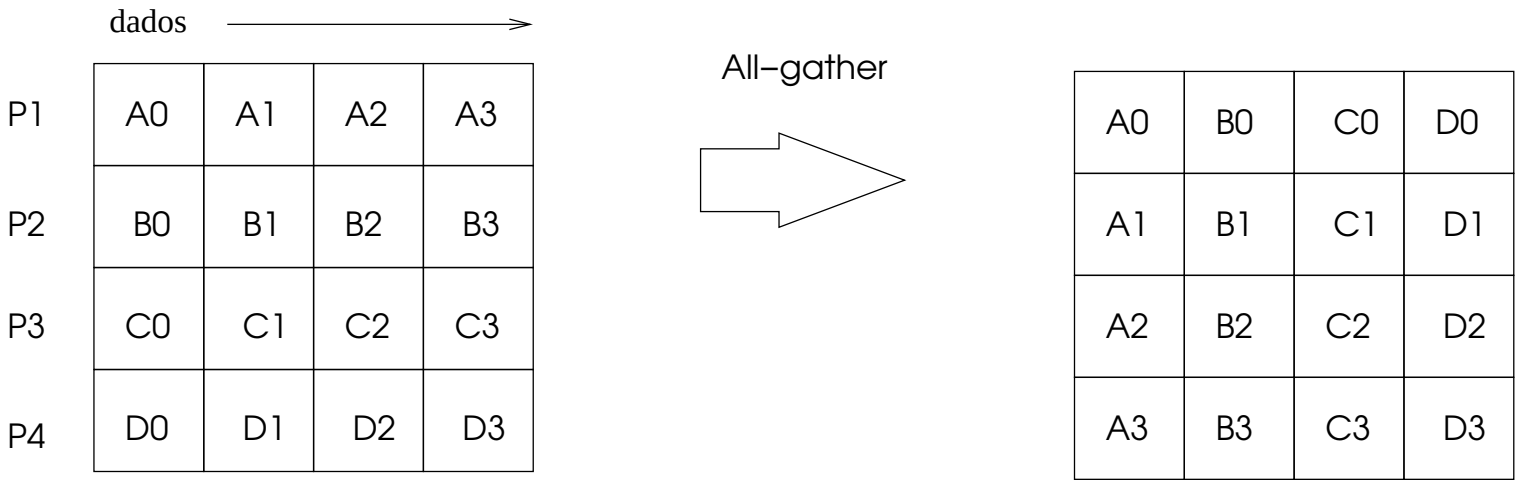

dados
P1
P2
P3
P4
A0
A1
A2
A3
All–gather
A0 A1 A2 A3
A0 A1 A2 A3
A0 A1 A2 A3
A0 A1 A2 A3

All-to-All

```
MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
             void *recvbuf, int recvcount, MPI_Datatype recvtype,
             MPI_Comm comm);
```

All-to-All é uma operação de comunicação coletiva do tipo muitos-para-muitos, em que cada processo envia seus dados para todos os processos da aplicação

All-to-All


dados
P1
P2
P3
P4
A0
A1
A2
A3
B0
B1
B2
B3
C0
C1
C2
C3
D0
D1
D2
D3
All–gather
A0
B0
C0
D0
A1
B1
C1
D1
A2
B2
C2
D2
A3
B3
C3
D3

Redução

Reduce

```
MPI_Reduce(void *sendbuf, void *recvbuf, int count,
           MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm);
```

