Programação Paralela com MPI

ELC139 - Programação Paralela

João Vicente Ferreira Lima (UFSM)

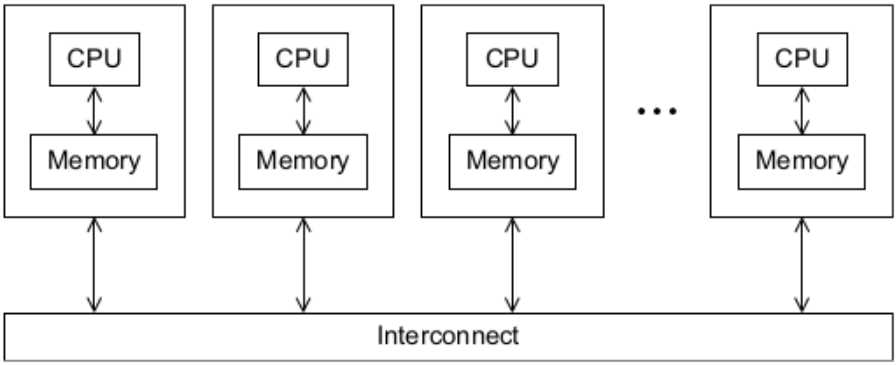
Universidade Federal de Santa Maria
jvlima@inf.ufsm.br
http://www.inf.ufsm.br/~jvlima

2023/1

Outline

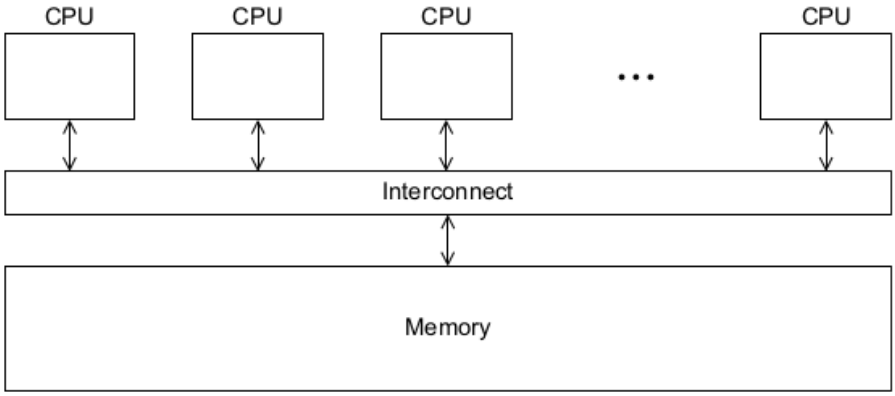
Outline

Sistema de memória distribuída



Introduction to Parallel Computing, Grama et al, 2003.

Sistema de memória compartilhada



Introduction to Parallel Computing, Grama et al, 2003.

Outline

Olá mundo

```c
#include <stdio.h>

int main(void)
{
  printf("hello, world\n");

  return 0;
}
```

informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
1960

Olá mundo

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

int main(void)
{
  /* Nenhuma chamada MPI pode acontecer antes */
  MPI_Init(NULL, NULL);

  /* Chamadas MPI */

  MPI_Finalize();
  /* Nenhuma chamada MPI pode acontecer depois */
  return 0;
}
```

informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
1960

Compilação

```
$ mpicc -g -Wall -o mpi_hello mpi_hello.c
```

- `mpicc` é um *wrapper* (script)
  - `mpicc --showme` – Mostra o comando real executado.
- As outras opções e flags são passadas diretamento ao compilador

Execução

```
$ mpiexec -n <n. de processos> <executável>
```

```
$ mpiexec -n 1 ./mpi_hello

$ mpiexec -n 4 ./mpi_hello
```

- `mpiexec` também é um *wrapper* (script) para executar o programa
- Uma vasta lista de opções disponíveis
  - Processadores e cores, threads por processo, variáveis de ambientes, etc.

MPI

MPI_Init

- Inicia o MPI, fazendo o setup inicial.

```
1 int MPI_Init(
2         int* argc,     /* in/out */
3         char*** argv   /* in/out */
4 );
```

- `argc` e `argv` são ponteiros recebidos pela `main`.

MPI_Finalize

- Sinalize que o programa terminou e libera recursos

```
1 int MPI_Finalize(void);
```

Olá mundo

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

int main(void)
{
  int rank, comm_size;
  MPI_Init(NULL, NULL);

  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf ("Hello from process %d of %d!\n", rank, comm_size);

  MPI_Finalize();
  return 0;
}
```

informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
1960

Executando

```
$ mpiexec -n 1 ./hello_mpi
Hello from process 0 of 1!
$ mpiexec -n 2 ./hello_mpi
Hello from process 0 of 2!
Hello from process 1 of 2!
$ mpiexec -n 6 ./hello_mpi
There are not enough slots available in the system to satisfy
the 6 slots that were requested by the application:

  ./hello_mpi

Either request fewer slots for your application, or make more
slots available for use.
```

