High Performance Programming: Multicore, Clusters and GPU

Message Passing Interface - MPI

Professor Marcelo Trindade Rebonatto
Curso de Ciência da Computação
UPF

Sumário

- Introdução
- Padrão MPI
- MPI Básico
- Uso de Coringas

Contexto Histórico

- Final da Década do 1980
  - Problemas com interoperabilidade de programas paralelos
- Abril de 1992
  - "Workshop" de padrões de "Message Passing"
  - Memória distribuída
  - Centro de Pesquisa em Computação Paralela, Williamsburg, Virginia
  - Criado um grupo de trabalho para dar continuidade ao processo de padronização

Messaging Passing Interface

- Novembro de 1992 (MPI-1)
  - MPI Forum
  - ± 175 pessoas e 40 organizações
  - Empresas, especialistas, grupos de desenvolvedores
  - Definiu uma especificação (lógica), modelo de trocas de mensagens
- Maio de 1994 (MPI-2)
  - Disponibilização da versão padrão do MPI, atualmente, o padrão MPI-2

Messaging Passing Interface

- Padrão
  - Programação de Alto Desempenho
  - Trocas de mensagens
  - Memória não compartilhada
- De direito
  - Aceito pela comunidade acadêmica e industria
- De fato
  - Realmente usado

MPI: razões de sucesso

- + de uma implementação de boa qualidade gratuita
- Grupos de comunicação sólidos, eficientes e determinísticos
- Gerenciamento eficiente dos buffers de mensagens
- Utilização eficiente em variadas arquiteturas - Portabilidade
- Especificação forte

MPI: Implementações

- Famílias de computadores
- Máquinas específicas
- Implementações de Universidades
  - MPICH - Chamaleon
  - LAM – Local Area Multicomputer
  - OpenMPI – Open Source High Performance Computing
- Implementações comerciais

MPICH

- Especificação estática de máquina virtual
- Base para várias implementações
  - Redes: MPI-SCI, MPI-BIP
  - SO Windows: WinMPICH, Nt-mpich
  - Famílias: SX-3, SX-4, SX-5, SX-6
  - Máquinas específicas: MPI/ES

LAM-MPI OpenMPI

- LAM-MPI
  - Oriunda de sistemas distribuídos
  - Grande popularização
  - Maior flexibilidade na manipulação da máquina virtual
  - Desempenho semelhante ao MPICH
  - Descontinuada, substituída pela OpenMPI
- OpenMPI: Versão a ser utilizada na disciplina
  - **sudo apt-get install openmpi-bin libopenmpi-dev**
  - Distribuições Ubuntu

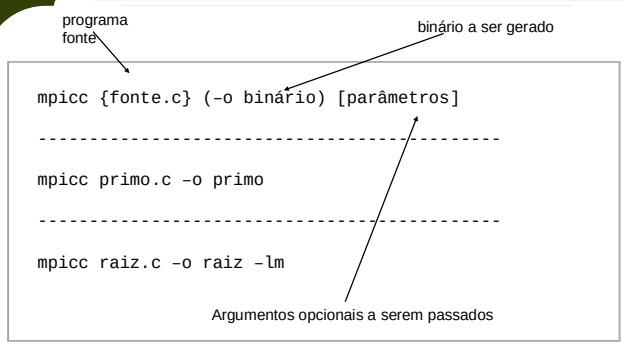
Funções do MPI

- Rotinas para
  - Gerenciamento de processos
  - Comunicação ponto a ponto
  - Comunicação em grupo
  - Comunicação assíncrona
  - Fins diversos

MPI: compilação de programas

- Implementações disponibilizam *script*
- *mpicc*
  - Compila e linkedita aplicação
  - Argumentos semelhantes ao gcc
    - Todos os parâmetros são aceitos
    - + parâmetros próprios
  - Pode ser usado em *Makefiles*

mpicc



MPI: máquina virtual

- Abstração de *Single System Image* (SSI) para execução de aplicações paralelas
- Transforma a visão de máquinas físicas isoladas em máquina paralela
  - Usa trocas de mensagens
  - Número de CPUs não limitado
  - Definido estaticamente

Máquina virtual: arquivo de hosts

- Especifica a localização das máquinas físicas que irão compor a máquina virtual
  - IP
    - ou
  - Hostname
- Hospedeira:
  - Máquina onde o usuário está fisicamente conectado
  - Presente, a primeira da lista

Exemplo de arquivo de hosts

```
192.168.9.1
192.168.9.5
192.168.9.4
192.168.9.10
192.168.9.7
192.168.9.14
```


