Introdução ao MPI

Message Passing Interface

O que é Message Passing?

O modelo de Message Passing é um conjunto de processos que possuem acesso à memória local. As informações são enviadas da memória local do processo para a memória local do processo remoto.

O que é o MPI?

- MPI é uma biblioteca de Message Passing desenvolvida

para ambientes de memória distribuída, máquinas

paralelas massivas, NOWs (network of workstations) e

redes heterogêneas.

- É uma biblioteca de rotina que fornece funcionalidade básica para que os processos se comuniquem.
- O paralelismo é explicito

Como funciona MPI?

- Problemas são divididos em pequenas partes e essas partes são distribuídas para que outras máquinas do cluster façam o cálculo em cimas dessas partes.
- Os resultados obtidos das outras máquinas são enviadas a uma receptora.

Conceitos de MPI

- ***Processo:*** Cada parte do programa quebrado é chamado de processo.

- ***Rank:*** Todo o processo tem uma identificação única atribuída pelo sistema quando o processo é inicializado. Essa identificação é contínua representada por um número inteiro, começando de zero até N-1, onde N é o número de processos. Cada processo tem um rank, é ele é utilizado para enviar e receber mensagens.

Conceitos de MPI

- ***Grupos:*** Grupo é um conjunto ordenado de N processos.

  Todo e qualquer grupo é associado a um comunicador muitas vezes já predefinido como "MPI_COMM_WORLD".

- ***Comunicador:*** O comunicador é um objeto local que representa o domínio (contexto) de uma comunicação (conjunto de processos que podem ser contactados).

- O MPI_COMM_WORLD é o comunicador predefinido que inclui todos os processos definidos pelo usuário numa aplicação MPI.

Conceitos de MPI

- ***Application Buffer:*** É o endereço de memória, gerenciado pela aplicação, que armazena um dado que o processo necessita enviar ou receber.
- ***System Buffer:*** É um endereço de memória reservado pelo sistema para armazenar mensagens.

Funções MPI

- Ele apresenta quatro funções chaves e uma outra função usável
- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Get_processor_name

Exemplo de MPI

```
#include "mpi.h“
#include <stdio.h>
Int main( int argc, char * argv[ ] )
{
int processId;          /* rank dos processos */
int noProcesses;    /* Número de processos */
int nameSize;          /* Tamanho do nome */
char computerName[MPI_MAX_PROCESSOR_NAME];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);
MPI_Comm_rank(MPI_COMM_WORLD, &processId);
MPI_Get_processor_name(computerName, &nameSize);
fprintf(stderr,"Hello from process %d on %s\n", processId,
computerName);
MPI_Finalize( );
return 0; }
```

Exemplo

[Mario@prjIntel mario]$ **mpicc hello.c -o hello**
[Mario@prjIntel mario]$ **mpirun -np 5 hello**
Hello from process 0 on amy
Hello from process 2 on oscarnode2.oscardomain
Hello from process 1 on oscarnode1.oscardomain
Hello from process 4 on oscarnode4.oscardomain
Hello from process 3 on oscarnode3.oscardomain

Mpi_Init

- Inicializa um processo MPI. Portanto, deve ser a primeira rotina a ser chamada por cada processo. Ela também sincroniza todos os processos na inicialização de uma aplicação MPI.

- Exeption:
- MPI_ERR_OTHER

MPI_Finalize

- MPI_Finalize é chamada para encerrar o MPI. Ela deve ser a última função a ser chamada.

MPI_Comm_size

- Retorna o número de processos dentro de um grupo.
- Ele pega o comunicador como seu primeiro argumento e o endereço de uma variável inteira usada para retornar o número de processos.

Exeptions:

- MPI_ERR_ARG.
- MPI_ERR_RANK.
- MPI_ERR_COMM.

MPI_Comm_rank

- Identifica um processo MPI dentro de um determinado grupo.