Comunicadores

- Um comunicador (*communicator*) é um conjunto de processos que podem mandar mensagens entre si
- A função `MPI_Init` é definir um comunicador que consiste em todos os processos criados quando o programa é executado
- O comunicador é o `MPI_COMM_WORLD` do tipo `MPI_Comm`
- Outro comunidor especial é o `MPI_COMM_SELF`

Comunicadores

MPI_Comm_size

- Consulta o número de processos no comunidador

```
1 int MPI_Comm_size(
2         MPI_Comm comm,   /* in */
3         int* size        /* out */
4 );
```

MPI_Comm_rank

- Retorna o rank do processo que chamou a função

```
1 int MPI_Comm_rank(
2           MPI_Comm comm,   /* in */
3           int* rank        /* out */
4   );
```

Olá mundo

```
1  int main (int argc, char *argv[])
2  {
3    int   rank, comm_size, len;
4    char hostname[MPI_MAX_PROCESSOR_NAME];
5    char msg[MPI_MAX_PROCESSOR_NAME+30];
6
7    MPI_Init(NULL, NULL);
8    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10   MPI_Get_processor_name(hostname, &len);
11   ...
```

Olá mundo

```
1     ...
2     if(rank != 0){
3       sprintf(msg, "Hello from %s rank %d", hostname, rank);
4       MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 0, 0,
          ↪  MPI_COMM_WORLD);
5     } else {
6       for(int p = 1; p < comm_size; p++){
7         MPI_Recv(msg, MPI_MAX_PROCESSOR_NAME+30, MPI_CHAR, p, 0,
            ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
8         printf("%s\n", msg);
9       }
10    }
11    MPI_Finalize();
12    return 0;
13  }
```

Comunicadores

MPI_Send

```
1  int MPI_Send(
2      const void   *buf,      /* in */
3      int          count,     /* in */
4      MPI_Datatype datatype,  /* in */
5      int          dest,      /* in */
6      int          tag,       /* in */
7      MPI_Comm     comm       /* in */
8  );
```

Comunicadores

Table 3.1 Some Predefined MPI Datatypes

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	

Introduction to Parallel Computing, Grama et al, 2003.

Comunicadores

MPI_Recv

```
1  int MPI_Recv(
2      void          *buf,      /* out */
3      int           count,     /* in */
4      MPI_Datatype  datatype,  /* in */
5      int           source,    /* in */
6      int           tag,       /* in */
7      MPI_Comm      comm,      /* in */
8      MPI_Status    *status    /* out */
9  );
```

Comunicação

- Vários parâmetros precisam casar (*match*) para uma mensagem ser entregue
- Assumindo que `p1` envia uma mensagem para `p2`
  - `comm1 == comm2`
  - `tag1 == tag2`
  - `dest == p2`
  - `source == p1`

```
1 int MPI_Send(const void *send_buf, int send_size, MPI_Datatype
   ↪ send_type, int dest, int tag1, MPI_Comm comm1);
2
3 int MPI_Recv(void *rec_buf, int rec_size, MPI_Datatype
   ↪ rec_type, int source, int tag2, MPI_Comm comm2, MPI_Status
   ↪ *status);
```

Comunicação

- As condições anteriores não garantem o sucesso do envio da mensagem.
- Os pares adicionais são:
  - `send_type == rec_type`
  - `rec_size >= send_size`

```
1 int MPI_Send(const void *send_buf, int send_size, MPI_Datatype
   ↪ send_type, int dest, int tag1, MPI_Comm comm1);
2
3 int MPI_Recv(void *rec_buf, int rec_size, MPI_Datatype
   ↪ rec_type, int source, int tag2, MPI_Comm comm2, MPI_Status
   ↪ *status);
```

Comunicação

- O `MPI_Recv` recebe também a variável do tipo `MPI_Status` contendo
  - O tamanho da mensagem
  - A origem da mensagem `MPI_SOURCE`
  - A tag da mensagem `MPI_TAG`
  - Erro se acontecer `MPI_ERROR`
- Os parâmetros `MPI_ANY_SOURCE` e `MPI_ANY_TAG` permitem aceitar mensagens de qualquer fonte com qualquer tag

Comunicação

```
1 MPI_Status status;
2 MPI_Recv(rec_buf, rec_size, rec_type, MPI_ANY_SOURCE,
   ↪  MPI_ANY_TAG, rec_comm, &status);
3
4 int src = status.MPI_SOURCE;
5 int tag = status.MPI_TAG;
6 int count;
7 MPI_Get_count(&status, rec_type, &count);
```

https://joao-ufsm.github.io/par2023a/