Introdução ao Message Passing Interface- MPI
MC-SD04-I

Carla Osthoff
LNCC/MCTI
osthoff@lncc.br

Material do curso:

www.cenapad-rj.lncc.br/tutoriais/materiais-hpc/semana-sdumont

Instalação do material na sua conta to SDUMONT:

$ sftp user@login.sdumont.lncc.br

$ put testes.tar

$ exit

ssh user@login.sdumont.lncc.br

cp testes.tar $SCRATCH

cd $SCRATCH

Tar xvf testes.tar

Instalação do material na maquina que você tem acesso:

$ sftp user@intelknl.lncc.br

$ put testes.tar

$ exit

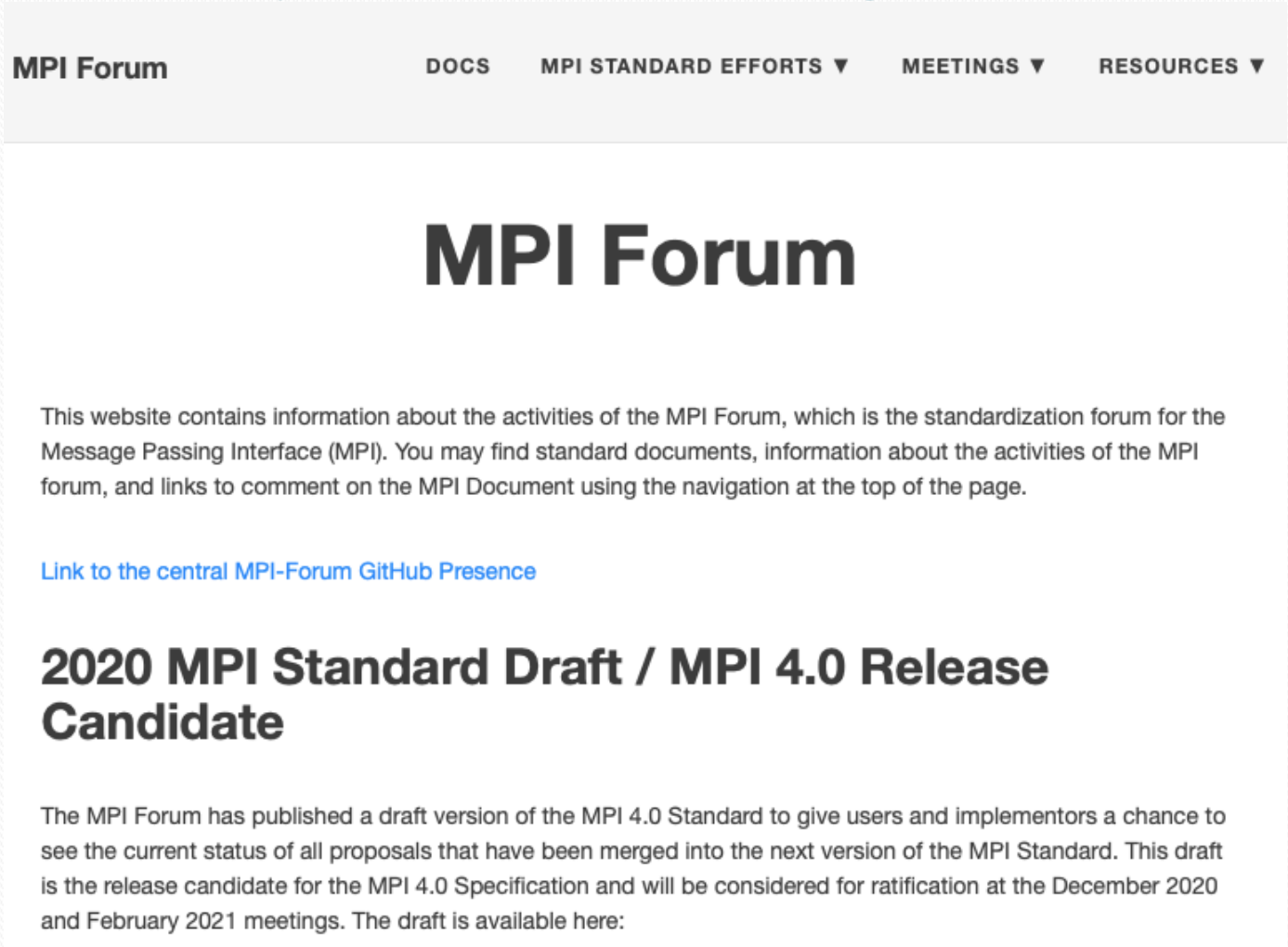
ssh user@intelknl.lncc.br

tar xvf testes.tar

Message Passing Interface

- **An Interface Specification:**
  - **M P I = Message Passing Interface**
  - MPI is a ***specification*** for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.
  - MPI primarily addresses the ***message-passing parallel programming model:*** data is moved from the address space of one process to that of another process through cooperative operations on each process.
  - Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:
    - Practical
    - Portable
    - Efficient
    - Flexible

www.mpi-forum.org



http://mpi-forum.org/docs/

MPI Documents

The official version of the MPI documents are the English Postscript versions (for MPI 1.0 and 1.1) and PDF (for the other versions). In several cases, a translation or HTML version is also available for convenience. The HTML version was made with automated tools. In case of a difference between these two sources, the Postscript or PDF version of MPI standard documents are always considered the official version. In the case of multiple PDF versions, only the one described as the "MPI x.y document as PDF" is the official version; the versions provided with alternate formatting are provided as a convenience and are not official (every effort has been taken to make them "the same", but no guarantee is made).

Those who prefer to get the documents via anonymous ftp may do so at ftp.mpi-forum.org in pub/docs/.

Some translations of MPI documents are available.

Draft Specification

Starting in 2018, the MPI Forum has decided to release draft specifications to allow users and implementors an early opportunity to see changes in upcoming versions of the MPI Standard. These draft specifications are not

http://mpi-forum.org/mpi-40/

MPI 4.0

Scope

The MPI 4.0 standardization efforts aim at adding new techniques, approaches, or concepts to the MPI standard that will help MPI address the need of current and next generation applications and architectures. In particular, the following additions are currently being proposed and worked on:

- Extensions to better support hybrid programming models
- Support for fault tolerance in MPI applications
- Persistent collectives
- Performance Assertions and Hints
- RMA/One-sided communication

Additionally, several working groups are working on new ideas and concepts, incl.

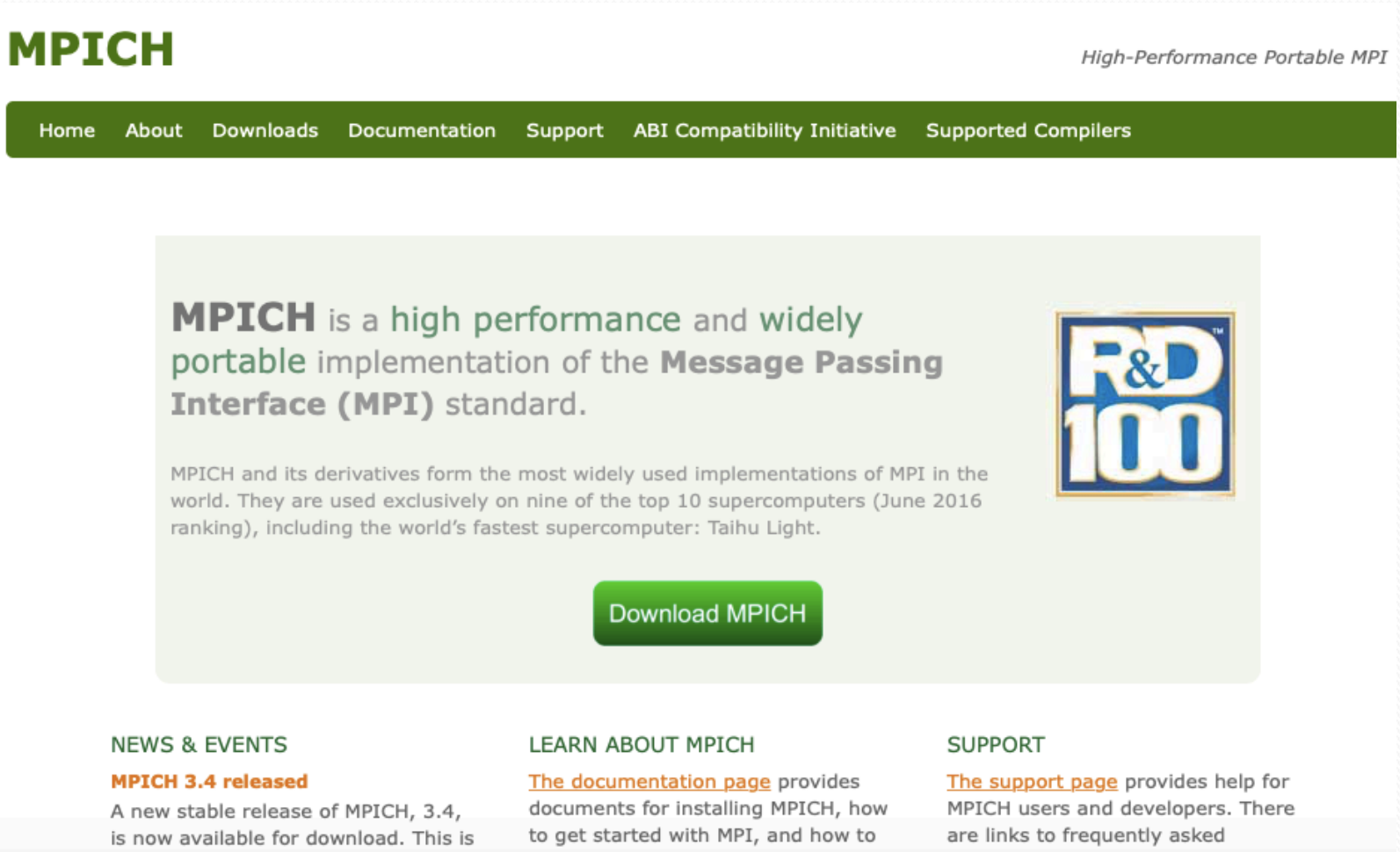
- Active messages
- Stream messaging
- Rework of the MPI profiling interface
- Extensions to MPI_T
- Generalized requests
- Hybrid MPI+X concerns (esp. MPI+CAF)
- Send cancelation
- Attribute callback
- Large count

Further, the tools WG is discussing additional 3rd party tool interfaces, which are generally published as side documents:

- Handle introspection from debuggers
- Debug DLL detection and identification

Note, though, that all of these efforts or new concepts are currently only being discussed or proposed and there is no guarantee that any particular one will be included in any upcoming MPI version.

Site da Implementação MPICH: http://www.mpich.org/



Manual online: http://www.mpich.org/static/docs/latest/

Web pages for MPI

MPI Commands

mpicc mpiexec mpifort
mpicxx mpif77

MPI Routines and Constants

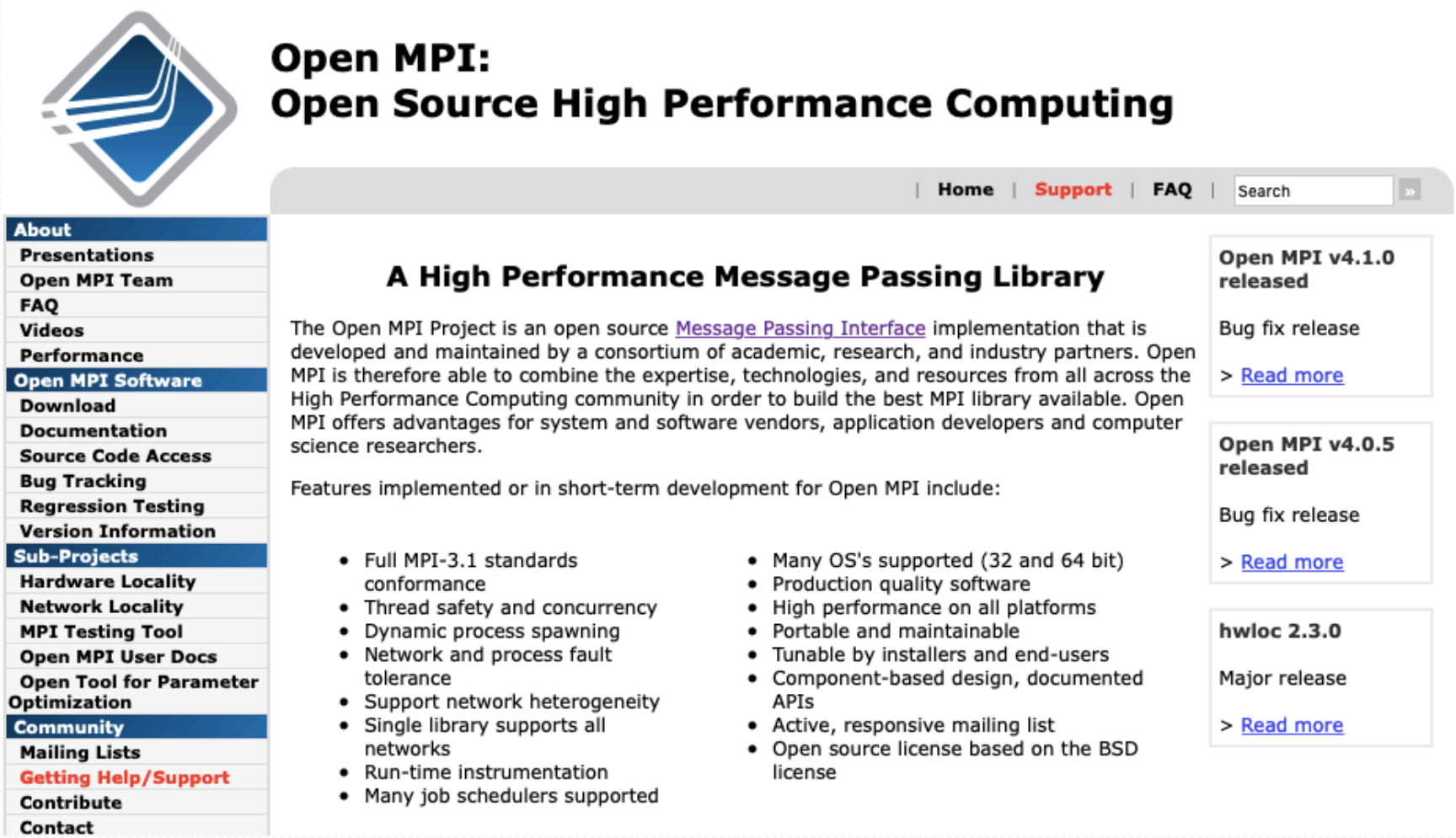
Constants
MPIX_Comm_agree
MPIX_Comm_failure_ack
MPIX_Comm_failure_get_acked
MPIX_Comm_revoke
MPIX_Comm_shrink
MPIX_GPU_query_support
MPI_2DOUBLE_PRECISION
MPI_2INT
MPI_2INTEGER
MPI_2REAL
MPI_AINT
MPI_File_iread_all
MPI_File_iread_at
MPI_File_iread_at_all
MPI_File_iread_shared
MPI_File_iwrite
MPI_File_iwrite_all
MPI_File_iwrite_at
MPI_File_iwrite_at_all
MPI_File_iwrite_shared
MPI_File_open
MPI_File_preallocate
MPI_File_read
MPI_SHORT_INT
MPI_SIGNED_CHAR
MPI_SIMILAR
MPI_SOURCE
MPI_STATUSES_IGNORE
MPI_STATUS_IGNORE
MPI_SUBVERSION
MPI_SUCCESS
MPI_SUM
MPI_Scan
MPI_Scatter
MPI_Scatterv

