Programação Paralela

Parte IV

Programação em Memória Distribuída

Ricardo Rocha
ricroc@ncc.up.pt
DCC-FCUP

MI/MIAC 2002/03

Programação em Memória Distribuída

As aplicações são vistas como um conjunto de programas que são executados de forma independente em diferentes processadores de diferentes máquinas.

- A sincronização e o modo de funcionamento da aplicação é da responsabilidade do programador. A semântica da aplicação é mantida por troca de informação entre os vários programas.
- No entanto, o programador deve desperdiçar o menor tempo possível 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 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 estandardizado de rotinas (biblioteca) definido para ser usado em programas C (ou Fortran).
- O MPI não é uma implementação. É apenas uma especificação.

Objectivos:

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

MPI: Um Pouco de História

- Nasceu em 1992 da cooperação entre universidades, empresas e utilizadores dos US e Europa (**MPI Forum** – `http://www.mpi-forum.org`) e foi publicado em Abril de 1994.
- Principais implementações:
  - **LAM**/Ohio Supercomputer Center
    `http://www.lam-mpi.org`
  - **MPICH**/Argonne National Lab & Mississippi State University
    `http://www.mcs.anl.gov/mpi/mpich`
  - **CHIMP**/Edinburgh Parallel Computing Centre
    `http://www.epcc.ed.ac.uk/chimp`
- Propostas de extensão foram entretanto estudadas e desenvolvidas:
  - **MPI-2**
  - **MPI-IO**

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) {
  <executa-tarefa-0>
} ...
...
} else if (my_rank == N) {
  <executa-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.

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.

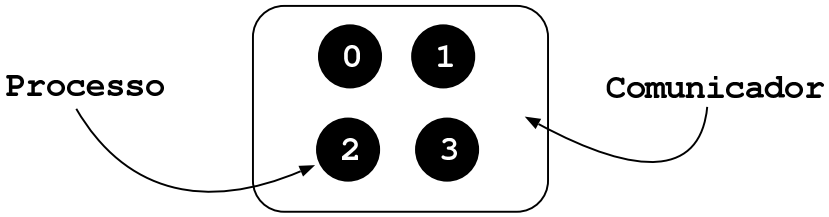
- Todas as funções MPI retornam 0 se OK, valor positivo se erro.

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
  ...
}
```


Comunicadores

- Uma aplicação MPI vê o seu ambiente de execução paralelo como um conjunto de grupos de processos.
- Um 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.



- 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.
- Por defeito, o ambiente de execução do MPI define um comunicador universal (MPI_COMM_WORLD) que engloba todos os processos em execução.

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.

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	

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.

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 (tem de corresponder aos dados que foram enviados).

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 fornece informação útil sobre o processo emissor (status.MPI_SOURCE) e a marca da mensagem recebida (status.MPI_TAG), quando se opta por usar *wildcards*. Se essa informação for desprezável indique MPI_STATUS_IGNORE.

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.

Exemplo: *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();
}
```

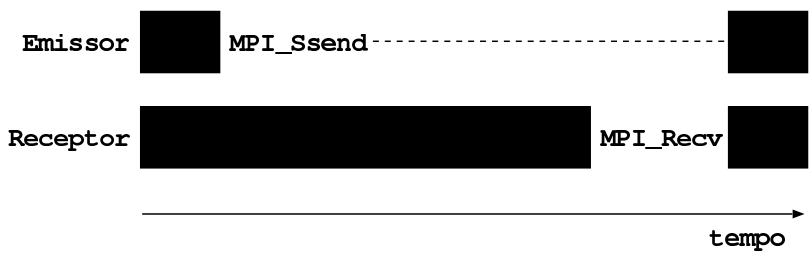
Modos de Comunicação

- Em MPI existem 4 modos de comunicação para envio de mensagens:
  - **Standard:** `MPI_Send()`
  - **Synchronous:** `MPI_Ssend()`
  - **Buffered:** `MPI_Bsend()`
  - **Ready:** `MPI_Rsend()`
- 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.

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.

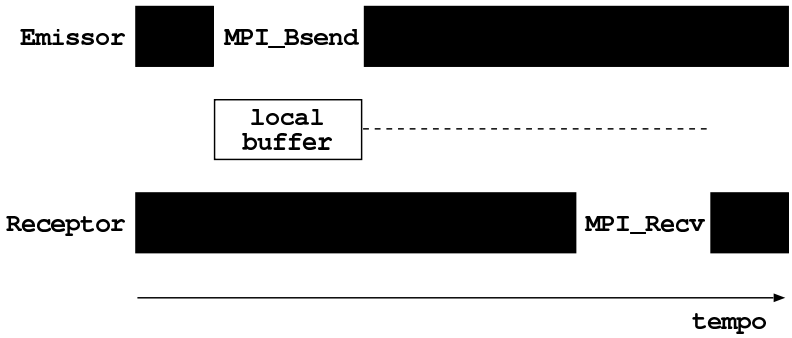


- Este tipo de comunicação só deve ser utilizado quando o processo emissor necessita de garantir a recepção antes de continuar a sua execução.
- Este método de comunicação pode ser útil para certas situações. No entanto, ele pode atrasar bastante a aplicação, pois enquanto o processo receptor não recebe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

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.

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.