Arquivo de hosts: slots

- Dependendo da forma de instalação do MPI, a execução pode ficar limitada a um número de processos
  - Em geral, o número de CPUs (núcleos do processador)
- Perceptível quando ocorre mensagem
  - "There are not enough slots available ..."
- Solução:
  - Executar passando arquivo de hosts
    - Mesmo com apenas um computador disponível
  - Acrescentar argumento '*slots*' ao lado de cada *host*

Arquivo de hosts: slots

- slots=n
  - n é o número de *slots*
  - Máximo de processos na máquina virtual

- Exemplo com vários computadores

```
192.168.9.1 slots=4
192.168.9.5 slots=4
192.168.9.4 slots=4
```

- Vai executar até 12 (4 x 3) processos

- Exemplo com um computador

```
127.0.0.1 slots=10
```

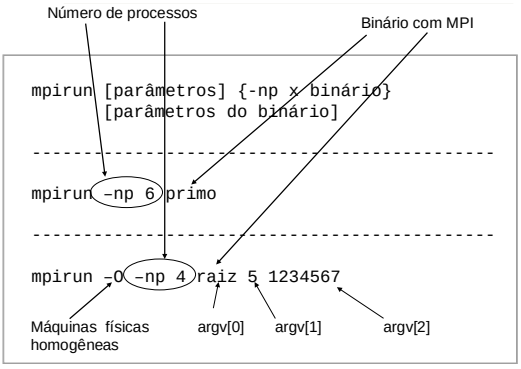
- Vai executar, na máquina local até 10 processos

MPI: execução de programas

- Para execução em vários computadores
  - Máquina paralela deve estar operante
- Utilização de *script*
  - *mpirun*
    - binário
    - -np processos
    - parâmetros MPI
    - parâmetros binário

Obrigatórios

mpirun



mpirun: vários computadores

- Para executar aplicações MPI em vários computadores, deve-se:
  - Ter permissão de executar aplicações de forma remota, sem senha
  - Configurar *Remote Shell* (RSH) ou *Secure Shell* (SSH)
- Todos os computadores devem ter acesso ao binário
  - Compilar na mesma pasta em todos os computadores
  - Usar *Network File System* (NFS)
    - Ou outra forma de compartilhamento de disco via rede (Samba, por exemplo)

mpirun: vários computadores

- Acrescentar arquivo de *hosts* ao comando de execução

```
--hostfile <arquivo_hosts>
```

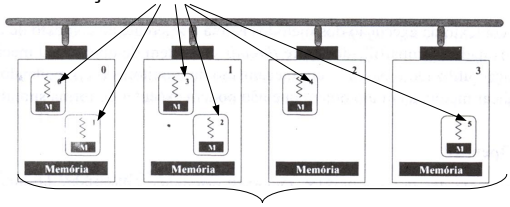
- Comando completo fica

```
mpirun -np x --hostfile hosts.txt binário
```

  - Onde
    - x: quantidade de processos sendo criado
    - hostfile: nome do arquivo de hosts (no caso hosts.txt)
    - binário: nome do binário executável

Alocação de processos a processadores

- Circular sobre máquinas físicas
- Máquina física: + de um processo
- Identificação: rank



Aplicações paralelas com MPI

- Utilizam de trocas de mensagens
- Número fixo de processos
- Identificação pelo rank
- Modelos
  - MPMD
  - **SPMD**

Programação com MPI

- Biblioteca
  - ± 120 funções
  - Pré-fixadas: “MPI_”
    - Próxima primeira letra: maiúscula
    - Exemplo: MPI_Init();
- Conceitos
  - Processos MPI
  - Comunicadores
  - Mensagens

Processos MPI

- Rank: identificador do processo
  - Numerados de 0 até “N-1”
    - N: número de processos
  - Único no ambiente paralelo
- Processo alocado em máquina
  - pid ≠ rank
- Rotinas para identificação
  - Identificação do rank
  - Número de processos

Comunicadores

- Processos MPI organizam-se em grupos
- Comunicação entre grupos: comunicadores
- Representa contexto de comunicação
- Todos os processos
  - **MPI_COMM_WORLD**

Mensagens MPI

- Divididas:
  - Dados: o que será comunicado (&)
  - Envelope: para quem

<table>
<tr><td colspan="6">Mensagem</td></tr>
<tr><td colspan="3">Envelope</td><td colspan="3">dado</td></tr>
<tr><td>origem
destino</td><td>tag</td><td>comunicador</td><td>endereço</td><td>quantidade</td><td>tipo</td></tr>
</table>