Implementações “OpenSource”

- Mpich: **MPICH** is a high performance and widely portable implementation of the **Message Passing Interface (MPI)** standard to provide feedback to MPI FORUM.

- OpenMPI: The Open MPI Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners

Site da Implementação OpenMPI: https://www.open-mpi.org/



Existem diversas implementações do padrão MPI desenvolvida por fabricantes:

- Intel MPI
- Bullx MPI
- Cray MPI ...

Livros sobre MPI recomendados: https://mpitutorial.com/recommended-books/

Programação Paralela com MPI: Um Curso Introdutório [Print Replica] eBook Kindle

por Gabriel P. Silva (Autor) | Formato: eBook Kindle

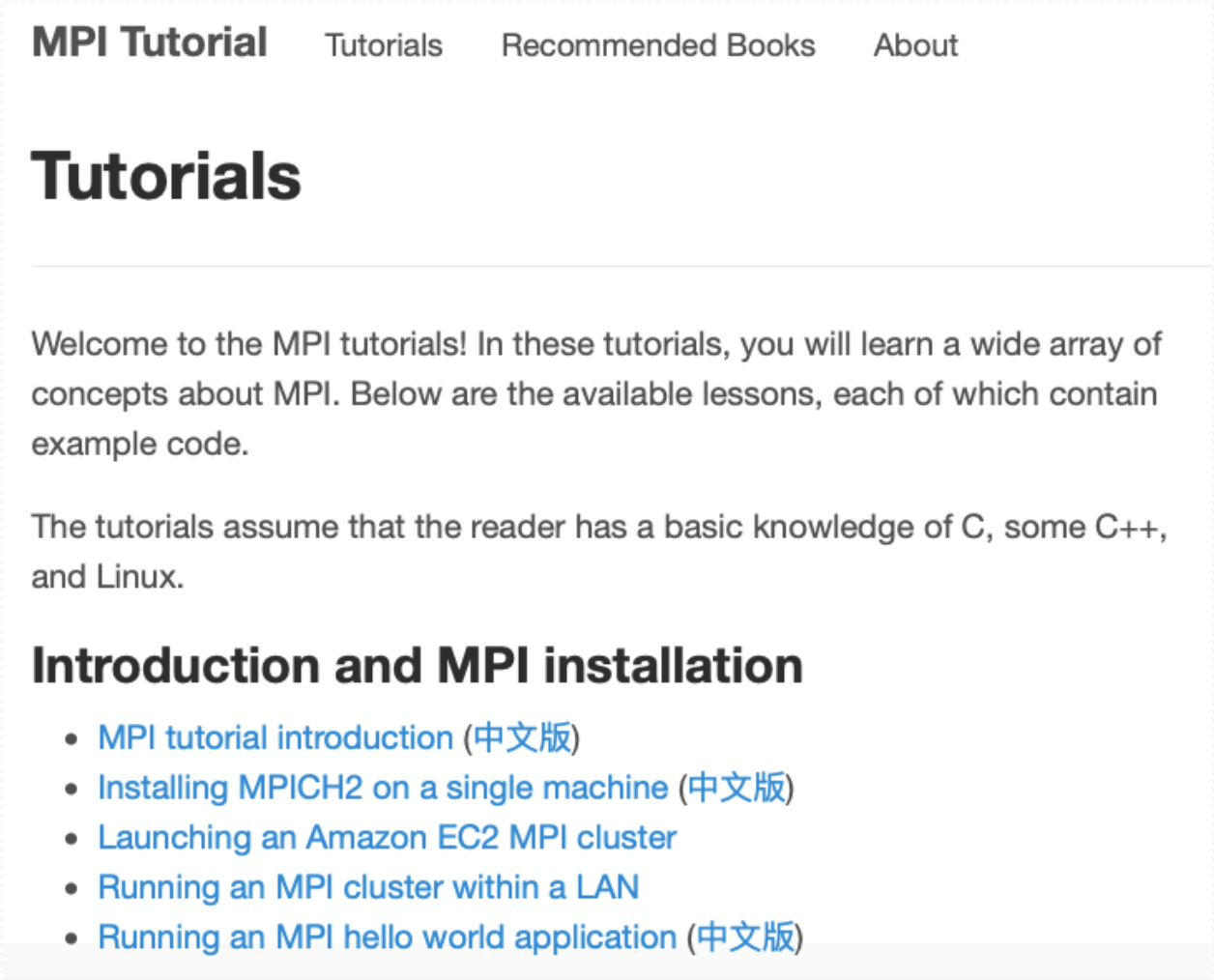
★★★★★ 1 classificação

› Ver todos os formatos e edições

Kindle
R$0,00 kindleunlimited

Este título e mais 1 milhão disponíveis com Kindle Unlimited
R$24,99 para comprar

Existem diversos sites sobre Tutoriais de MPI: http://mpitutorial.com/tutorials/



Os exercícios deste tutorial são baseadas no Tutorial do LLNL: https://computing.llnl.gov/tutorials/mpi/

Message Passing Interface (MPI)

Author: Blaise Barney, Lawrence Livermore National Laboratory

Table of Contents

Histórico do padrão MPI:

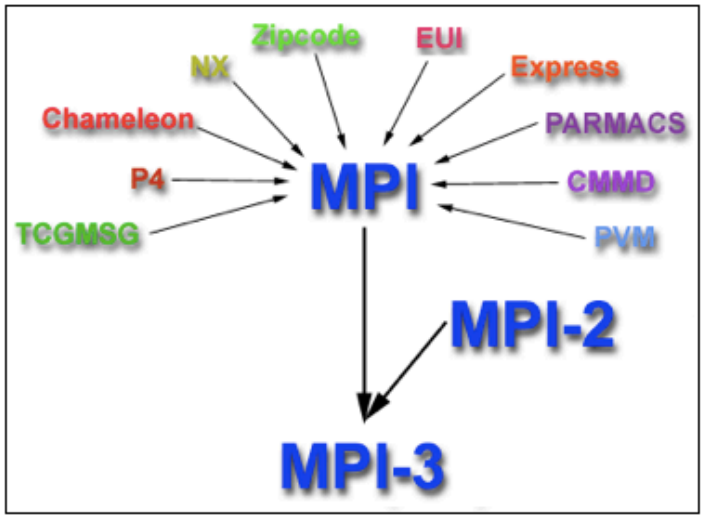
- **1980s - early 1990s:** Distributed memory, parallel computing develops, as do a number of incompatible software tools for writing such programs - usually with tradeoffs between portability, performance, functionality and price. Recognition of the need for a standard arose.

- **Apr 1992:** Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, Williamsburg, Virginia. The basic features essential to a standard message passing interface were discussed, and a working group established to continue the standardization process. Preliminary draft proposal developed subsequently.

- **Nov 1992:** Working group meets in Minneapolis. MPI draft proposal (MPI1) from ORNL presented. Group adopts procedures and organization to form the MPI Forum. It eventually comprised of about 175 individuals from 40 organizations including parallel computer vendors, software writers, academia and application scientists.

- **Nov 1993:** Supercomputing 93 conference - draft MPI standard presented.

- **May 1994:** Final version of MPI-1.0 released
  - MPI-1.1 (Jun 1995)
  - MPI-1.2 (Jul 1997)
  - MPI-1.3 (May 2008).



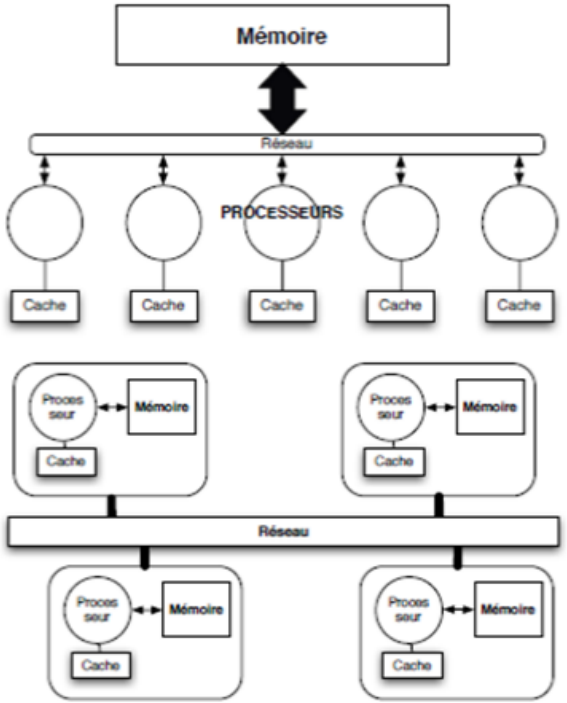
- **1998:** MPI-2 picked up where the first MPI specification left off, and addressed topics which went far beyond the MPI-1 specification.
  - MPI-2.1 (Sep 2008)
  - MPI-2.2 (Sep 2009)

- **Sep 2012:** The MPI-3.0 standard was approved.
  - MPI-3.1 (Jun 2015)

- **Current:** The MPI-4.0 standard is under development.

Segundo a classificação de sistemas computacionais baseada no modelo de memória, o MPI é um modelo desenvolvido para sistemas computacionais de memória distribuída:

- **Shared memory computer**
  Several processors sharing the same global memory space via a fast interconnect



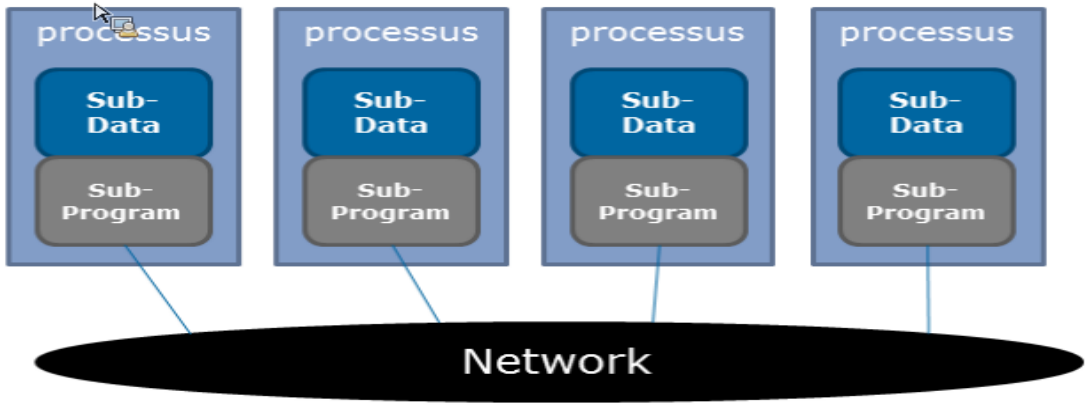
- **Distributed memory computer**
  - Each node with its own memory
    Each node reaches other nodes memory via the network (call to communications routines)

- **Hybrid computer**
  - Most common case: a set of shared memory computers (eventually equipped with coprocessors or accelarators) linked by a network

MPI segue o modelo de programação por troca de mensagens:

Um programa é dividido em diversos sub programas para serem executados por processos.

Os processos se comunicam através da rede de comunicação utilizando mensagens.

