MPI
(Message Passing Interface)

- Standard desenvolvido por académicos e indústria.
- Define rotinas, não a implementação.
- Não define como se criam os processos (depende da implementação)
- Existem várias implementações open source.

Communicators

- Definem o âmbito de uma operação de comunicação.
- Cada processo tem um identificar (rank) associado com o *communicator*.
- Inicialmente todos os processos pertencem a um "universo" designado MPI_COMM_WORLD, e cada processo tem um *rank* único de 0 a *p* - 1, com p processos.
- Podem ser criados outros communicators para grupos de processos.

Modelo de computação SPMD

```
main (int argc, char *argv[])
{
MPI_Init(&argc, &argv);      // Inicializa o MPI
         .
         .
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
                          /* obtém o identificador do processo  */

if (myrank == 0)
        master();
else
        slave();
         .
         .
MPI_Finalize();    // termina o mpi
}
```

O código de master() e de slave() são executados pelos processos master e slave respetivamente.

Modelo de computação SPMD

Exemplo em Python:

```
# helloWorld.py
from mpi4py import MPI

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    print ("hello world from process ", rank)
```

Notas: ***`MPI_Init`*** *é invocado quando o módulo MPI é importado do package mpi4py;*

`MPI_Finalize()` *é invocado quando o processo python termina.*

Modelo de computação SPMD

Se executarmos o programa anterior com:

mpiexec –np 4 python helloWorld.py

em msmpi

Output:

```
hello world from process  1
hello world from process  3
hello world from process  0
hello world from process  2
```

Modelo de computação SPMD

Exemplo em Python:

```
# variables.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
      # variável global, vai ser replicada em cada processo

if rank != 0:
   x = 123                          # variável local
   print ("I'm process %s , my variable is %d" % (rank, x ))
else:
   x = 321
   print ("I'm process %s , my variable is %d" % (rank, x ) )
```

Modelo de computação SPMD

Se executarmos o programa anterior com:

mpiexec –np 6 python variables.py

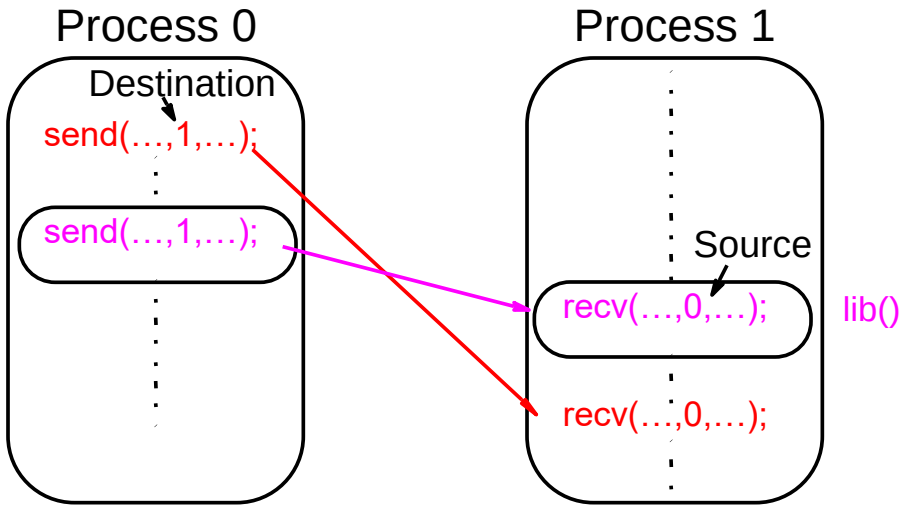
em msmpi

Output:

