Centro de Ciências Exatas e Tecnológicas Universidade Federal de Viçosa

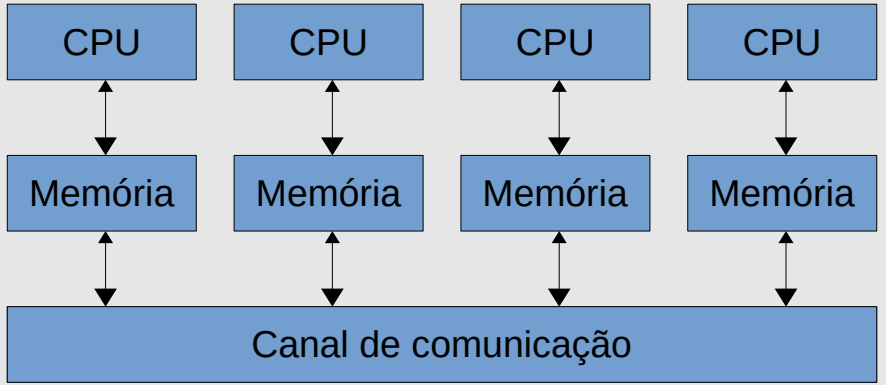
INF 310 – Programação Concorrente e Distribuída

Message Passing Interface (MPI)

Professor: Vitor Barbosa Souza
vitor.souza@ufv.br

Introdução

- Permite a programação paralela baseada na troca de mensagens
- Programa MPI é formado por um conjunto fixo de processos, criados no momento da inicialização
- Cada processo pode executar programas diferentes mas quase sempre o paralelismo é implementado como SPMD
- MPI define funções para:
  - Comunicação ponto-a-ponto
  - Operações coletivas
  - Grupos de processos
  - Contextos de comunicação
  - Ligação para programas C/C++ e Fortran
    - OpenMPI tem interface para uso em Java
  - Topologia de processos



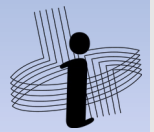
Introdução

- Biblioteca

```
mpi.h
```

- Funções básicas

```
MPI_Init(&argc, &argv)                          //inicializa uma execução MPI
MPI_Finalize()                                  //termina uma execução MPI
MPI_Comm_size(communicator, &size)              //obtém número de processos
MPI_Comm_Rank(communicator, &pid)               //obtém identificador do processo
MPI_Send(&buf,count,datatype,dest,tag,comm)       //envia msg(não-bloqueante)
MPI_Recv(&buf,count,datatype,source,tag,comm,status)//recebe msg(bloqueante)
```

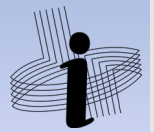


UFV

Introdução

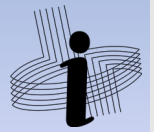
- Tipos de dados pré-definidos na biblioteca MPI

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long 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_BYTE	
MPI_PACKED	



Introdução

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
    int myrank, comm_sz, source, dest, tag=0;
    char message[100];                          //buffer da mensagem
    MPI_Status status;                          //status da recepção
    MPI_Init(&argc, &argv);                     //inicializa o MPI criando os processos
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);     //cada processo obtém “myrank”
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);    //número total de processos
    if (myrank != 0) {                          //se é um processo filho
        sprintf(message,“Hello do processo %d!”,my_rank); //cria mensagem
        dest = 0;                               //define processo pai como destinatário
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
                 tag, MPI_COMM_WORLD);
    } else                                      //se é o pai, recebe 1 msg de cada filho
        for (source = 1; source < comm_sz; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf(”%s \n”, message);
        }
    MPI_Finalize();                             //encerra MPI
    return 0;
}
```



Introdução

- Compilando

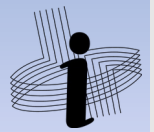
```
$ mpicc -g -Wall -o helloworld helloworld.cpp
```

- Executando

```
$ mpiexec -n <número_de_processos> ./helloworld
```

- Exemplo

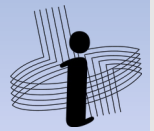
```
$ mpiexec -n 10 ./helloworld
Hello do processo 1!
Hello do processo 2!
Hello do processo 3!
Hello do processo 4!
Hello do processo 5!
Hello do processo 6!
Hello do processo 7!
Hello do processo 8!
Hello do processo 9!
```



UFV

Comunicação entre processos

- *Communicator* é um grupo de processos que podem comunicar entre si
  - *MPI_COMM_WORLD* é o *communicator* que engloba todos os processos iniciados pelo *MPI_Init*
  - O MPI fornece funções para criação de *communicators*
- *Tag* é um inteiro não negativo utilizado para definir um contexto
  - Exemplo: valores que devem ser impressos / valores que devem ser armazenados



UFV

Comunicação entre processos

- MPI_Send
  - Parâmetros destino, tag e communicator definem conjuntamente o receptor da msg
  - Exemplo
    - processo q

