Programação Paralela e Distribuída

MPI – Message Passing Interface

Prof. Maglan Cristiano Diemer
Univates - Centro Universitário

O que é MPI?

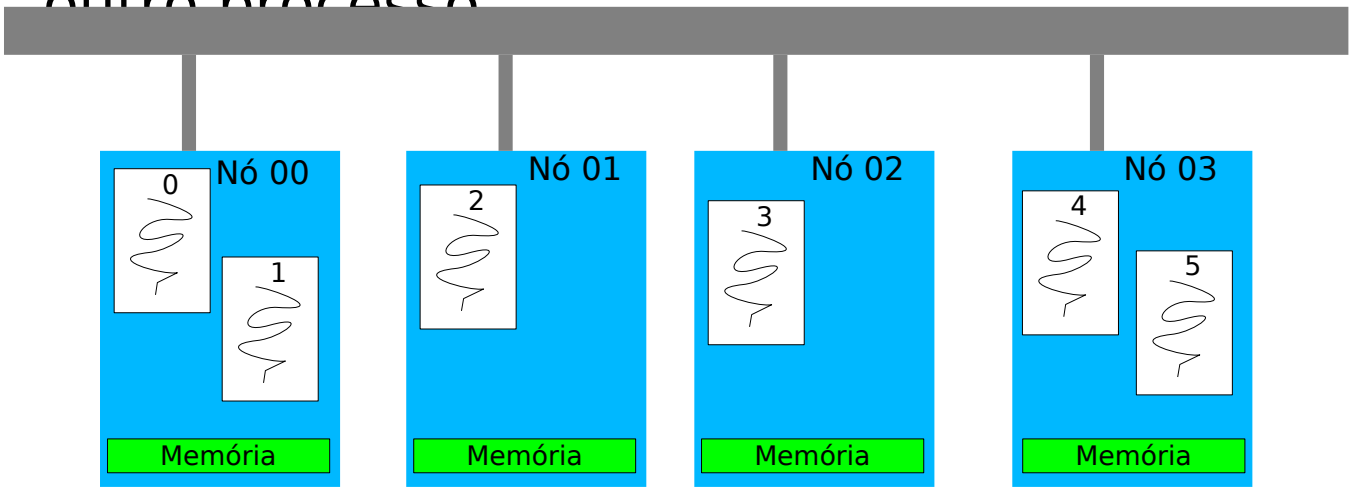
- MPI (*Message Passing Interface*) é uma biblioteca de comunicação que permite a programação paralela baseada em troca de mensagens
- Foi definida pela MPI Fórum (www.mpi-forum.org), com a participação de Universidades, empresas e laboratórios de pesquisa
- Um dos objetivos do MPI é oferecer possibilidade de uma implementação eficiente da comunicação:
  - Evitando cópias de memória para memória;
  - Permitindo superposição de comunicação e computação.
- Permitir implementações em ambientes heterogêneos.
- Supõe que a interface de comunicação é confiável:
  - Falhas de comunicação devem ser tratadas pelo subsistema de comunicação da plataforma.

O que é MPI ?

- A solução proposta por MPI para a criação remota de processos é a construção de uma *máquina virtual*, composta de *nós virtuais,* sobre uma arquitetura real
- Cada nó virtual consiste em um processo, responsável por executar a aplicação.
- Segundo o padrão MPI, a configuração da máquina virtual, número de nós, permanece inalterada durante toda a execução da aplicação.
- A princípio, cada nó real suporta a execução de um nó virtual, porém nós reais são autorizados a suportar a execução de mais de um nó virtual
- Esta característica é útil pois permite codificar a aplicação sem que o número de nós da máquina real seja conhecido

Programação SPMD

- É necessário conhecer o modelo de programação empregado pelo MPI: o modelo SPMD – *Single Program, Multiple Data*
- Neste modelo, a aplicação define um conjunto de processos que deverão executar o mesmo código de forma concorrente.
  - Note que, devido ao fato de diferentes processo manipularem diferentes conjuntos de dados, a porção de código sendo executada por um processo não é necessariamente a mesma que a executada por um outro processo