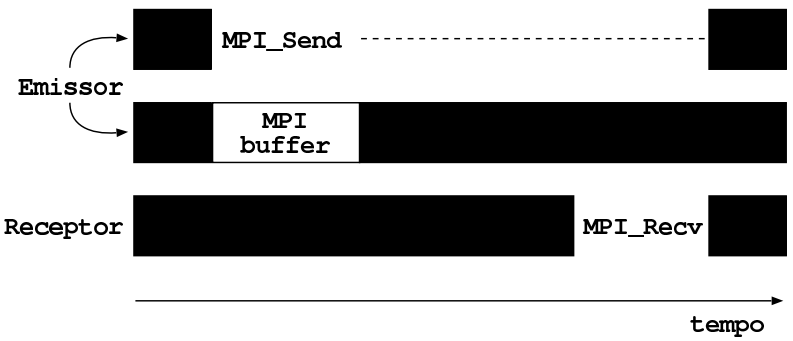
Exemplo: *Welcome!* (mpi_welcome.c)

```
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();
}
```


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.

Ready Send

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

- Esta forma de envio completa imediatamente. A comunicação sucede normalmente se aparecer um receptor já preparado para receber. Se não existir um receptor, o resultado da comunicação é indefinido. O programa deve estar logicamente correcto para que funcione.

- Esta forma de comunicação é útil em casos muitos restritos, daí a maior parte dos programadores raramente a utilizarem.

Comunicações Bloqueantes Versus 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. Exemplos:
  - MPI_Send()
  - MPI_Ssend()
  - MPI_Rsend()
  - MPI_Recv()
  - MPI_Probe()
- 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. Exemplo:
  - MPI_Bsend()
- O MPI não considera no entanto a função MPI_Bsend() puramente não bloqueante porque esta utiliza um *buffer* local ao processo.

Comunicações Não Bloqueantes

- 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.
- 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.
- Quando o mais cedo e o mais tarde possível são o mesmo momento, então a utilidade deste tipo de comunicações não é nenhum.
- O MPI define um conjunto próprio de funções para lidar com comunicações não bloqueantes.

Envio Não Bloqueante de Mensagens

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

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

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

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

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

Recepção Não Bloqueante de Mensagens

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

- MPI_Irecv() devolve 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.

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.

Exemplo: *Hello!* (mpi_hello.c)

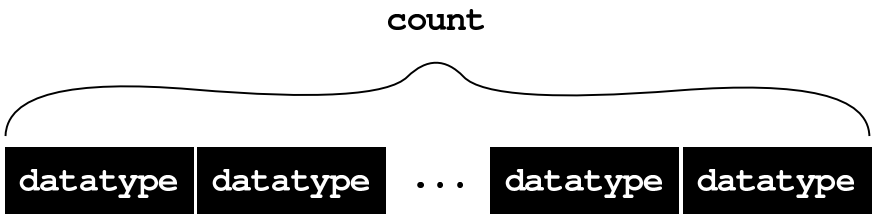
```
main(int argc, char **argv) {
  char recv_msg[100]; MPI_Request req[100];
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
  sprintf(msg, "Hello!");
  for (i = 0; i < n_procs; i++)
    if (my_rank != i) {
      MPI_Irecv(recv_msg, 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 *buffered*, *synchronous* e *synchronous* 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.

Agrupar Dados para Comunicação

- Uma heurística natural em programação com troca de mensagens 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*.

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.

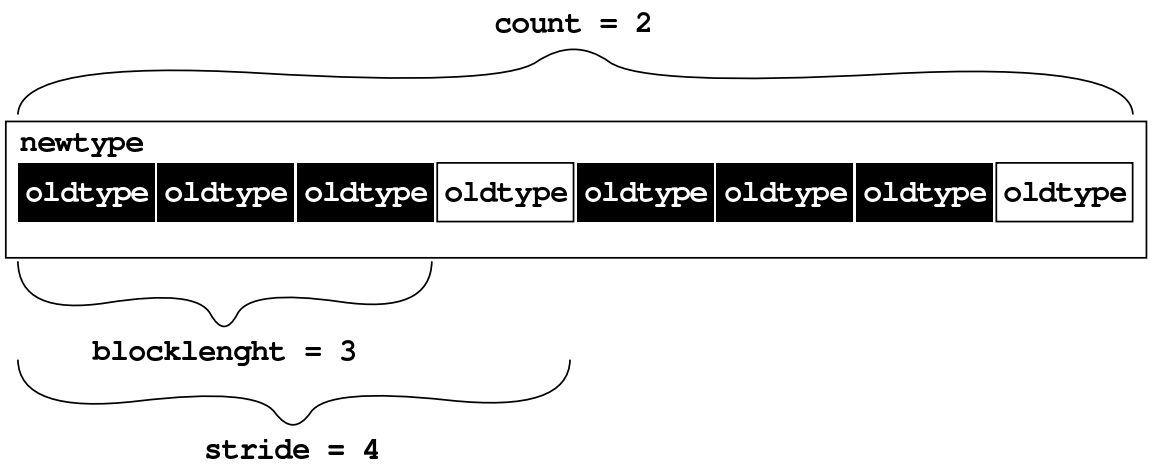
Construção de Tipos Derivados

```
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.

Construção de Tipos Derivados

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



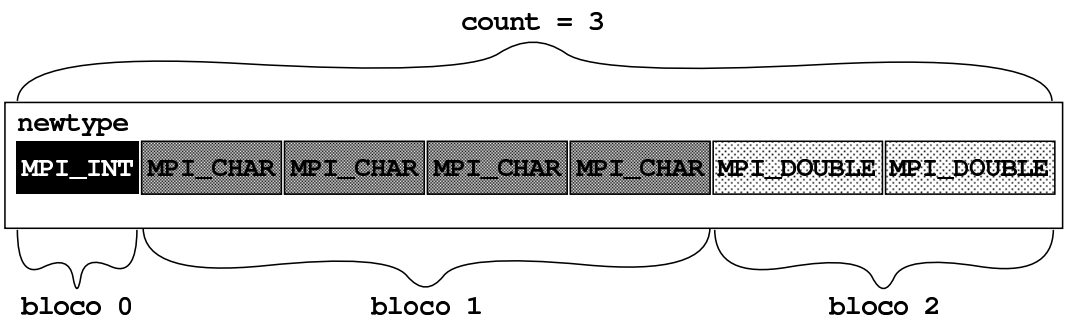
Construção de Tipos Derivados

```
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.

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}
```

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.

