Sistemas Distribuídos

Prof. André Nasserala

andre.nasserala@ufac.br

Será visto nessa parte:

- Programação Distribuída.

Introdução

- Programa Distribuído: Conjunto de processos que trabalham em conjunto para solucionar um problema e executam em sistemas distribuídos

- Sistema Distribuído:
- Não há compartilhamento de memória
- Troca de informação através de troca de mensagens

Introdução

- Áreas de demanda:
- Otimização combinatória
- Mineração de dados
- Simulações
- Meteorologia
- Bioinformática
- Computação gráfica

Sequencial ≠ Distribuído

- Determinação de estado não é trivial

Sequencial

if (x=1) then

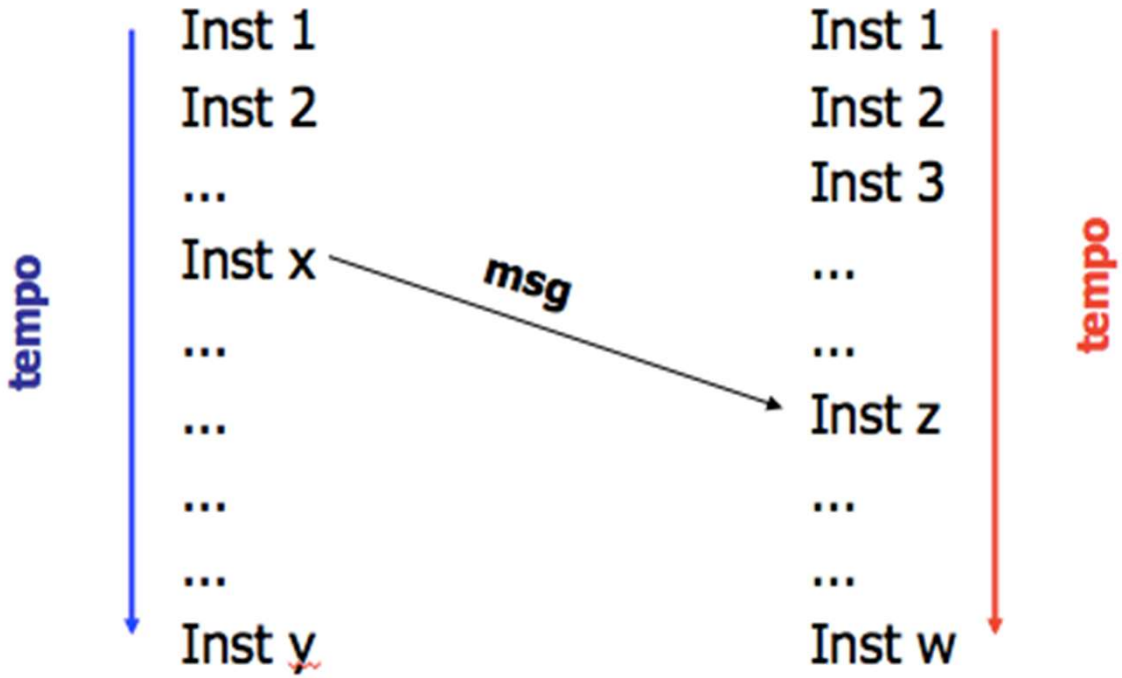
Distribuído

if ($x_{\text{proc a}}$ = 1) and
($y_{\text{proc b}}$ = 2) then

Não é trivial !!!