Mensagens MPI: envelope

- Origem/destino: rank
  - Envio: destino (para quem)
  - Recebimento: origem (de quem)
    - Permite recebimento múltiplo
- Tag: etiqueta (assunto)
  - Filtros para recebimento
- Comunicador: grupo comunicação

Mensagens MPI: dados

- Endereço: referência a memória
  - Envio: de onde
  - Recebimento: para onde
- Quantidade
  - Número de elementos que serão comunicados
    - 1: variável
    - N: vetor
- Tipo: tipos padrão do MPI

MPI: Tipos de dados (principais)

Tipos de dados do MPI	Tipos de dados em C
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long 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	

MPI básico

- 6 funções
  - 2 para inicio e término
  - 2 para controle de processos
  - 2 comunicação ponto a ponto
    - Envio
    - Recebimento
- Grande número de aplicações podem ser implementadas

MPI básico

- MPI_Init: inicializa
- MPI_Finalize: termina
- MPI_Comm_rank: rank atual
- MPI_Comm_size: número de processos
- MPI_Send: envia mensagem
- MPI_Recv: recebe mensagem

MPI_Init

- Primeira função a ser chamada
- Apenas 1 vez
- Sincroniza todos os processos
- Protótipo

```
int MPI_Init(
      int*   argc   /* entrada/saída */
      char** argv[] /* entrada/saída */)
```


MPI_Finalize

- Última função a ser chamada
- Sincroniza os processos
- Apenas 1 vez
- Protótipo

```
int MPI_Finalize(void)
```


MPI_Comm_rank

- Descobre a identificação (rank) do processo sendo executado
- Protótipo

```
int MPI_Comm_rank(
    MPI_Comm comm          /* entrada */
    int*     rank_processo /* saída   */)
```


MPI_Comm_size

- Descobre o número de processos sendo executados
- Protótipo

```
int MPI_Comm_size(
    MPI_Comm comm         /* entrada */
    int*     num_de_procs /* saída */)
```

1o. Programa exemplo

- primeiro.c
- Algoritmo
  - Inicialização dos processos
  - Cada processo descobre o número de processos ativos e quem é (rank)
  - Cada processo mostra uma mensagem
  - Finalização dos processos

1o. Programa exemplo Compilação e Execução

```
$ mpicc primeiro.c –o primeiro

$ mpirun -np 4 primeiro

Oi Mundo!! Sou o processo 0 de 4 sendo executado
Oi Mundo!! Sou o processo 2 de 4 sendo executado
Oi Mundo!! Sou o processo 1 de 4 sendo executado
Oi Mundo!! Sou o processo 3 de 4 sendo executado
```


MPI_Send

- Envia mensagens no modelo
  - Síncrono: bloqueia até mensagem ser copiada *buffer*
- Protótipo

```
int MPI_Send(
    void*        mensagem     /* entrada */
    int          elementos    /* entrada */
    MPI_Datatype tipo_MPI     /* entrada */
    int          destino      /* entrada */
    int          tag          /* entrada */
    MPI_Comm     comunicador) /* entrada */
```


MPI_Send: dados

- mensagem: void*
  - **DEVE** ser uma referência
  - A partir de onde os dados serão enviados
  - void*: uso com qualquer tipo de dados
- elementos: int
  - Quantidade de elementos a serem enviados
    - Variável: 1
    - Conjunto de dados (contínuo): “n”
- tipo_MPI: MPI_Datatype
  - Tipo de dado padrão do MPI

MPI_Send: envelope

- destino: int
  - Destinatário (um)
  - Identificação (rank) de quem irá receber a mensagem
- tag: int
  - Assunto
- comunicador: MPI_Comm
  - Padrão: MPI_COMM_WORLD

MPI_Recv

- Recebe mensagens no modelo
- Síncrono: bloqueia até que mensagem nos padrões do envelope seja recebida
- Protótipo

```
int MPI_Recv(
        void*        mensagem    /* saída */
        int          elementos   /* entrada */
        MPI_Datatype tipo_MPI    /* entrada */
        int          origem      /* entrada */
        int          tag         /* entrada */
        MPI_Comm     comunicador /* entrada */
        MPI_Status*  status)     /* saída */
```

MPI_Recv: dados

- mensagem: void*
  - DEVE ser uma referência
  - A partir de onde os dados serão recebidos
- elementos: int
  - Quantidade de elementos a serem recebidos
  - Pode ser ≠ do enviado
  - 1 variável ou conjunto de dados
- tipo_MPI: MPI_Datatype
  - Tipo de dado padrão do MPI

MPI_Recv: envelope

- origem: int
  - Quem enviou: um ou vários
  - Identificação (rank) de quem enviou a mensagem
- tag: int
  - Assunto
- comunicador: MPI_Comm
  - Padrão: MPI_COMM_WORLD