Exemplo: 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, ROWS, MPI_INT, source, tag, comm, &status);
...
// libertar o tipo derivado
MPI_Type_free(&col_matrix);
```


Exemplo: 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);
```

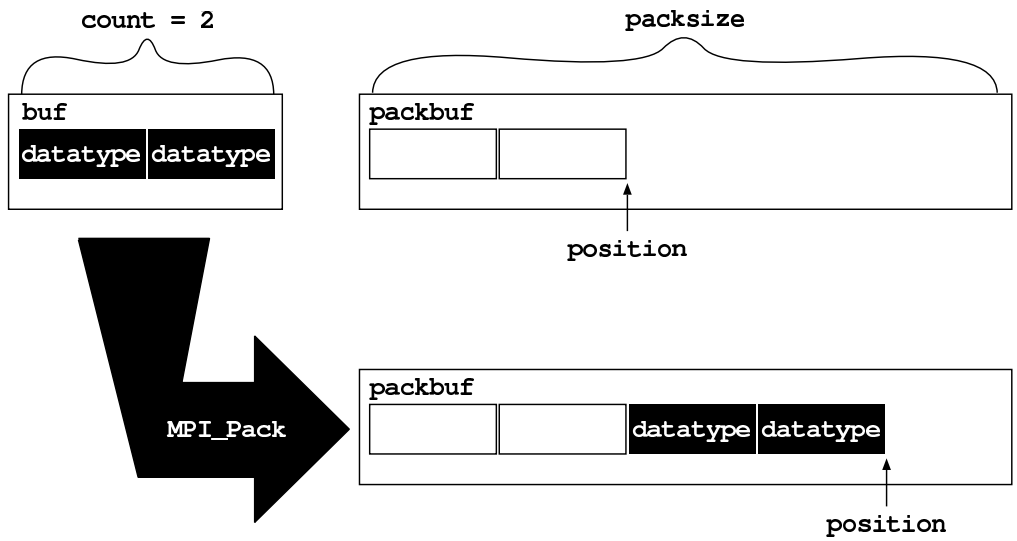
Empacotar Dados

```
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)
```



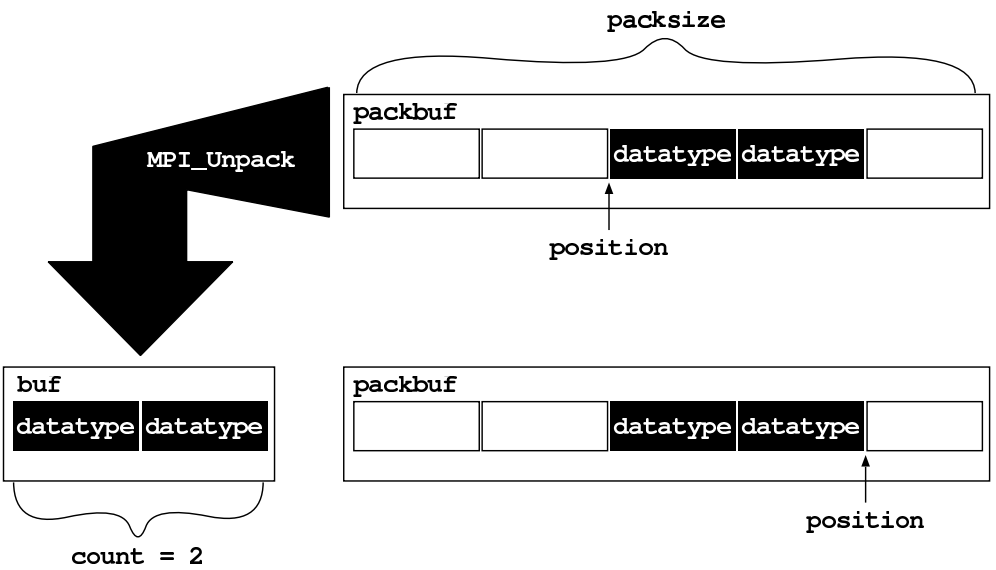
Desempacotar Dados

```
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)
```



Exemplo: Matriz de Tamanho Variável (mpi_pack.c)