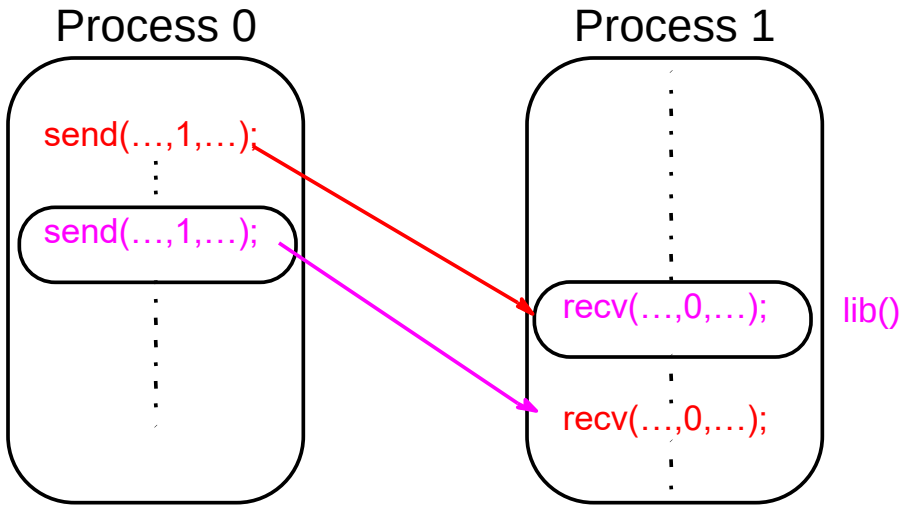
```
I'm process 5 , my variable is 123
I'm process 4 , my variable is 123
I'm process 1 , my variable is 123
I'm process 0 , my variable is 321
I'm process 3 , my variable is 123
I'm process 2 , my variable is 123
```

Comunicação não segura - Example

(a) Comportamento desejado



(b) Comportamento possível



Solução em MPI: “Communicators”

- Definem um domínio de comunicação – conjunto de processos que podem comunicar entre si.

- Domínio de comunicação das bibliotecas pode ser diferente do domínio do programador

- Usados em todas as comunicações ponto a ponto e coletivas

Default Communicator

MPI_COMM_WORLD

- Existe como primeiro comunicador para todos os processos de uma aplicação.
- Existe um conjunto de rotinas MPI para formar communicators.
- Os processos têm um rank num communicator.
- Existem communicators para comunicar dentro de um grupo e communicators para comunicar entre grupos

Comunicação MPI Point-to-Point

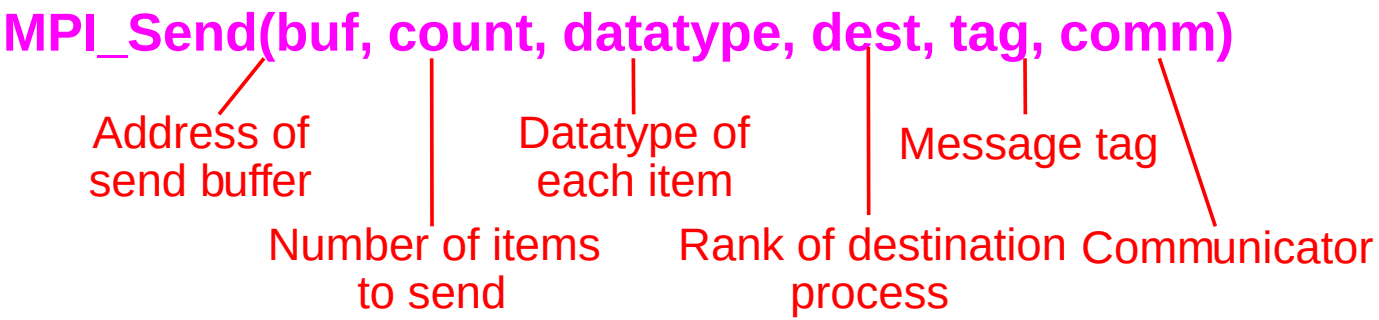
- Rotinas de send e receive com “message tags” e communicator.
- É possível usar wild cards, para o identificador do processo e para a tag

Rotinas MPI bloqueantes

- Retornam quando estão completas localmente. – isto é, quando a localização usada para guardar a mensagem pode ser usada novamente ou alterada sem afetar a mensagem enviada.