MPI_Recv: MPI_Status

- status: MPI_Status
  - Estrutura de controle de recebimento
  - Permite identificações após o recebimento
  - Uso com coringas
- Protótipo do MPI_Status

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
}
```


2o. Programa exemplo: algoritmo

- segundo.c
- Inicialização dos processos
  - Descoberta de rank e # de processos
- Divisão de funções devido ao rank: SPMD
- rank != 0 (n-1 processos)
  - Gera um número e envia ao rank == 0
- rank == 0 (1 processo)
  - Laço de 1 a n processos (para receber todas msgs)
  - Recebe cada uma das mensagens (em ordem)
  - Mostra o que foi recebido e de quem foi recebido
- Finalização dos processos

2o. Programa exemplo: Compilação e execução

```
$ mpicc segundo.c –o segundo

$ mpirun -np 4 segundo
Recebido o valor 2 do processo 1
Recebido o valor 4 do processo 2
Recebido o valor 6 do processo 3
```

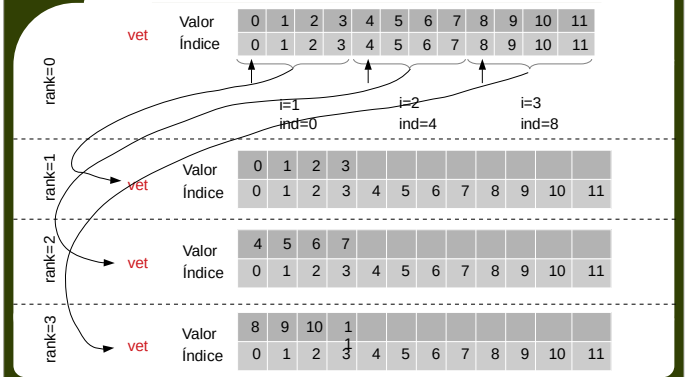

Comunicando conjuntos de dados

- Distribuição de volume de dados a serem processados
  - Individualmente: muitas mensagens
- Dados alocados continuamente na memória
- Uso de MPI_Send e MPI_Recv
  - Número de elementos com valor > 1

3o. Programa exemplo: algoritmo

- terceiro.c
- Inicialização semelhante aos outros
  - Vetor (vet) com tamanho especificado: #define
  - Cálculo da quantidade de posições: “parte”
- rank == 0
  - Preenche vetor com valores
  - Envia “parte” elementos de “vet” para os demais
- rank != 0
  - Recebe “parte posições” a partir de vet[0]
    - Não ocupa todo o vetor
  - Mostra os valores recebidos (ao final quem mostra)
- Finalização dos processos

3o. Programa exemplo Execução



3o. Programa exemplo: Compilação e execução

```
$ mpirun -np 4 terceiro

Proc. 1 recebeu [ 0 1 2 3 ] Final do proc. 1
Proc. 2 recebeu [ 4 5 6 7 ] Final do proc. 2
Proc. 3 recebeu [ 8 9 10 11 ] Final do proc. 3

-------------------------------------------------

$ mpirun -np 5 terceiro

Proc. 1 recebeu [ 0 1 2 ] Final do proc. 1
Proc. 2 recebeu [ 3 4 5 ] Final do proc. 2
Proc. 3 recebeu [ 6 7 8 ] Final do proc. 3
Proc. 4 recebeu [ 9 10 11 ] Final do proc. 4
```


Recebendo mensagens com coringas

- Recebimento de mensagens
  - Fora de ordem (laço)
  - Filtros de assuntos
- Coringas
  - Na origem: **MPI_ANY_SOURCE**
  - No assunto (tag): **MPI_ANY_TAG**
- Uso após recebimento
  - MPI_Status (estrutura)

4o. Programa exemplo: algoritmo

- Inicialização e finalização: iguais
- rank == 0
  - Laço 1
    - Envia msg. aos demais com tag = msg = destino
  - Laço 2
    - Recebe msg. em qq. ordem com qq. tag
    - Mostra de quem recebeu e qual tag recebida
- rank != 0
  - Recebe msg. do 0 com qq. tag (MPI_ANY_TAG)
  - Calcula valor usando tag de recebimento
  - Envia msg. com valor calculado

4o. Programa exemplo: execução

```
$ mpirun -np 6 quarto

Recebido 1 do proc. 1 com tag 1
Recebido 4 do proc. 2 com tag 2
Recebido 25 do proc. 5 com tag 5
Recebido 16 do proc. 4 com tag 4
Recebido 9 do proc. 3 com tag 3
```