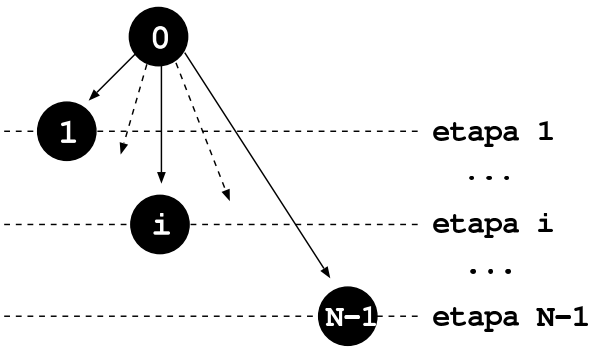
```
// inicialmente ROWS e COLS não são do conhecimento do processo 1
int *my_matrix, ROWS, COLS, pos;
char pack_buf[BUF_SIZE];
...
if (my_rank == 0) {
  // empacotar e enviar ROWS, COLS e my_matrix
  pos = 0;
  MPI_Pack(&ROWS, 1, MPI_INT, pack_buf, BUF_SIZE, &pos, comm);
  MPI_Pack(&COLS, 1, MPI_INT, 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).

Comunicações Colectivas

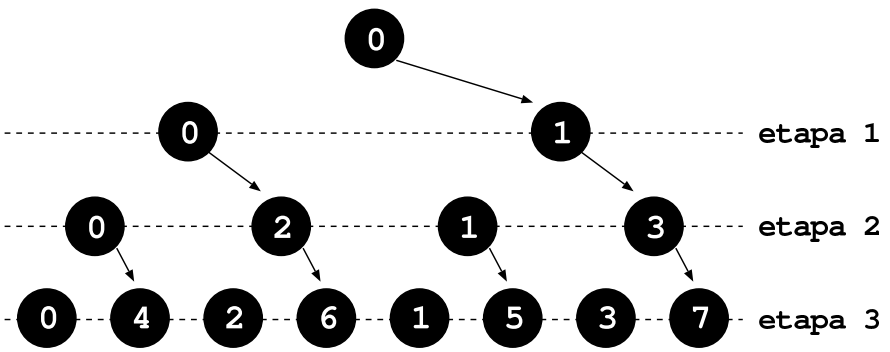
➤ Em programação paralela é habitual que, em determinadas partes do programa, um dado processo distribua o mesmo conjunto de dados pelos restantes processos (e.g. iniciar dados ou tarefas).




```
...
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);
...
```


Comunicações Colectivas

➤ A topologia de comunicação do esquema anterior é inerentemente 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.

Comunicações Colectivas

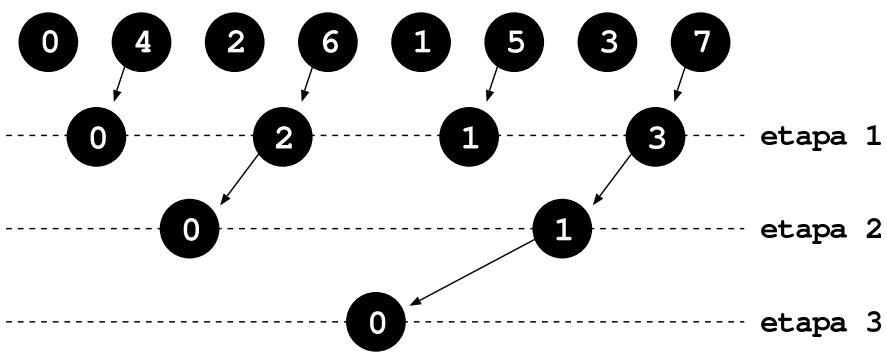
- Para implementar a topologia em árvore, cada processo precisa de calcular em cada etapa se é um processo emissor/receptor e qual o destino/origem dos dados a enviar/receber.
  - Se $my_rank < 2^{stage-1}$, então envio para $my_rank + 2^{stage-1}$.
  - Se $2^{stage-1} \leq my_rank < 2^{stage}$, então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



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:
  - ***Point-to-point:*** 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 *point-to-point* concorrentes e envolvendo todos os processos de um comunicador. As mensagens colectivas são chamadas com os mesmos argumentos por todos os processos.
- As mensagens colectivas são variações ou combinações das seguintes 4 operações primitivas:
  - **Broadcast**
  - **Reduce**
  - **Scatter**
  - **Gather**

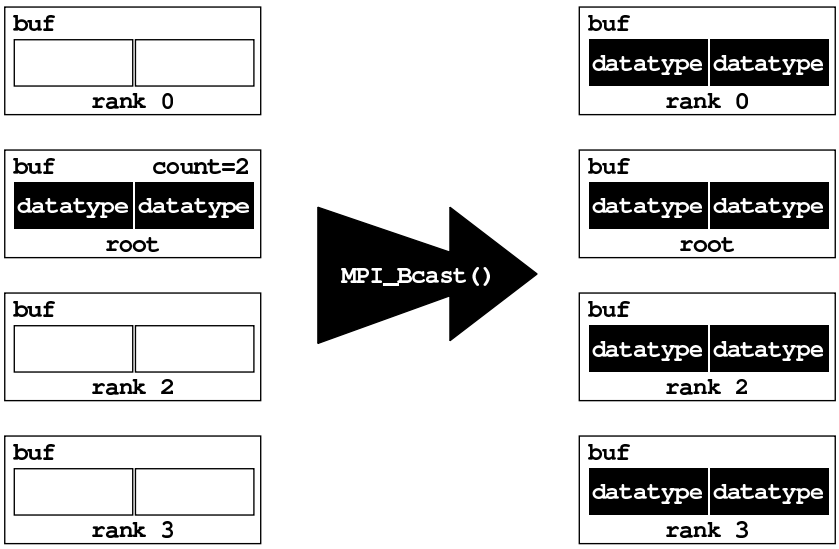
Broadcast

```
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.