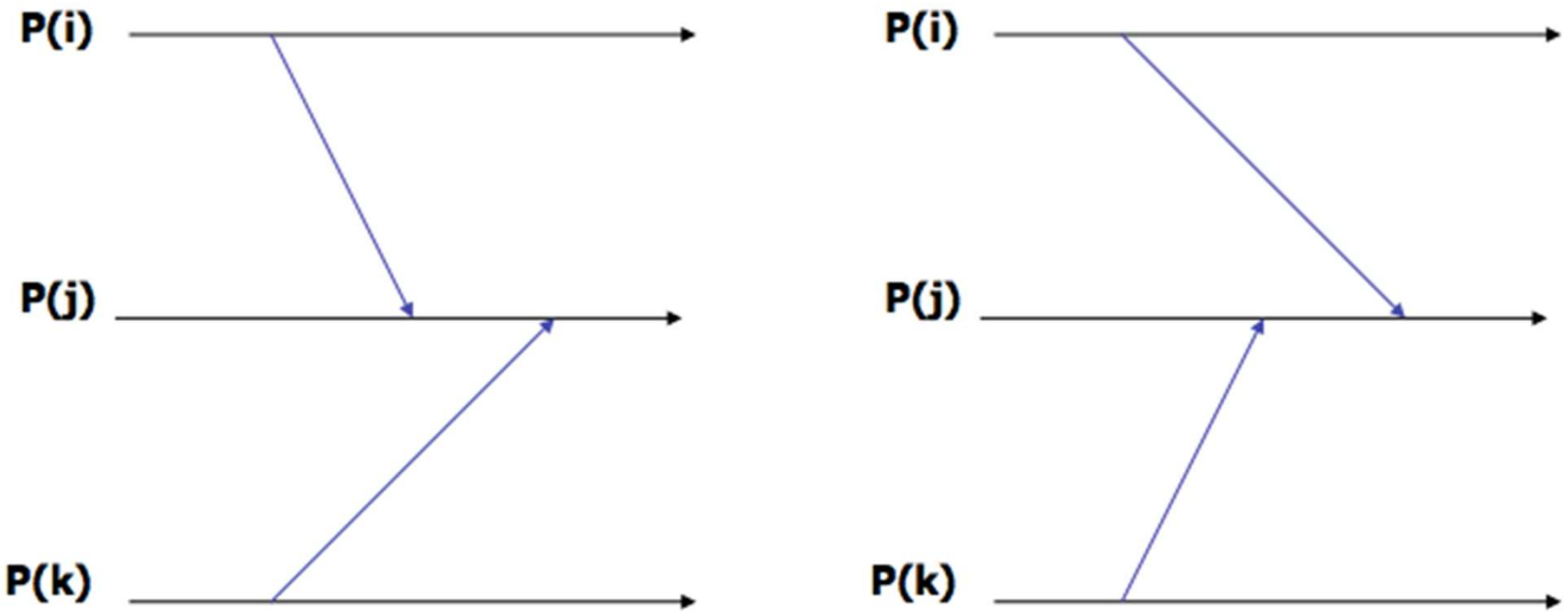
Sequencial ≠ Distribuído

- Ausência de uma base de tempo global



Sequencial ≠ Distribuído

- Não determinístico



Modelos de Computação

- Assíncrono:
  - Sem base de tempo global
  - Atraso na transmissão das mensagens é finito mas não determinado
  - Desenvolvimento mais complexo
  - Realístico
- Síncrono:
  - Com base de tempo global
  - Comunicação em “pulsos”
  - Desenvolvimento mais simples
  - Menos realístico

Introdução ao MPI

Ufac

- O MPI é um padrão para desenvolvimento de aplicações distribuídas
- Disponível na forma de bibliotecas para linguagens C, C++ e Fortran
- SPMD – Single Program Multiple Data – todos os processos executam o MESMO programa
- Síntese de diversos sistemas anteriores
- Desenvolvido por um fórum aberto internacional composto por representantes da indústria, universidade e entidades governamentais
- Influenciado por outras plataformas e comunidades: Zipcode, Chimp, PVM, Chamaleon e PICL
- Padronização iniciada em abril de 1992

Introdução ao MPI

Ufac

- Objetivos:
  - Eficiência na comunicação
  - Ambientes para desenvolvimento e execução heterogêneos
  - Fácil aprendizado para atuais programadores de aplicações distribuídas (interface parecida com a do PVM)
- Características:
  - Supõe que a subcamada de comunicação é confiável
  - Garante ordem de entrega das mensagens
  - Trabalha com o conceito de COMUNICADORES, que definem o universo de processos envolvidos em uma operação de comunicação
  - Cada processo ganha uma identificação numérica (rank)

Inicialização/Finalização

- MPI_Init : Inicializa o ambiente de execução
- MPI_Comm_rank: Determina o rank (identificação) do processo no comunicador
- MPI_Comm_size: Determina o número de processos no comunicador
- MPI_Finalize: Termina o ambiente de execução

Comunicação Ponto a Ponto

Ufac

- O que acontece se um processo tentar receber uma mensagem que ainda não foi enviada?