Programação SPMD

```
main() {
   int id;
   id = fork();
   if (id !=0 ) {
      // código do pai
   } else {
      // código filho
   }
}
```

- A chamada de uma primitiva `fork` por um processo força a duplicação em dois processo independentes: u processo *pai* e um processo *filho*.
- Apesar de cada processo conter um cópia completa do código da aplicação, instruções de controle de fluxo, como o `if` no exemplo, decidem qual o trecho a ser executado em cada um
- Na programação SPMD, não apenas dois, mas um grupo de *n* processos executam um mesmo programa.
- De forma semelhante ao *id* do `fork`, cada processo tem acesso a identificação de sua posição no grupo, ou seja, saber que ele é o *i*-ésimo nó de uma máquina virtual de *n* nós; em função da posição de seu nó, o processo pode selecionar a porção do código a ser executado

Programação SPMD

- Durante a execução dos processo não existe nenhuma forma de sincronização implícita entre os processos
  - A introdução de pontos de sincronização entre os processos é de responsabilidade da aplicação
  - O programador deve, explicitamente, utilizar mecanismos de troca de mensagens para possibilitar a cooperação entre as tarefas
- Outra diferença fundamental entre o *fork* e a programação SPMD é que, no momento da execução do *fork* o processo filho consiste em uma cópia do processo original, implicando que a área de dados seja igualmente duplicada, enquanto processos SPMD consistem em instâncias totalmente autônomas desde o momento em que a execução é iniciada
- Os processos na programação SPMD são iniciados todos a partir do mesmo ponto, o início do programa, cada um responsável por inicializar sua própria área de dados

OpenMPI: : Open Source High Performance Computing

- O projeto OpenMPI (www.open-mpi.org) integra várias tecnologias e recursos de outros projetos (FT-MPI, LA-MPI, LAM/MPI, PACX-MPI) para criar e disponibilizar a melhor biblioteca MPI
- Totalmente compatível com a especificação MPI-2
- Versão 1.0.1 foi disponibilizada em Dezembro de 2005

Como executar tarefas paralelas

- O OpenMPI oferece o binário `orterun` para criar tarefas MPI
  Por questões de compatibilidade, há links simbólicos `mpirun` e `mpiexec` para o `orterun`
- Algumas opções do `mpirun/orterun`:
  - `-np` *`numerotarefas`*
    Define o número de tarefas paralelas
  - `-host` *`host1,host2, ...`*
    Define uma lista de hosts do cluster onde as tarefas devem ser executadas
  - `-hostfile` *`nomearquivo`*
    Define um arquivo com a lista de hosts do cluster onde as tarefas devem ser executadas
  - `-app` *`nomearquivo`*
    Define um arquivo com os parâmetros que devem ser executados pelo `mpirun`

```
# mpirun -np 4 programa
# mpirun -np 4 -host a,b programa
```

Como configurar o SSH para não pedir senha

- Gerar um par de chaves DSA com o comando `ssh-keygen`:

  ```
  # ssh-keygen -t dsa
  ```

  Aceitar o valor default onde deve ser armazenado a chave `[/.ssh/id_dsa]` e digitar uma passphrase para o par de chaves

- Copiar o arquivo `~/.ssh/id_dsa.pub` gerado pelo `ssh-keygen`

  ```
  # ssh-copy-id -i ~/.ssh/id_dsa.pub <ip destino>
  ```

Escrevendo programas MPI

- Todo programa em MPI deve realizar include no header `mpi.h`
  `#include “mpi.h”`
  - Este arquivo, `mpi.h`, contém as definições, macros e funções de protótipos de funções necessários para a compilação de um programa MPI.
- Antes de qualquer outra função MPI ser chamada, a função `MPI_Init` deve ser chamada pelo menos uma vez.
  - Seus argumentos são os ponteiros para os parâmetros do programa principal, `argc` e `argv`.
  - Esta função permite que o sistema realize as operações de preparação necessárias para que a biblioteca MPI seja utilizada.
- Ao término do programa a função `MPI_Finalize` deve ser chamada.
  - Esta função limpa qualquer pendência deixada pelo MPI, p. ex, recepções pendentes que nunca foram completadas.