Broadcast

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



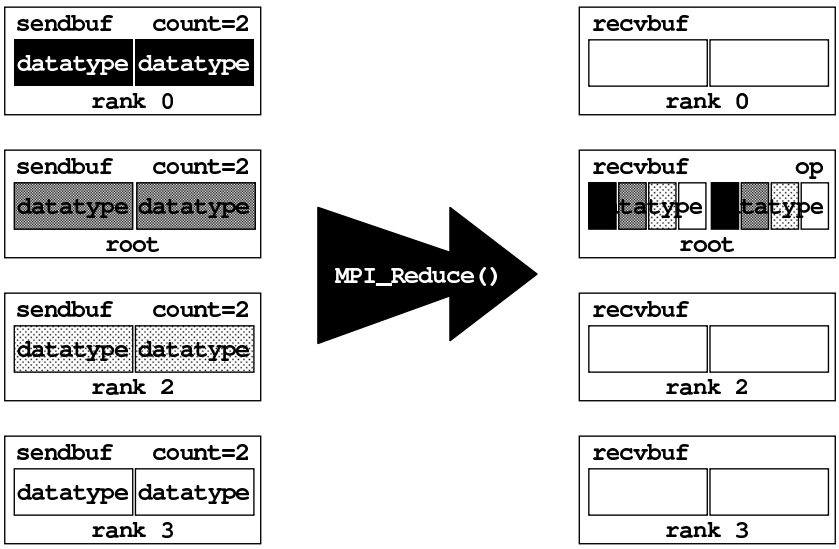
Reduce

```
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.

Reduce

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

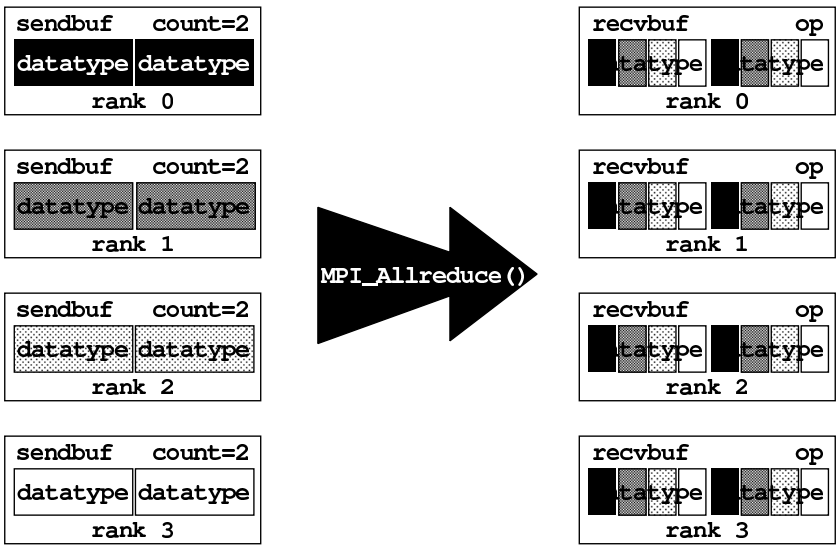


Operações de Redução

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

Allreduce

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

- 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).

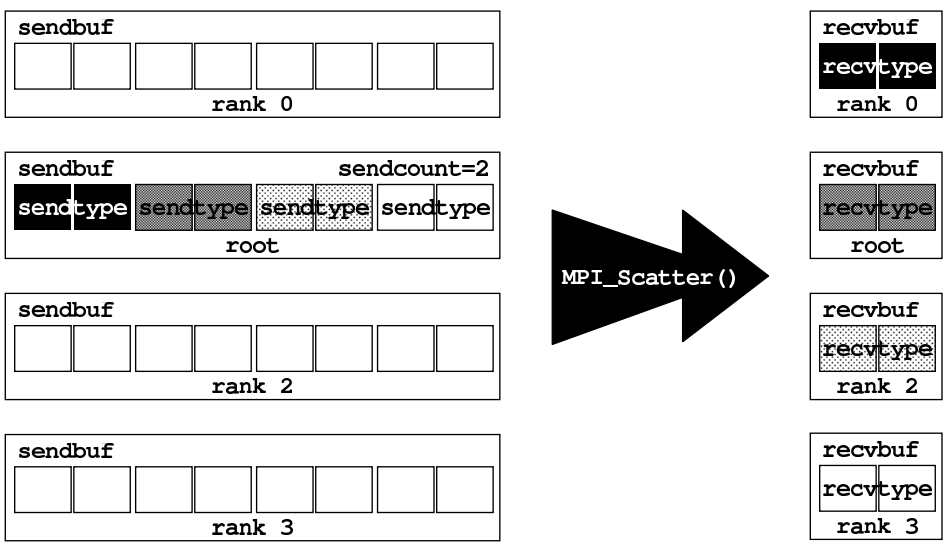
Scatter

```
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.

Scatter

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



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.