- Função BLOQUEANTE de recebimento (MPI_Recv) : bloqueia a aplicação até que o buffer de recepção contenha a mensagem

- Função NÃO BLOQUEANTE de recebimento (MPI_Irecv) : retorna um handle request, que pode ser testado ou usado para ficar em espera pela chegada da mensagem

MPI_Send

- int MPI_Send(void* message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- message: endereço inicial da informação a ser enviada
- count: número de elementos do tipo especificado a enviar
- datatype: MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_BYTE, MPI_LONG, MPI_UNSIGNED_CHAR, etc
- dest: rank do processo destino
- tag: identificador do tipo da mensagem
- comm: especifica o contexto da comunicação e os processos participantes do grupo. O padrão é MPI_COMM_WORLD

MPI_Recv

- int MPI_Recv(void* message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)
- message: Endereço inicial do buffer de recepção
- count: Número máximo de elementos a serem recebidos
- datatype: MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_BYTE, MPI_LONG, MPI_UNSIGNED_CHAR, etc.
- source: rank do processo origem (* = MPI_ANY_SOURCE)
- tag: tipo de mensagem a receber (* = MPI_ANY_TAG)
- comm: comunicador
- status: Estrutura com três campos: MPI_SOURCE, MPI_TAG, MPI_ERROR

MPI_Irecv

- Menos utilizada

- Parâmetros iguais ao bloqueante, acrescido de uma estrutura (request) que armazena informações que possibilitam o bloqueio posterior do processo usando a função MPI_Wait(&request, &status)

