1a. Lista de Exercícios*

1 Programando em MPI

O objetivo desta lista de exercícios é o de iniciar o estudo *prático* de computação paralela utilizando troca de mensagens (`MPI_Send` e `MPI_Recv`. Utilizando esses dois comandos vamos escrever um programa bastante simples.

2 O Programa Congratulações

Da mesma forma que nosso primeiro programa em C foi provavelmente "hello, world", vamos apresentar uma variante desse programa, utilizando multiplos processos. No nosso sistema, os p processos são rotulados com $0, 1, \ldots, p-1$. O programa que vamso apresentar, todos os processos cujos rótulos são diferentes de 0 enviam uma mensagem de congratulações para o processo 0. O processo 0 recebe essas mensagens e as imprime. A seguir o programa MPI que executa isso.

```
// Programa: greetings.c
/*
 * Envia uma mensagem de todos os processadores com rank != 0 para o
 *       processo 0. O processo 0 imprime as mensagens recebidas.
 *
 * Input: nenhuma.
 * Output: conteúdo das mensagens recebidas pelo processo 0.
 *
```

*Este material é para o uso exclusivo do da disciplina MAC 5705 - Topicos de Algoritmos Paralelos usando MPI e BSP/CGM - 2003 do IME-USP e refere-se ao livro Parallel Programming with MPI de Peter Pacheco

```
 * Veja Capítulo 3, pp. 41 & ff em PPMPI.
 */
// Declaração das Bibliotecas uilizadas
#include <stdio.h>
#include <string.h>
#include "mpi.h"

main(int argc, char* argv[]) {
// Declaração das variáveis locais
   int         my_rank;       /* rank do processo             */
   int         p;             /* numero de processos          */
   int         source;        /* rank do emissário            */
   int         dest;          /* rank do recebedor            */
   int         tag = 0;       /* tag para as menssages        */
   char        message[100];  /* armazenamento para messagem */
   MPI_Status  status;        /* return status do receive     */

// Passo 1. Inicialização
// Passo 1.1. Inicialize o MPI
   MPI_Init(&argc, &argv);
// Passo 1.2. Determine o rank do processo
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
// Passo 1.3. Determine o número de processos
   MPI_Comm_size(MPI_COMM_WORLD, &p);
// Passo 2. Envie/receba a Mensagem
// Passo 2.1. Se o rank != 0
   if (my_rank != 0) {
// Passo 2.1.1. Crie a mensagem
      sprintf(message, "Congratulações do processo %d!", my_rank);
// Passo 2.1.1. Defina o número do processo que receberá a MSG
      dest = 0;
// Passo 2.1.2. Use strlen+1 tal que '\0' seja transmitido
      MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   }
// Passo 2.2. Se o rank == 0
   else {
// Passo 2.2.1. receba as MSG's dos demais processos
      for (source = 1; source < p; source++) {
```

```
         MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
// Passo 2.2.2. Imprima a Mensagem
         printf("%s\n", message);
      }
   }
// Passo 3. Finalize o MPI
   MPI_Finalize();
} /* fim main */
```

3 Compilação e Execução

Os detalhes da compilação e execução do programa acima depenedem do sistema que estiver sendo usado. Vamos descrever os detalhes para o ambiente LAM MPI.

A compilação pode ser feita utilizando o mpicc

```
% mpicc -o greetings greetings.c
```

Após o programa ser compilado, podemos executa-lo. Para isso, temo que inicializar o LAM MPI. Isso pode ser feito da seguinte maneira:

```
% lamboot -v lamhosts
```

onde `lamhosts` é o arquivo contendo as estações de trabalho que comporão a máquina paralela virtual.

Após inicializar a máquina virtual, o programa pode ser executado (em todas as estações de trabalho) da máquina virtual usando o seguinte comando:

```
% mpirun N greetings
```

ou de uma forma mais geral

```
% mpirun n0-x [-c tarefas] greetings
```

onde `0-x` é o intervalo de estações de trabalho da máquina paralela virtual que serão utilizadas e `-c tarefas` é o número de cópias do programa `greetings` que serão executos em cada processador (estação de trabalho).

Quando o programa é executado em dois processos, a saída é a seguinte:

```
Congratulações do processo 1!
```

Se o programa é executado em quatro processos, a saída deve ser

```
Congratulações do processo 1!
Congratulações do processo 2!
Congratulações do processo 3!
```

Alguns aspectos relativos a execução do programa devem ser destacados.

1. O usuário emite uma diretiva para o sistema operacional cujo efeito é o de colocar uma cópia (ou cópias) do programa executável em cada processador.

