

Programação Paralela e Distribuída

Programação em Memória Distribuída com o MPI

Programação Paralela e Distribuída 2010/2011

Ricardo Rocha DCC-FCUP

Programação em Memória Distribuída

- As aplicações são vistas como um conjunto de programas que são executados de forma independente em diferentes processadores de diferentes computadores. A semântica da aplicação é mantida por troca de informação entre os vários programas.
- A sincronização e o modo de funcionamento da aplicação é da responsabilidade do programador. No entanto, o programador não quer desperdiçar muito tempo com os aspectos relacionados com a comunicação propriamente dita.
- Existem diferentes bibliotecas que implementam os pormenores da comunicação. Essas bibliotecas permitem executar programas remotamente, monitorizar o seu estado, e trocar informação entre os diferentes programas, sem que o programador precise de saber explicitamente como isso é conseguido.

Message-Passing Interface (MPI)

O que não é o MPI:

- O MPI não é um modelo revolucionário de programar máquinas paralelas. Pelo contrário, ele é um modelo de programação paralela baseado na **troca de mensagens** que pretendeu recolher as melhores funcionalidades dos sistemas existentes, aperfeiçoá-las e torná-las um **standard**.
- O MPI não é uma linguagem de programação. É um conjunto de rotinas (biblioteca) definido inicialmente para ser usado em programas C ou Fortran.
- O MPI não é a implementação. É apenas a especificação!

Principais objectivos:

- Aumentar a portabilidade dos programas.
- Aumentar e melhorar a funcionalidade.
- Conseguir implementações eficientes numa vasta gama de arquitecturas.
- Suportar ambientes heterogéneos.

2

Um Pouco de História

- O MPI nasceu em 1992 da cooperação entre universidades, empresas e utilizadores dos Estados Unidos e Europa (**MPI Forum** – <http://www.mpi-forum.org>) e foi publicado em Abril de 1994.
- Principais implementações:
 - ◆ **LAM**
<http://www.lam-mpi.org>
 - ◆ **MPICH**
<http://www.mcs.anl.gov/mpi/mpich>
 - ◆ **CHIMP**
<http://www.epcc.ed.ac.uk/chimp>
- Propostas de extensão foram entretanto estudadas e desenvolvidas:
 - ◆ **MPI-2**
 - ◆ **MPI-IO**

3

Single Program Multiple Data (SPMD)

- SPMD é um modelo de programação em que os vários programas que constituem a aplicação são incorporados num único executável.
- Cada processo executa uma cópia desse executável. Utilizando condições de teste sobre o **ranking** dos processos, diferentes processos executam diferentes partes do programa.

```
...
if (my_rank == 0) {
    // código tarefa 0
} ...
...
} else if (my_rank == N) {
    // código tarefa N
}
...
...
```

- O MPI não impõe qualquer restrição quanto ao modelo de programação (isso depende do suporte oferecido por cada implementação particular). Sendo assim, o modelo SPMD é aquele que oferece a aproximação mais portável.

4

Iniciar e Terminar o Ambiente de Execução do MPI

MPI_Init(int *argc, char ***argv)

- MPI_Init() inicia o ambiente de execução do MPI.

MPI_Finalize(void)

- MPI_Finalize() termina o ambiente de execução do MPI.
- Todas as funções MPI retornam 0 se OK, valor positivo se erro.

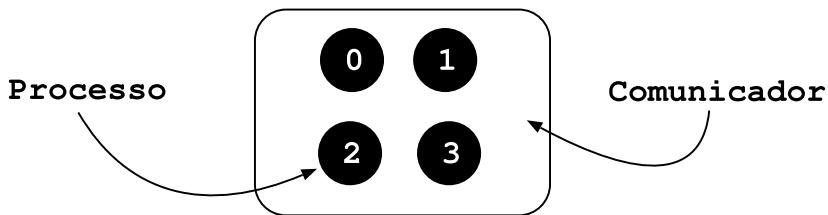
Estrutura Base de um Programa MPI

```
// incluir a biblioteca de funções MPI
#include <mpi.h>
...
main(int argc, char **argv) {
    ...
    // nenhuma chamada a funções MPI antes deste ponto
    MPI_Init(&argc, &argv);
    ...
    MPI_Finalize();
    // nenhuma chamada a funções MPI depois deste ponto
    ...
}
```

6

Comunicadores

- Uma aplicação MPI vê o seu ambiente de execução paralelo como um conjunto de grupos de processos.
- O **comunicador** é a estrutura de dados MPI que abstrai o conceito de grupo e define quais os processos que podem trocar mensagens entre si. Todas as funções de comunicação têm um argumento relativo ao comunicador.



- Por defeito, o ambiente de execução do MPI define um comunicador universal (**MPI_COMM_WORLD**) que engloba todos os processos em execução.
- Todos os processos possuem um identificador único (**rank**) que determina a sua posição (de 0 a N-1) no comunicador. Se um processo pertencer a mais do que um comunicador ele pode ter *rankings* diferentes em cada um deles.

7

Informação Relativa a um Comunicador

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

- MPI_Comm_rank() devolve em rank a posição do processo corrente no comunicador comm.

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

- MPI_Comm_size() devolve em size o total de processos no comunicador comm.

8

Mensagens MPI

- Na sua essência, as mensagens não são nada mais do que pacotes de informação trocados entre processos.
- Para efectuar uma troca de mensagens, o ambiente de execução necessita de conhecer no mínimo a seguinte informação:
 - ◆ Processo que envia.
 - ◆ Processo que recebe.
 - ◆ Localização dos dados na origem.
 - ◆ Localização dos dados no destino.
 - ◆ Tamanho dos dados.
 - ◆ Tipo dos dados.
- O tipo dos dados é um dos itens mais relevantes nas mensagens MPI. Daí, uma mensagem MPI ser normalmente designada como uma sequência de tipo de dados.

Tipos de Dados Básicos

MPI	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_PACKED	

10

Envio Standard de Mensagens

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

- MPI_Send() é a funcionalidade básica para envio de mensagens.
- buf é o endereço inicial dos dados a enviar.
- count é o número de elementos do tipo datatype a enviar.
- datatype é o tipo de dados a enviar.
- dest é a posição do processo, no comunicador comm, a quem se destina a mensagem.
- tag é uma marca que identifica a mensagem a enviar. As mensagens podem possuir idênticas ou diferentes marcas por forma a que o processo que as envia/recebe as possa agrupar/diferenciar em classes.
- comm é o comunicador dos processos envolvidos na comunicação.

11

Recepção Standard de Mensagens

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

- MPI_Recv() é a funcionalidade básica para recepção de mensagens.
- buf é o endereço onde devem ser colocados os dados recebidos.
- count é o número máximo de elementos do tipo datatype a receber (tem de ser maior ou igual ao número de elementos enviados).
- datatype é o tipo de dados a receber (não necessita de corresponder aos dados que foram enviados).

12

Recepção Standard de Mensagens

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

- source é a posição do processo, no comunicador comm, de quem se pretende receber a mensagem. Pode ser MPI_ANY_SOURCE para receber de qualquer processo no comunicador comm.
- tag é a marca que identifica a mensagem que se pretende receber. Pode ser MPI_ANY_TAG para receber qualquer mensagem.
- comm é o comunicador dos processos envolvidos na comunicação.
- status devolve informação sobre o processo emissor (status.MPI_SOURCE) e a marca da mensagem recebida (status.MPI_TAG). Se essa informação for desprezável indique MPI_STATUS_IGNORE.

13

Informação Relativa à Recepção

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

- `MPI_Get_count()` devolve em `count` o número de elementos do tipo `datatype` recebidos na mensagem associada com `status`.

```
MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- `MPI_Probe()` sincroniza a recepção da próxima mensagem, retornando em `status` informação sobre a mesma sem contudo proceder à sua recepção.
- A recepção deverá ser posteriormente feita com `MPI_Recv()`.
- É útil em situações em que não é possível conhecer antecipadamente o tamanho da mensagem e assim evitar que esta exceda o *buffer* de recepção.

14

I'm Alive! (`mpi_alive.c`)

```
#include <mpi.h>
#define STD_TAG 0

main(int argc, char **argv) {
    int i, my_rank, n_procs;
    char msg[100];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
    if (my_rank != 0) {
        sprintf(msg, "I'm alive!");
        MPI_Send(msg, strlen(msg) + 1, MPI_CHAR, 0, STD_TAG, MPI_COMM_WORLD);
    } else {
        for (i = 1; i < n_procs; i++) {
            MPI_Recv(msg, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                     MPI_COMM_WORLD, &status);
            printf("Proc %d: %s\n", status.MPI_SOURCE, msg);
        }
    }
    MPI_Finalize();
}
```

Modos de Comunicação

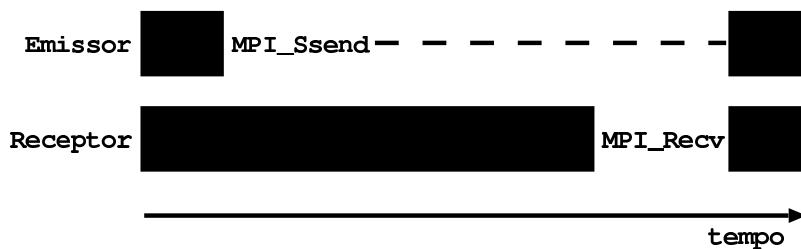
- Em MPI existem diferentes modos de comunicação para envio de mensagens:
 - ◆ **Standard:** MPI_Send()
 - ◆ **Synchronous:** MPI_Ssend()
 - ◆ **Buffered:** MPI_Bsend()
- Independentemente do modo de envio, a recepção é sempre feita através da chamada MPI_Recv().
- Em qualquer um dos modos a ordem das mensagens é sempre preservada:
 - ◆ O processo A envia N mensagens para o processo B fazendo N chamadas a qualquer uma das funções MPI_Send().
 - ◆ O processo B faz N chamadas a MPI_Recv() para receber as N mensagens.
 - ◆ O ambiente de execução garante que a 1^a chamada a MPI_Send() é emparelhada com a 1^a chamada a MPI_Recv(), a 2^a chamada a MPI_Send() é emparelhada com a 2^a chamada a MPI_Recv(), e assim sucessivamente.

16

Synchronous Send