- Send bloqueante vai enviar a mensagem e retorna -- não significa que a mensagem foi recebida, apenas que o processo emissor pode prosseguir.

Blocking send

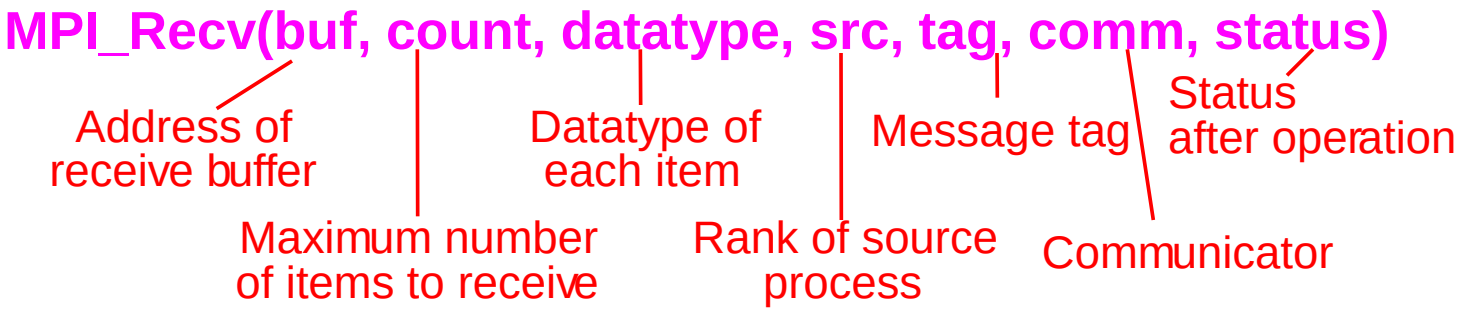


Em python:

```
Send(self, buf, int dest, int tag=0)
```

Exemplo: comm.send(data, dest=1, tag=11)

Parameters of blocking receive



Em Python:

```
Recv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG, Status status=None)
```

Exemplo: data = comm.recv(source=0, tag=11)

Example

To send an integer x from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */

if (myrank == 0) {
   int x = 100;
   MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
   int x;
   MPI_Recv(&x, 1, MPI_INT, 0,msgtag,MPI_COMM_WORLD,status);
}
```

Example em python:

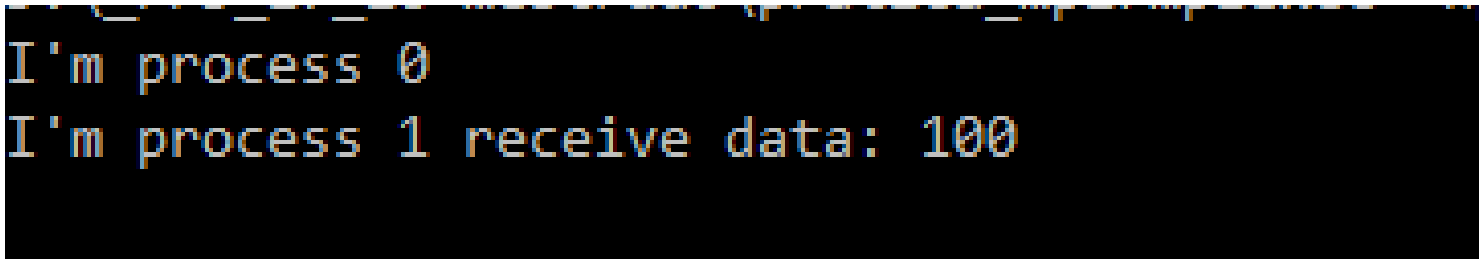
```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    x = 100
    comm.send(x, dest=1, tag=11)
    print ("I'm process %s " % rank )
elif rank ==1 :
    x = comm.recv(source=0, tag=11)
    print ("I'm process %s receive data: %s" % (rank , x))
```

Example em python:

Executando:

mpiexec –np 2 python exemplo.py



MPI Nonblocking Routines

- Nonblocking send - MPI_Isend() – retorna imediatamente mesmo antes de ser seguro altera a localização.

- Nonblocking receive - MPI_Irecv() – retorna mesmo que não haja mensagem para aceitar

Nonblocking Routine Formats