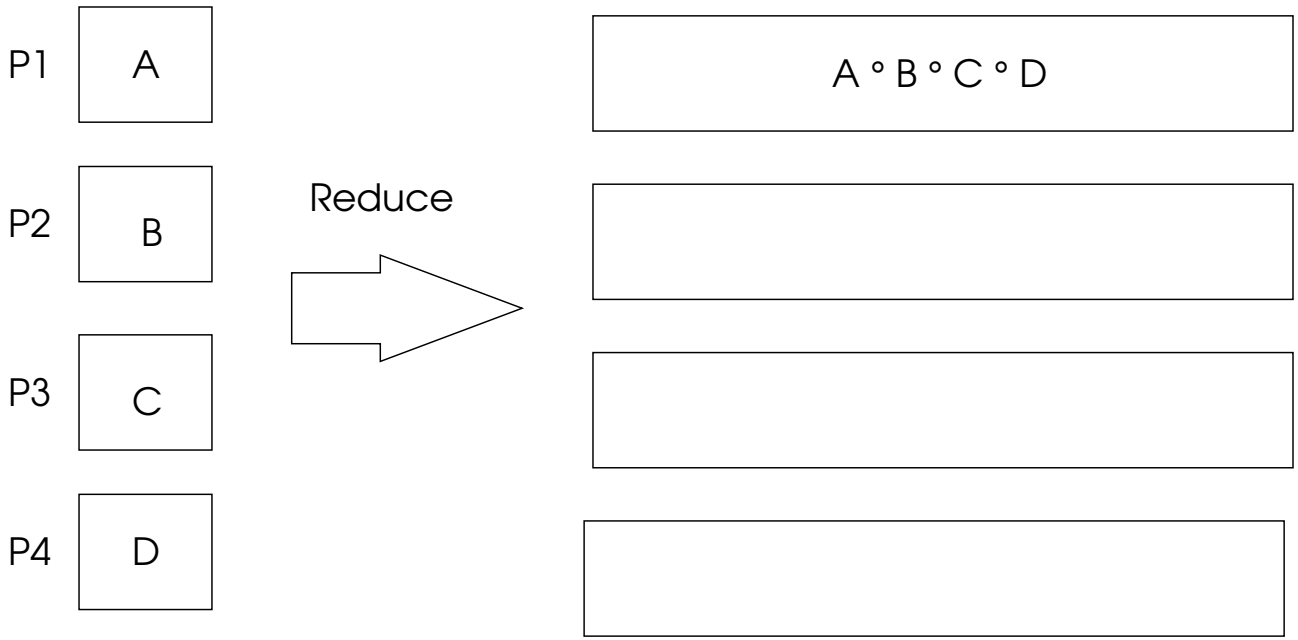
Elementos dos buffers de envio são combinados par a par para um único elemento correspondente no buffer de recepção do root.

Operaçoes de redução:

`MPI_MAX` (maximum), `MPI_MIN` (minimum), `MPI_SUM` (sum), `MPI_PROD` (product), `MPI_LAND` (logical and), `MPI_BAND` (bitwise and) `MPI_LOR` (logical or), `MPI_BOR` (bitwise or), `MPI_LXOR` (logical exclusive or), `MPI_BXOR` (bitwise exclusive or)

Reduce

° : operador associativo



All-Reduce

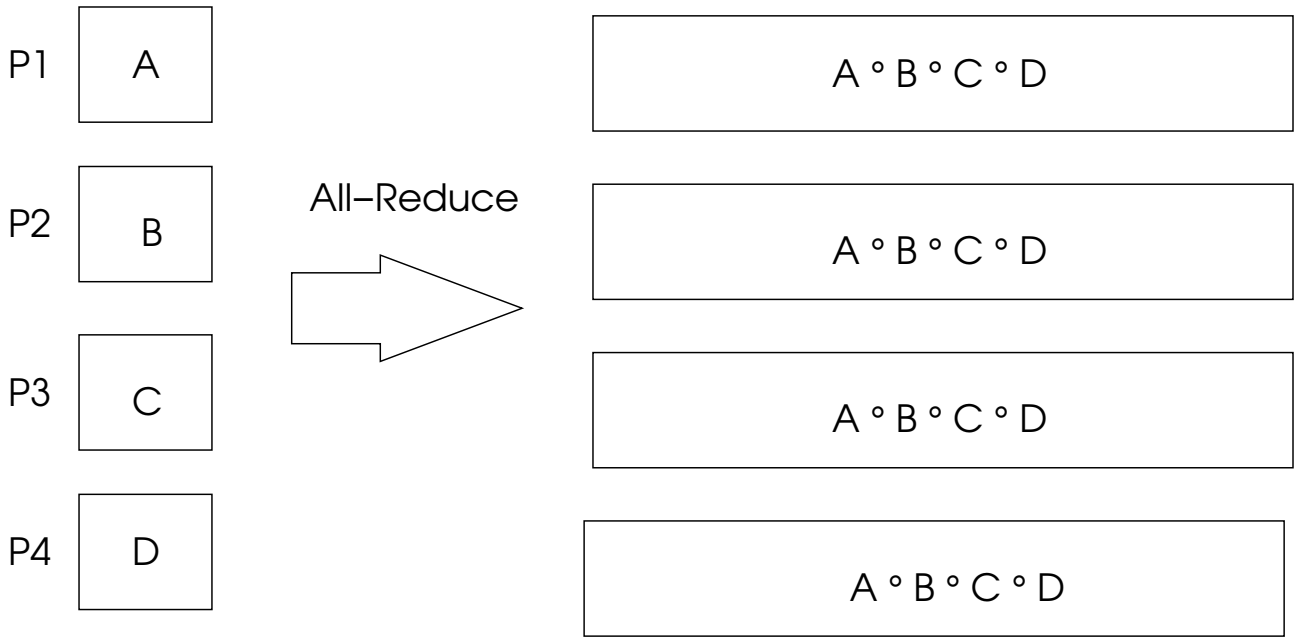
Executa da mesma maneira que o Reduce, porém os resultados são transmitidos para todos os nós do grupo.

```
MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
              MPI_Datatype dtype, MPI_Op op, MPI_Comm comm);
```

Assim o argumento root é omitido do protótipo da função.