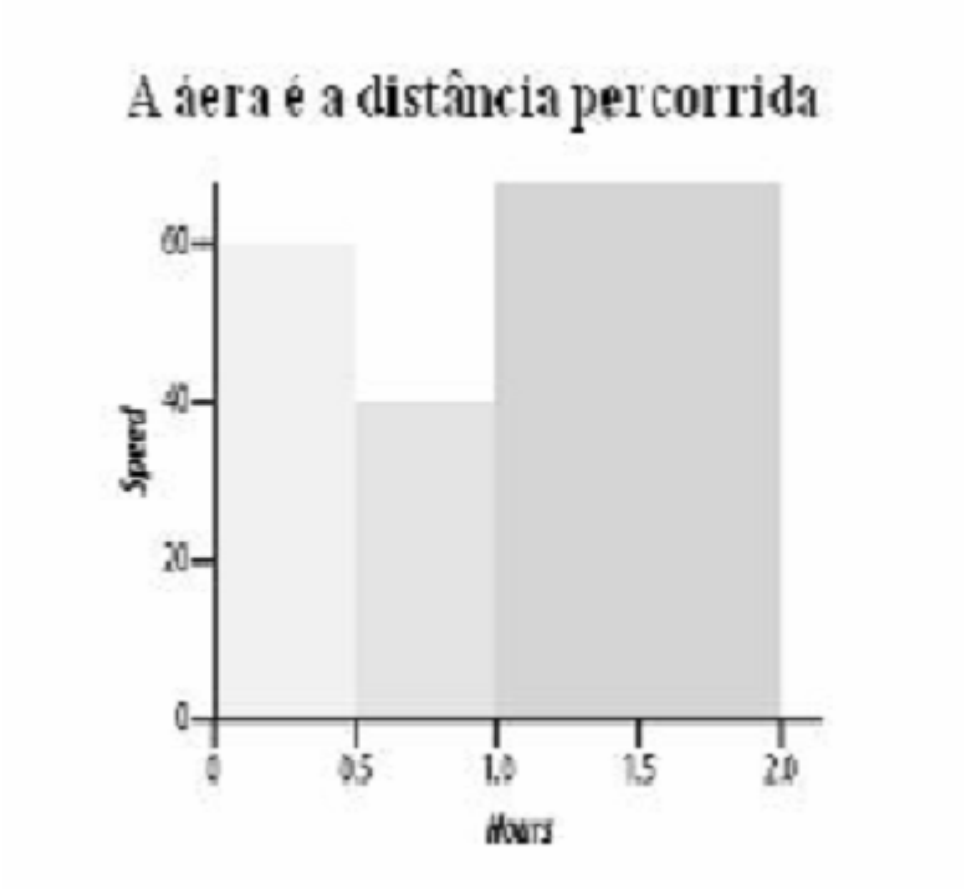
Exeption:

- MPI_ERR_COMM.

MPI_Get_processor_name

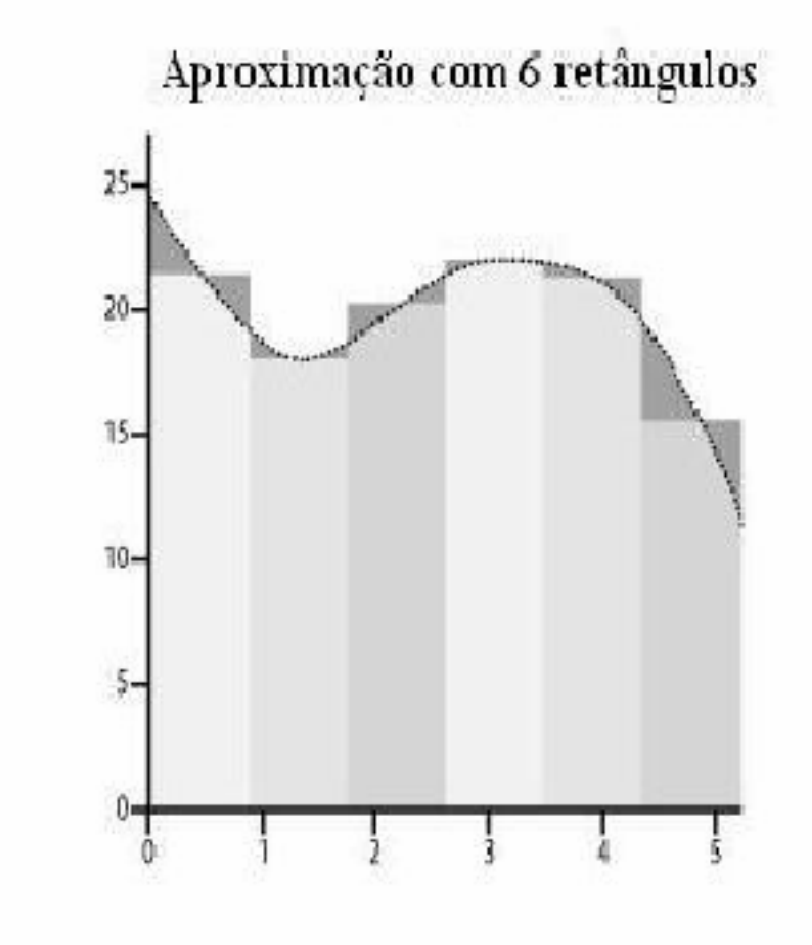
- Função Importante para fazer debug de código, mas não é tão usável.
- Ela retorna o nome do nó cujo processo individual está rodando.
- Usa um argumento para guardar o nome da máquina e outro para o tamanho do nome