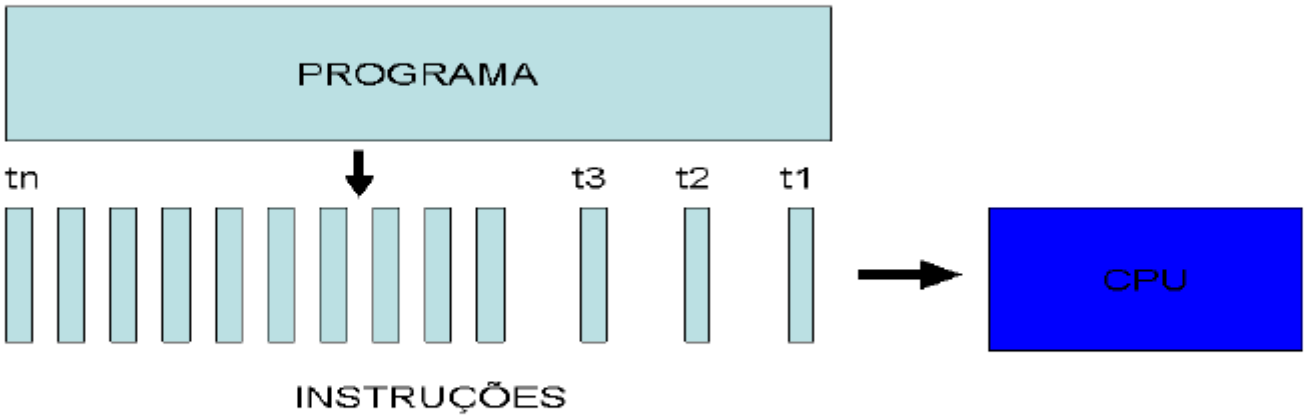


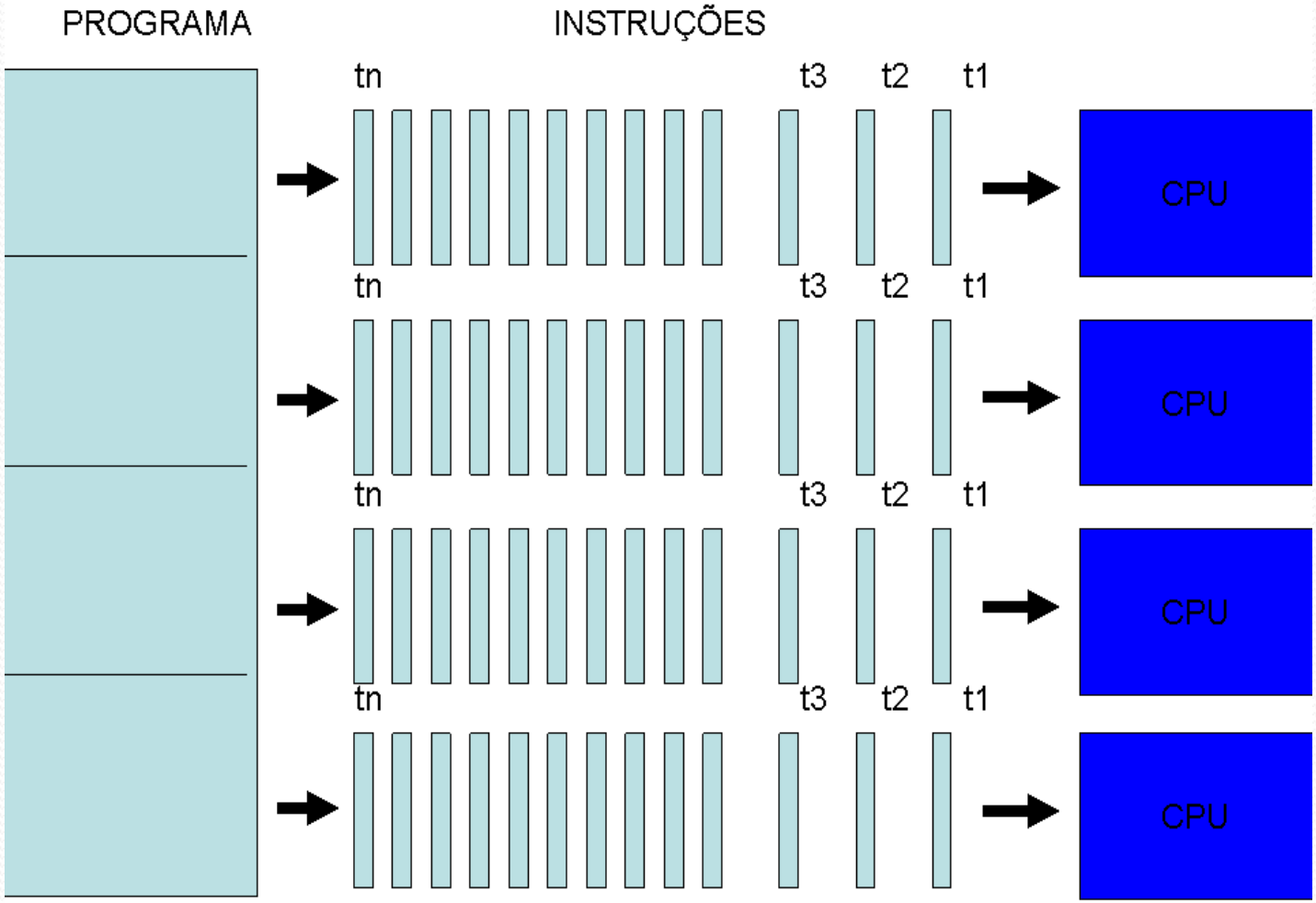
MPI segue o modelo de Programação SPMD:

- SPMD

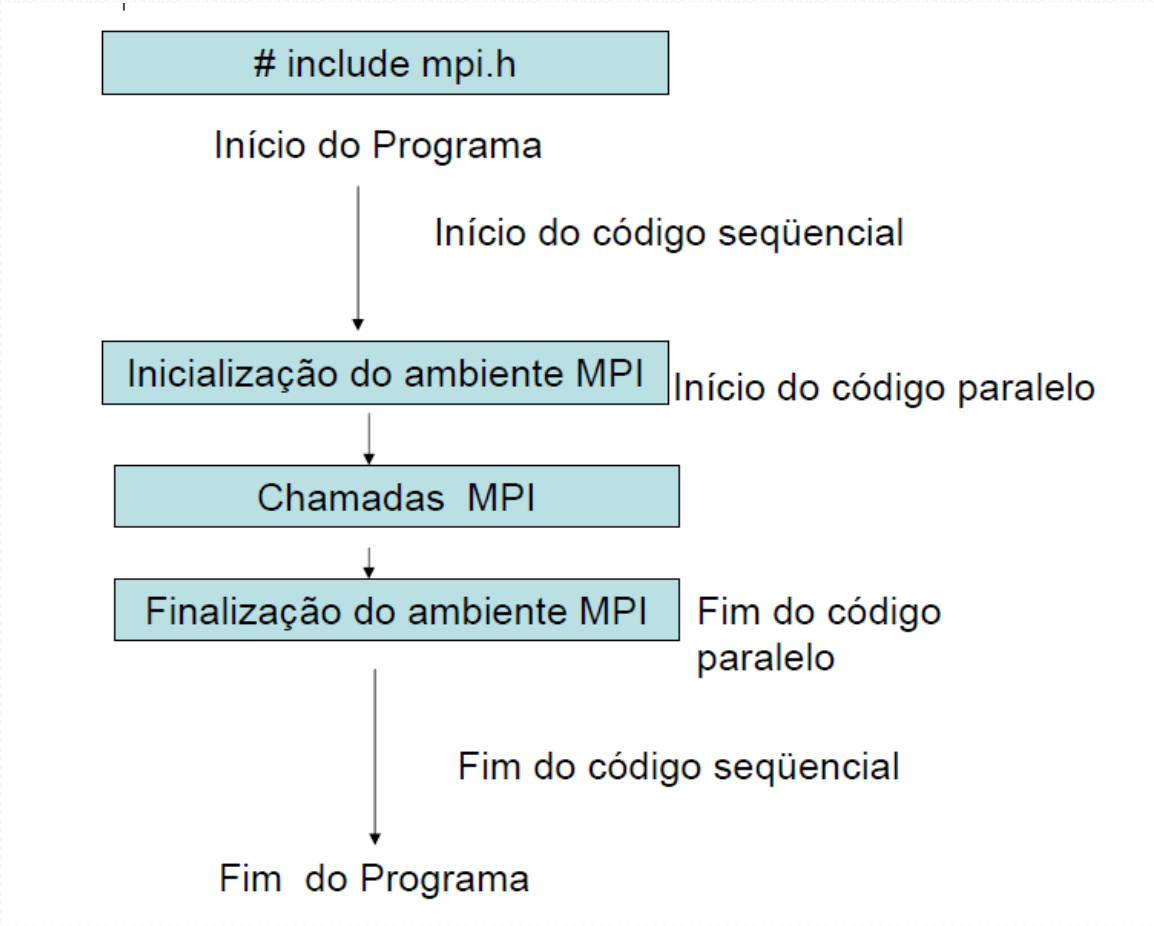
  (Single programa Multiple data)
- O processador gerente envia e gerencia um **“mesmo programa”** em todas as máquinas do sistema distribuído.

Como Funciona o MPI?




PROGRAMA
INSTRUÇÕES
tn
t3
t2
t1
CPU
CPU
CPU
CPU

Estrutura de um Programa MPI



Gerenciamento do Ambiente da biblioteca MPI:

- Todo programa MPI escrito em C tem que inicializar com a chamada à biblioteca : **#include "mpi.h"**
- Um programa MPI apresenta quatro funções básicas:
  - MPI_Init,
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank

Variáveis utilizadas pela biblioteca MPI:

- **Processo :** Cada parte do programa quebrado é chamada de processo. Os processos podem ser executados em uma única máquina ou em várias máquinas.
- **RANK:** Todo processo tem uma identificação única atribuída pelo sistema quando o processo é inicializado.
- **Comunicador**: é um objeto local que representa o domínio de uma comunicação.O MPI_COMM_WORLD é o comunicador predefinido que inclui todos os processos definidos pelo usuário numa aplicação MPI

Primeiro exemplo: Hello World

C Language - Environment Management Routines

```c
// required MPI include file
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
int  numtasks, rank, len, rc;
char hostname[MPI_MAX_PROCESSOR_NAME];

// initialize MPI
MPI_Init(&argc,&argv);

// get number of tasks
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);

// get my rank
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

// this one is obvious
MPI_Get_processor_name(hostname, &len);
printf ("Number of tasks= %d My rank= %d Running on %s\n", numtasks,rank,hostname);


     // do some work with message passing


// done with MPI
MPI_Finalize();
}
```

Execução do teste01.c no SDumont

- MPI "build script" para compilar
  - **$compilar.sh**
- MPI "build script" para submeter na fila de execução:
  - **$sbatch sub.sh teste01**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

Execução do teste01.c na sua máquina:

- MPI "build script" para compilar:
  - **mpicc teste01.c –o teste01**

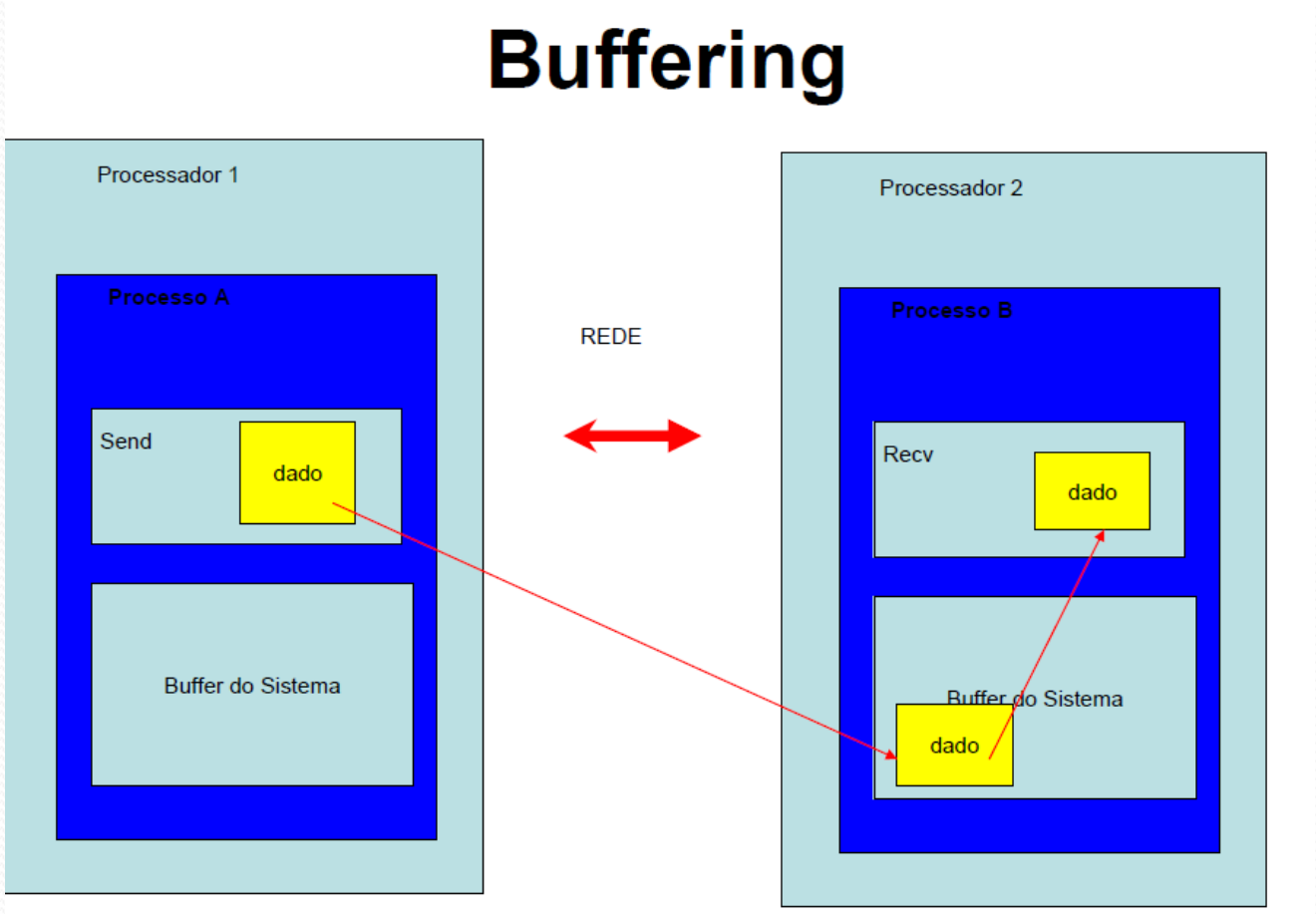
- MPI "build script" para executar:
  - **mpirun –np 4 ./teste01**