Escrevendo programas MPI

```
...
#include “mpi.h”
...
main(int argc, char** argv) {

   ...
   /* Nenhuma função MPI pode ser chamada antes deste ponto */

   MPI_Init(&argc, &argv);
   ...
   /* Código principal do programa MPI */
   ...
   MPI_Finalize();

   /* Nenhuma função MPI pode ser chamada depois deste ponto*/
   ...
}
```

Primeiro exemplo MPI

```
#include <stdio.h>
#include <string.h>
#include “mpi.h”

main(int argc, char** argv)
{
   int rank, nroRanks;
   char hostName[MPI_MAX_PROCESSOR_NAME];
   int tamanhoHostName;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &nroRanks);
   MPI_Get_processor_name(hostName, tamanhoHostName);

   printf(“Meu rank %d de %d executando no host %s”,
       rank, nroRanks, hostName);

   MPI_Finalize();
}
```

Trocando mensagens

```
char msg[100];
MPI_Status status;

...

if (rank != 0) {
   sprintf(msg, “Oi! Eu sou o processo %d do host %s”,
       rank, hostname);
   MPI_Send(msg,strlen(msg)+1,MPI_CHAR,0,0,MPI_COMM_WORLD);
} else {
   for (i = 1; i < nroRanks; i++) {
      MPI_Recv(msg, 100, MPI_CHAR, i, 0, MPI_COMM_WORLD, &status);
      printf(“%s\n”,msg);
   }
}
...
```

MPI_Comm_Size e MPI_Comm_rank

- `MPI_Comm_rank(MPI_Comm comm, int *rank)`

  Obter a classificação do processo (o número do nó virtual)

- `MPI_Comm_size(MPI_Comm comm, int *size)`

  Obter o número de processos (de nós virtuais)

- **`MPI_Comm`**: Identifica o grupo de processos que está participando de uma operação de comunicação
  - O comunicador mais usado é `MPI_COMM_WORLD`, que é um comunicador definido pela MPI para denotar todos os processo que estão executando um programa MPI.
  - MPI permite que se defina comunicadores adicionais para subconjuntos de processos. Isso dá a opção de atribuir subconjuntos dos processos para executar tarefas especializadas dentro de um programa paralelo

MPI_Send e MPI_Recv

```
int MPI_Send( void *buff, int cont, MPI_Datatype tipo,
   int dest, int tag, MPI_Comm grupo )
```

```
int MPI_Recv( void *buff, int cont, MPI_Datatype tipo,
   int origem, int tag, MPI_Comm grupo, MPI_Status *status )
```

- **buff**: corresponde à área de dados a ser transmitida ou onde os dados recebidos devem ser armazenados.
- **tipo**: descreve o tipo do dados que a mensagem contém. Além dos tipos primitivos, outros podem ser introduzidos pelo usuário.
- **cont**: o número de elementos a serem enviados ou recebidos, sendo cada elemento do tipo *tipo*. Em outras palavras, *buff* é um array contendo *cont* elementos do tipo *tipo*
- **origem** e **dest** identificam, respectivamente o nó origem e destino da mensagem.
- **tag**: possibilita a classificação de mensagens através de um rótulo: somente uma mensagem que possua o *tag* informado será recebida.
- **grupo**: informa o grupo a que pertencem os nós *origem* e *destino*. tipicamente `MPI_COMM_WORLD` para todos os nós da máquina virtual
- **status**: permite que o receptor tenha acesso a uma séria de informações a respeito da mensagem recebida (por exemplo, o tamanho em bytes)

MPI_Send e MPI_Recv

- Principais dados MPI predefinidos

```
MPI_CHAR      char
MPI_INT       int
MPI_LONG      long
MPI_FLOAT     float
MPI_DOUBLE    double
MPI_PACKED    tipo a ser informado
```