Exemplo


A área é a distância percorrida
Speed
60
40
20
0
0
0.5
1.0
1.5
2.0
Hours

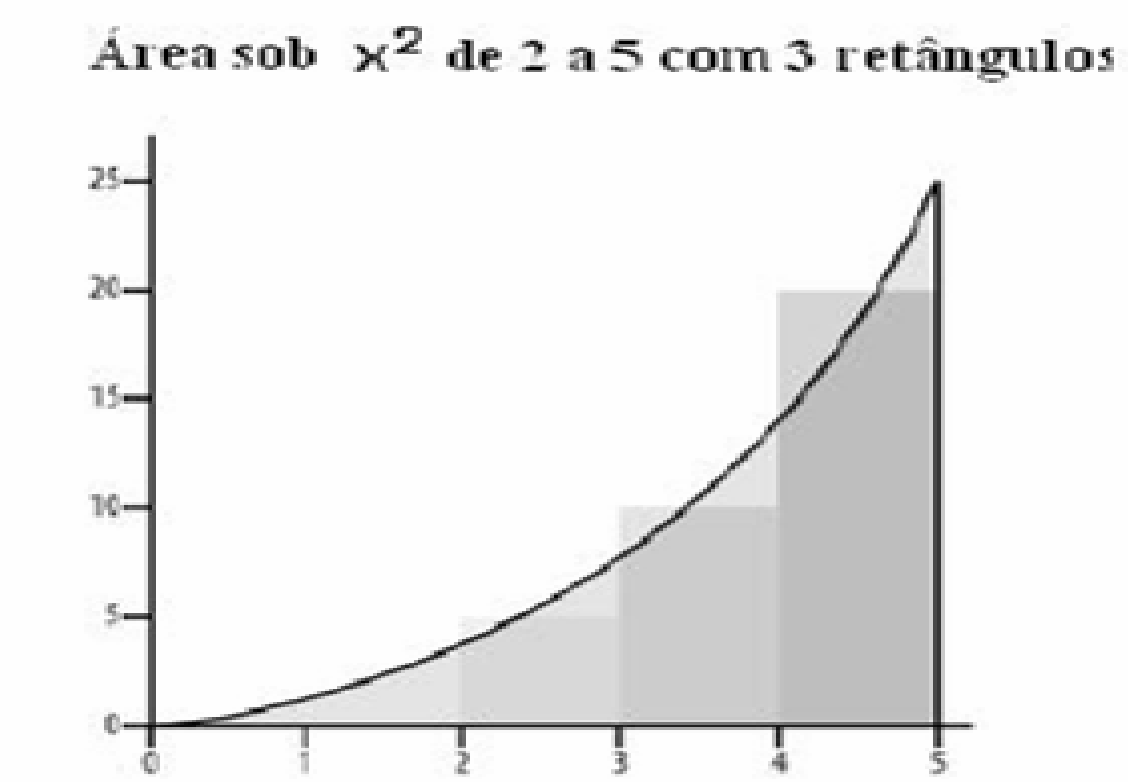
Importância da divisão

- Dividir a área sob a curva em vários retângulos, nos proporciona um resultado mais próximo da realidade. Quanto mais retângulo, maior será a aproximação.