```
MPI_Isend(buf,count,datatype,dest,tag,comm,request)
```

```
MPI_Irecv(buf,count,datatype,source,tag,comm, request)
```

Para detetar se a operação já completou: **`MPI_Wait()`** and **`MPI_Test()`**.

`MPI_Wait()` Bloqueia até que a operação esteja concluída.

`MPI_Test()` Retorna uma flag que indica se a operação já está completa ou não.

MPI_Wait e MPI_Test são operações do parâmetro **`request`**.

Em Python

```
Isend(self, buf, int dest, int tag=0)
```

```
Irecv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG)
```

```
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
...
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    ...
    data = req.wait()
    print (data)
```

Example

To send an integer x from process 0 to process 1 and allow process 0 to continue,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
   int x;
   MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
   compute();
   MPI_Wait(req1, status);
} else if (myrank == 1) {
   int x;
   MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```

Modos de comunicação (Send)

- Standard Mode Send - Não assume que o método receive tenha começado. O MPI não define um buffer. Dependendo da implementação, o send pode terminar antes do receive começar se existir um buffer.

- Buffered Mode – O Send pode começar e retornar antes do receive correspondente. É cecessário fornecer o tamanho do buffer com a rotina MPI_Buffer_attach(). Pode ser removido com MPI_Buffer_detach()

Modos de comunicação (Send)

- Synchronous Mode - Send e receive podem começar antes um do outro, mas só podem completar ao mesmo tempo.

- Ready Mode – O send só pode começar quando o receive correspondente é atingido.

- Qualquer do 4 modos pode ser aplicado aos dois sends, bloqueante ou não bloqueante

- Os modos não standard são identificados pelas letras,

buffered: -b; synchronous –s; ready: -r

MPI_Issend() – "nonblocking synchronous send routine"

Comunicação Coletiva

Envolve um conjunto de processos definidos por um intra-comunicador. Não existem tags nas mensagens.

- **`Operações principais:`**

- **`MPI_Bcast()`** - Broadcast from root to all other processes
- **`MPI_Gather()`** - Gather values from group of processes
- **`MPI_Scatter()`** - Scatters buffer in parts to group of processes
- **`MPI_Alltoall()`** - Sends data from all processes to all processes
- **`MPI_Reduce()`** - Combine values on all processes to single value
- **`MPI_Reduce_scatter()`** - Combine values and scatter results
- **`MPI_Scan()`** - Compute prefix reductions of data on processes

Barreira (Barrier)

Quando cada processo atinge a barreira, é suspenso até que todos atinjam o ponto de sincronização.

- Em python: `Barrier(self)`

Sample MPI program

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000
void main(int argc, char *argv)
{
     int myid, numprocs;
     int data[MAXSIZE], i, x, low, high, myresult, result;
     char fn[255];
     char *fp;
     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
     if (myid == 0) {  /* Open input file and initialize data */
            strcpy(fn,getenv("HOME"));
            strcat(fn,"/MPI/rand_data.txt");
            if ((fp = fopen(fn,"r")) == NULL) {
                        printf("Can't open the input file: %s\n\n", fn);
                        exit(1);
            }
            for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d", &data[i]);
     }
     MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD); /* broadcast data */
     x = n/nproc; /* Add my portion Of data */
     low = myid * x;
     high = low + x;
     for(i = low; i < high; i++)
            myresult += data[i];
     printf("I got %d from %d\n", myresult, myid); /* Compute local sum */
     MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
     if (myid == 0) printf("The sum is %d.\n", result);
     MPI_Finalize();
}
```

Avaliação de programas paralelos

Tempo de execução sequencial, t_s:

Estimado contando o número de passos computacionais do melhor algoritmo sequencial

Tempo de execução paralela, t_p:

Além do número de passos computacionais, t_{comp}, é necessário estimar o tempo de comunicação, t_{comm}:

$$t_p = t_{comp} + t_{comm}$$

Tempo de computação

Contar o número de passos computacionais.