Rotinas básicas de comunicação: Ponto a Ponto

- Enquanto um processo realiza uma operação de envio o outro processo realiza uma operação de recebimento da mensagem

- Existem duas rotinas básicas para fazer a troca de mensagens entre dois processadores, MPI_SEND e MPI_RECV:
  - Estas rotinas permitem a troca de mensagem de forma bloqueante
  - Não deixam o programa seguir em frente enquanto não obtiverem confirmação do recebimento da mensagem.
  - Após o retorno, libera o "system buffer" e permite o acesso ao "application buffer

As rotinas de troca de mensagens utilizam o conceito de :System Buffer e Application Buffer



MPI_SEND

```
MPI_Send(&outmsg, 1,
  MPI_CHAR, dest, tag,
  MPI_COMM_WORLD);
```

- 1º: Endereço do dado a ser transmitido
- 2º: Número de itens a ser enviado
- 3º: Tipo de Dados
- 4º: Destino
- 5º: Comunicador

MPI_RCV

MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);

1º: Endereço do dado a ser transmitido

2º: Número de itens a ser enviado

3º: Tipo de Dados

4º: Destino

5º: Comunicador

6º: Status da mensagem

C Language - Blocking Message Passing Example

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

main(int argc, char *argv[])  {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;   // required variable for receive routines

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

// task 0 sends to task 1 and waits to receive a return message
if (rank == 0) {
  dest = 1;
  source = 1;
  MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

// task 1 waits for task 0 message then returns a message
else if (rank == 1) {
  dest = 0;
  source = 0;
  MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

// query recieve Stat variable and print message details
MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n",
       rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

MPI_Finalize();
}
```

Execução do teste02.c na sua máquina:

- MPI "build script" para compilar:
  - **mpicc teste02.c –o teste02**

- MPI "build script" para executar:
  - **mpirun –np 4 ./teste02**

Execução do teste02.c no SDumont

- MPI "build script" para compilar
  - **$compilar.sh**
- MPI "build script" para submeter na fila de execução:
  - **$sbatch sub.sh teste02**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

teste02.c:

- **EXEMPLO COM SEND E RECEIVE BLOQUEANTES**
- Processo 0 envia mensagem para o processo 1 e espera pela mensagem de recebimento.
- Experimente trocar a ordem das operações de mensagens do ranck=0. Quando ambos os nós executam primeiro RCV ocorre um “deadlock”, porque ambos processos ficam parados na operação de RCV esperando a operação de recebimento do buffer ser executada.
- Experimente trocar a ordem das operações de mensagens do ranck=1. O que acontece quando ambos os nós executam primeiro SEND? Não ocorre bloqueio porque o buffer do sistema recebe a mensagem e responde ok.
- Experimente executar para mais de 2 processos. Observe que do processo 3 para cima não tem envio de dados e eles imprimem “lixo" relativo ao buffer de recebimento.

- **4 modes of send** for point to point communications:
  - Standard (MPI implementation dependant)
  - Buffered (copy in a buffer ; the send is done later, asynchroneously ; no need to wait receiving) => should probably give better results, but requires copy in memory
  - Synchronous (with receiving ; the program takes back the hand when the send is complete)
  - Ready (started only if the matching receive is already posted)
- Each mode has blocking and non-blocking implementation:

	Mode	Blocking	Non-blocking
Send	Standard	MPI_Send	MPI_Isend
	Buffered	MPI_Bsend	MPI_Ibsend
	Synchronous	MPI_Ssend	MPI_Issend
	Ready	MPI_Rsend	MPI_Irsend
Receive		MPI_Recv	MPI_Irecv

Rotinas não Bloqueante: MPI_Isend e MPI_Irecv

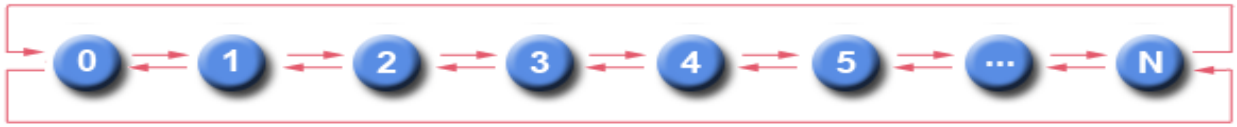
- Identifica uma área na memória para ser utilizada como buffer para o envio das mensagens.
- A execução do programa continua sem esperar que a mensagem seja copiada do buffer da aplicação para o sistema
- A instrução de comunicação devolve uma mensagem com um status pendente.
- O programa não deve alterar o buffer até que as rotinas de teste de recebimento tais como a rotina MPI_Wait ou a rotina MPI_Test indiquem o término do envio

Mensagens não Bloqueantes:

- Não é seguro modificar o buffer da aplicação até que se tenha a confirmação de que a operação foi efetivamente realizado pela biblioteca.
- Para isto a biblioteca MPI fornece operações de espera, chamadas de wait
- As comunicações não-bloqueantes são importantes para sobrepor comunicação com computação e explorar possíveis ganhos de desempenho

MPI_Wait

- Fica em estado de espera bloqueante até que a operação seja concluída.
- Para o caso de várias operações bloqueantes, o programador pode especificar os parâmetros “nenhum, alguns ou todos”.



C Language - Non-blocking Message Passing Example

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

main(int argc, char *argv[])  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];   // required variable for non-blocking calls
MPI_Status stats[4];   // required variable for Waitall routine

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

// determine left and right neighbors
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

// post non-blocking receives and sends for neighbors
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

   // do some work while sends/receives progress in background

// wait for all non-blocking operations to complete
MPI_Waitall(4, reqs, stats);

   // continue - do more work

MPI_Finalize();
}
```

Execução do teste03.c na sua máquina:

- MPI "build script" para compilar:
  - **mpicc teste03.c –o teste03**

- MPI "build script" para executar:
  - **mpirun –np 4 ./teste03**

Execução do teste03.c no SDumont

- MPI "build script" para compilar
  - **$compilar.sh**
- MPI "build script" para submeter na fila de execução:
  - **$sbatch sub.sh teste03**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

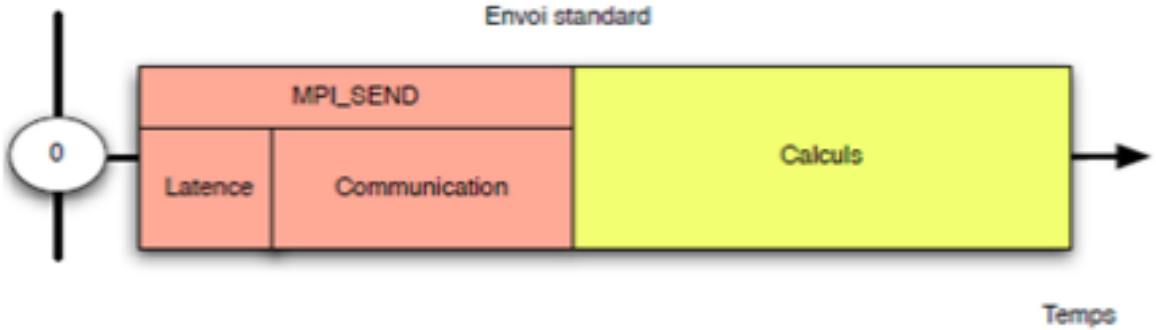
Exemplo não bloqueante: teste03.c

- Observe que se a operação MPI_SEND for colocada antes das operações MPI_IRCV, existe a possibilidade de uma máquina receber os dados antes de ter alocado um espaço no seu buffer de recebimento, causando a perda dos dados de recebimento.
- As operações não bloqueastes são utilizadas para possibilitar a execução de operações de computação enquanto ocorre uma transferência de dados.

Desempenho

- What are decisive factors?
  - System architecture and network between cores and nodes.
  - MPI implementation.
  - The code: choice of algorithms, memory management, communication/computing ratio in the code, load balancing...
- Time sharing during the execution of a MPI program
  - Latency: time to begin an exchange ≈ time needed to send an empty message
  - Communications
  - Computations

Example:

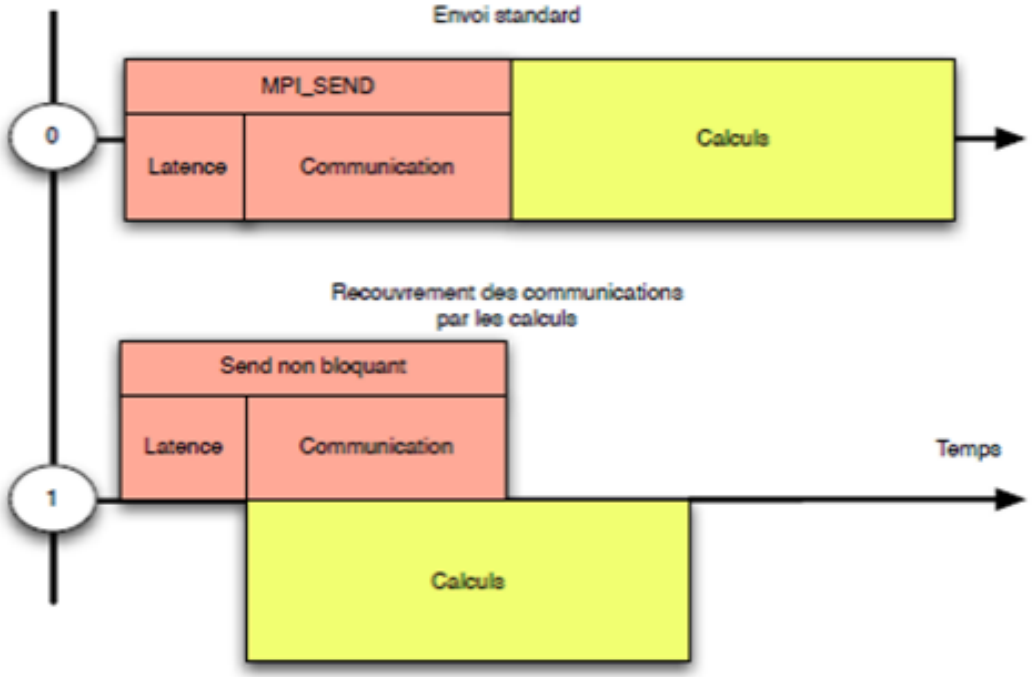


Alternar Computação com Comunicação:

- How to improve the implementation?
  - Use the good algorithms…
  - Use specialized libraries (fftw, scalapack…).
  - Overlap communications with computations.
  - Change communication mode.
  - Balance load between different processes.