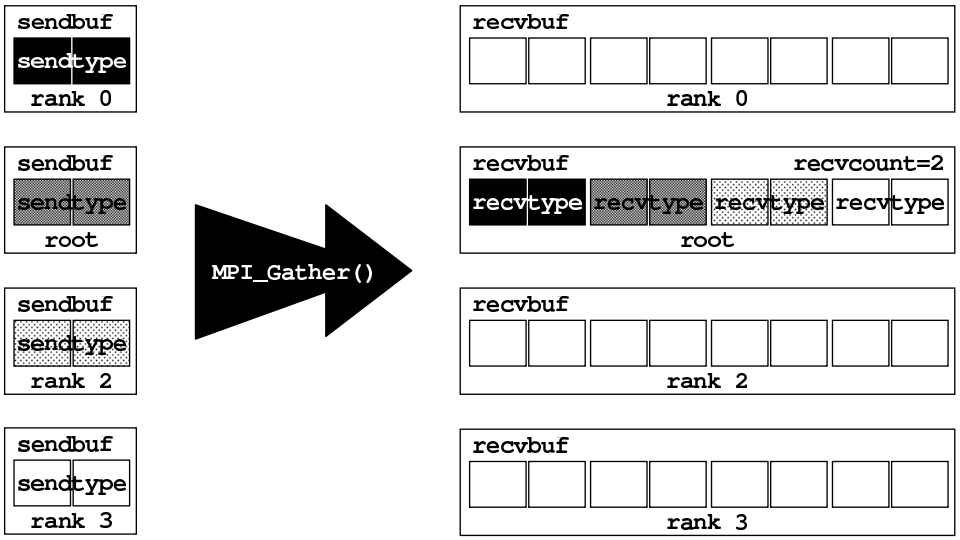
Gather

```
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.

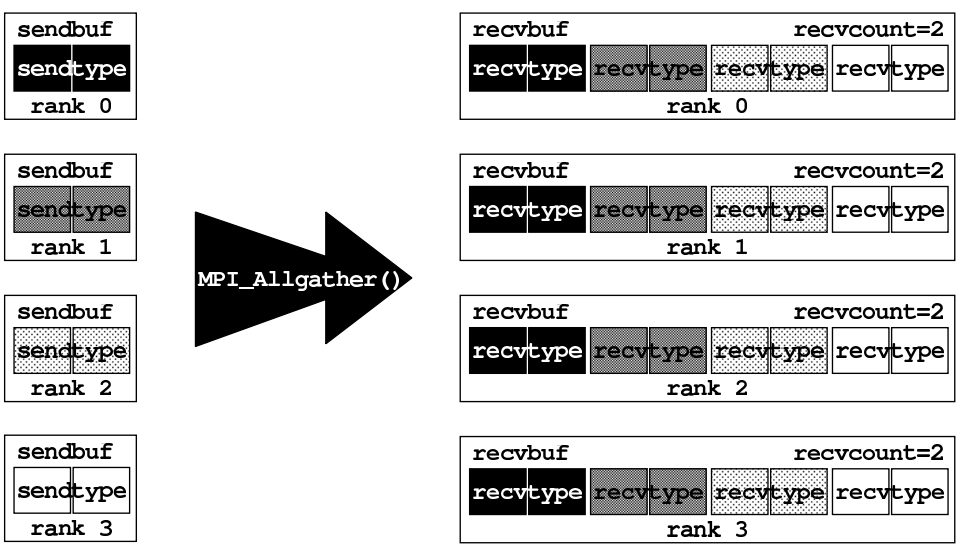
Gather

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



Allgather

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



Exemplo: Produto Escalar

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

$$x \cdot y = x_0 y_0 + x_1 y_1 + \cdots + x_{n-1} y_{n-1}$$

- Se tivermos `P` processos, cada um deles pode calcular `K` (`N/P`) componentes do produto escalar:

Processo	Componentes
0	$x_0 y_0 + x_1 y_1 + \cdots + x_{k-1} y_{k-1}$
1	$x_k y_k + x_{k+1} y_{k+1} + \cdots + x_{2k-1} y_{2k-1}$
...	...
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;
}
```


Exemplo: 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);
...
```

Exemplo: 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]);
}
...
```


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.

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.

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 excluí os size processos referenciados em ranks[].

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.

Exemplo: 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 processo 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`.

Exemplo: 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
```


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().

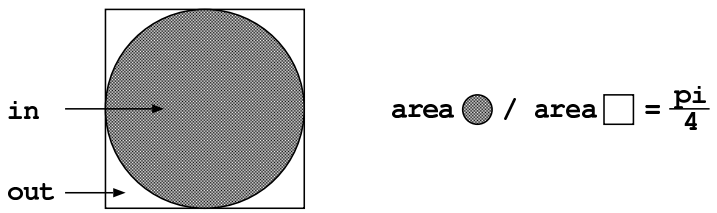
Exemplo: Medir o Tempo de Execução (mpi_time.c)

```
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 do método 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 $4 * in/(in + out)$.

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.

Exemplo: 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);
}
...
```

Exemplo: Cálculo de π (mpi_pi.c)

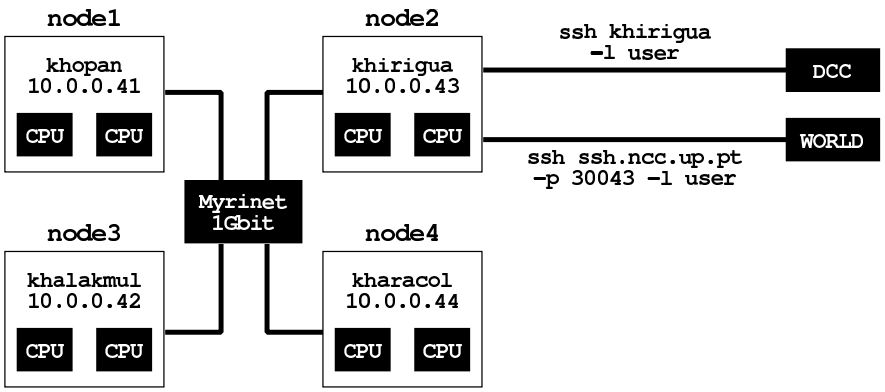
```
// servidor
if (my_rank == SERVER) {
  do {
    MPI_Recv(&req_points, 1, MPI_INT, MPI_ANY_SOURCE,
             REQUEST, MPI_COMM_WORLD, &status);
    if (req_points) {
      for (i = 0; i < req_points; i++)
        rands[i] = random();
      MPI_Send(rands, req_points, MPI_INT, status.MPI_SOURCE,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