```
MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int
          dest, int tag, MPI_Comm comm)
```

- Só quando o processo receptor confirmar que está pronto a receber é que o envio acontece. Até lá o processo emissor fica à espera.



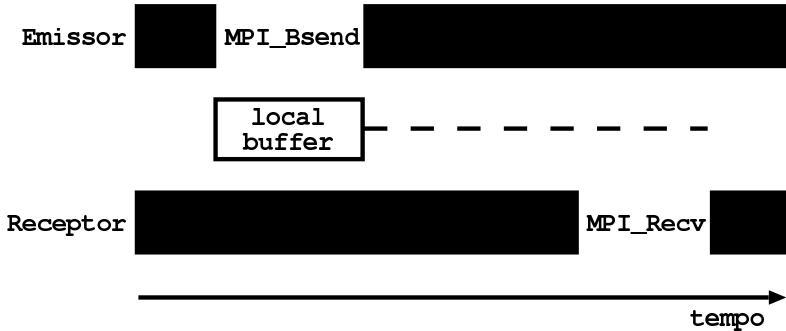
- Este tipo de comunicação só deve ser utilizado quando o processo emissor necessita de garantir a recepção antes de continuar a sua execução.
- Este método de comunicação pode ser útil para certas situações. No entanto, ele pode atrasar bastante a aplicação, pois enquanto o processo receptor não recebe a mensagem, o processo emissor poderia estar a executar trabalho útil.

17

Buffered Send

```
MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- A mensagem é copiada para um *buffer* local do programa e só depois enviada. O processo emissor não fica dependente da sincronização com o processo receptor, e pode desde logo continuar a sua execução.



- Tem a vantagem de não requerer sincronização, mas o inconveniente de se ter de definir explicitamente um *buffer* associado ao programa.

18

Buffered Send

```
MPI_Buffer_attach(void *buf, int size)
```

- MPI_Buffer_attach() informa o ambiente de execução do MPI que o espaço de tamanho *size bytes* a partir do endereço *buf* pode ser usado para *buffering* local de mensagens.

```
MPI_Buffer_detach(void **buf, int *size)
```

- MPI_Buffer_detach() informa o ambiente de execução do MPI que o actual *buffer* local de mensagens não deve ser mais utilizado. Se existirem mensagens pendentes no *buffer*, a função só retorna quando todas elas forem entregues.
- Em cada instante da execução só pode existir um único *buffer* associado a cada processo.
- A função MPI_Buffer_detach() não liberta a memória associada ao *buffer*, para tal é necessário invocar explicitamente a função free() do sistema.

Welcome! (mpi_welcome.c)

```

main(int argc, char **argv) {
    int buf_size; char *local_buf;
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
    buf_size = BUF_SIZE; local_buf = (char *) malloc(buf_size);
    MPI_Buffer_attach(local_buf, buf_size);
    sprintf(msg, "Welcome!");
    for (i = 0; i < n_procs; i++)
        if (my_rank != i)
            MPI_Bsend(msg, strlen(msg) + 1, MPI_CHAR, i, STD_TAG, MPI_COMM_WORLD);
    for (i = 0; i < n_procs; i++)
        if (my_rank != i) {
            sprintf(msg, "Argh!");
            MPI_Recv(msg, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                     MPI_COMM_WORLD, &status);
            printf("Proc %d->%d: %s\n", status.MPI_SOURCE, my_rank, msg);
        }
    MPI_Buffer_detach(&local_buf, &buf_size);
    free(local_buf);
    MPI_Finalize();
}

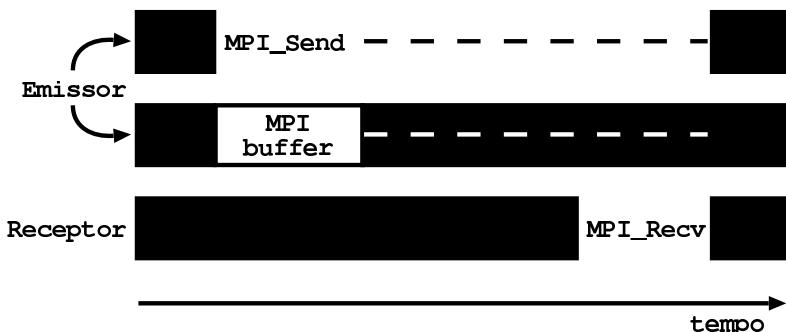
```

20

Standard Send

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

- Termina assim que a mensagem é enviada, o que não significa que tenha sido entregue ao processo receptor. A mensagem pode ficar pendente no ambiente de execução durante algum tempo (depende da implementação do MPI).



- Tipicamente, as implementações fazem *buffering* de mensagens pequenas e sincronizam nas grandes. Para escrever código portável, o programador não deve assumir que o envio termina antes nem depois de começar a recepção.

Envio e Recepção Simultânea de Mensagens

```
MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype
            sendtype, int dest, int sendtag, void *recvbuf, int
            recvcount, MPI_Datatype recvtype, int source, int recvtag,
            MPI_Comm comm, MPI_Status *status)
```

- MPI_Sendrecv() permite o envio e recepção simultânea de mensagens. É útil para quando se pretende utilizar comunicações circulares sobre um conjunto de processos, pois permite evitar o problema de ordenar correctamente as comunicações de modo a não ocorrerem situações de *deadlock*.
- sendbuf é o endereço inicial dos dados a enviar.
- sendcount é o número de elementos do tipo sendtype a enviar.
- sendtype é o tipo de dados a enviar.
- dest é a posição do processo no comunicador comm a quem se destina a mensagem.
- sendtag é uma marca que identifica a mensagem a enviar.

22

Envio e Recepção Simultânea de Mensagens

```
MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype
            sendtype, int dest, int sendtag, void *recvbuf, int
            recvcount, MPI_Datatype recvtype, int source, int recvtag,
            MPI_Comm comm, MPI_Status *status)
```

- recvbuf é o endereço onde devem ser colocados os dados recebidos.
- recvcount é o número máximo de elementos do tipo recvtype a receber.
- recvtype é o tipo de dados a receber.
- source é a posição do processo no comunicador comm de quem se pretende receber a mensagem.
- recvtag é a marca que identifica a mensagem que se pretende receber.
- comm é o comunicador dos processos envolvidos na comunicação.
- status devolve informação sobre o processo emissor.

23

Envio e Recepção Simultânea de Mensagens

- Os *buffers* de envio `sendbuf` e de recepção `recvbuf` devem ser necessariamente diferentes.
- As marcas `sendtag` e `recvtag`, os tamanhos `sendcount` e `recvcount`, e os tipos de dados `sendtype` e `recvtype` podem ser diferentes.
- Uma mensagem enviada por uma comunicação `MPI_Sendrecv()` pode ser recebida por qualquer outra comunicação usual de recepção de mensagens.
- Uma mensagem recebida por uma comunicação `MPI_Sendrecv()` pode ter sido enviada por qualquer outra comunicação usual de envio de mensagens.

24

Envio e Recepção Simultânea de Mensagens

```
MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype
datatype, int dest, int sendtag, int source, int recvtag,
MPI_Comm comm, MPI_Status *status)
```

- `MPI_Sendrecv_replace()` permite o envio e recepção simultânea de mensagens utilizando o mesmo *buffer* para o envio e para a recepção. No final da comunicação, a mensagem a enviar é substituída pela mensagem recebida.
- O *buffer* `buf`, o tamanho `count` e o tipo de dados `datatype` são utilizados para definir tanto a mensagem a enviar como a mensagem a receber.
- Uma mensagem enviada por uma comunicação `MPI_Sendrecv_replace()` pode ser recebida por qualquer outra comunicação usual de recepção de mensagens.
- Uma mensagem recebida por uma comunicação `MPI_Sendrecv_replace()` pode ter sido enviada por qualquer outra comunicação usual de envio de mensagens.

25

Comunicações Não Bloqueantes

- Uma comunicação diz-se **bloqueante** se a execução é interrompida enquanto a comunicação não sucede. Uma comunicação bloqueante só sucede quando o *buffer* da mensagem associado à comunicação pode ser reutilizado.
- Uma comunicação diz-se **não bloqueante** se a continuação da execução não depende do sucesso da comunicação. O *buffer* da mensagem associado a uma comunicação não bloqueante não deve, no entanto, ser reutilizado pela aplicação até que a comunicação suceda.
 - ◆ A ideia das comunicações não bloqueantes é iniciar o envio das mensagens o mais cedo possível, continuar de imediato com a execução, e verificar o mais tarde possível o sucesso das mesmas.
 - ◆ Uma chamada a uma função não bloqueante apenas anuncia ao ambiente de execução a existência de uma mensagem para ser enviada ou recebida. A função completa de imediato.
 - ◆ A comunicação fica completa quando num momento posterior o processo toma conhecimento do sucesso da comunicação.

26

Envio e Recepção Não Bloqueante de Mensagens

```
MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *req)
```

```
MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *req)
```

- Ambas as funções devolvem em req o identificador que permite a posterior verificação do sucesso da comunicação.

```
MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
```

- MPI_Iprobe() testa a chegada de uma mensagem associada com source, tag e comm sem contudo proceder à sua recepção. Retorna em flag o valor lógico que indica a chegada de alguma mensagem, e em caso positivo retorna em status informação sobre a mesma.
- A recepção deverá ser posteriormente feita com uma função de recepção.

27

Verificar o Sucesso duma Comunicação Não Bloqueante

```
MPI_Wait(MPI_Request *req, MPI_Status *status)
```

- `MPI_Wait()` bloqueia até que a comunicação identificada por `req` suceda. Retorna em `status` informação relativa à mensagem.

```
MPI_Test(MPI_Request *req, int *flag, MPI_Status *status)
```

- `MPI_Test()` testa se a comunicação identificada por `req` sucedeu. Retorna em `flag` o valor lógico que indica o sucesso da comunicação, e em caso positivo retorna em `status` informação relativa à mensagem.