2. Cada processador inicia a execução de sua cópia (ou cópias) do executável.

3. Processos diferentes podem executar instruções diferentes através da ramificação do programa baseado nos ranks dos processos.

A última observação é muito importante. A forma mais geral de programação MIMD, cada processo executa um programa diferente. Contudo, na prática, isto não é necessário, e a aparência de "cada processo executando um programa diferente" é obtido através da inclusão de instruções de ramificação dentro de um único problema. Desta forma, no programa "Congratulações!", temos que as instruções executadas pelo processo 0 são diferentes das instruções executadas pelos demais processos, e evitamos a escrita de diversos programas através da utilização da instução de ramificação

```
if (my_rank != 0)
      .
      .
      .
 else
      .
      .
      .
```

Esta formo de programação MIMD é frequentemente denominada de programação **single-program multiple-data** (**SPMD**). Neste nosso curso, utilizaremos a programação SPMD.

Após a execução do programa, a máquina virtual paralela é finalizada com os comandos:

```
% lamclean -v
% lamhalt -v
```

4 MPI

Notamos que nosso programa consiste inteiramente de instruçõe convencionais em C e diretivas para o preprocessador. MPI não é uma nova linguagem de programação, é simplesmente uma biblioteca de definições e funções que podem ser usadas em programas C. Todo programa MPI deve conter a diretiva

```
#include ``mpi.h''
```

para o preprocessador. Este arquivo, `mpi.h`, contém as definições e declarações necessárias para compilar um programa MPI.

Antes que qualquer função MPI seja chamada, a função `MPI_Init` deve ter sido chamada. Após o programa terminar de usar as funções da biblioteca MPI, ele deve fazer uma chamada da função `MPI_Finalize`.

O rank de cada processopode ser obtido através da função `MPI_Comm_rank` e o número de processos que estão sendo executados é obtido através da função `MPI_Comm_size`.

O envio e o recebimento de mensagens é efetuado pelas funções `MPI_Send` e `MPI_Recv`, respectivamente. Uma mensagem contém pelo menos as seguintes informações:

1. O rank do recebedor
2. O rank do emissor
3. Uma tag (etiqueta)
4. Um comunicador

Pra mais detalhes, sugerimos a leitura das páginas 45 a 47 do livro do Peter Pacheco.

A sintaxe das funções de envio e recebimento é a seguinte:

```
int MPI_Send( void*       message /* in */,
              int         count   /* in */,
              MPI_Datatypedatatype/* in */,
              int         destination/* in */,
              int         tag     /* in */,
              MPI_Comm    comm    /* in */)

int MPI_Recv(  void*       mensagem/* out */,
               int         count   /* in */,
               MPI_Datatypedatatype/* in */,
               int         source  /* in */,
               int         tag     /* in */,
               MPI_Comm    comm    /* in */,
               MPI_Status* status  /* out */)
```

Os parâmetros `dest` e `source` são, respectivamente, os ranks dos processos de recebimento e envio. O MPI permite que `source` contenha um coringa, `MPI_ANY_SOURCE`. Não existem coringas para `dest`. Podemos usar um coringa tamb em `tag`, `MPI_ANY_TAG`.

Para obter informação com relação ao tamanho da mensagem recebida, podemos utilizar a função

```
int MPI_Get_Count(MPI_Status*   status    /* in */,
                  MPI_Datatype  datatype  /* in */,
                  int*          count_ptr /* in */)
```

5 Exercícios

1. Crie um arquivo fonte C contendo o programa "Congratulações!". Descubra a forma de compila-lo e executa-la em um número diferente de processadores. Qual é a saída do programa se ele é executado em um único processador? Quantos processadores você pode usar?

2. Modifique o programa "Congratulações" de tal forma que ele utilize coringas para `source` e `tag`. Existe alguma diferença na saída do programa?

3. Modifique o programa "Congratulações" tal que todos os processos enviem uma mensagem para o processo $p - 1$. Em vários sistemas paralelos, todo processo pode imprimir na tela do terminal na qual o programa foi inicializado. O processo $p - 1$ consegue imprimir na tela?

6 Exercício de Programação

1. Escreva um programa na qual o processo i envia congratulações para o processo $(i + 1)\%p$. (Analise bem como i calcula de quem ele deve receber a mensagem). O processo i deve primeiro enviar sua mensagem para o processo $i+1$ e então receber a mensagem do processo $i-1$? Ou ele deve primeiro recer e então enviar? Essa ordem envio/recebimento é importante? O que acontece quando o programa é executado em um único processador?