Exemplo: 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);
}
```

O Cluster Dolphin2

- O cluster Dolphin2 é composto por 4 máquinas de 2 CPUs cada, ligadas entre si por uma rede Myrinet de alta velocidade com capacidade até 1Gbit/segundo.



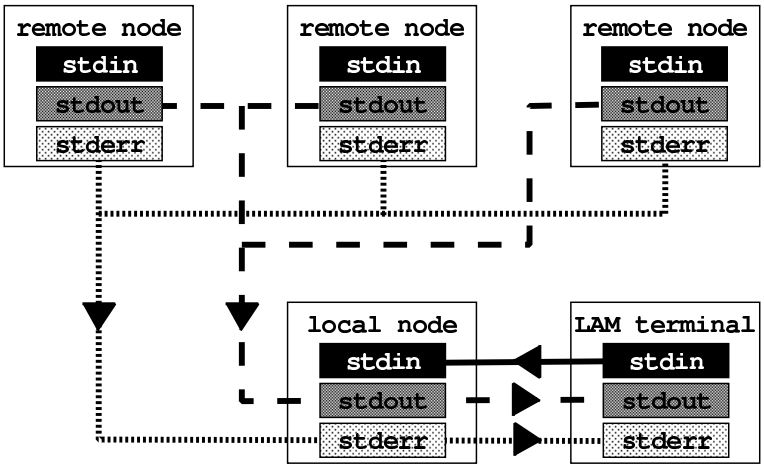
- Na rede interna do DCC as máquinas são conhecidas como `khopan`, `khirigua`, `khalakmul` e `kharacol`. Entre elas são conhecidas como `node1`, `node2`, `node3` e `node4`. A consola do cluster é a máquina `khirigua`. A partir do DCC o acesso deve ser feito com `ssh khirigua -l user`. Do exterior deve fazer-se `ssh ssh.ncc.up.pt -p 30043 -l user`.

LAM/MPI no Cluster Dolphin2

- O LAM é uma implementação do MPI inicialmente desenvolvida no Ohio Supercomputer Center. Para além do suporte à execução de aplicações MPI em conjuntos de máquinas em rede, possui ferramentas elaboradas de monitorização e *debugging* da execução.
- A versão instalada no cluster Dolphin2 é a LAM 6.5.9 que implementa o standard MPI 2. A instalação foi feita a partir dum *rpm* pelo que os executáveis, *libraries* e *man-pages* encontram-se nos locais habituais do sistema operativo.
- Para obter a descrição sumária da sintaxe de qualquer um dos comandos do LAM deve fazer man nome-do-comando ou nome-do-comando -h.

Standard I/O no LAM

- 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.



Esquema de Arranque

- Para iniciar e terminar uma sessão LAM é necessário definir a topologia da máquina paralela num esquema de arranque.

```
node2:~/> cat boot_schema
# cluster com 4 máquinas e 8 CPUs
node2 cpu=2
node1 cpu=2
node3 cpu=2
node4 cpu=2
```

- No esquema de arranque deve especificar-se o nome das máquinas a utilizar, o número de CPUs por máquina, se mais do que 1 (`cpu=2`), assim como o nome do utilizador numa dada máquina, se diferente do utilizador local (`user=guest`).
- A cada máquina no esquema de arranque é atribuído um identificador de posição, sendo `n0` para a primeira máquina, `n1` para a segunda, etc.
- O esquema de arranque é utilizado pelos comandos que permitem iniciar e terminar uma sessão LAM.

Comandos de Sessão do LAM

- recon: permite verificar se o LAM pode ser iniciado em cada uma das máquinas indicadas no esquema de arranque.

```
node2:~/> recon -v boot_schema
recon: -- testing n0 (node2)
recon: -- testing n1 (node1)
recon: -- testing n2 (node3)
recon: -- testing n3 (node4)
-----------------------------------------------------------------------
Woo hoo!

recon has completed successfully.  This means that you will most likely
be able to boot LAM successfully with the "lamboot" command (but this
is not a guarantee).  See the lamboot(1) manual page for more
information on the lamboot command.

If you have problems booting LAM (with lamboot) even though recon
worked successfully, enable the "-d" option to lamboot to examine each
step of lamboot and see what fails.  Most situations where recon
succeeds and lamboot fails have to do with the hboot(1) command (that
lamboot invokes on each host in the hostfile).
-----------------------------------------------------------------------
```


Comandos de Sessão do LAM

- lamboot: inicia uma sessão LAM para um dado utilizador.

```
node2:~/> lamboot -v boot_schema
LAM 6.5.9/MPI 2 C++/ROMIO - Indiana University

Executing hboot on n0 (node2 - 2 CPUs)...
Executing hboot on n1 (node1 - 2 CPUs)...
Executing hboot on n2 (node3 - 2 CPUs)...
Executing hboot on n3 (node4 - 2 CPUs)...
topology done
```

- wipe: termina uma sessão LAM.

```
node2:~/> wipe -v boot_schema
LAM 6.5.9/MPI 2 C++/ROMIO - Indiana University

Executing tkill on n0 (node2)...
Executing tkill on n1 (node1)...
Executing tkill on n2 (node3)...
Executing tkill on n3 (node4)...
```

Comandos de Sessão do LAM

➤ tping: permite verificar se um ou vários nós estão activos.