```
MPI_Send(send_buf, sbuf_size, send_type, dst, send_tag, send_comm);
```

    - processo r

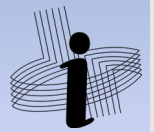
```
MPI_Recv(recv_buf, rbuf_size, recv_type, src, recv_tag, recv_comm,&status);
```

    - condição para entrega da mensagem

```
recv_comm==send_comm && recv_tag==send_tag && dst==r && src==q
```

    - possível erro

```
send_type != recv_type
rbuf_size < sbuf_size
```



UFV

Comunicação entre processos

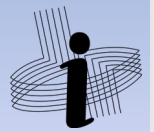
- MPI_Recv
  - Não-determinismo pode ser implementado através de constantes

```
MPI_ANY_SOURCE
MPI_ANY_TAG
```

    - Parâmetro `status` pode ser utilizado para obter emissor, tag e quantidade de dados na mensagem

```
MPI_Status status;
...
status.MPI_SOURCE
status.MPI_TAG
MPI_Get_count(&status,recv_type,&count)
```

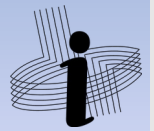
    - Quando o status não é necessário, a constante MPI_STATUS_IGNORE pode ser passada
  - O *communicator* deve ser sempre especificado por ambos emissor e receptor



UFV

Comunicação entre processos

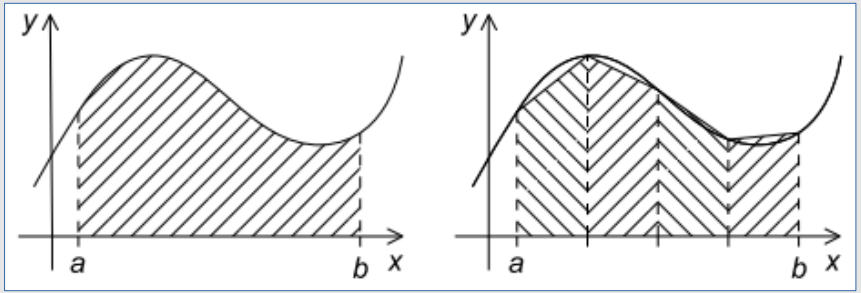
- Mensagens trocadas entre cada par de processos são sempre entregues em ordem
- Não há garantia de ordem de recebimento entre mensagens vindas de 2 ou mais processos distintos



UFV

Regra do trapézio

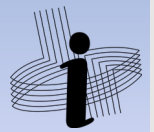
```
/* obtenção dos valores de a, b e n foi omitida (por enquanto) */
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
float h = (b-a)/n;
int local_n = n/comm_sz;
float local_a = a + my_rank * local_n * h;
float local_b = local_a + local_n * h;
float local_int = regra_trapezio(local_a, local_b, local_n, h);
if (my_rank != 0) {
    MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
} else {
    total_int = local_int;
    for (int p = 1; p < comm_sz; p++) {
        MPI_Recv(&local_int, 1, MPI_DOUBLE, p, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        total_int += local_int;
    }
}
if (my_rank == 0)
    printf("%f",total_int);
MPI_Finalize();
```



Regra do trapézio

- Obtendo valores de *a*, *b* e *n*
  - Muitas implementações permitem que apenas o processo 0 acesse o *stdin*

```
void Get_input(int my_rank,int comm_sz,double *a,double *b,int *n) {
    int dest;
    if (my_rank == 0) {
        printf("Digite valores de a, b e n\n");
        scanf("%lf %lf %d", a, b, n);
        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(a, 1, MPI_DOUBLE,0,0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(b, 1, MPI_DOUBLE,0,0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(n, 1, MPI_INT, 0,0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```