All-Reduce

° : operador associativo



Redução

```
const int ROOT = 0;
sum = 0;
MPI_Reduce(&rank, &sum, 1, MPI_INT, MPI_SUM, ROOT, MPI_COMM_WORLD);
printf("Reduce: process %d has %3d \n", rank, sum);
sum = 0;
MPI_Allreduce(&rank, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
printf("Allreduce: process %d has %3d \n", rank, sum );
```

Redução

Exemplo de saída com 4 processos:

```
Reduce : process 1 has 0
Reduce : process 2 has 0
Reduce : process 3 has 0
Reduce : process 0 has 6
Allreduce : process 1 has 6
Allreduce : process 2 has 6
Allreduce : process 0 has 6
Allreduce : process 3 has 6
```

Soma de prefixos paralelo

Computação de prefixos paralelo

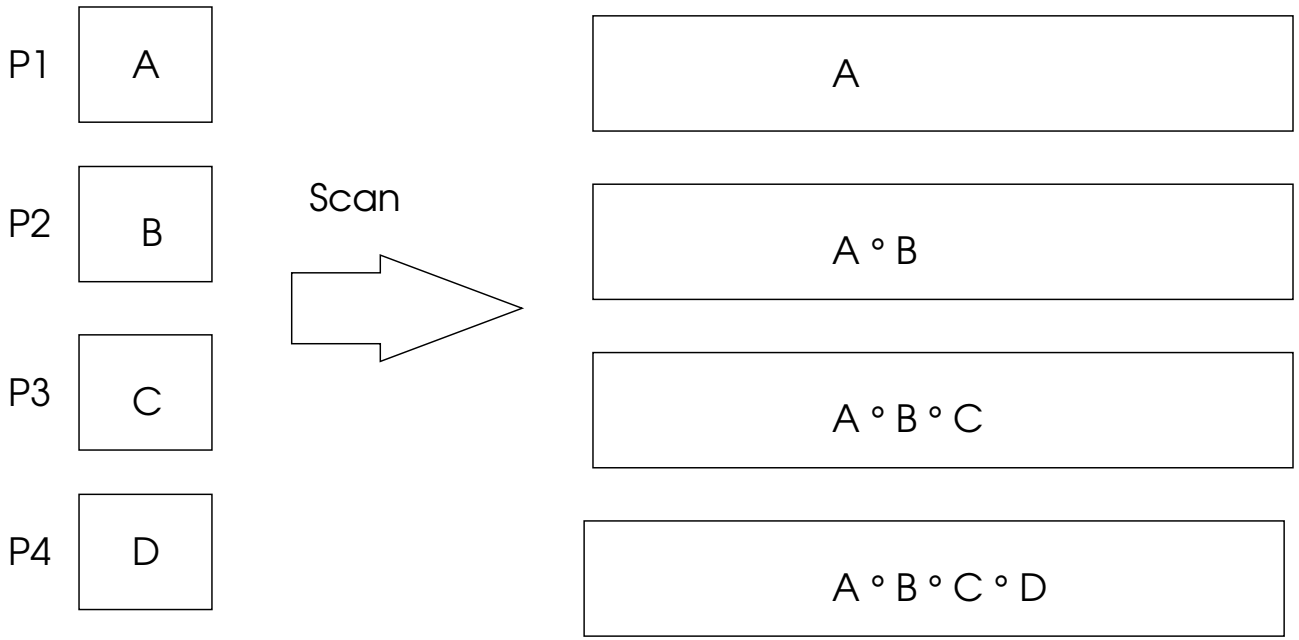
A função `MPI_Scan()` executa computação de prefixos em paralelo.

```
MPI_Scan(void *sendbuf, void *recvbuf, int count,
         MPI_Datatype dtype, MPI_Op op, MPI_Comm comm);
```

No processo i, a computação de prefixo calcula $v[0]$ op $v[1]$ op $\cdots$ op $v[i]$, que será armazenado em `recvbuf`.

Scan

° : operador associativo



Barreira

O MPI implementa uma barreira de comunicação que possibilita a sincronização de processos através da emissão de um sinal de controle que indica que todos os processos do grupo chamaram a função.

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

O único argumento do `MPI_Barrier()` é o comunicador que define o grupo de processos a serem sincronizados.

Fim