```
node2:~/> tping n0
1 byte from n0 (o): 0.000 secs
1 byte from n0 (o): 0.000 secs
1 byte from n0 (o): 0.000 secs
1 byte from n0 (o): 0.000 secs
1 byte from n0 (o): 0.000 secs

5 messages, 5 bytes (0.005K), 0.001 secs (14.638K/sec)
roundtrip min/avg/max: 0.000/0.000/0.000

node2:~/> tping N
1 byte from 3 remote nodes and 1 local node: 0.004 secs
1 byte from 3 remote nodes and 1 local node: 0.002 secs
1 byte from 3 remote nodes and 1 local node: 0.002 secs
1 byte from 3 remote nodes and 1 local node: 0.002 secs
1 byte from 3 remote nodes and 1 local node: 0.002 secs

5 messages, 5 bytes (0.005K), 0.014 secs (0.714K/sec)
roundtrip min/avg/max: 0.002/0.003/0.004
```


Mnemónicas sobre os Nós

➤ Muitos dos comandos do LAM necessitam da indicação dos nós sobre os quais devem operar. Essa indicação deve ser dada no formato `n<list>` em que `<list>` pode referenciar nós específicos ou intervalo de nós.

➤ Exemplos:

- n1 (referência ao nó 1)
- n1-3 (referência aos nós 1, 2 e 3)
- n1,3,5-8 (referência aos nós 1, 3, 5, 6, 7 e 8)

➤ Para além da identificação explícita dos nós, o LAM permite a utilização de mnemónicas especiais:

- h (referência ao nó onde o comando é invocado)
- o (referência ao nó onde o LAM foi iniciado com `lamboot`)
- N (referência a todos os nós)
- C (referência a todos os CPUs)

Compilação

- O LAM providencia um conjunto de *scripts* para compilar programas MPI escritos em C, C++ ou Fortran.
  - hcc (*script* de compilação para programas MPI escritos em C)
  - hcp (*script* de compilação para programas MPI escritos em C++)
  - hf77 (*script* de compilação para programas MPI escritos em Fortran)
- Cada um dos *scripts* invoca o respectivo compilador nativo e fornece os caminhos dos *headers* e *libraries* do LAM necessários à compilação. No entanto, a biblioteca do MPI deve ser ligada explicitamente.
  - hcc mpi_pi.c -lmpi -o pi
  - hcp mpi_pi.c -lmpi -o pi
  - hf77 mpi_pi.c -lmpi -o pi

Iniciar a Execução

- mpirun: inicia a execução de uma dada aplicação na sessão corrente do LAM. Uma das seguintes opções deve ser indicada:
  - -s <nodeid> nó onde se encontra o executável do programa.
  - <nodeids> nós onde o programa deve ser executado (todos por defeito).
  - -c <#> número de processos a lançar por nó (1 por defeito).

```
node2:~/> mpirun -v n0-1 pi
2345 pi running on n0 (o)
2346 pi running on n1
```

- Para além da invocação directa da aplicação a executar, é possível indicar um ficheiro de configuração com o esquema da aplicação (conjunto de programas a executar, um por linha).

```
node2:~/> cat application_schema
# esquema master-slave
master n0
slave n1 -s h
node2:~/> mpirun -v application_schema
3478 master running on n0 (o)
3479 slave running on n1
```

Terminar a Execução

- lamclean: permite restaurar o ambiente de execução do LAM quando algo corre mal, sem reiniciar uma nova sessão. Elimina todos os processos em execução e liberta todos os recursos em utilização, incluindo as mensagens por entregar.

```
node2:~/> lamclean -v
killing processes, done
closing files, done
sweeping traces, done
cleaning up registered objects, done
sweeping messages, done
```


Monitorizar a Execução

- mpitask: dá informação sobre o estado dos processos em execução (para activar o modo de monitorização é necessário iniciar a execução com a opção `-lamd`).

```
node2:~/> mpirun -lamd C pi
node2:~/> mpitask
TASK (G/L)   FUNCTION   PEER|ROOT   TAG   COMM   COUNT   DATATYPE
0 pi         <running>
1/1 pi       Recv       0/0         15    <2>*   1       INT
2/2 pi       Recv       3/3         86    <2>*   1       INT
3 pi         <running>
4/4 pi       Recv       0/0         15    <2>*   1       INT
5/5 pi       Recv       7/7         1     WORLD  1000    INT
6 pi         <running>
7/7 pi       Recv       ANY         0     WORLD  1       INT
```

- O LAM permite mostrar em qualquer momento o estado de sincronização ou da troca de mensagens de todos os processos em execução, incluindo identificadores dos processos origem e destino; função em execução; marca e comunicador da mensagem; número e tipo dos dados.

Monitorizar a Execução

- mpimsg: dá informação sobre mensagens em *buffer*.

```
node2:~/> mpirun -lamd C pi
node2:~/> mpimsg
SRC (G/L)   DEST (G/L)   TAG   COMM   COUNT   DATATYPE   MSG
1/1         7/7          0     WORLD  1       INT        n0,#240
5/5         7/7          0     WORLD  1       INT        n0,#241
6/6         7/7          0     WORLD  1       INT        n0,#242
2/2         7/7          0     WORLD  1       INT        n0,#243
3/3         7/7          0     WORLD  1       INT        n0,#246
...and more buffered messages...
```

- O LAM permite mostrar em qualquer momento as mensagens por entregar, incluindo identificadores dos processos origem e destino; marca e comunicador da mensagem; número e tipo dos dados.

Bibliografia

- ***Parallel Programming with MPI.*** P. Pacheco. Morgan Kaufmann. **(Capítulos 3, 4, 5, 6, 7 (págs 111–121) e 13. Biblioteca ref. 159)**
- ***A User's Guide to MPI.*** P. Pacheco. University of San Francisco. **(Página da Cadeira)**
- ***MPI Primer/Developing With LAM.*** Ohio Supercomputer Center. The Ohio State University. **(Página da Cadeira)**