Hello! (mpi_hello.c)

```
main(int argc, char **argv) {
    char recv_msg[100]; MPI_Request req[100];
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
    sprintf(msg, "Hello!");
    for (i = 0; i < n_procs; i++)
        if (my_rank != i) {
            MPI_Irecv(recv_msg, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                      MPI_COMM_WORLD, &(req[i]));
            MPI_Isend(msg, strlen(msg) + 1, MPI_CHAR, i, STD_TAG,
                      MPI_COMM_WORLD, &(req[i + n_procs]));
        }
    for (i = 0; i < n_procs; i++)
        if (my_rank != i) {
            sprintf(recv_msg, "Argh!");
            MPI_Wait(&(req[i + n_procs]), &status);
            MPI_Wait(&(req[i]), &status);
            printf("Proc %d->%d: %s\n", status.MPI_SOURCE, my_rank, recv_msg);
        }
    MPI_Finalize();
}
```

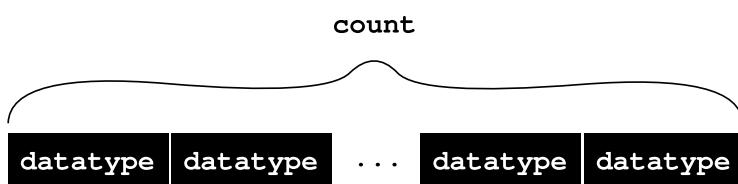
Que Tipo de Mensagens Devo Utilizar?

- Grande parte dos utilizadores utiliza as versões *standard* para envio e recepção de mensagens quando o ambiente de execução possui implementações eficientes dessas funções. No entanto, o uso dessas funções nem sempre garante a portabilidade da aplicação.
- Em alternativa, a utilização das versões *synchronous* e *standard* não bloqueante é suficiente para construir aplicações robustas e ao mesmo tempo portáveis.
- As versões não bloqueantes nem sempre levam a melhores resultados. A sua utilização só deve ser considerada quando existe uma clara e substancial sobreposição da computação.
- É perfeitamente possível enviar mensagens com funções não bloqueantes e receber com funções bloqueantes. O contrário é igualmente possível.

30

Agrupar Dados para Comunicação

- Em programação com troca de mensagens, uma heurística natural para maximizar a performance do sistema é **minimizar o número de mensagens a trocar**.
- Por defeito, todas as funções de envio e recepção de mensagens permitem agrupar numa mesma mensagem dados do mesmo tipo guardados em posições contíguas de memória.



- Para além desta funcionalidade básica o MPI permite:
 - ◆ Definir novos tipos de dados que agrupam dados de vários tipos.
 - ◆ Empacotar e desempacotar dados para/de um *buffer*.

31

Tipos Derivados

- A definição de novos tipos de dados é feita em tempo de execução:
 - ◆ Inicialmente, os processos devem construir o novo tipo derivado. A construção de tipos derivados é feita a partir dos tipos de dados básicos que o MPI define.
 - ◆ Em seguida, devem certificar perante o ambiente de execução do MPI a existência do novo tipo derivado.
 - ◆ Após a sua utilização, cada processo liberta a certificação feita.
- A construção de tipos derivados é dispendiosa, pelo que só deve ser utilizada quando o número de mensagens a trocar é significativo.
- Os novos tipos de dados são utilizados nas funções de envio e recepção de mensagens tal como os restantes tipos básicos. Para tal é necessário que ambos os processos emissor e receptor tenham certificado o novo tipo derivado. Normalmente, isso é feito na parte do código que é comum a ambos os processos.

32

Construção de Tipos Derivados

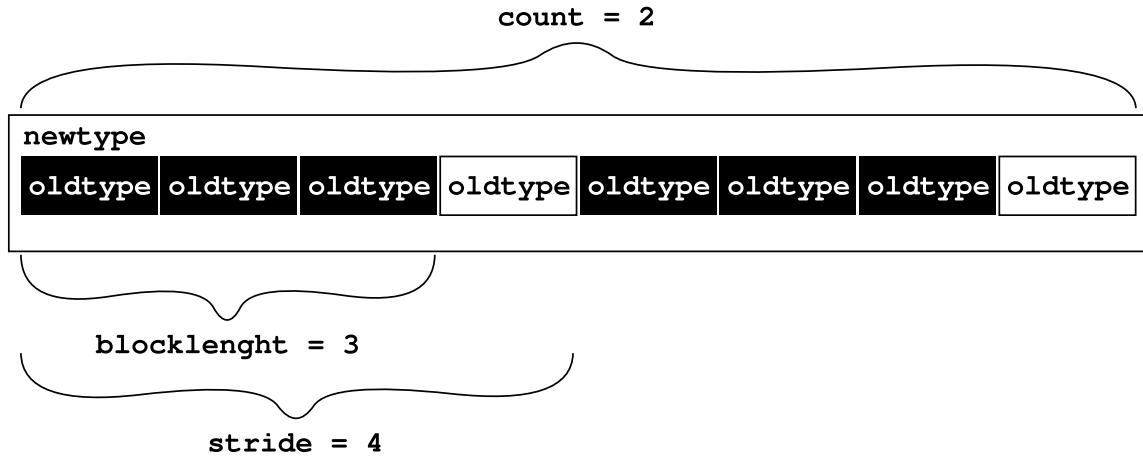
```
MPI_Type_vector(int count, int blocklength, int stride,
                 MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- `MPI_Type_vector()` constrói um novo tipo de dados a partir de um vector de dados.
- `count` é o número de blocos de dados do novo tipo derivado.
- `blocklength` é o número de elementos contíguos de cada bloco.
- `stride` é o número de elementos contíguos que separa o início de cada bloco (deslocamento).
- `oldtype` é o tipo de dados dos elementos do vector.
- `newtype` é o identificador do novo tipo derivado.

33

Construção de Tipos Derivados

```
MPI_Type_vector(int count, int blocklength, int stride,
                 MPI_Datatype oldtype, MPI_Datatype *newtype)
```



34

Construção de Tipos Derivados

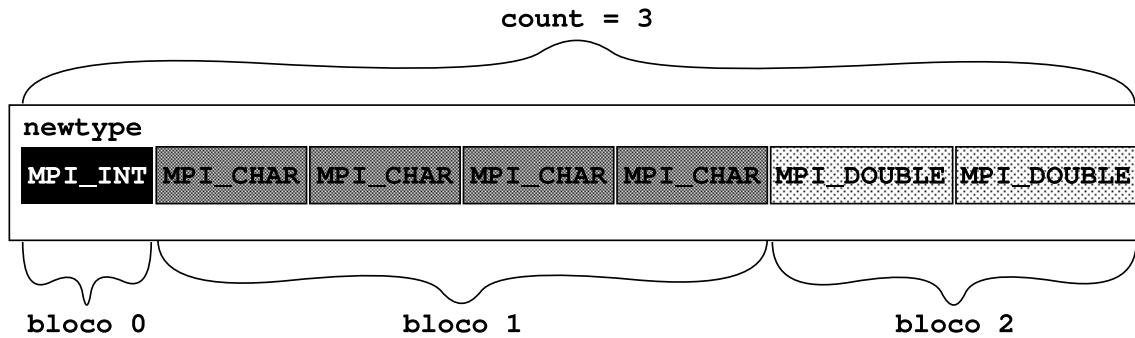
```
MPI_Type_struct(int count, int blocklengths[], MPI_Aint
                 displacements[], MPI_Datatype oldtypes[], MPI_Datatype
                           *newtype)
```

- **MPI_Type_struct()** constrói um novo tipo de dados a partir de uma estrutura de dados.
- **count** é o número de blocos de dados do novo tipo derivado. Representa igualmente o número de entradas nos vectores **blocklengths[]**, **displacements[]** e **oldtypes[]**.
- **blocklengths[]** é o número de elementos contíguos de cada bloco.
- **displacements[]** é o deslocamento em *bytes* de cada bloco.
- **oldtypes[]** é o tipo de dados dos elementos de cada bloco.
- **newtype** é o identificador do novo tipo derivado.

35

Construção de Tipos Derivados

```
MPI_Type_struct(int count, int blocklengths[], MPI_Aint
    displacements[], MPI_Datatype oldtypes[], MPI_Datatype
    *newtype)
```



```
blocklengths[3] = {1, 4, 2}
displacements[3] = {0, int_length, int_length + 4 * char_length}
oldtypes[3] = {MPI_INT, MPI_CHAR, MPI_DOUBLE}
```

36

Construção de Tipos Derivados

- Um tipo derivado é um objecto que especifica um conjunto de tipos básicos e os respectivos deslocamentos. Os tipos básicos indicam ao MPI como interpretar os bits, enquanto que os deslocamentos indicam onde se encontram esses bits.
- Para instanciar os deslocamentos dum novo tipo derivado deve utilizar-se uma das seguintes funções:

```
MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

- `MPI_Type_extent()` devolve em `extent` o tamanho em *bytes* do tipo de dados `datatype`.

```
MPI_Address(void *location, MPI_Aint *address)
```

- `MPI_Address()` devolve em `address` o endereço de memória de `location`.

Certificar e Libertar um Tipo Derivado

```
MPI_Type_commit(MPI_Datatype *datatype)
```

- `MPI_Type_commit()` certifica perante o ambiente de execução do MPI a existência de um novo tipo derivado identificado por `datatype`.

```
MPI_Type_free(MPI_Datatype *datatype)
```

- `MPI_Type_free()` liberta do ambiente de execução o tipo derivado identificado por `datatype`.