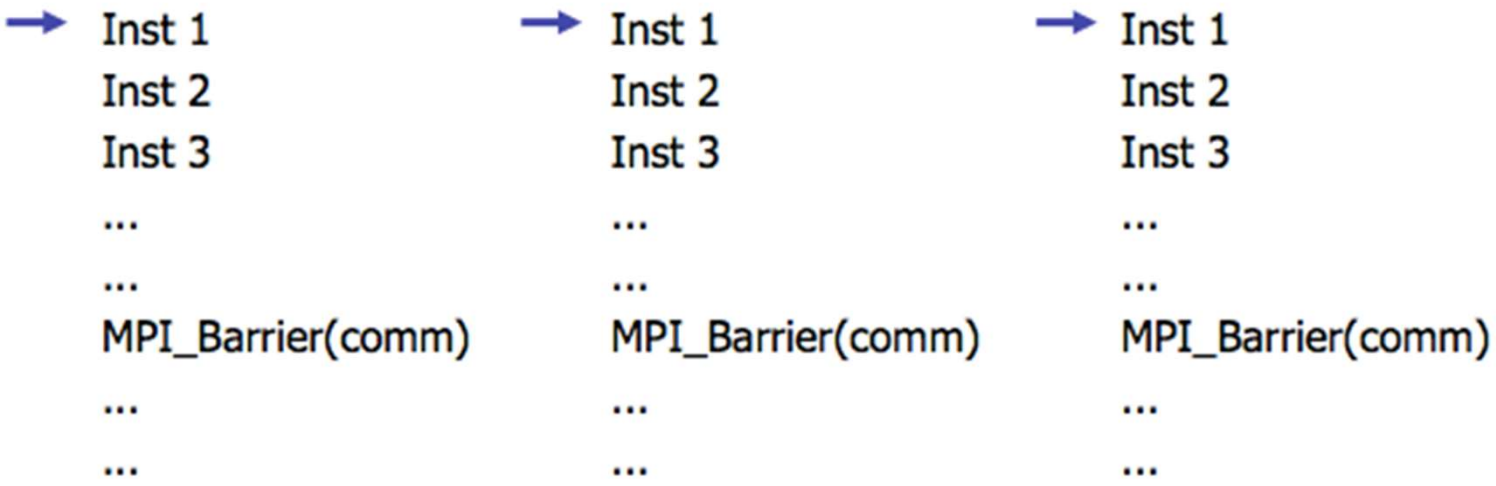
Comunicação Coletiva

- Mais restritivas que as comunicações ponto a ponto:
- quantidade de dados enviados deve casar exatamente com a quantidade de dados especificada pelo receptor
- Apenas a versão bloqueante das funções está disponível
- O argumento tag não existe!

- Todos os processos participantes da comunicação coletiva chamam a mesma função com argumentos compatíveis

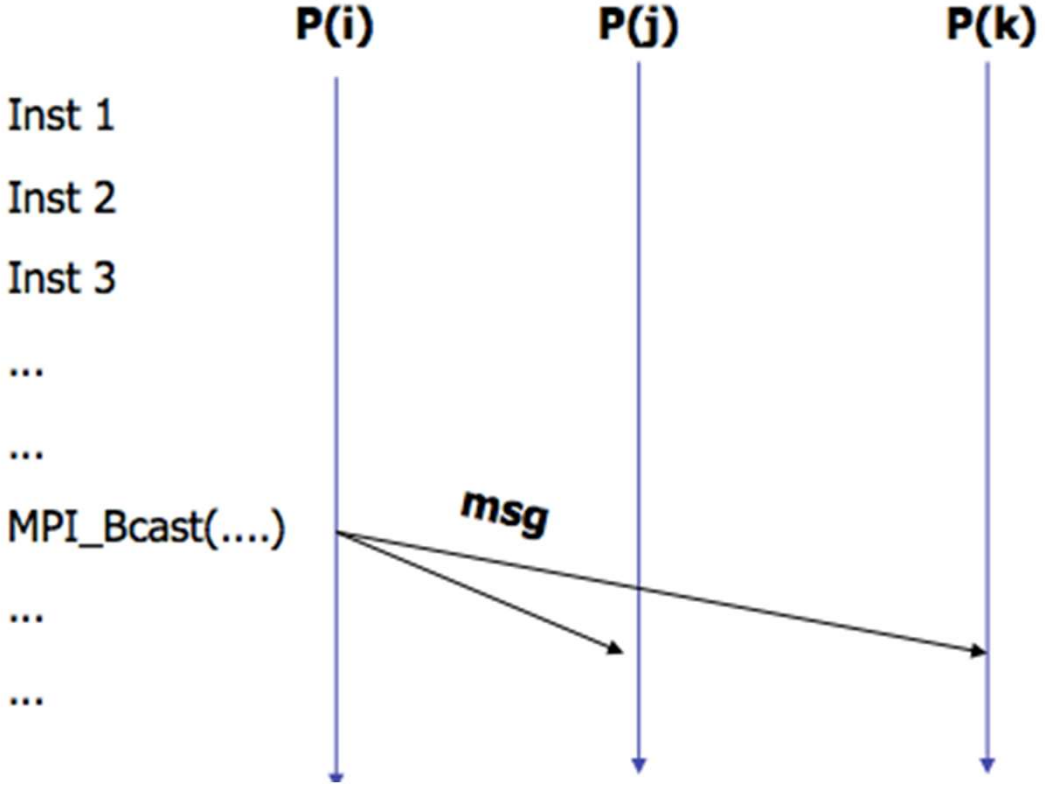
Comunicação Coletiva

- MPI_Barrier: Bloqueia o processo até que todos os processos associados ao comunicador chamem essa função.