Quando mais do que um processo executa simultaneamente, contar apenas as instruções do processo mais complexo. Geralmente é função do *n* e *p*.

(n – número de itens de dados, p – número de processos)

$$t_{comp} = f\,(n, p)$$

Geralmente este tempo é dividido em vários componentes:

$$t_{comp} = t_{comp1} + t_{comp2} + t_{comp3} + \ldots$$

A análise geralmente é feita supondo que os processadores são iguais e operam a igual velocidade

Tempo de comunicação

Depende do tipo de rede ou redes utilizada.

Como primeira aproximação usa-se

$$t_{comm} = t_{startup} + n\, t_{data}$$

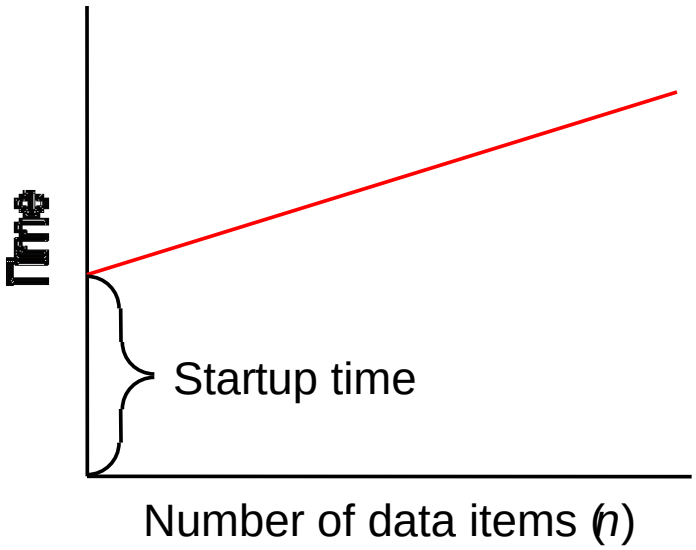
$t_{startup}$ - tempo de *startup*, corresponde ao tempo necessário para enviar uma mensagem sem dados.

Assume-se constante.

t_{data} – tempo de transmissão de um item de dados (data word)

n – número de itens de dados

Tempo ideal de comunicação



Tempo final de comunicação, t_{comm}

Soma dos tempos de comunicação de todas as mensagens sequenciais enviadas por um processo.

$$t_{comm} = t_{comm1} + t_{comm2} + t_{comm3} + \ldots$$

Assume-se que o padrão de comunicação é igual para todos os processos e ocorre em simultâneo. Basta considerar um processo.

Se $t_{startup}$ e t_{data}, forem medidos em unidades de passos computacionais, podemos adicionar t_{comp} com t_{comm} e obter o tempo de execução paralela.

Fatores de Benchmark

Com t_s, t_{comp}, and t_{comm}, podemos calcular o speedup e rácio computação/comunicação:

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{t_s}{t_{comp} + t_{comm}}$$

$$\text{Computation/communication ratio} = \frac{t_{comp}}{t_{comm}}$$

Ambos são função do número de processadores, p, e do número de itens de dados, n.

Estes fatores dão indicação da escalabilidade da solução paralela com o aumento do número de processadores e do tamanho do problema.