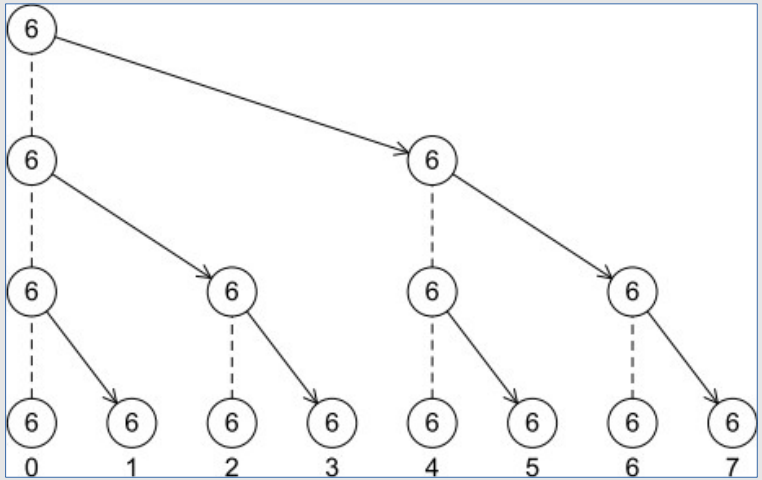
Broadcast

- Forma mais eficiente de difundir para todos os processos os dados obtidos por um processo específico

```
MPI_Bcast(&in_out_data,count,datatype,source,comm);
```

  - O parâmetro *in_out_data* e do tipo entrada/saída
  - Uso de *broadcast* para distribuir valores *a, b* e *n* na função Get_input(...)

```
MPI_Bcast(a,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(b,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Bcast(n,1,MPI_INT,0,MPI_COMM_WORLD);
```



Redução

- MPI_Reduce
  - Evitar que o processo 0 faça todo o cálculo final

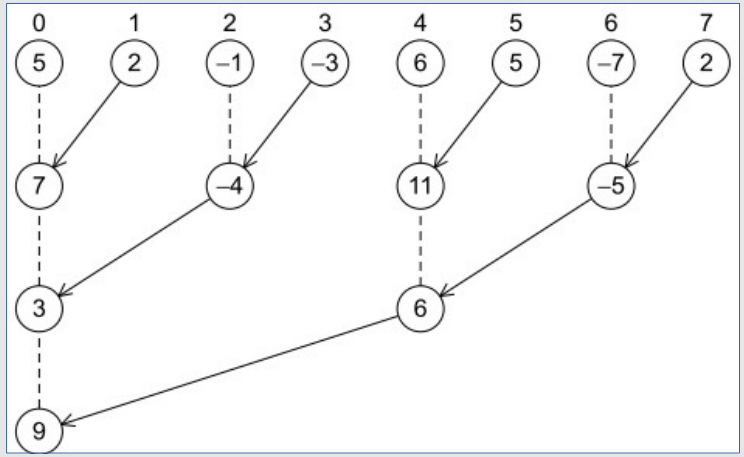
```
MPI_Reduce(&inData,&outData,count,dataType,operator,dest,comm)
```

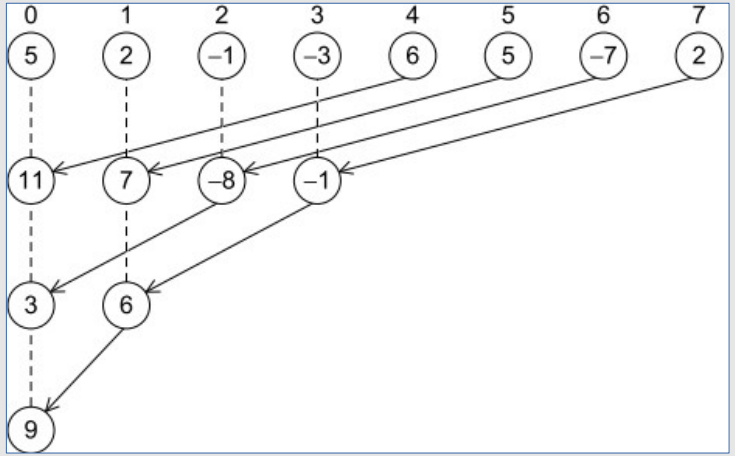
  - O código respectivo no exemplo da regra do trapézio pode ser substituído por

```
MPI_Reduce(&local_int,&total_int,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

  - Atenção: o código abaixo tem resultados não previsíveis