38

Extrair Colunas de Matrizes (mpi_vector.c)

```
int my_matrix[ROWS][COLS];
int my_vector[ROWS];
MPI_Datatype col_matrix;
...
// construir um tipo derivado com ROWS blocos
// de 1 elemento separados por COLS elementos
MPI_Type_vector(ROWS, 1, COLS, MPI_INT, &col_matrix);
MPI_Type_commit(&col_matrix);
...
// enviar a coluna 1 de my_matrix
MPI_Send(&my_matrix[0][1], 1, col_matrix, dest, tag, comm);
...
// receber uma dada coluna na coluna 3 de my_matrix
MPI_Recv(&my_matrix[0][3], 1, col_matrix, source, tag, comm, &status);
...
// receber uma dada coluna em my_vector
MPI_Recv(&my_vector[0], ROWS, MPI_INT, source, tag, comm, &status);
...
// libertar o tipo derivado
MPI_Type_free(&col_matrix);
```

Estruturas de Dados (mpi_struct.c)

```

struct { int a; char b[4]; double c[2]; } my_struct;
...
int blocklengths[3];
MPI_Aint int_length, char_length, displacements[3];
MPI_Datatype oldtypes[3];
MPI_Datatype struct_type;
...
// construir o tipo derivado representando my_struct
blocklengths[0] = 1; blocklengths[1] = 4; blocklengths[2] = 2;
oldtypes[0] = MPI_INT; oldtypes[1] = MPI_CHAR; oldtypes[2] = MPI_DOUBLE;
MPI_Type_extent(MPI_INT, &int_length);
MPI_Type_extent(MPI_CHAR, &char_length);
displacements[0] = 0;
displacements[1] = int_length;
displacements[2] = int_length + 4 * char_length;
MPI_Type_struct(3, blocklengths, displacements, oldtypes, &struct_type);
MPI_Type_commit(&struct_type);
...
// enviar my_struct
MPI_Send(&my_struct, 1, struct_type, dest, tag, comm);
...
// receber em my_struct
MPI_Recv(&my_struct, 1, struct_type, source, tag, comm, &status);

```

40

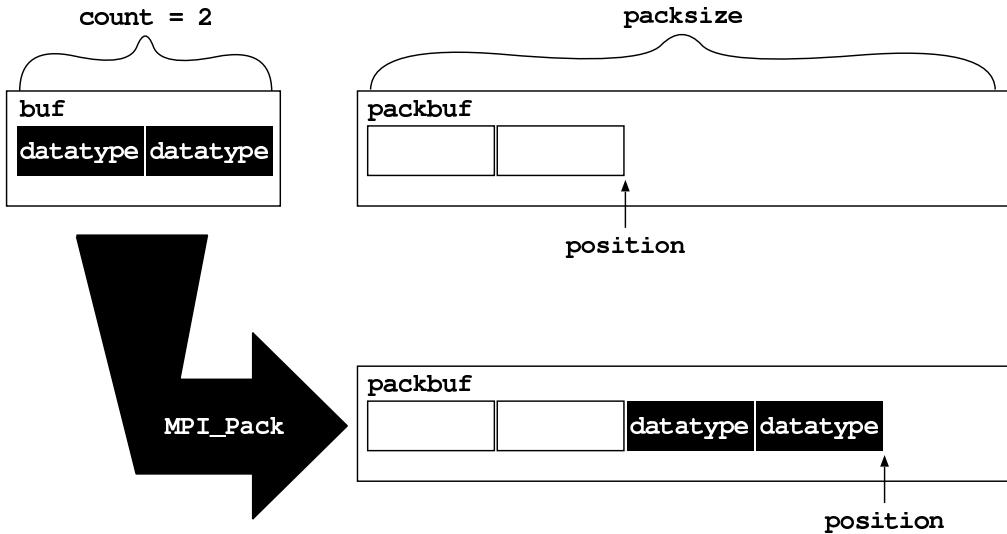
Empacotar Dados

```
MPI_Pack(void *buf, int count, MPI_Datatype datatype, void
         *packbuf, int packsize, int *position, MPI_Comm comm)
```

- MPI_Pack() permite empacotar dados não contíguos em posições contíguas de memória.
- buf é o endereço inicial dos dados a empacotar.
- count é o número de elementos do tipo datatype a empacotar.
- datatype é o tipo de dados a empacotar.
- packbuf é o endereço do *buffer* onde devem ser colocados os dados a empacotar.
- packsize é o tamanho em *bytes* do *buffer* de empacotamento.
- position é a posição (em *bytes*) do *buffer* a partir da qual os dados devem ser empacotados.
- comm é o comunicador dos processos envolvidos na comunicação.

Empacotar Dados

```
MPI_Pack(void *buf, int count, MPI_Datatype datatype, void
         *packbuf, int packsize, int *position, MPI_Comm comm)
```



42

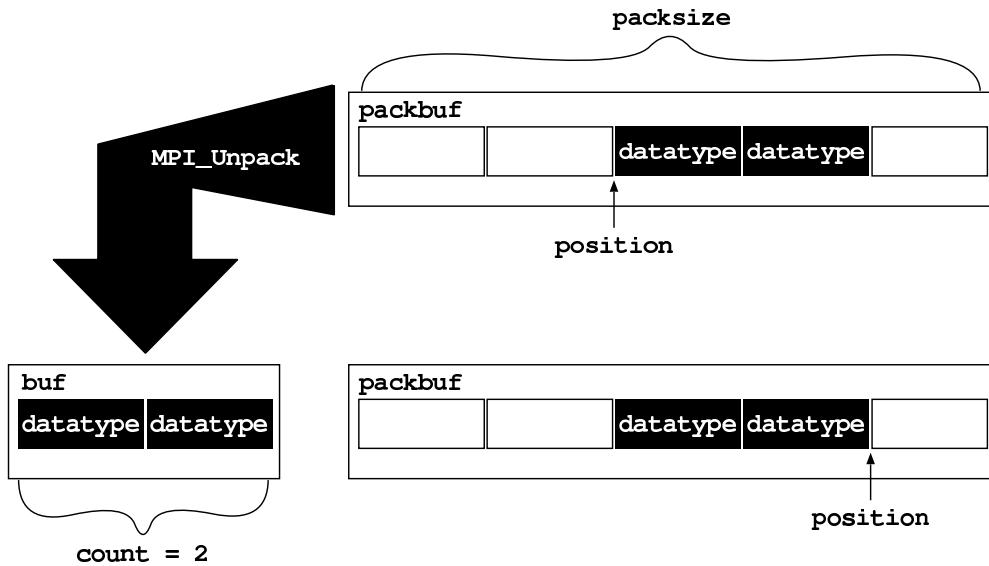
Desempacotar Dados

```
MPI_Unpack(void *packbuf, int packsize, int *position, void
           *buf, int count, MPI_Datatype datatype, MPI_Comm comm)
```

- `MPI_Unpack()` permite desempacotar dados contíguos em posições não contíguas de memória.
- `packbuf` é o endereço do *buffer* onde estão os dados a desempacotar.
- `packsize` é o tamanho em *bytes* do *buffer* de empacotamento.
- `position` é a posição (em *bytes*) do *buffer* a partir da qual estão os dados a desempacotar.
- `buf` é o endereço inicial para onde os dados devem ser desempacotados.
- `count` é o número de elementos do tipo `datatype` a desempacotar.
- `datatype` é o tipo de dados a desempacotar.
- `comm` é o comunicador dos processos envolvidos na comunicação.

Desempacotar Dados

```
MPI_Unpack(void *packbuf, int packsize, int *position, void
           *buf, int count, MPI_Datatype datatype, MPI_Comm comm)
```



44

Matriz de Tamanho Variável (mpi_pack.c)

```
// inicialmente ROWS e COLS não são do conhecimento do processo 1
int *my_matrix, ROWS, COLS, pos;
char pack_buf[BUF_SIZE];
...
if (my_rank == 0) {
    // empacotar e enviar ROWS, COLS e my_matrix
    pos = 0;
    MPI_Pack(&ROWS, 1, MPI_INT, pack_buf, BUF_SIZE, &pos, comm);
    MPI_Pack(&COLS, 1, MPI_INT, pack_buf, BUF_SIZE, &pos, comm);
    MPI_Pack(my_matrix, ROWS * COLS, MPI_INT, pack_buf, BUF_SIZE, &pos, comm);
    MPI_Send(pack_buf, pos, MPI_PACKED, 1, tag, comm);
} else if (my_rank == 1) {
    // receber e desempacotar ROWS, COLS e my_matrix
    MPI_Recv(&pack_buf, BUF_SIZE, MPI_PACKED, 0, tag, comm, &status);
    pos = 0;
    MPI_Unpack(&pack_buf, BUF_SIZE, &pos, &ROWS, 1, MPI_INT, comm);
    MPI_Unpack(&pack_buf, BUF_SIZE, &pos, &COLS, 1, MPI_INT, comm);
    // aloca espaço para representar my_matrix
    my_matrix = (int *) malloc(ROWS * COLS * sizeof(int));
    MPI_Unpack(&pack_buf, BUF_SIZE, &pos, my_matrix, ROWS * COLS, MPI_INT, comm);
}
...
}
```

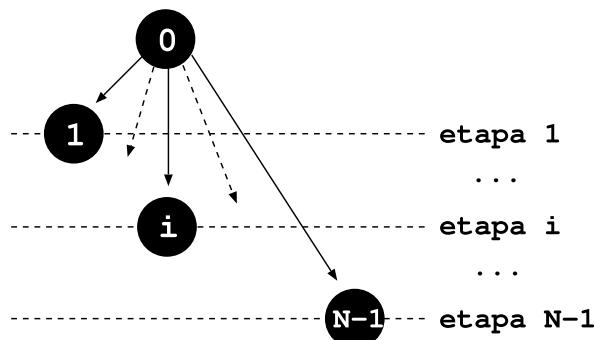
Que Tipo de Dados Devo Utilizar?

- Se os dados forem todos do mesmo tipo e se encontrarem em posições contíguas de memórias então devemos utilizar o argumento count das funções de envio e recepção de mensagens.
- Se os dados forem todos do mesmo tipo mas não se encontrarem em posições contíguas de memória, então devemos criar um tipo derivado utilizando as funções MPI_Type_vector() (para dados separados por intervalos regulares) ou MPI_Type_indexed() (para dados separados por intervalos irregulares).
- Se os dados forem heterogéneos e possuírem um determinado padrão constante então devemos criar um tipo derivado utilizando a função MPI_Type_struct().
- Se os dados forem heterogéneos mas não possuírem padrões regulares então devemos utilizar as funções MPI_Pack()/MPI_Unpack().
- As funções MPI_Pack()/MPI_Unpack() também podem ser utilizadas para trocar dados heterogéneos apenas uma vez (ou relativamente poucas vezes).

46

Comunicações Colectivas

- Em programação paralela é habitual que, em determinadas partes do programa, um dado processo distribua o mesmo conjunto de dados pelos restantes processos (e.g. iniciar dados ou tarefas).