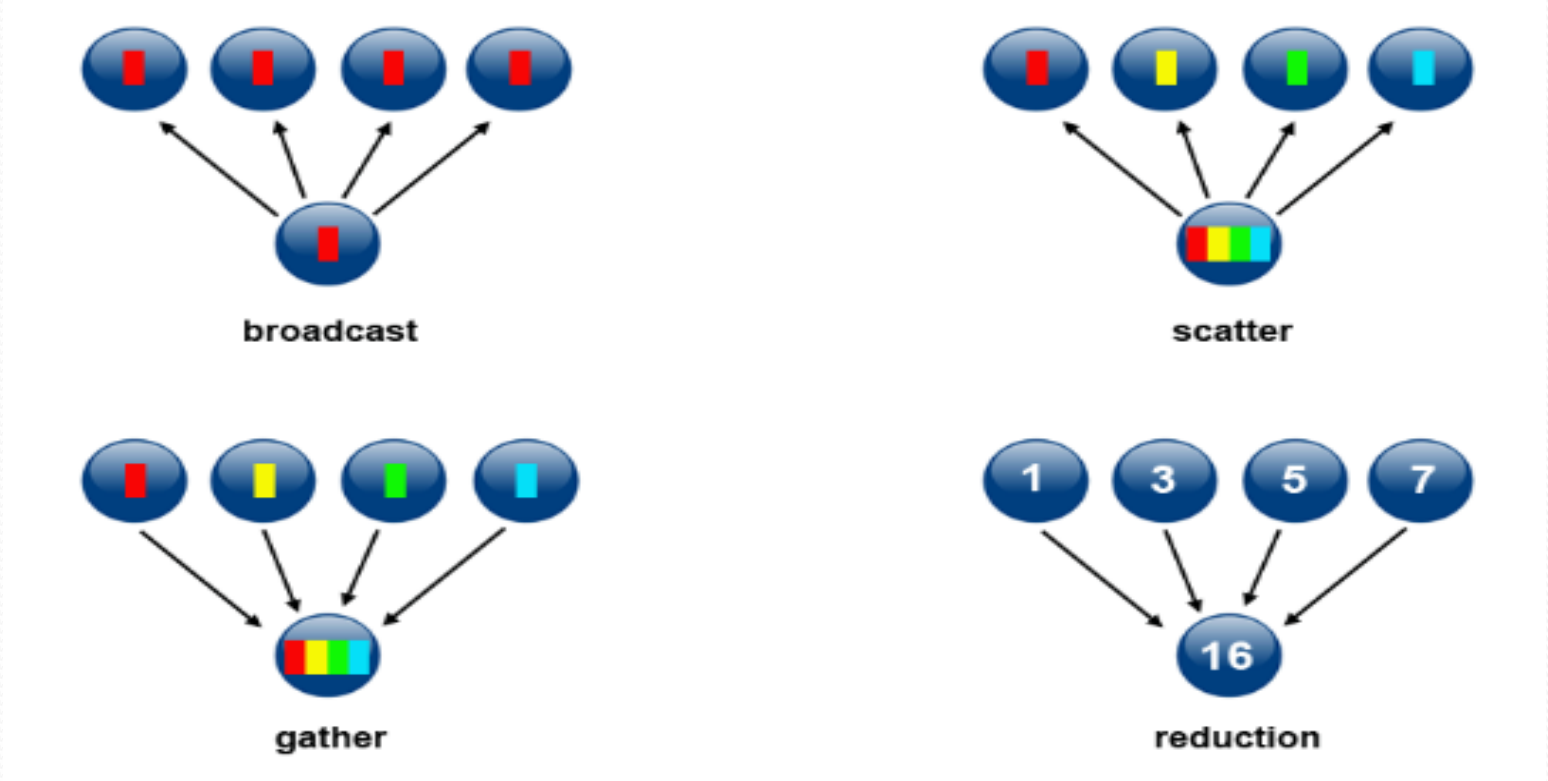
Example:

-



Rotinas de Comunicação Coletivas

- Envolve todos os processos no âmbito de um comunicador MPI_COMM_WORLD .



Tipos de Operações Coletivas:

- **Sincronização -** processos de esperar até que todos os membros do grupo tenham chegado ao ponto sincronização**.**
- **Movimento de Dados -** broadcast, scatter, gather, tudo para todos.
- **Computação Coletivas** (reduções) - um membro do grupo executa a coleta dos dados dos outros membros e exerce uma operação (min, max, adicionar, multiplicar, etc) sobre esses dados

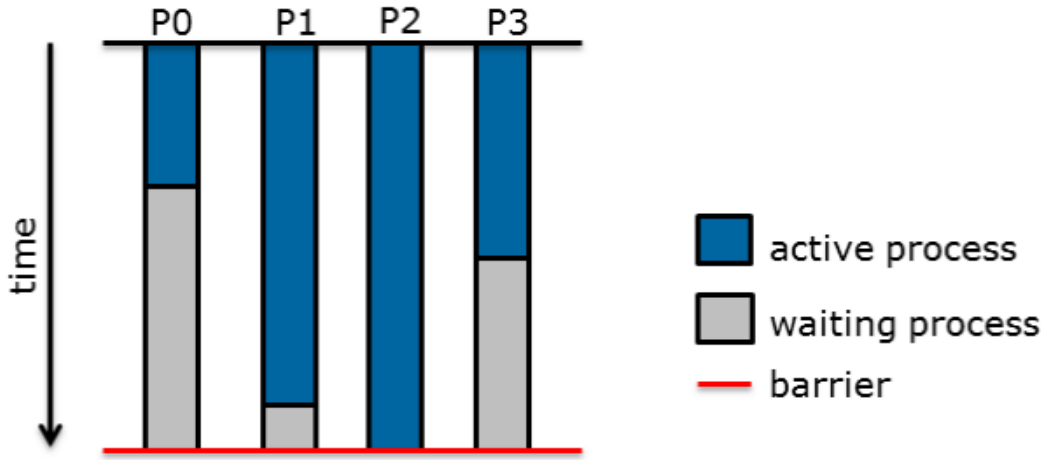
Considerações e Restrições sobre a Programação:

- As mensagens de comunicação coletivas não possuem tag.
- A partir de MPI-3 elas pode ser bloquantes e não bloqueantes
- Só podem ser utilizadas para MPI dataypes predefinidos

Sincronização através de operação de barreira:

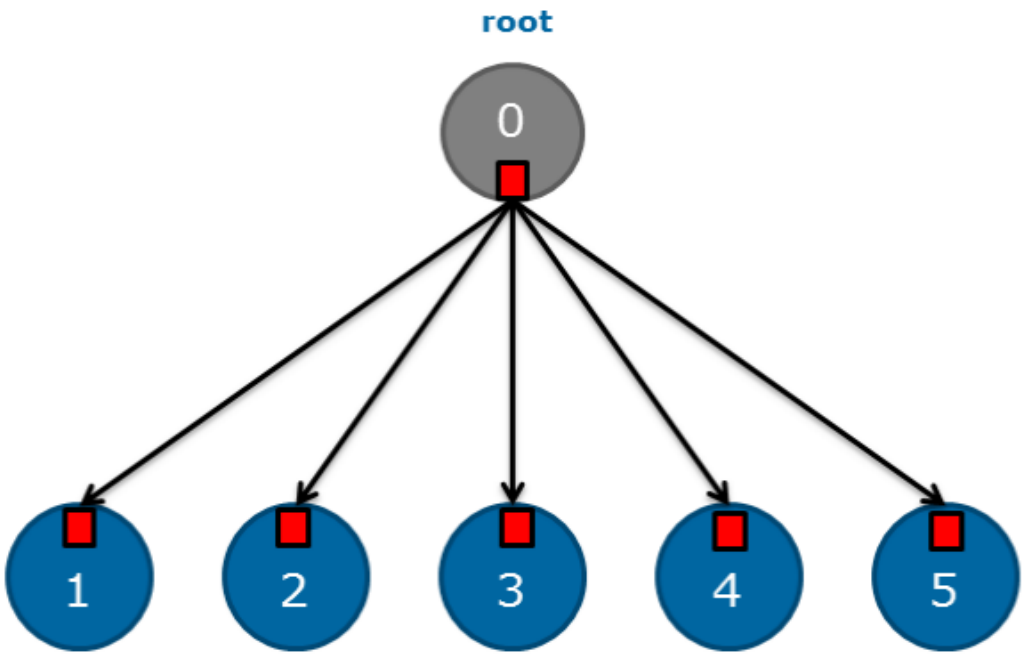
```
int MPI_Barrier(MPI_Comm comm)
```

- Barrier synchronization across all members of a *comm.*
- Blocks the caller until all group members have called it
- Returns at any process only after all processes in *comm* have entered the call.



Operação de Broadcast:

int **MPI_Bcast**(&buffer, count, datatype, root, comm)

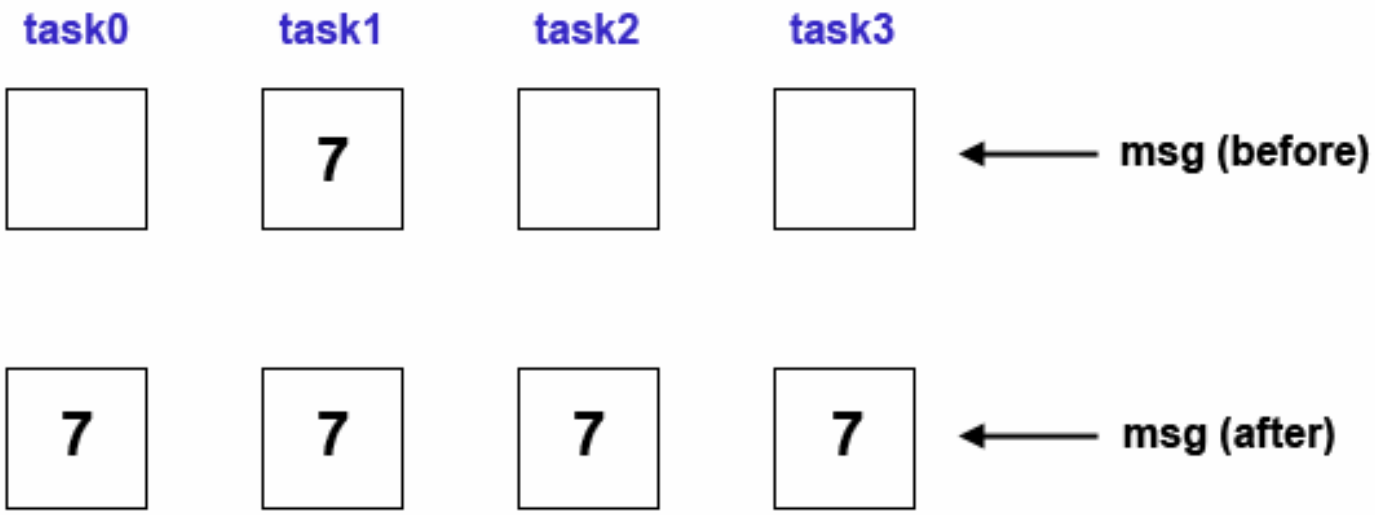


MPI_BCAST

Broadcasts a message from one task to all other tasks in communicator

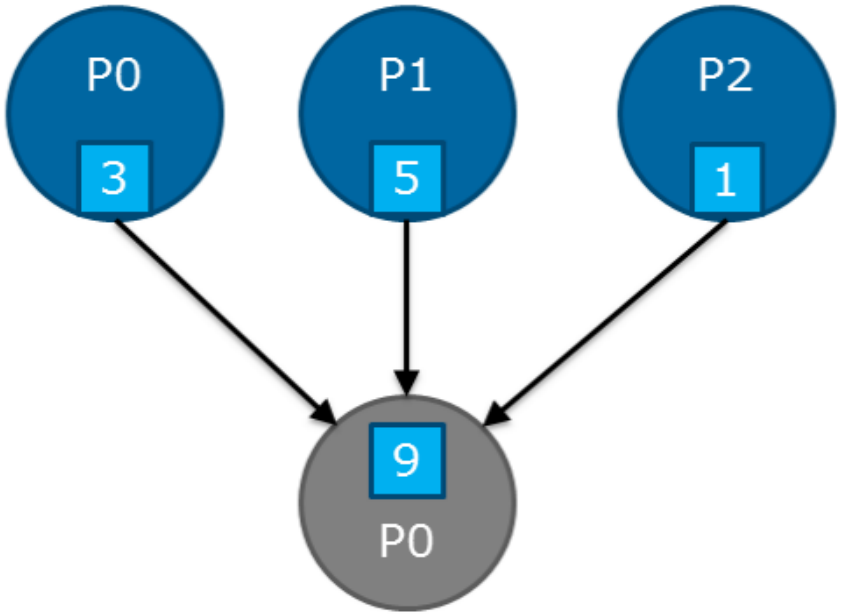
```
count = 1;
source = 1;
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

task1 contains the message to be broadcast



Operação de Reduction:

- Performs a global reduce operation (for example sum, maximum, and logical and) across all members of a group
- Example with *SUM* operation

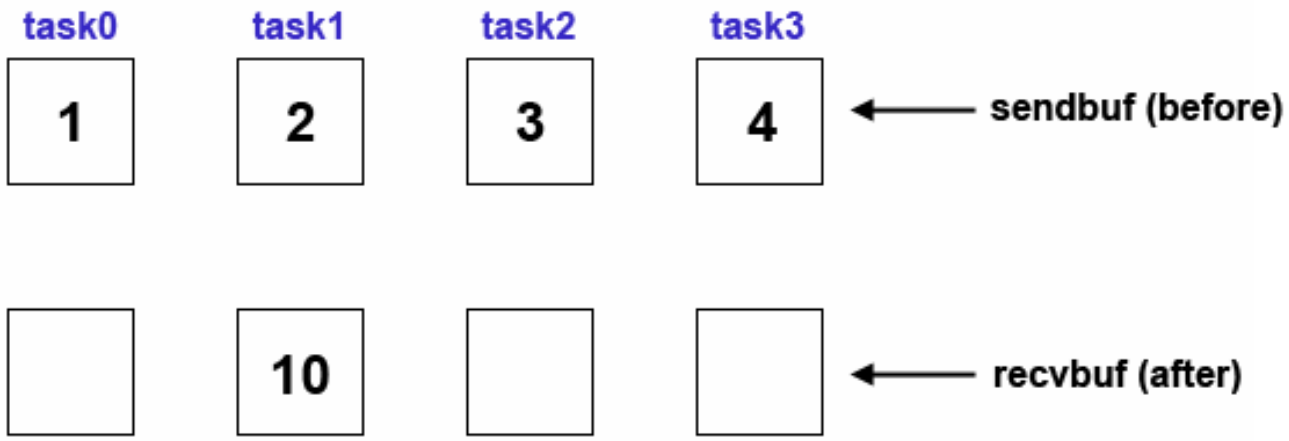


MPI_Reduce

Perform reduction across all tasks in communicator and store result in 1 task

```
count = 1;
dest = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT,
           MPI_SUM, dest, MPI_COMM_WORLD);
```

task1 will contain result



Operações pré-definidas:

Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	Min value and location

Exemplo teste04.c