```
MPI Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```



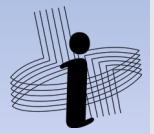


Redução

- MPI_Reduce
  - Várias operações são pré-definidas para a operação de redução

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

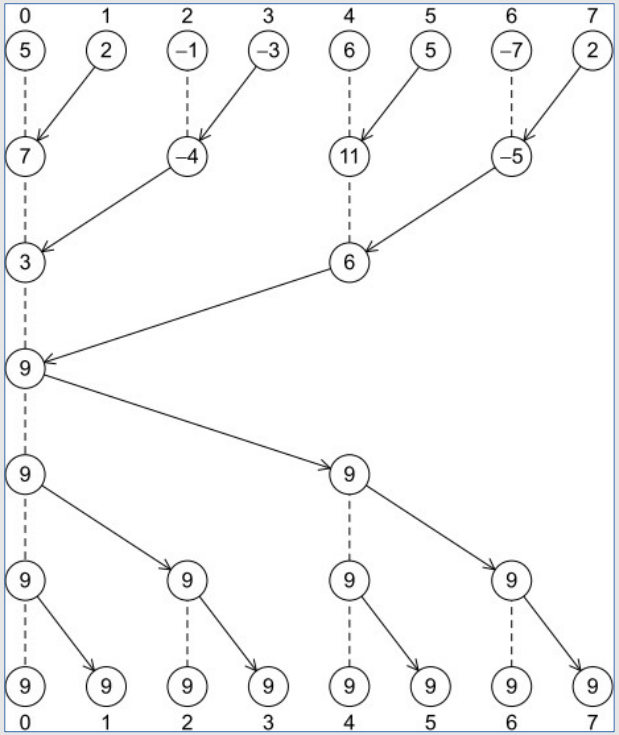
  - O MPI permite que o usuário defina suas próprias operações

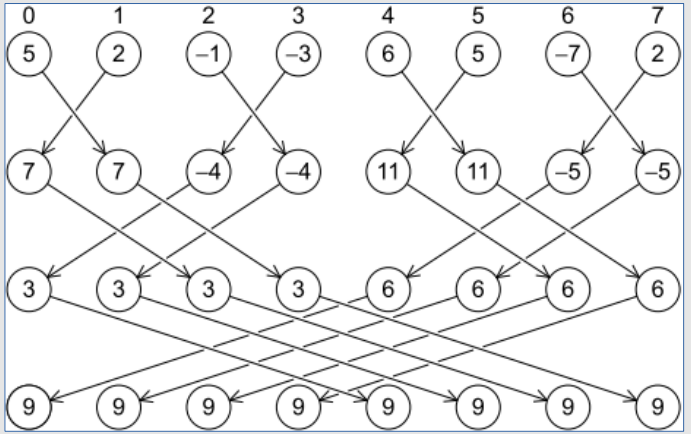


Redução

- MPI_Allreduce
  - O MPI fornece uma variante do MPI_Reduce que armazena o resultado em todos os processos do *communicator*

```
MPI_Allreduce(&inData,&outData,count,dataType,operator,comm)
```





Blocos de dados

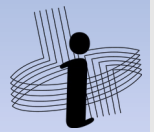
- Alguns problemas exigem o processamento de um conjunto grande de dados
  - Ex: Soma de 2 vetores

```
void vectorSum(int *a, int *b, int *r, int n) {

    for(int i=0; i<n; ++i)
        r[i] = a[i] + b[i];
}
```

- Copiar os dados para cada processo é ineficiente

	Components											
									Block-Cyclic			
Process	Block				Cyclic				Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11



UFV

Blocos de dados

- MPI_Scatter
  - Distribui os dados a serem processados em blocos iguais entre todos os processos

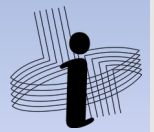
```
MPI_Scatter(&sendBuf,sendCount,sendType,
            &recvBuf,recvCount,recvType,
            source,comm);
```

  - *sendCount* é o tamanho das partições, e não do vetor original
  - Lendo e distribuindo um vetor de tamanho *n* para todos os processos

```
double *a = NULL;
if (my_rank == 0) {
    a = malloc(n*sizeof(double));
    for (int i = 0; i < n; i++)
        scanf("%lf", &a[i]);
}
MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,comm);

if (my_rank == 0)
    free(a);
```

  - Se existem *p* processos, cada processo recebe um vetor de tamanho *n/p*



Blocos de dados

- MPI_Gather
  - Une as partições contendo resultados em um único processo

```
MPI_Gather(&sendBuf,sendCount,sendType,
           &recvBuf,recvCount,recvType,
           destination,comm);
```

  - *recvCount* é o tamanho das partições recebidas, e não o tamanho total dos dados
  - Obtendo e imprimindo um vetor distribuído de tamanho *n*

```
double *b = NULL; int i;
if (my_rank == 0)
    b = malloc(n*sizeof(double));
MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0,comm);
if (my_rank == 0) {
    for (i = 0; i < n; i++)
        print("%f", &b[i]);
    free(b);
}
```

  - MPI_Scatter e MPI_Gather funcionam bem apenas se o tamanho dos blocos são iguais
    - MPI_Scatterv e MPI_Gatherv podem ser utilizados, caso contrário