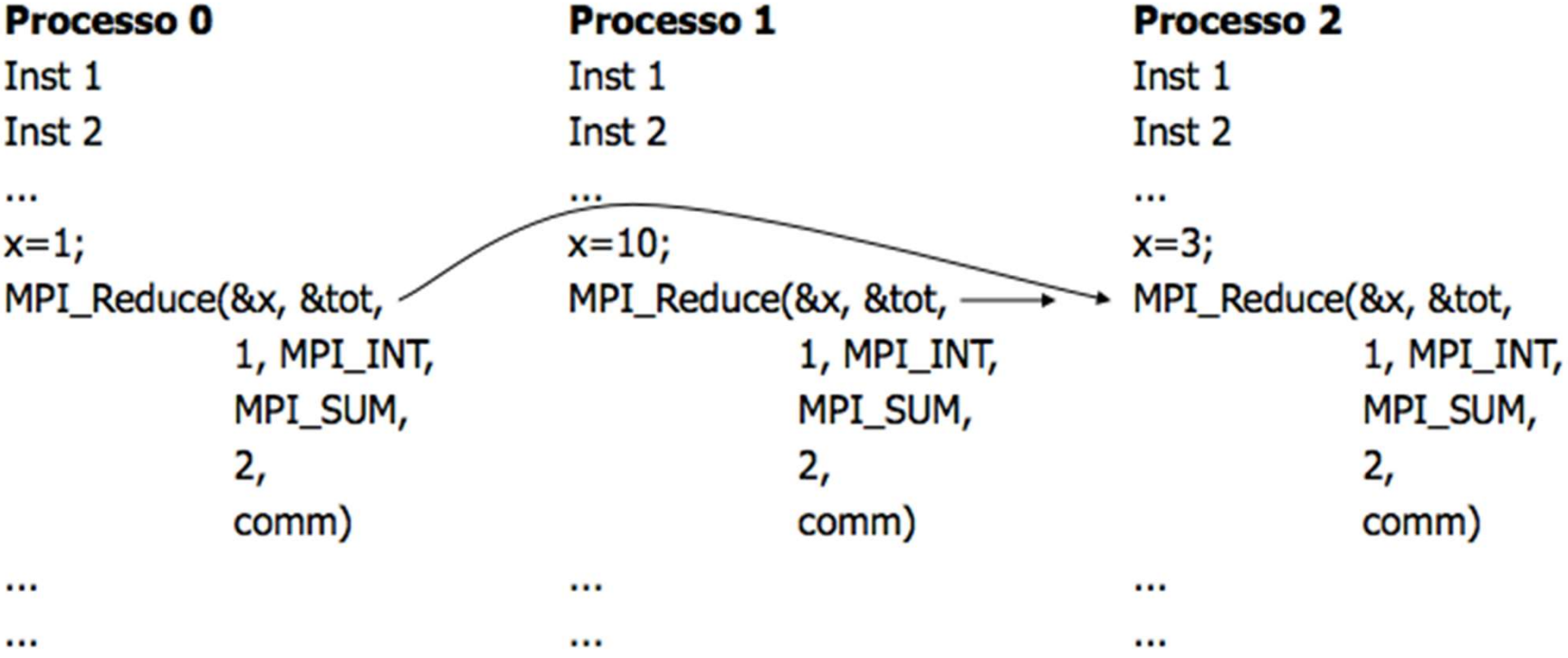
Comunicação Coletiva

- MPI_Bcast: Faz a difusão de uma mensagem do processo raiz para todos os processos associados ao comunicador



Comunicação Coletiva

- MPI_Reduce: Combina todos os elementos presentes no buffer de cada processo do grupo usando a operação definida como parâmetro e coloca o valor resultante no buffer do processo especificado. O exemplo abaixo soma todas as variáveis “x” armazenando o total na variável “tot” do processo 2.