Um problema prático

- Calcular a área sob a curva f(x) = x2



Programa em C

```
#include <stdio.h> /* Parâmetros */
#define f(x) ((x) * (x))
#define numberRects 50
#define lowerLimit 2.0
#define upperLimit 5.0
int main ( int argc, char * argv[ ] )
{
 int i;
double area, at, height, width; area = 0.0;
width = (upperLimit - lowerLimit) / numberRects;
for (i = 0; i < numberRects; i++) {
        at = lowerLimit + i * width + width / 2.0;
        height = f(at);
        area = area + width * height; }
        printf("A area entre %f e %f e: %f\n", lowerLimit, upperLimit, area );
        return 0;
}
```

Resultado

[Mario@prjIntel mario]$ **gcc rect.c -o rect**

[Mario@prjIntel mario]$ **./rect**

A area entre 2.000000 e 5.000000 e: 38.999100

Programa usando MPI

```
#include "mpi.h“
#include <stdio.h>
/* problem parameters */
#define f(x) ((x) * (x))
#define numberRects 50
#define lowerLimit 2.0
#define upperLimit 5.0
int main( int argc, char * argv[ ] ){
/*Variáveis MPI */
int dest, noProcesses, processId, src, tag; MPI_Status status;
/* Variáveis do problema */
int i;
double area, at, height, lower, width, total, range;
```

Programa usando MPI

(Parte2)

```
/* Inicializacção do MPI */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);
MPI_Comm_rank(MPI_COMM_WORLD, &processId);
/*Ajustando o tamanho para o subproblema*/
range = (upperLimit - lowerLimit) / noProcesses;
width = range / numberRects;
lower = lowerLimit + range * processId;
/*calculando área para o subproblema*/
area = 0.0;
for (i = 0; i < numberRects; i++) {
at = lower + i * width + width / 2.0;
height = f(at);
area = area + width * height;
}
```

Programa usando MPI

(Parte3)

```
/* coletando informações e imprimindo resultado */
tag = 0;
if (processId = = 0)
/* Se o rank é zero ele coleta os resultados */
{ total = area;
for (src=1; src < noProcesses; src++) {
MPI_Recv(&area, 1, MPI_DOUBLE, src, tag, MPI_COMM_WORLD,
&status);
total = total + area; }
fprintf(stderr, “A area entre %f e %f e: %f\n", lowerLimit, upperLimit,
total ); }
```

Programa usando MPI
(Parte4)

- else
- /* Todos os outros processos somente enviam os resultados */
- {
- dest = 0;
- MPI_Send(&area, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
- };
- /* Finalizando */
- MPI_Finalize( );
- return 0; }

Resultado

- [Mario@prjIntel mario]$ **mpicc mpi-rect.c -o mpi-rect**
- [Mario@prjIntel mario]$ **mpirun -np 5 mpi-rect**
- The area from 2.00000 to 5.000000 is: 38.999964

MPI_Send

- Rotina básica para envio de mensagens no MPI, utiliza o modo de comunicação "blocking send" (envio bloqueante), o que traz maior segurança na transmissão da mensagem. Após o retorno, libera o "system buffer" e permite o acesso ao "application buffer".

Exeptions:

- MPI_ERR_COMM
- MPI_ERR_COUNT
- MPI_ERR_RANK
- MPI_ERR_TYPE
- MPI_ERR_TAG

MPI_Send

Parâmetros

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

MPI_Recv

Parâmetros

- 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**

Broadcast

- É a comunicação coletiva em que um único processo envia (send) os mesmos dados para todos os processos com o mesmo communicator.