```
osthoff:lab5-03$ cat teste04.c
# include "mpi.h"
# include <math.h>
# include <stdio.h>
int main(argc, argv)
        int argc;
        char *argv[];
{
        int n, myid, numprocs, i;
        double mypi, pi, h, x, sum = 0.0;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);

        /* Calculo de Pi */
        if ( myid == 0 ){
                printf("Entre com o numero de intervalos: ");
                scanf("%d", &n);
        }

        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if ( n != 0 ){
                h = 1.0/(double)n;
                for ( i = myid + 1; i <= n; i += numprocs ){
                        x = h * ((double)i - 0.5 );
                        sum += (4.0/(1.0 + x*x));
                }
        }

        /* Fim calculo Pi */

        mypi = h * sum;

        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

        if ( myid == 0 ){
                printf("Valor aproximado de Pi: %.16f\n", pi);
        }

MPI_Finalize();

}
```

Execução do teste04.c na sua máquina:

- MPI "build script" para compilar:
  - **mpicc teste04.c –o teste04**

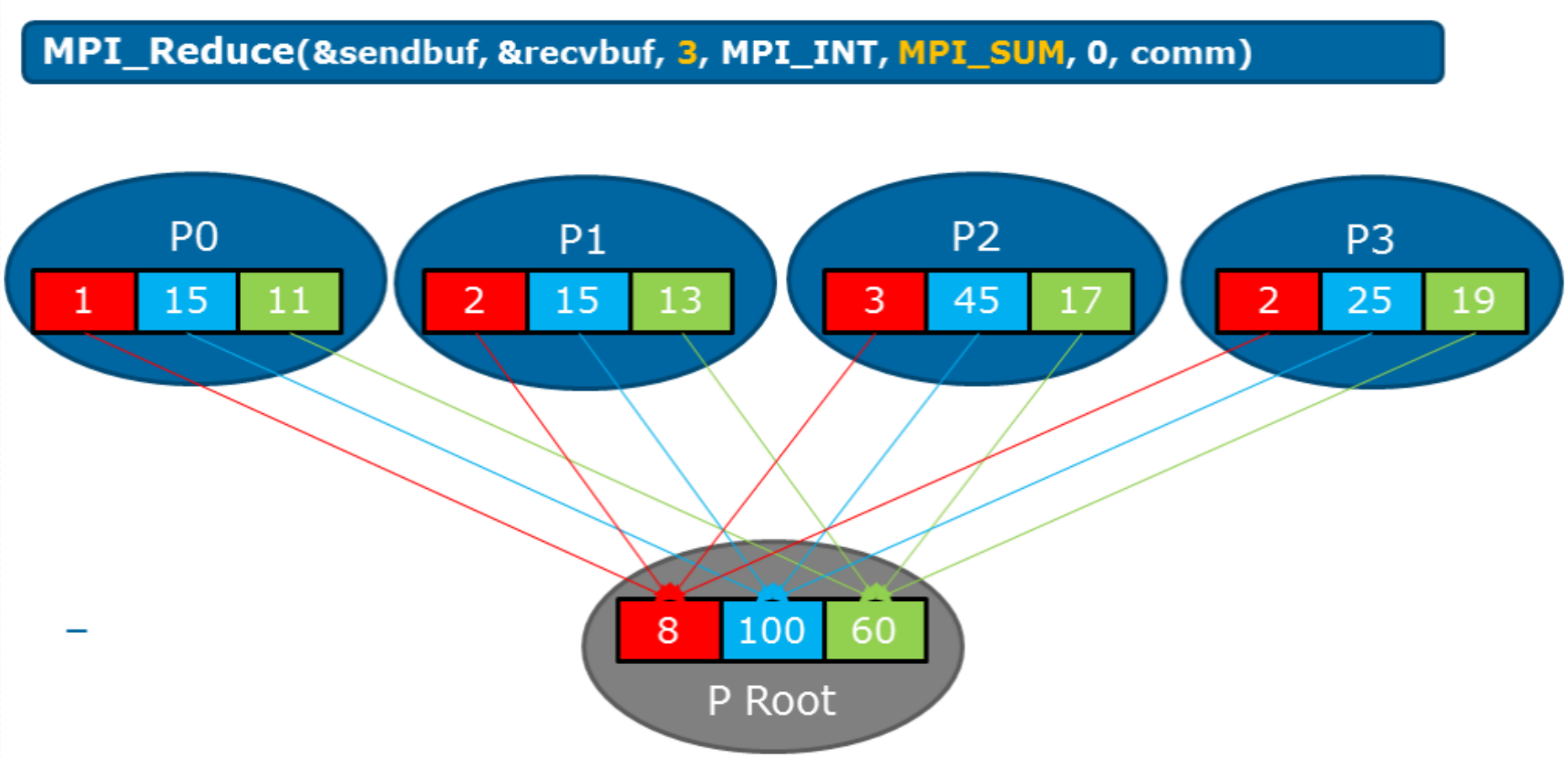
- MPI "build script" para executar:
  - **mpirun –np 16 ./teste04 10000000000**

  - **OBS: O numero de cores físicos do seu processor pode ser obtido pelo comando: $cat /proc/spuinfo**

Execução do teste04.c no SDumont

- MPI “build script” para compilar
  - **$compilar.sh**
- MPI “build script” para submeter na fila de execução:
  - **$sbatch sub_cal_pi.sh teste04 10000000000**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

Exemplo de Reduce:

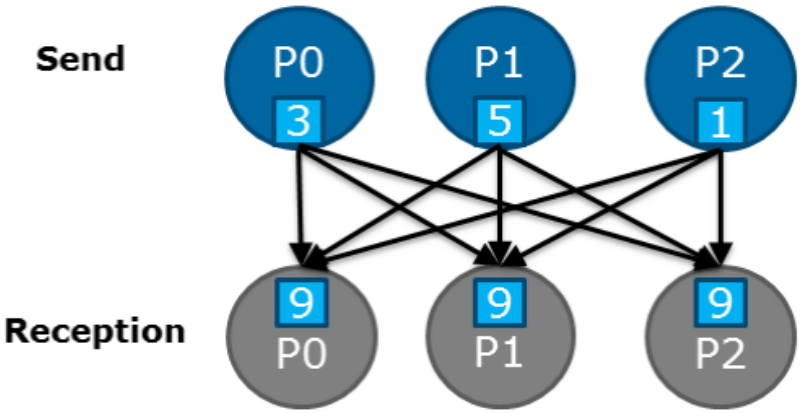


Operação MPI_ALLreduce:

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

- A variant of the reduce operations where the result is returned to all processes in a group.

- Example with *SUM* operation

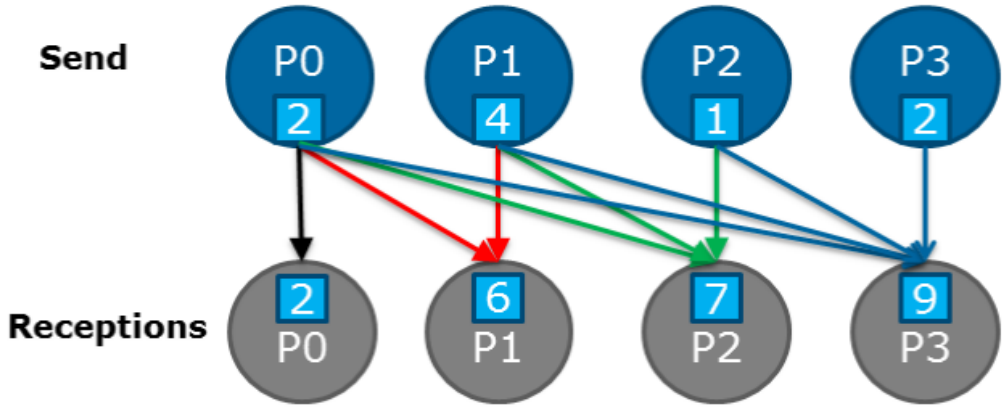


Operação SCAN:

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

- The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0,. . .,i

- Example with *SUM* operation

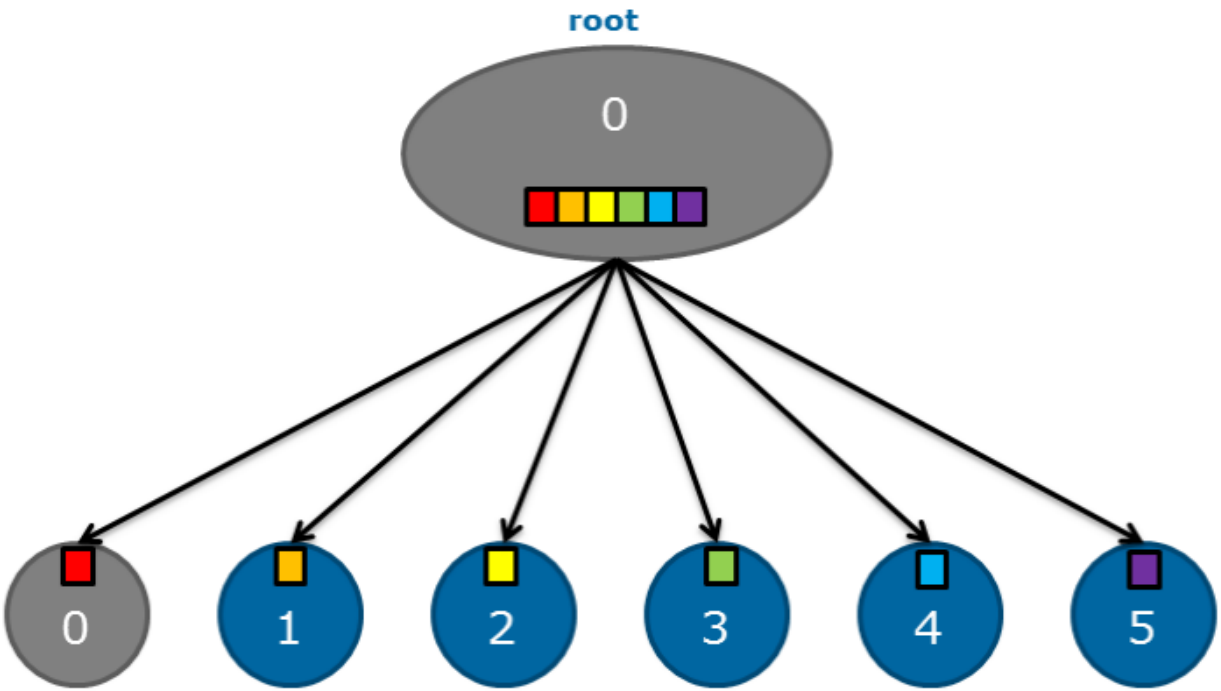


Operação de Reduce definida pelo usuário:

```
int MPI_Op_create(MPI_User_function* user_fct, int commute, MPI_Op* op)
int MPI_Op_free(MPI_Op* op)
void MPI_User_function(void* invec, void* inoutvec, int *len, MPI_Datatype *datatype)
```

Operação Scatter:

```
int MPI_Scatter(&sendbuf, sendcnt, sendtype,
                &recvbuf, recvcnt, recvtype, root, comm)
```



Operação SCATTER

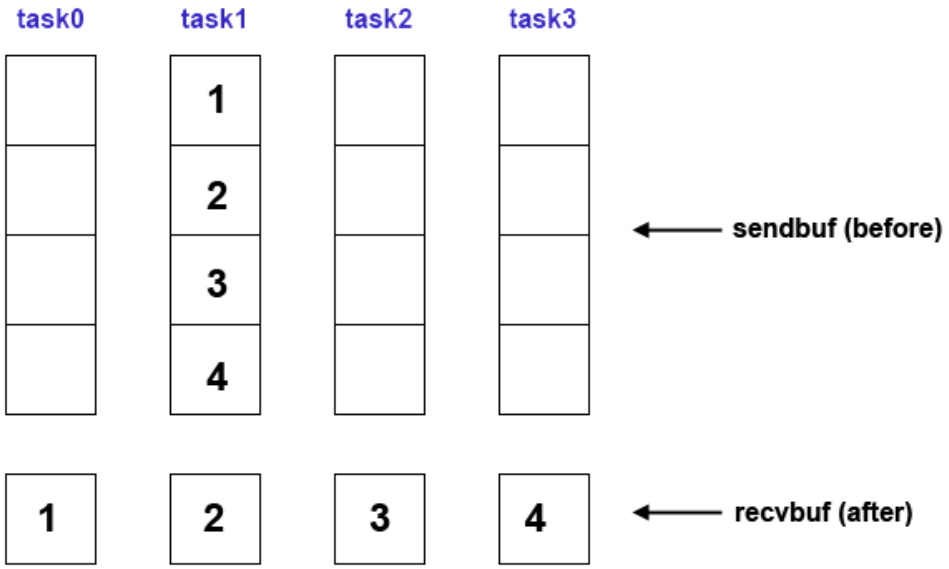
- Muito utilizada para aumentar a eficiência da transferência dos dados do disco para os nós computacionais. O nó master executa a operação de leitura do disco de forma “seqüencial e contígua” e a seguir envia os dados para demais através da rede de alto desempenho.

MPI_Scatter

Sends data from one task to all other tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Scatter(sendbuf, sendcnt, MPI_INT
            recvbuf, recvcnt, MPI_INT
            src, MPI_COMM_WORLD);
```