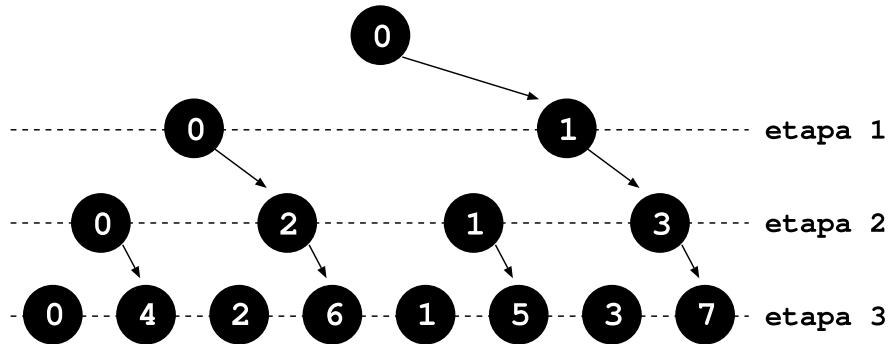
```
...
if (my_rank == 0)
    for (dest = 1; dest < n_procs; dest++)
        MPI_Send(data, count, datatype, dest, tag, MPI_COMM_WORLD);
    else
        MPI_Recv(data, count, datatype, 0, tag, MPI_COMM_WORLD, &status);
...

```

47

Comunicações Colectivas

- A topologia de comunicação do esquema anterior é inherentemente sequencial pois todas as comunicações são realizadas a partir do processo 0. Se mais processos colaborarem na distribuição da informação podemos reduzir significativamente o tempo total de comunicação.



- Se usarmos uma topologia em árvore, tal como na figura acima, podemos distribuir os dados em $\lceil \log_2 N \rceil$ etapas em vez de $N - 1$ como na situação anterior.

48

Comunicações Colectivas

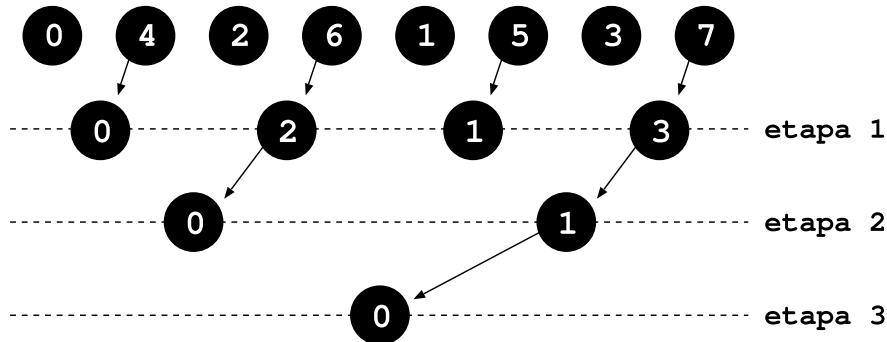
- Para implementar a topologia em árvore, cada processo precisa de calcular em cada etapa se é um processo emissor/receptor e qual o destino/origem dos dados a enviar/receber.
 - ◆ Se $my_rank < 2^{stage-1}$, então envio para $my_rank + 2^{stage-1}$.
 - ◆ Se $2^{stage-1} \leq my_rank < 2^{stage}$, então recebo de $my_rank - 2^{stage-1}$.
- Segue-se uma possível implementação:

```

...
for (stage = 1; stage <= upper_log2(n_procs); stage++)
  if (my_rank < pow(2, stage - 1))
    send_to(my_rank + pow(2, stage - 1));
  else if (my_rank >= pow(2, stage - 1) && my_rank < pow(2, stage))
    receive_from(my_rank - pow(2, stage - 1));
...
  
```

Comunicações Colectivas

- Em programação paralela é igualmente habitual que, em determinadas partes do programa, um dado processo (normalmente o processo 0) recolha informação dos restantes processos e calcule resumos dessa informação.
- Se invertermos a topologia de comunicação em árvore, podemos aplicar a mesma ideia para resumir dados em $\lceil \log_2 N \rceil$ etapas.



50

Mensagens Colectivas

- De forma a libertar o programador dos detalhes inerentes à topologia e eficiência das comunicações colectivas, o MPI define um conjunto de funções para lidar especificamente com este tipo de comunicações.
- Podemos então classificar as mensagens em:
 - ◆ **Ponto-a-ponto:** a mensagem é enviada por um processo e recebida por um outro processo (e.g. todo o tipo de mensagens que vimos anteriormente).
 - ◆ **Colectivas:** podem consistir de várias mensagens ponto-a-ponto concorrentes e envolvendo todos os processos de um comunicador (as mensagens colectivas têm de ser chamadas por todos os processos do comunicador).
- As mensagens colectivas são variações ou combinações das seguintes 4 operações primitivas:
 - ◆ **Broadcast**
 - ◆ **Reduce**
 - ◆ **Scatter**
 - ◆ **Gather**

51

Broadcast

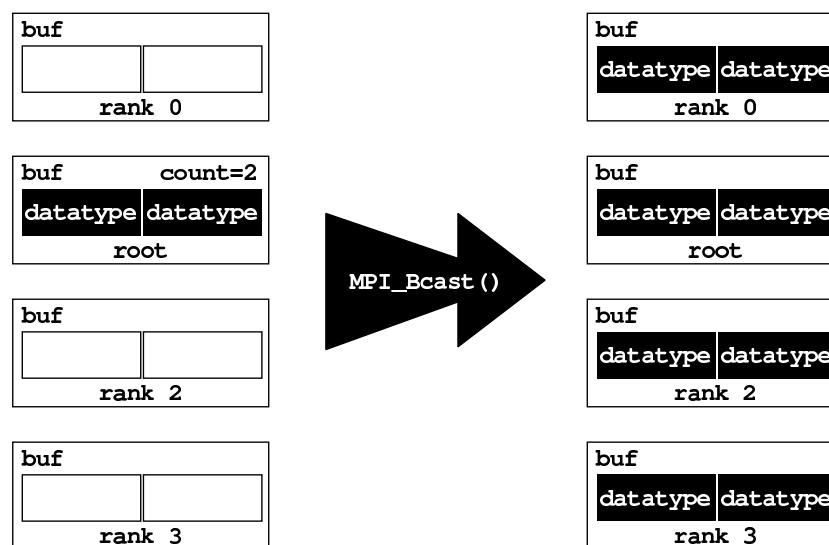
```
MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int
          root, MPI_Comm comm)
```

- MPI_Bcast() faz chegar uma mensagem de um processo a todos os outros processos no comunicador.
- buf é o endereço inicial dos dados a enviar/receber.
- count é o número de elementos do tipo datatype a enviar/receber.
- datatype é o tipo de dados a enviar/receber.
- root é a posição do processo, no comunicador comm, que possui à partida a mensagem a enviar.
- comm é o comunicador dos processos envolvidos na comunicação.