MPI_Bcast

- Parâmetros:
- 1º: Buffer que contém os dados
- 2º: Número de itens do buffer
- 3º: Tipo de dado
- 4º: rank do processo
- 5º: Comunicador

- EX: MPI_Bcast(&numberRects, 1, MPI_INT, 0, MPI_Comm_WORLD)

MPI_Reduce

- É a comunicação coletiva onde cada processo no comunicador contém um operador, e todos eles são combinados usando um operador binário que será aplicado sucessivamente.

MPI_Reduce

- **int MPI_Reduce(void* operand, void* result, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)**

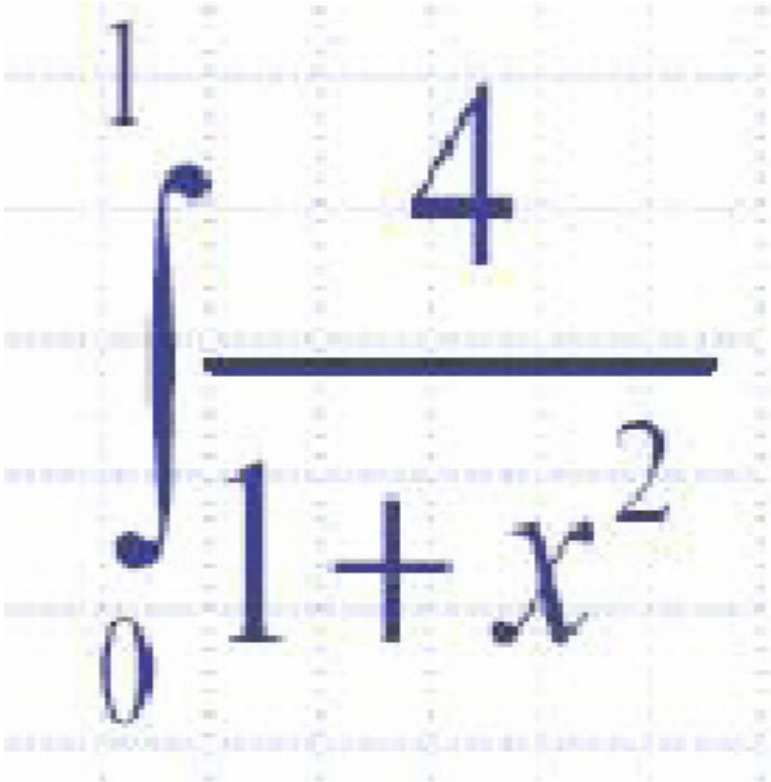
- **A operação MPI_Reduce combina os operandos armazenados em *operand usando a operação op e armazena o resultado em *result no processo root.**

MPI_Reduce

```
. . .
/* Adiciona as integrais calculadas por
cada processo */
MPI_Reduce(&integral, &total, 1,
MPI_FLOAT,
MPI_SUM, 0, MPI_COMM_WORLD);
/* Imprime o resultado */
```

Cálculo de Pi

- **O valor de pode ser obtido pela integração numérica**



Cáculo de PI

- **Calculada em paralelo, dividindo-se o intervalo de integração entre os processos.**

```
#include “mpi.h”
#include <math.h>
int main (argc, argv)
int argc;
char argv[ ];
{ int n, myid, numprocs, i, rc;
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);
```

Cálculo de Pi

(Parte2)

```
    if (myid == 0) {
            printf (“Entre com o número 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));
            }
            mpi = h* sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_WORLD_COMM);
            if (myid == 0)
                    printf (“valor aproximado de pi: %.16f \n”, pi);
    }
    MPI_Finalize( );
}
```

Implementações

- IBM MPI: Implementação IBM para SP e clusters
- **MPICH**: Argonne National Laboratory/Mississipi State University
- UNIFY: Mississipi State University
- CHIMP: Edinburgh Parallel Computing Center
- **LAM**: Ohio Supercomputer Center
- PMPIO: NASA
- MPIX: Mississippi State University NSF Engineering Research Center

- Páginas Sobre MPI
- Laboratório Nacional de Argonne : http://www-unix.mcs.anl.gov/mpi/
- **MPICH:** http://www-unix.mcs.anl.gov/mpi/mpich/
- **LAM/MPI:** **http://www.lam-mpi.org/**