task1 contains the data to be scattered



Exemplo MPI_Scatter: teste05.c

C Language - Collective Communications Example

```c
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

main(int argc, char *argv[])  {
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

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

if (numtasks == SIZE) {
  // define source task and elements to send/receive, then perform collective scatter
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
              MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Execução do teste05.c na sua máquina:

- MPI "build script" para compilar:
  - **mpicc teste05.c –o teste05**

- MPI "build script" para executar:
  - **mpirun –np 4   ./teste05**

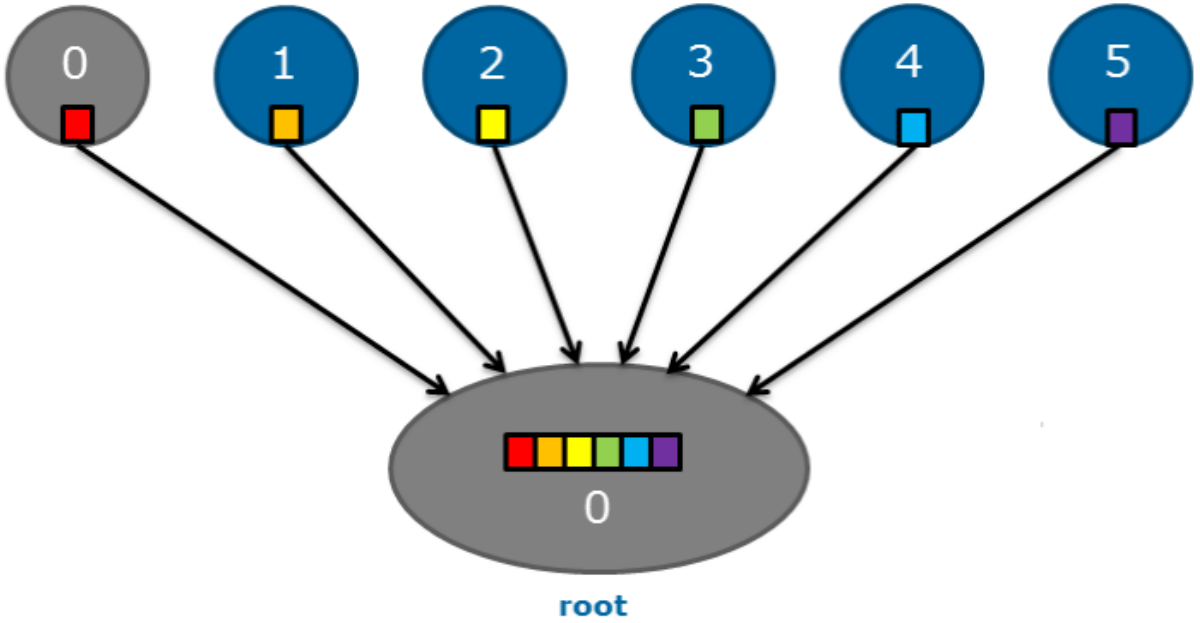
  - **(observe que tenho que se executar com 5 processadores o quinta processador não receberá nenhum dado. Isto faz a operação scatter ser pouco flexível)**

Execução do teste05.c no SDumont

- MPI "build script" para compilar
  - **$compilar.sh**
- MPI "build script" para submeter na fila de execução:
  - **$sbatch sub.sh teste05**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

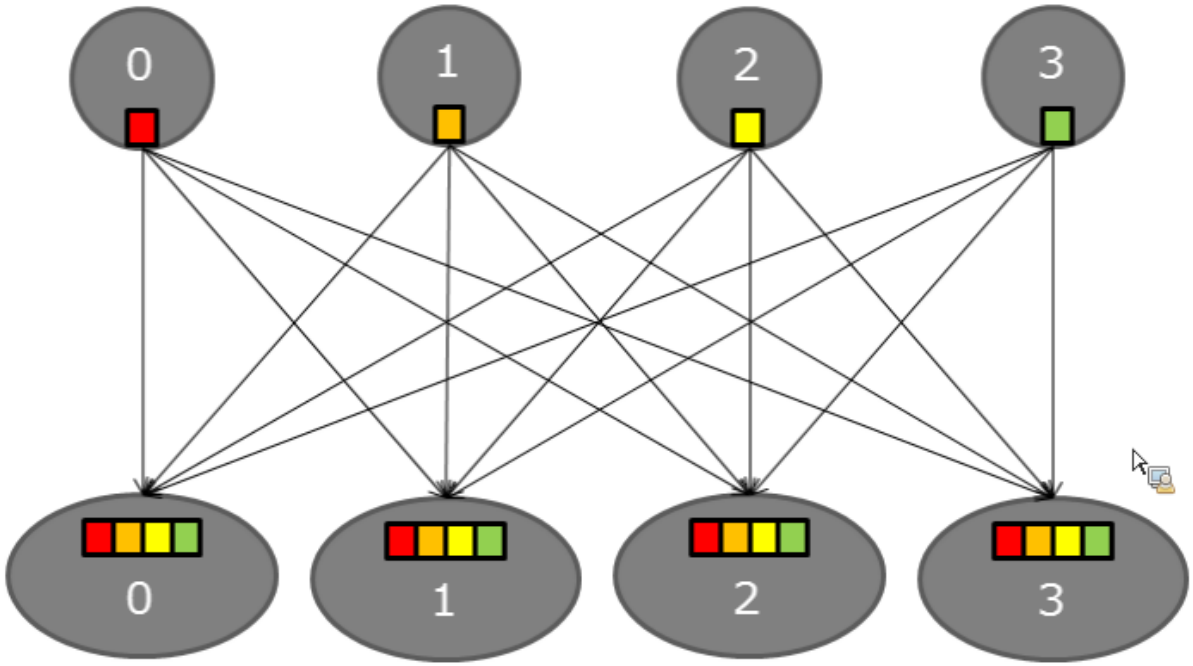
Operação Gather:

```
int MPI_Gather(&sendbuf, sendcnt, sendtype,
               &recvbuf, recvcount, recvtype, root, comm)
```



Operação Allgather:

```
int MPI_Allgather(&sendbuf, sendcount, sendtype,
                  &recvbuf, recvcount, recvtype, comm)
```



Tipos de dados Derivados

- MPI fornece ferramentas para que o programador possa definir as suas próprias estruturas de dados baseadas em sequencias de tipos de dados primitivos de MPI.

Tipos de dados do C

MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED_LONG

MPI_UNSIGNED
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_BYTE
MPI_PACKED

Rotinas para a construção de tipos de dados derivados

- Contínua (**MPI_Type_contiguous)**
- • Vetor (**MPI_Type_vector)**
- • Indexado (**MPI_Type_indexed)**
- • Estruturado (**MPI_Type_struct)**

TIPO: MPI_Type_contiguous

- É o construtor mais simples.
- Produz um novo tipo de dado contínuo, fazendo cópias de um tipo de dado existente

MPI_Type_commit

- Informa o novo datatype aos processadores da comunicação coletiva.
- Necessita ser executado antes da execução de um construtor de tipos de dados derivado.

MPI_Type_free

- Libera o objeto especificado pelo tipo de dado.
- O uso desta rotina é importante para evitar o gasto de memória quando muitos objetos de tipos de dados são criados como por exemplo em um loop.

Exemplo: teste06.c

- Cria um tipo de dado representando a linha de um array e distribui linhas diferentes do array para os processos.
- Observe que ao utilizar uma operação MPI_Send, temos mais flexibilidade para o envio dos dados.

C Language - Contiguous Derived Data Type Example

```c
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] =
  {1.0, 2.0, 3.0, 4.0,
   5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0,
   13.0, 14.0, 15.0, 16.0};
float b[SIZE];

MPI_Status stat;
MPI_Datatype rowtype;    // required variable

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

// create contiguous derived data type
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

if (numtasks == SIZE) {
   // task 0 sends one element of rowtype to all tasks
   if (rank == 0) {
      for (i=0; i<numtasks; i++)
        MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
      }

   // all tasks receive rowtype data from task 0
   MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
          rank,b[0],b[1],b[2],b[3]);
   }
else
   printf("Must specify %d processors. Terminating.\n",SIZE);

// free datatype when done using it
MPI_Type_free(&rowtype);
MPI_Finalize();
}
```

Execução do teste06.c na sua máquina:

- MPI "build script" para compilar:
  - **mpicc teste06.c –o teste06**

- MPI "build script" para executar:
  - **mpirun –np 4   ./teste06**

Execução do teste06.c no SDumont

- MPI "build script" para compilar
  - **$compilar.sh**
- MPI "build script" para submeter na fila de execução:
  - **$sbatch sub.sh teste06**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

TIPO: MPI_Type_vector

count = 4; blocklength = 1; stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT, &columntype);

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);

2.0	6.0	10.0	14.0

1 element of columntype

Exemplo: teste07.c

- Permite uma transferencia eficiente e mais amigável de colunas de uma matriz.
- Permite regular as lacunas (strides) nos deslocamentos.

C Language - Vector Derived Data Type Example

```c
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] =
  {1.0, 2.0, 3.0, 4.0,
   5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0,
  13.0, 14.0, 15.0, 16.0};
float b[SIZE];

MPI_Status stat;
MPI_Datatype columntype;    // required variable


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

// create vector derived data type
MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
MPI_Type_commit(&columntype);

if (numtasks == SIZE) {
   // task 0 sends one element of columntype to all tasks
   if (rank == 0) {
      for (i=0; i<numtasks; i++)
         MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
      }

   // all tasks receive columntype data from task 0
   MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
          rank,b[0],b[1],b[2],b[3]);
   }
else
   printf("Must specify %d processors. Terminating.\n",SIZE);

// free datatype when done using it
MPI_Type_free(&columntype);
MPI_Finalize();
}
```

Execução do teste07.c na sua máquina:

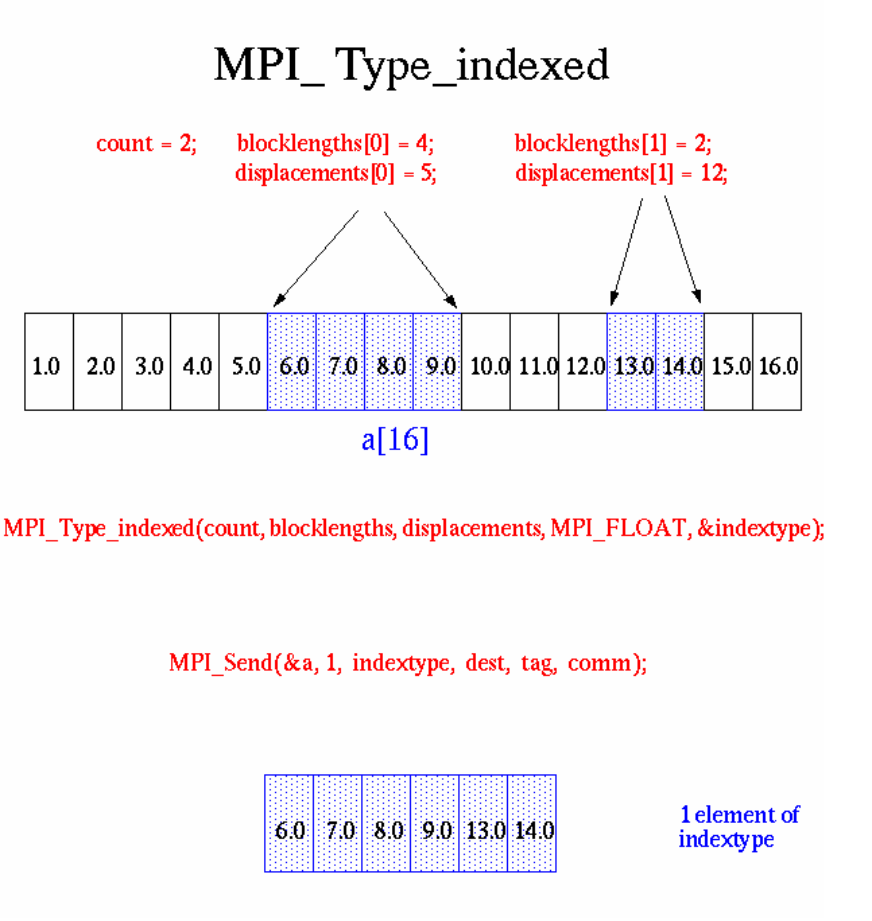
- MPI "build script" para compilar:
  - **mpicc teste06.c –o teste07**

- MPI "build script" para executar:
  - **mpirun –np 4   ./teste07**

Execução do teste07.c no SDumont

- MPI "build script" para compilar
  - **$compilar.sh**
- MPI "build script" para submeter na fila de execução:
  - **$sbatch sub.sh teste07**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

TIPO: MPI_Type_indexed



Exemplo com Indexed Derived Data Type: Teste08.c

- Cria um tipo de dado extraindo porções variáveis de um array e distribui para todas as processos
- Permite uma transferência eficiente para dados que não estão armazenados de forma continua na memória.
- Permite uma transferência eficiente para processamento de dados de uma matriz esparsa.

C Language - Indexed Derived Data Type Example

```c
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6

main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
int blocklengths[2], displacements[2];
float a[16] =
  {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[NELEMENTS];

MPI_Status stat;
MPI_Datatype indextype;    // required variable

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

blocklengths[0] = 4;
blocklengths[1] = 2;
displacements[0] = 5;
displacements[1] = 12;

// create indexed derived data type
MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT, &indextype);
MPI_Type_commit(&indextype);

if (rank == 0) {
  for (i=0; i<numtasks; i++)
    // task 0 sends one element of indextype to all tasks
      MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
  }