52

Broadcast

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



53

Reduce

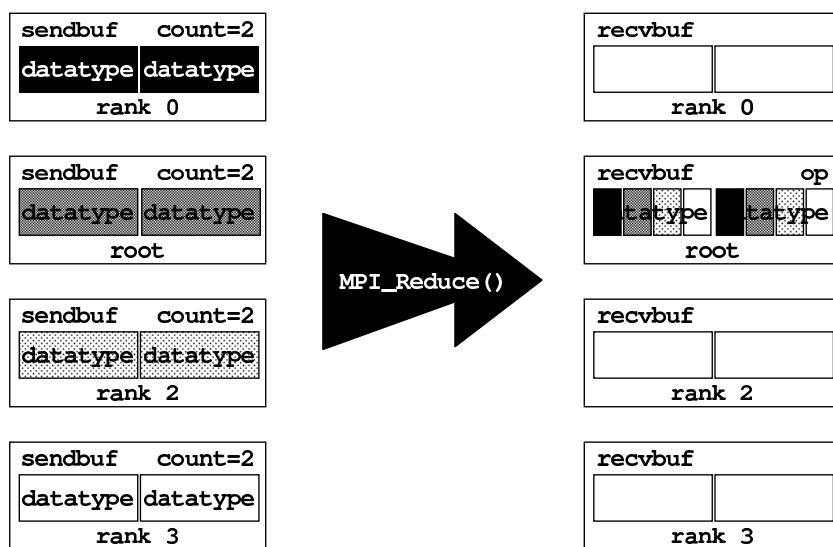
```
MPI_Reduce(void *sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

- `MPI_Reduce()` permite realizar operações globais de resumo fazendo chegar mensagens de todos os processos a um único processo no comunicador.
- `sendbuf` é o endereço inicial dos dados a enviar.
- `recvbuf` é o endereço onde devem ser colocados os dados recebidos (só é importante para o processo `root`).
- `count` é o número de elementos do tipo `datatype` a enviar.
- `datatype` é o tipo de dados a enviar.
- `op` é a operação de redução a aplicar aos dados recebidos.
- `root` é a posição do processo, no comunicador `comm`, que recebe e resume os dados.
- `comm` é o comunicador dos processos envolvidos na comunicação.

54

Reduce

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



55

Operações de Redução

Operação	Significado
MPI_MAX	máximo
MPI_MIN	mínimo
MPI_SUM	soma
MPI_PROD	produto
MPI_LAND	e lógico
MPI_BAND	e dos bits
MPI_LOR	ou lógico
MPI_BOR	ou dos bits
MPI_LXOR	ou exclusivo lógico
MPI_BXOR	ou exclusivo dos bits

56

Allreduce

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



57

Scatter

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

- MPI_Scatter() divide em partes iguais os dados de uma mensagem e distribuí ordenadamente cada uma das partes por cada um dos processos no comunicador.
- sendbuf é o endereço inicial dos dados a enviar (só é importante para o processo root).
- sendcount é o número de elementos do tipo sendtype a enviar para cada processo (só é importante para o processo root).
- sendtype é o tipo de dados a enviar (só é importante para o processo root).

58

Scatter

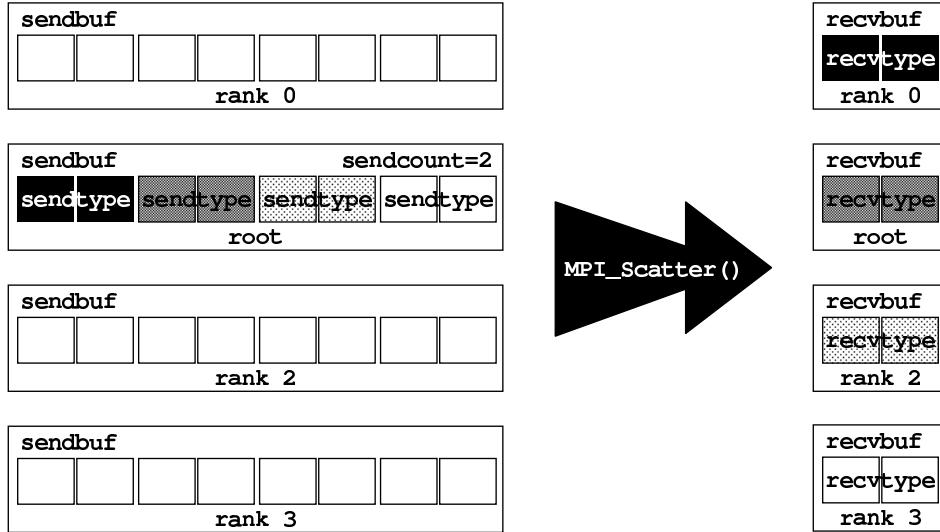
```
MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype  
           sendtype, void *recvbuf, int recvcount, MPI_Datatype  
           recvtype, int root, MPI_Comm comm)
```

- recvbuf é o endereço onde devem ser colocados os dados recebidos.
- recvcount é o número de elementos do tipo recvtype a receber (normalmente o mesmo que sendcount).
- recvtype é o tipo de dados a receber (normalmente o mesmo que sendtype).
- root é a posição do processo, no comunicador comm, que possui à partida a mensagem a enviar.
- comm é o comunicador dos processos envolvidos na comunicação.

59

Scatter

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



60

Gather

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

- MPI_Gather() recolhe ordenadamente num único processo um conjunto de mensagens oriundo de todos os processos no comunicador.
- sendbuf é o endereço inicial dos dados a enviar.
- sendcount é o número de elementos do tipo sendtype a enviar por cada processo.
- sendtype é o tipo de dados a enviar.

Gather

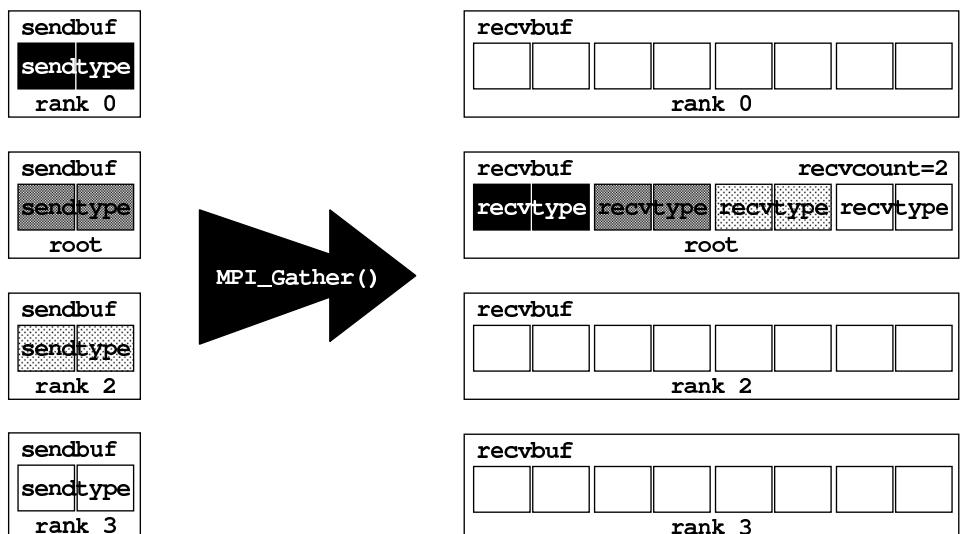
```
MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype
           sendtype, void *recvbuf, int recvcount, MPI_Datatype
           recvtype, int root, MPI_Comm comm)
```

- recvbuf é o endereço onde devem ser colocados os dados recebidos (só é importante para o processo root).
- recvcount é o número de elementos do tipo recvtype a receber de cada processo (normalmente o mesmo que sendcount; só é importante para o processo root).
- recvtype é o tipo de dados a receber (normalmente o mesmo que sendtype; só é importante para o processo root).
- root é a posição do processo, no comunicador comm, que recebe os dados.
- comm é o comunicador dos processos envolvidos na comunicação.

62

Gather

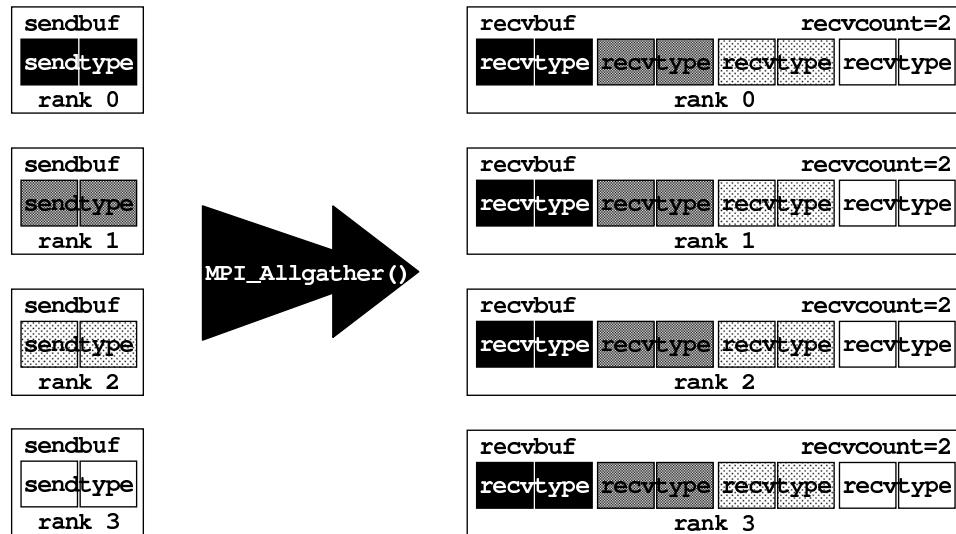
```
MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype
           sendtype, void *recvbuf, int recvcount, MPI_Datatype
           recvtype, int root, MPI_Comm comm)
```



63

Allgather

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



64

Produto Escalar

- O produto escalar de 2 vectores de dimensão N é definido por:

$$x \cdot y = x_0y_0 + x_1y_1 + \cdots + x_{n-1}y_{n-1}$$
- Se tivermos P processos, cada um deles pode calcular K (N/P) componentes do produto escalar:

Processo	Componentes
0	$x_0y_0 + x_1y_1 + \cdots + x_{k-1}y_{k-1}$
1	$x_ky_k + x_{k+1}y_{k+1} + \cdots + x_{2k-1}y_{2k-1}$
...	...
P - 1	$x_{(p-1)k}y_{(p-1)k} + x_{(p-1)k+1}y_{(p-1)k+1} + \cdots + x_{n-1}y_{n-1}$

- Segue-se uma possível implementação:

```
int produto_escalar(int x[], int y[], int n) {
    int i, pe = 0;
    for (i = 0; i < n; i++)
        pe = pe + x[i] * y[i];
    return pe;
}
```

65

Produto Escalar (mpi_escalar.c)

```

int *vector_x, *vector_y;
int K, pe, loc_pe, *loc_x, *loc_y;
...
if (my_rank == ROOT) {
    ... // calcular K e iniciar os vectores X e Y
}
// enviar K a todos os processos
MPI_Bcast(&K, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
// alocar espaço para os vectores locais
loc_x = (int *) malloc(K * sizeof(int));
loc_y = (int *) malloc(K * sizeof(int));
// distribuir as componentes dos vectores X e Y
MPI_Scatter(vector_x, K, MPI_INT, loc_x, K, MPI_INT, ROOT, MPI_COMM_WORLD);
MPI_Scatter(vector_y, K, MPI_INT, loc_y, K, MPI_INT, ROOT, MPI_COMM_WORLD);
// calcular o produto escalar
loc_pe = produto_escalar(loc_x, loc_y, K);
MPI_Reduce(&loc_pe, &pe, 1, MPI_INT, MPI_SUM, ROOT, MPI_COMM_WORLD);
// apresentar o resultado
if (my_rank == ROOT)
    printf("Produto Escalar = %d\n", pe);
...

```

66

Produto Matriz-Vector (mpi_produto.c)

- Sejam `matrix[ROWS,COLS]` e `vector[COLS]` respectivamente uma matriz e um vector coluna. O produto matriz-vector é um vector linha `result[ROWS]` em que cada `result[i]` é o produto escalar da linha `i` da matriz pelo vector.
- Se tivermos `ROWS` processos, cada um deles pode calcular um elemento do vector resultado.