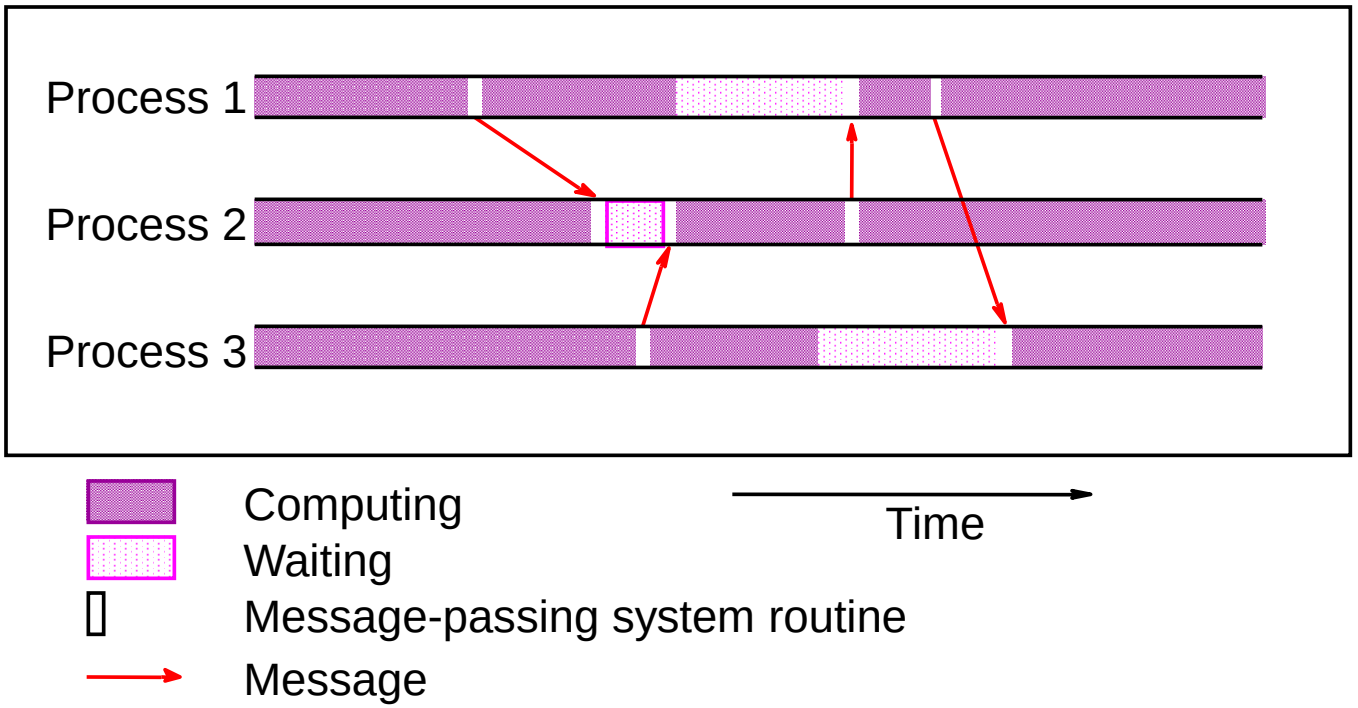
Exemplo:

Processador de 1 GFLOPS (10^9 operações de virgula flutuante por segundo e um tempo de startup de 1µs (10^{-6})

- Quantos operações pode fazer no tempo de startup?

Ferramentas de visualização

Programas que permitem analisar a execuçao de um programa paralelo num diagrama processo/tempo



Avaliar empiricamente os programas
Medindo o tempo de execução.

Para medir o tempo de execução entre os pontos L1 e L2 do código, usamos uma construção do tipo:

```
L1: time(&t1);                  /* start timer */
                .
                .
L2: time(&t2);                  /* stop timer */
                .
elapsed_time = difftime(t2, t1);  /* elapsed_time = t2 - t1 */
printf("Elapsed time = %5.2f seconds", elapsed_time);
```

MPI fornece a rotina **`MPI_Wtime()`** que devolve o tempo (em segundos).