// all tasks receive indextype data from task 0
MPI_Recv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
        rank,b[0],b[1],b[2],b[3],b[4],b[5]);

// free datatype when done using it
MPI_Type_free(&indextype);
MPI_Finalize();
}
```

Execução do teste08.c na sua máquina:

- MPI "build script" para compilar:
  - **mpicc teste08.c –o teste08**

- MPI "build script" para executar:
  - **mpirun –np 4   ./teste08**

Execução do teste08.c no SDumont

- MPI "build script" para compilar
  - **$compilar.sh**
- MPI "build script" para submeter na fila de execução:
  - **$sbatch sub.sh teste08**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

TIPO: MPI_Type_struct

- O novo tipo de dado é formado de acordo com tipos de dado de cada componente da estrutura de dados.

Exemplo com Struct Derived Data Type – Teste09.c

- Cria o tipo de dado que representa uma partícula e distribui um array de partículas para todos os processos.

- Permite uma transferência eficiente e mais amigável de dados compostos por tipos diferente de dados.

C Language - Struct Derived Data Type Example

```
#include "mpi.h"
#include <stdio.h>
#define NELEM 25

main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;

typedef struct {
  float x, y, z;
  float velocity;
  int  n, type;
  }            Particle;
Particle      p[NELEM], particles[NELEM];
MPI_Datatype particletype, oldtypes[2];   // required variables
int           blockcounts[2];

// MPI_Aint type used to be consistent with syntax of
// MPI_Type_extent routine
MPI_Aint    offsets[2], extent;

MPI_Status stat;

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

// setup description of the 4 MPI_FLOAT fields x, y, z, velocity
offsets[0] = 0;
oldtypes[0] = MPI_FLOAT;
blockcounts[0] = 4;

// setup description of the 2 MPI_INT fields n, type
// need to first figure offset by getting size of MPI_FLOAT
MPI_Type_extent(MPI_FLOAT, &extent);
offsets[1] = 4 * extent;
oldtypes[1] = MPI_INT;
blockcounts[1] = 2;

// define structured type and commit it
MPI_Type_struct(2, blockcounts, offsets, oldtypes, &particletype);
MPI_Type_commit(&particletype);
```

Continuação:

```
// task 0 initializes the particle array and then sends it to each task
if (rank == 0) {
  for (i=0; i<NELEM; i++) {
     particles[i].x = i * 1.0;
     particles[i].y = i * -1.0;
     particles[i].z = i * 1.0;
     particles[i].velocity = 0.25;
     particles[i].n = i;
     particles[i].type = i % 2;
     }
  for (i=0; i<numtasks; i++)
     MPI_Send(particles, NELEM, particletype, i, tag, MPI_COMM_WORLD);
  }

// all tasks receive particletype data
MPI_Recv(p, NELEM, particletype, source, tag, MPI_COMM_WORLD, &stat);

printf("rank= %d   %3.2f %3.2f %3.2f %3.2f %d %d\n", rank,p[3].x,
     p[3].y,p[3].z,p[3].velocity,p[3].n,p[3].type);

// free datatype when done using it
MPI_Type_free(&particletype);
MPI_Finalize();
}
```

MPI_Type_extent

- Retorna o tamanho em bytes do tipo de dado especificado.
- É útil para sub-rotinas MPI que necessitam especificar os deslocamentos em bytes

Execução do teste09.c na sua máquina:

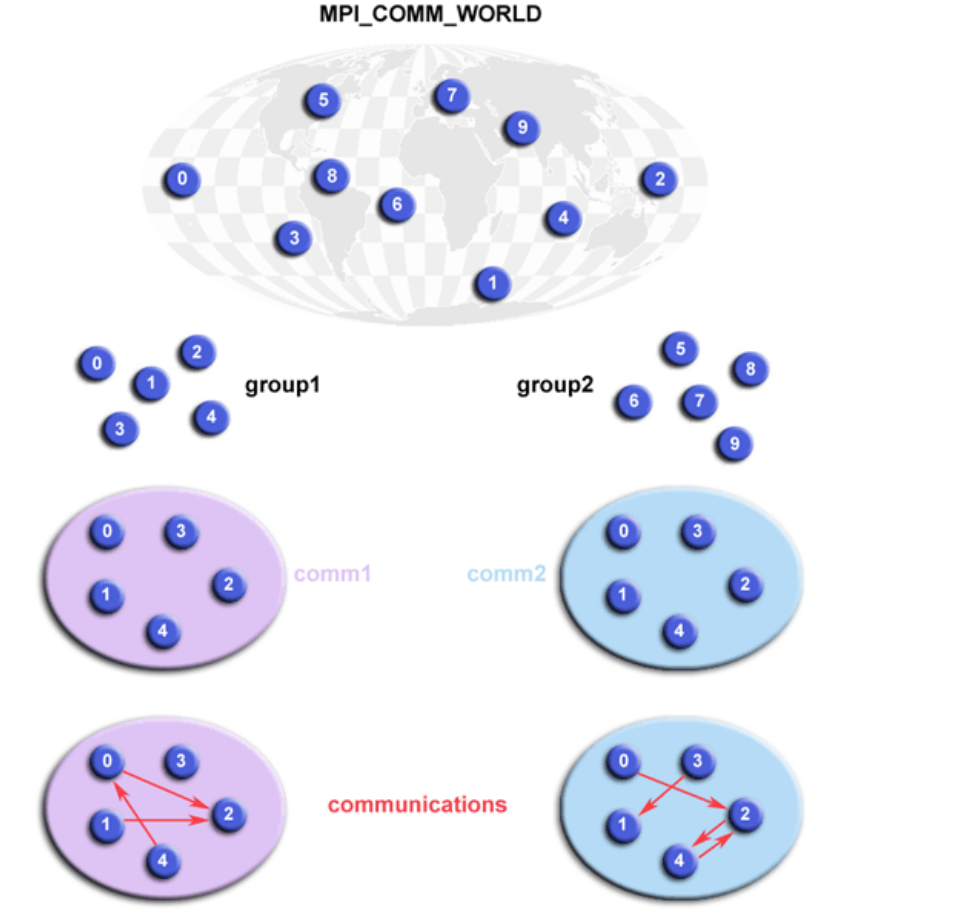
- MPI "build script" para compilar:
  - **mpicc teste09.c –o teste09**

- MPI "build script" para executar:
  - **mpirun –np 4   ./teste09**

Execução do teste09.c no SDumont

- MPI "build script" para compilar
  - **$compilar.sh**
- MPI "build script" para submeter na fila de execução:
  - **$sbatch sub.sh teste09**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

Grupos e Comunicadores



Comunicadores

- Engloba um grupo de processos que podem se comunicar.
- Todas as mensagens MPI devem especificar um comunicador.
- Implementam operações de comunicações coletivas través de um subconjunto de processos relacionados.

Grupo

- Um grupo é um conjunto ordenado de processos, onde cada processo é associado à um rank.
- As rotinas do grupo rotinas são utilizadas principalmente para especificar quais processos devem ser usados para construir um comunicador
- **Do ponto de vista do programador, um comunicador e um grupo são iguais**.

Considerações e Restrições

- Os Grupos e os comunicadores são dinâmicos; eles podem ser criados e destruídos durante a execução do programa.
- Os Processos podem pertencer a mais de um grupo e de um comunicador.
- Eles possuirão um único rank dentro de cada grupo e comunicador.
- O padrão MPI fornece mais de 40 rotinas relacionadas aos grupos, comunicadores, e topologias virtuais.

C Language - Group and Communicator Example

```c
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8

main(int argc, char *argv[])  {
int         rank, new_rank, sendbuf, recvbuf, numtasks,
            ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
MPI_Group   orig_group, new_group;    // required variables
MPI_Comm    new_comm;    // required variable

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

if (numtasks != NPROCS) {
  printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
  MPI_Finalize();
  exit(0);
  }

sendbuf = rank;

// extract the original group handle
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

//  divide tasks into two distinct groups based upon rank
if (rank < NPROCS/2) {
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
  }
else {
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
  }

// create new new communicator and then perform collective communications
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

// get rank in new group
MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

MPI_Finalize();
}
```

Exemplo de Grupo e de Comunicador-Teste10.c

- Cria dois grupos de processos distintos para troca de comunicação coletiva. Necessita da criação de novos grupos de comunicação.
- Saída:

```
rank= 7 newrank= 3 recvbuf= 22
rank= 0 newrank= 0 recvbuf= 6
rank= 1 newrank= 1 recvbuf= 6
rank= 2 newrank= 2 recvbuf= 6
rank= 6 newrank= 2 recvbuf= 22
rank= 3 newrank= 3 recvbuf= 6
rank= 4 newrank= 0 recvbuf= 22
rank= 5 newrank= 1 recvbuf= 22
```

Execução do teste10 na sua máquina:

- MPI "build script" para compilar:
  - **mpicc teste10.c –o teste10**

- MPI "build script" para executar:
  - **mpirun –np 8 ./teste10**

Execução do teste10 no SDumont

- MPI “build script” para compilar
  - **$compilar.sh**
- MPI “build script” para submeter na fila de execução:
  - **$sbatch sub_teste10.sh teste10**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

Topologias Virtuais

- Em termos de MPI, uma topologia virtual descreve a ordenação de um mapeamento de processos MPI em uma forma geométrica.
- Os principais tipos de topologias MPI são o cartesiano (malha) e o gráfico.
- As topologias MPI são virtuais - pode não haver relação entre a estrutura física da máquina, e os processos paralelos da topologia

Exemplo para Topologia Cartesiana: envio dados para 4 vizinhos

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

```c
#include "mpi.h"
#include <stdio.h>
#define SIZE 16
#define UP    0
#define DOWN  1
#define LEFT  2
#define RIGHT 3

main(int argc, char *argv[])  {
int numtasks, rank, source, dest, outbuf, i, tag=1,
   inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
   nbrs[4], dims[2]={4,4},
   periods[2]={0,0}, reorder=0, coords[2];

MPI_Request reqs[8];
MPI_Status stats[8];
MPI_Comm cartcomm;   // required variable

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
   // create cartesian virtual topology, get rank, coordinates, neighbor ranks
   MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
   MPI_Comm_rank(cartcomm, &rank);
   MPI_Cart_coords(cartcomm, rank, 2, coords);
   MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
   MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

   printf("rank= %d coords= %d %d  neighbors(u,d,l,r)= %d %d %d %d\n",
          rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],
          nbrs[RIGHT]);

   outbuf = rank;

   // exchange data (rank) with 4 neighbors
   for (i=0; i<4; i++) {
      dest = nbrs[i];
      source = nbrs[i];
      MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
                MPI_COMM_WORLD, &reqs[i]);
      MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
                MPI_COMM_WORLD, &reqs[i+4]);
      }

   MPI_Waitall(8, reqs, stats);

   printf("rank= %d                    inbuf(u,d,l,r)= %d %d %d %d\n",
          rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);  }
else
   printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Exemplo com Topologia Virtual: tete11.c

- Exemplo de rotina que gera uma topologia Cartesiana de 4 x 4 para 16 processadores onde cada processo envia o número do seu rank para os seus vizinhos.

Execução do teste11 na sua máquina:

- MPI "build script" para compilar:
  - **mpicc teste11.c –o teste11**

- MPI "build script" para executar:
  - **mpirun –np 16   ./teste11**

Execução do teste11 no SDumont

- MPI “build script” para compilar
  - **$compilar.sh**
- MPI “build script” para submeter na fila de execução:
  - **$sbatch sub_teste11.sh teste11**
    - **$squeue -u $USER (para visualizar o job na fila de execução**
    - **$scancel JOB_ID (para cancelar o job da fila de execução, caso seja necessário)**
    - **$more slurm_JOB_ID.out**

MPI-2:

- Intentionally, the MPI-1 specification did not address several "difficult" issues. For reasons of expediency, these issues were deferred to a second specification, called MPI-2 in 1998.
- MPI-2 was a major revision to MPI-1 adding new functionality and corrections.
- Key areas of new functionality in MPI-2:
  - **Dynamic Processes** - extensions that remove the static process model of MPI. Provides routines to create new processes after job startup.
  - **One-Sided Communications** - provides routines for one directional communications. Include shared memory operations (put/get) and remote accumulate operations.
  - **Extended Collective Operations** - allows for the application of collective operations to inter-communicators
  - **External Interfaces** - defines routines that allow developers to layer on top of MPI, such as for debuggers and profilers.
  - **Additional Language Bindings** - describes C++ bindings and discusses Fortran-90 issues.
  - **Parallel I/O** - describes MPI support for parallel I/O.

MPI-3:

- The MPI-3 standard was adopted in 2012, and contains significant extensions to MPI-1 and MPI-2 functionality including:
  - **Nonblocking Collective Operations** - permits tasks in a collective to perform operations without blocking, possibly offering performance improvements.
  - **New One-sided Communication Operations** - to better handle different memory models.
  - **Neighborhood Collectives** - extends the distributed graph and Cartesian process topologies with additional communication power.
  - **Fortran 2008 Bindings** - expanded from Fortran90 bindings
  - **MPIT Tool Interface** - allows the MPI implementation to expose certain internal variables, counters, and other states to the user (most likely performance tools).
  - **Matched Probe** - fixes an old bug in MPI-2 where one could not probe for messages in a multi-threaded environment.

More Information on MPI-2 and MPI-3:

- MPI Standard documents: http://www.mpi-forum.org/docs/