```

int ROWS, COLS, *matrix, *vector, *result;
int pe, *linha;
... // iniciar ROWS, COLS e o vector
if (my_rank == ROOT) { ... // iniciar a matriz }
// distribuir a matriz
MPI_Scatter(matrix, COLS, MPI_INT, linha, COLS, MPI_INT, ROOT, MPI_COMM_WORLD);
// calcular o produto matriz-vector e apresentar o resultado
pe = produto_escalar(linha, vector, COLS);
MPI_Gather(&pe, 1, MPI_INT, result, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
if (my_rank == ROOT) {
    printf("Produto Matriz-Vector: ");
    for (i = 0; i < ROWS; i++)
        printf("%d ", result[i]);
}
...

```

67

Comunicadores

- Um comunicador pode ser descrito como um grupo de processos que podem trocar mensagens entre si. Associado a um comunicador temos:
 - ◆ **Um grupo:** conjunto ordenado de processos.
 - ◆ **Um contexto:** estrutura de dados que o identifica de forma única.
- Para além do comunicador universal, o ambiente de execução do MPI permite criar novos comunicadores. O MPI distingue 2 tipos de comunicadores:
 - ◆ **Intra-comunicadores:** permitem a troca de mensagens e realização de operações colectivas.
 - ◆ **Inter-comunicadores:** permitem a troca de mensagens entre processos pertencentes a intra-comunicadores disjuntos.

68

Comunicadores

- A sequência básica para criar novos comunicadores é a seguinte:
 - ◆ Obter o grupo de processos de um comunicador existente.
 - ◆ Criar um novo grupo a partir do anterior, indicando quais os processos que dele devem fazer parte.
 - ◆ Criar um novo comunicador com base no novo grupo.
 - ◆ Após a sua utilização, libertar os grupos e o comunicador.
- Para além desta sequência de utilização, o ambiente de execução do MPI disponibiliza funções específicas para criação automática de comunicadores.

69

Criar Grupos

```
MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

- `MPI_Comm_group()` devolve em `group` o grupo de processos do comunicador `comm`.

```
MPI_Group_incl(MPI_Group old_group, int size, int ranks[],  
               MPI_Group *new_group)
```

- `MPI_Group_incl()` cria um novo grupo `new_group` a partir de `old_group` constituído pelos `size` processos referenciados em `ranks[]`.

```
MPI_Group_excl(MPI_Group old_group, int size, int ranks[],  
               MPI_Group *new_group)
```

- `MPI_Group_excl()` cria um novo grupo `new_group` a partir de `old_group` e exclui os `size` processos referenciados em `ranks[]`.

70

Criar Comunicadores

```
MPI_Comm_create(MPI_Comm old_comm, MPI_Group group, MPI_Comm  
                *new_comm)
```

- `MPI_Comm_create()` cria um novo comunicador `new_comm` constituído pelo grupo de processos `group` do comunicador `old_comm`.
- `MPI_Comm_create()` é uma comunicação colectiva, pelo que deve ser chamada por todos os processos, incluindo aqueles que não aderem ao novo comunicador.
- No caso de serem criados vários comunicadores, a ordem de criação deve ser a mesma em todos os processos.

Libertar Grupos e Comunicadores

```
MPI_Group_free(MPI_Group *group)
```

- MPI_Group_free() liberta o grupo group do ambiente de execução.

```
MPI_Comm_free(MPI_Comm *comm)
```

- MPI_Comm_free() liberta o comunicador comm do ambiente de execução.

72

Processos Pares (mpi_even.c)

```

MPI_Group world_group, even_group;
MPI_Comm even_comm;
...
for (i = 0; i < n_procs; i += 2)
    ranks[i/2] = i;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
MPI_Group_incl(world_group, (n_procs + 1)/2, ranks, &even_group);
MPI_Comm_create(MPI_COMM_WORLD, even_group, &even_comm);
MPI_Group_free(&world_group);
MPI_Group_free(&even_group);
if (my_rank % 2 == 0) {
    MPI_Comm_rank(even_comm, &even_rank);
    printf("Rank: world %d even %d\n", my_rank, even_rank);
    MPI_Comm_free(&even_comm);
}
...

```

Criar Comunicadores

```
MPI_Comm_dup(MPI_Comm old_comm, MPI_Comm *new_comm)
```

- `MPI_Comm_dup()` cria um novo comunicador `new_comm` idêntico a `old_comm`.

```
MPI_Comm_split(MPI_Comm old_comm, int split_key, int rank_key,
                MPI_Comm *new_comm)
```

- `MPI_Comm_split()` cria um ou mais comunicadores `new_comm` a partir de `old_comm` agrupando em cada novo comunicador os processos com idênticos valores de `split_key` e ordenando-os por `rank_key`.
- Os *ranks* dos processos nos novos comunicadores são atribuídos por ordem crescente do argumento `rank_key`. Ou seja, o processo com o menor `rank_key` terá *rank* 0, o segundo menor *rank* 1, e assim sucessivamente.
- Os processos que não pretendam aderir a nenhum novo comunicador devem indicar em `split_key` a constante `MPI_UNDEFINED`.

74

Processos Pares e Ímpares (`mpi_split.c`)

```
MPI_Comm split_comm;
...
MPI_Comm_split(MPI_COMM_WORLD, my_rank % 2, my_rank, &split_comm);
MPI_Comm_rank(split_comm, &split_rank);
printf("Rank: world %d split %d\n", my_rank, split_rank);
MPI_Comm_free(&split_comm);
...
```

- Se executarmos o exemplo com 5 processos obtemos o seguinte *output*:

```
Rank: world 0 split 0
Rank: world 1 split 0
Rank: world 2 split 1
Rank: world 3 split 1
Rank: world 4 split 2
```

Topologias

- O modelo de programação do MPI é independente da topologia física de comunicação existente entre os processadores disponíveis no sistema.
- No entanto, de modo a aumentar o desempenho das comunicações duma aplicação, a topologia física do hardware deverá coincidir o mais possível com a topologia de comunicação da aplicação.
- O MPI permite que o programador defina a topologia de um comunicador na esperança de que o ambiente de execução do MPI use essa informação de modo a optimizar o desempenho das comunicações nesse comunicador. O MPI permite definir dois tipos de topologias:
 - ◆ **Grelhas cartesianas**
 - ◆ **Grafos arbitrários**

76

Criar Grelha Cartesiana

```
MPI_Cart_create(MPI_Comm old_comm, int ndims, int dims[], int
                 periods[], int reorder, MPI_Comm *new_comm)
```

- `MPI_Cart_create()` cria um novo comunicador idêntico a `old_comm` em que o grupo de processos é organizado como sendo uma grelha cartesiana.
- `old_comm` é o identificador do comunicador a partir do qual será criado o novo comunicador representando a grelha cartesiana.
- `ndims` é o número de dimensões da grelha cartesiana. Representa igualmente o número de entradas nos vectores `dims[]` e `periods[]`.
- `dims[]` especifica o número de processos em cada dimensão da grelha cartesiana.
- `periods[]` especifica se as dimensões são periódicas ou não, ou seja, se o último processo de cada dimensão comunica ou não com o primeiro processo da mesma dimensão. Valor 1 indica que a dimensão é periódica, valor 0 indica que não é.

77

Criar Grelha Cartesiana

```
MPI_Cart_create(MPI_Comm old_comm, int ndims, int dims[], int
                periods[], int reorder, MPI_Comm *new_comm)
```

- reorder indica se as posições dos processos da grelha cartesiana devem ser iguais às do comunicador old_comm ou se estas podem ser reordenadas pelo ambiente de execução do MPI de modo a optimizar o desempenho das futuras comunicações. Valor 0 indica que as posições devem ser as mesmas, valor 1 indica que podem ser reordenadas.
- new_comm é o identificador do novo comunicador que representa a grelha cartesiana.
- MPI_Cart_create() é uma comunicação colectiva, pelo que deve ser chamada por todos os processos do comunicador old_comm.

78

Informação Relativa a uma Grelha Cartesiana

```
MPI_Cart_coords(MPI_Comm comm, int rank, int ndims, int coords[])
```

- MPI_Cart_coords() devolve em coords[] as ndims coordenadas do processo que tem posição rank na grelha cartesiana representada pelo comunicador comm.

```
MPI_Cart_rank(MPI_Comm comm, int coords[], int *rank)
```

- MPI_Cart_rank() devolve em rank a posição do processo de coordenadas coords[] na grelha cartesiana representada pelo comunicador comm.

79

Grelha Cartesiana (mpi_cart.c)

```

MPI_Comm grid_comm;
int up, down, right, left, reorder, dims[2], periods[2], coords[2];
...
dims[0] = 4; dims[1] = 2; periods[0] = 1; periods[1] = 1; reorder = 0;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &grid_comm);
MPI_Comm_rank(grid_comm, &grid_rank);
MPI_Cart_coords(grid_comm, grid_rank, 2, coords);
coords[0] -= 1; MPI_Cart_rank(grid_comm, coords, &up);
coords[0] += 2; MPI_Cart_rank(grid_comm, coords, &down); coords[0] -= 1;
coords[1] -= 1; MPI_Cart_rank(grid_comm, coords, &left);
coords[1] += 2; MPI_Cart_rank(grid_comm, coords, &right); coords[1] -= 1;
printf("Rank: world %d grid %d (%d,%d) --> (up %d down %d left %d right %d)\n",
      my_rank, grid_rank, coords[0], coords[1], up, down, left, right);
...

```

- Se executarmos o exemplo com 8 processos obtemos o seguinte *output*:

```

Rank: world 0 grid 0 (0,0) --> (up 6 down 2 left 1 right 1)
Rank: world 1 grid 1 (0,1) --> (up 7 down 3 left 0 right 0)
Rank: world 2 grid 2 (1,0) --> (up 0 down 4 left 3 right 3)
Rank: world 3 grid 3 (1,1) --> (up 1 down 5 left 2 right 2)
Rank: world 4 grid 4 (2,0) --> (up 2 down 6 left 5 right 5)
Rank: world 5 grid 5 (2,1) --> (up 3 down 7 left 4 right 4)
Rank: world 6 grid 6 (3,0) --> (up 4 down 0 left 7 right 7)
Rank: world 7 grid 7 (3,1) --> (up 5 down 1 left 6 right 6)

```

80

Criar Comunicadores a Partir de uma Grelha Cartesiana

```
MPI_Cart_sub(MPI_Comm old_comm, int dims[], MPI_Comm *new_comm)
```

- `MPI_Cart_sub()` cria um ou mais comunicadores `new_comm` a partir da grelha cartesiana representada pelo comunicador `old_comm` agrupando em cada novo comunicador os processos segundo as dimensões especificadas em `dims[]`. Valor 1 indica que a dimensão faz parte dos novos comunicadores, valor 0 indica que não.
- As coordenadas dos processos nos novos comunicadores `new_comm` são as mesmas que no comunicador `old_comm` para as dimensões que fazem parte dos novos comunicadores.
- `MPI_Cart_sub()` é uma comunicação colectiva, pelo que deve ser chamada por todos os processos do comunicador `old_comm`.
- A funcionalidade conseguida com `MPI_Cart_sub()` é semelhante à conseguida com `MPI_Comm_split()`. A diferença é que `MPI_Cart_sub()` só pode ser usado sobre um comunicador relativo a uma grelha cartesiana e os novos comunicadores só podem ser criados por combinação das suas dimensões.