- O par de primitivas MPI_Send e MPI_Recv permite realizar comunicações síncronas entre processos. Isto quer dizer que um processo ao invocar uma primitiva MPI_Send fica bloqueado até que a comunicação seja concluída. De forma semelhante, ao invocar MPI_Recv, o processo fica bloqueado até que a mensagem desejada esteja presente na fila de mensagens

MPI_Isend e MPI_Irecv

- Caso o sincronismo não seja desejado, é possível optar por primitivas de comunicação assíncronas (não bloqueantes), tipo `MPI_Isend` e `MPI_Irecv`, cujos parâmetros são os mesmos das sua homólogas síncronas
- No envio assíncrono, a mensagem é postada na rede o processo é desbloqueado, podendo continuar suas operações
- Na recepção assíncrona, caso a mensagem já tenha sido recebida, ela é lida para o *buffer* de recepção e o processo pode continuar suas operações; caso a mensagem ainda não tenha sido recebida, o processo é também liberado para continuar sua execução, devendo em um momento mais tarde tentar uma nova recepção

Comunicação de grupo

- Outra possibilidade de comunicação em MPI é a comunicação de grupo, onde um processo pode enviar, ou receber, mensagens a, ou de, todos outros processo através de mecanismos de **Broadcast** (`MPI_Bcast`) e **Redução** (`MPI_Reduce`)
- Estas rotinas não aceitam tags para filtrar mensagens
- A comunicação envolve necessariamente todo o grupo de processos envolvidos (`MPI_Comm == MPI_COMM_WORLD`)
- No entanto, um destes processos é assumido como raiz da comunicação
  - quando da ocorrência de um broadcast o processo raiz é responsável pelo envio da mensagem
  - no caso de uma redução, pelo recebimento e tratamento das mensagens

MPI_Brodcast

- Existe apenas uma primitiva para realização do broadcast e esta deve ser realizada por todos os processo participantes, tanto pelo processo raiz, que envia a mensagem, como pelos processo que receberão

```
int MPI_Bcast( void *buf, int cont, MPI_Datatype tipo,
    int raiz, MPI_Comm grupo)
```

- Ao utilizar `MPI_Bcast`, a semântica associada ao parâmetro **`buff`** é obtida pelo valor fornecido ao parâmetro **`raiz`**.
  - Caso o valor informado para **`raiz`** correponda a própria posição do processo no grupo (seu rank), **`buff`** contém os dados a serem enviados: este processo é considerado a raiz da comunicação
  - Nos demais processos, **`buff`** corresponde a área de dados onde deve ser armazenado os dados recebidos

MPI_Reduce

- É um mecanismo inverso ao broadcast, em que vários nós enviam mensagens (uma mensagem por nó) a um nó raiz.

```
int MPI_Reduce(void *operando, void *resultado, int cont,
   MPI_Datatype tipo, MPI_Op operador, int raiz, MPI_Comm
grupo);
```

- É necessário especificar a *ação de redução* a ser tomada quando os dados forem recebidos pelo nó raiz.
- A operação (**operador**), executada pelo nó raiz, é realizada sobre o valor recebido em **operando** e em **resultado**, armazenando o resultado em **resultado**.
- Em outro nó que não seja raiz esta operação não é realizada, sendo o dado armazenado em **operando** transmitido na mensagem

MPI_Reduce

- Operações de redução predefinidas em MPI

```
MPI_MAX      o maior valor entre operando e resultado
MPI_MIN      o menor valor entre operando e resultado
MPI_SUM      a soma de operando e resultado
MPI_PROD     o produto de operando e resultado
MPI_LAND     o E lógico entre operando e resultado
MPI_BAND     o E byte a byte entre operando e resultado
MPI_LOR      o OU lógico entre operando e resultado
MPI_BOR      o OU byte a byte entre operando e resultado
MPI_LXOR     o XOR lógico entre operando e resultado
MPI_BXOR     o XOR byte a byte entre operando e resultado
```