Criar Comunicadores a Partir de uma Grelha Cartesiana

```
MPI_Cart_sub(MPI_Comm old_comm, int dims[], MPI_Comm *new_comm)
```

- Consideremos uma grelha cartesiana tri-dimensional de dimensões $4 \times 2 \times 5$.
 - ◆ Se $\text{dims}[]$ for igual a $\{1,0,1\}$ isso significa que serão criados 2 novos comunicadores bi-dimensionais de dimensões 4×5 . O processo com coordenados $(2,1,3)$ na grelha tri-dimensional terá coordenadas $(2,3)$ no novo comunicador.
 - ◆ Se $\text{dims}[]$ for igual a $\{0,0,1\}$ isso significa que serão criados 8 novos comunicadores uni-dimensionais de dimensão 5. O processo com coordenados $(2,1,3)$ na grelha tri-dimensional terá coordenadas (3) no novo comunicador.

82

Processos em Coluna (mpi_subcart.c)

```

MPI_Comm grid_comm, col_comm;
...
dims[0] = 4; dims[1] = 2; periods[0] = 1; periods[1] = 1; reorder = 0;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &grid_comm);
MPI_Comm_rank(grid_comm, &grid_rank);
dims[0] = 1; // a dimensão 0 faz parte dos novos comunicadores
dims[1] = 0; // a dimensão 1 não faz parte dos novos comunicadores
MPI_Cart_sub(grid_comm, dims, &col_comm);
MPI_Comm_rank(col_comm, &row_rank);
printf("Rank: world %d grid %d row %d\n", my_rank, grid_rank, row_rank);
...

```

- Se executarmos o exemplo com 8 processos obtemos o seguinte *output*:

```

Rank: world 0 grid 0 row 0
Rank: world 1 grid 1 row 0
Rank: world 2 grid 2 row 1
Rank: world 3 grid 3 row 1
Rank: world 4 grid 4 row 2
Rank: world 5 grid 5 row 2
Rank: world 6 grid 6 row 3
Rank: world 7 grid 7 row 3

```

83

Medir o Tempo de Execução

```
double MPI_Wtime(void)
```

- `MPI_Wtime()` retorna o tempo em segundos que passou desde um determinado ponto arbitrário no passado.

```
double MPI_Wtick(void)
```

- `MPI_Wtick()` retorna a precisão da função `MPI_Wtime()`. Por exemplo, se `MPI_Wtime()` for incrementado a cada microsegundo então `MPI_Wtick()` retorna 0.000001.

```
MPI_Barrier(MPI_Comm comm)
```

- `MPI_Barrier()` sincroniza todos os processos no comunicador. Cada processo bloqueia até que todos os processos no comunicador chamem `MPI_Barrier()`.

84

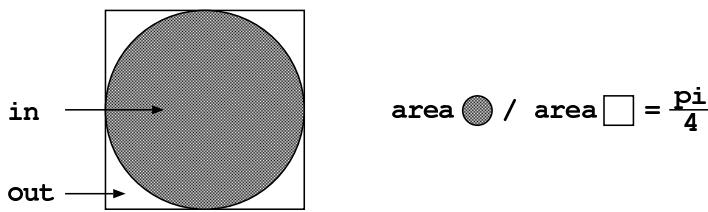
Medir o Tempo de Execução (`mpi_time.c`)

```
double start, finish;
...
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
...
// parte da execução a medir
...
MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();
if (my_rank == 0)
    printf("Tempo de Execução: %f segundos\n", finish - start);
...
```

- Os valores devolvidos por `MPI_Wtime()` são em tempo real, ou seja, todo o tempo que o processo possa ter estado interrompido pelo sistema é igualmente contabilizado.

Cálculo de π

- O valor de π pode ser calculado por aproximação utilizando o método de Monte Carlo.



- A ideia é a seguinte:
 - ◆ Gerar N pontos aleatórios (x, y) .
 - ◆ Para cada ponto (x, y) verificar se $x * x + y * y < 1$ e em função disso incrementar *in* ou *out*.
 - ◆ Calcular o valor aproximado de π como $4 * \text{in}/(\text{in} + \text{out})$.

86

Cálculo de π

Como proceder em paralelo?

- Definir um dos processos como servidor de sequências de números aleatórios.
- Com os restantes processos definir um novo comunicador de clientes.
- Os clientes pedem sucessivamente sequências de números ao servidor, verificam onde caem os pontos resultantes de cada par de números e propagam essa informação aos restantes clientes.
- Quando o total de pontos processados for superior a N a computação termina e um dos processos escreve o resultado aproximado do cálculo de π .

Cálculo de π (mpi_pi.c)

```

...
// define novo comunicador para os clientes
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
ranks[0] = SERVER;
MPI_Group_excl(world_group, 1, ranks, &worker_group);
MPI_Comm_create(MPI_COMM_WORLD, worker_group, &workers_comm);
MPI_Group_free(&worker_group);
MPI_Group_free(&world_group);
// obtém o número de pontos e propaga a todos
if (my_rank == ROOT) scanf("%d", &total_points);
MPI_Bcast(&total_points, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
// inicia a contagem do tempo
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
// calcula o valor aproximado de PI
if (my_rank == SERVER) { ... // servidor } else { ... // cliente }
// termina a contagem do tempo e escreve resultado
MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();
if (my_rank == ROOT) {
    printf("PI = %.20f\n", (4.0 * total_in) / (total_in + total_out));
    printf("Tempo de Execução = %f segundos\n", finish - start);
}
...

```

88

Cálculo de π (mpi_pi.c)

```

// servidor
if (my_rank == SERVER) {
    do {
        MPI_Recv(&req_points, 1, MPI_INT, MPI_ANY_SOURCE,
                 REQUEST, MPI_COMM_WORLD, &status);
        if (req_points) {
            for (i = 0; i < req_points; i++)
                rands[i] = random();
            MPI_Send(rands, req_points, MPI_INT, status.MPI_SOURCE,
                     REPLY, MPI_COMM_WORLD);
        }
    } while (req_points);
}

```

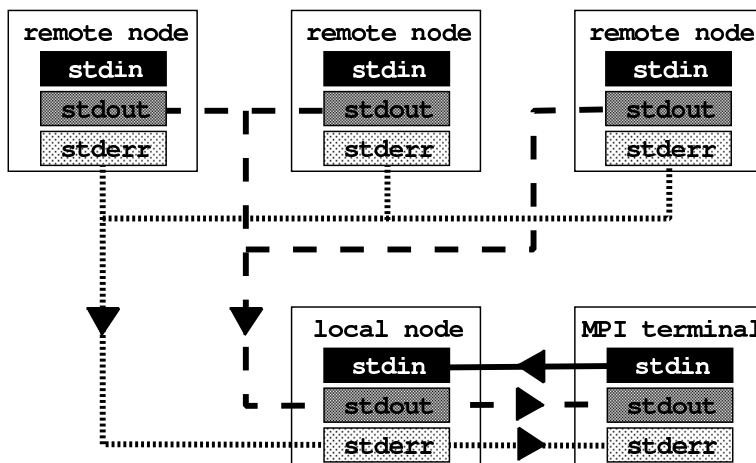
Cálculo de π (mpi_pi.c)

```
// cliente
if (my_rank != SERVER) {
    in = out = 0;
    do {
        req_points = REQ_POINTS;
        MPI_Send(&req_points, 1, MPI_INT, SERVER,
                 REQUEST, MPI_COMM_WORLD);
        MPI_Recv(rands, req_points, MPI_INT, SERVER,
                 REPLY, MPI_COMM_WORLD, &status);
        for (i = 0; i < req_points; i += 2) {
            x = (((double) rands[i]) / RAND_MAX) * 2 - 1;
            y = (((double) rands[i+1]) / RAND_MAX) * 2 - 1;
            (x * x + y * y < 1.0) ? in++ : out++;
        }
        MPI_Allreduce(&in, &total_in, 1, MPI_INT, MPI_SUM, workers_comm);
        MPI_Allreduce(&out, &total_out, 1, MPI_INT, MPI_SUM, workers_comm);
        req_points = (total_in + total_out < total_points);
        if (req_points == 0 && my_rank == ROOT)
            MPI_Send(&req_points, 1, MPI_INT, SERVER, REQUEST, MPI_COMM_WORLD);
    } while (req_points);
    MPI_Comm_free(&workers_comm);
}
```

90

Standard I/O

- O *standard input* é redireccionado para `/dev/null` em todos os nós remotos. O nó local (aquele onde o utilizador invoca o comando que inicia a execução) herda o *standard input* do terminal onde a execução é iniciada.
- O *standard output* e o *standard error* são redireccionados em todos os nós para o terminal onde a execução é iniciada.



91

Compilação e Execução de Programas

- De forma a facilitar o processo de compilação, as implementações MPI disponibilizam um conjunto de *scripts* para tratar dos caminhos dos *headers* e *libraries* necessários à compilação.
 - ◆ mpicc (*script* de compilação para programas MPI escritos em C)
 - ◆ mpiCC (*script* de compilação para programas MPI escritos em C++)
 - ◆ mpif77 (*script* de compilação para programas MPI escritos em Fortran)
- O comando mpirun permite iniciar a execução distribuída de um dado programa MPI. Para tal é necessário indicar a seguinte informação:
 - ◆ A topologia do conjunto de máquinas a considerar (esquema de arranque).
 - ◆ O número de unidades de execução a lançar por máquina ou por CPU.

92

Esquema de Arranque

- No esquema de arranque deve especificar-se o nome das máquinas a utilizar e o número de CPUs por máquina, se mais do que 1 (cpu=2).

```
# cluster com 4 máquinas e 6 CPUs
node1
node2
node3 cpu=2
node4 cpu=2
```

- A cada máquina no esquema de arranque é atribuído um identificador de posição, sendo n0 para a primeira máquina, n1 para a segunda, etc. Para além da identificação explícita das máquinas, é usual os comandos MPI permitirem a utilização de mnemónicas especiais, tais como:
 - ◆ N (referência a todas as máquinas)
 - ◆ C (referência a todos os CPUs)

93