Programa Exemplo

```
#include <stdio.h>
#include <string.h>
#include “mpi.h”
main(int argc, char** argv)
{
        int meu_rank, np, origem, destino, tag=0;
        char msg[100];
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
        MPI_Comm_size(MPI_COMM_WORLD,&np);
```

Programa Exemplo

```
if (meu_rank != 0) {
   sprintf(msg, “Processo %d está vivo!”, meu_rank);
   destino = 0;
   MPI_Send(msg,
                        strlen(msg)+1,
                        MPI_CHAR,
                        destino,
                        tag,
                        MPI_COMM_WORLD);
}
```

Programa Exemplo

```
else {    // if (meu_rank == 0)
   for (origem=1; origem<np; origem++) {
      MPI_Recv(msg,
                                  100,
                                  MPI_CHAR,
                                  origem,
                                  tag,
                                  MPI_COMM_WORLD,
                                  &status);
      printf("%s\n",msg);
   }
}
MPI_Finalize( );
}
```

Compilando e Executando

- Escreva o código fonte em um editor de sua preferência(vi, nano ou pico em modo texto, Gnutext, notes, bloco de notas ou outro em modo gráfico) salve com extensão *.c;
- Compilando:
- mpicc -o hello hello.c
- Onde: -o hello é o arquivo binário gerado(em Windows .exe) e hello.c o arquivo com o código fonte;
- Executando:
- mpirun -np 8 ./hello
- Onde –np indica o número de processos que serão gerados, nesse exemplo 8.

Programa Exemplo01

- Objetivo:

- Fazer um programa em C com MPI que faça os n-processos entrantes enviarem uma mensagem “Processo X está VIVO!” para o processo inicial
- 0(ZERO) em ordem de processos.

Programa Exemplo01

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char** argv)
{
      int meu_rank, np, origem, destino, tag=0;
      char msg[100];
      MPI_Status status;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
      MPI_Comm_size(MPI_COMM_WORLD,&np);
```

Programa Exemplo01

```
if (meu_rank != 0) {
   sprintf(msg, "Processo %d está vivo!", meu_rank);
   destino = 0;
   MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
}
else {    // if (meu_rank == 0)
   for (origem=1; origem<np; origem++) {
      MPI_Recv(msg, 100, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
      printf("%s\n",msg);
   }
}
MPI_Finalize( );
}
```

Programa Exemplo02

- Objetivo:

- Fazer um programa em C com MPI que faça os n-processos entrantes enviarem uma mensagem “Processo X está VIVO!” para o processo inicial 0(ZERO).

Programa Exemplo02

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char** argv)
{
      int meu_rank, np, origem, destino, tag=0;
      char msg[100];
      MPI_Status status;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
      MPI_Comm_size(MPI_COMM_WORLD,&np);
```

Programa Exemplo02

```
if (meu_rank != 0) {
   sprintf(msg, "Processo %d está vivo!", meu_rank);
   destino = 0;
   MPI_Send(msg, strlen(msg)+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
}
else {    // if (meu_rank == 0)
   for (origem=1; origem<np; origem++) {
      MPI_Recv(msg, 100, MPI_CHAR, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
      printf("%s\n",msg);
   }
}
MPI_Finalize( );
}
```

Programa Exemplo03

- Objetivo:

- Fazer um programa em MPI que some dois números e envie os resultados ao processo 0, enquanto outro processo multiplique esses mesmo números e envie ao processo 0, no final o processo zero deve mostrar a soma e a multiplicação dos números.

Programa Exemplo03

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"
main(int argc, char** argv)
{
      int meu_rank, tag=0;
      int a = 3, b = 2, soma, mult;
      MPI_Status status;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
```

Programa Exemplo03

```
if (meu_rank == 0) {
        MPI_Recv(&soma, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&mult, 1, MPI_INT, 2, tag, MPI_COMM_WORLD, &status);
        printf("A soma e %d\n", soma);
        printf("A multiplicacao e %d\n", mult);
    }
    else if (meu_rank == 1) {
        soma = a + b;
        MPI_Send(&soma, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    }
    else {
        mult = a * b;
        MPI_Send(&mult, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    }
    MPI_Finalize( );
}
```

Programa Exemplo04

- Objetivo:

- Fazer um programa com 4 processos para calcular as raízes de uma equação do 2o Grau, o processo 0 calculará o delta e enviará o resultado aos processos 1 e 2 que calcularão as raízes X1 e X2 e enviarão o resultado ao processo 3 que vai imprimi-la em tela.

Programa Exemplo04

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char** argv)
{
      int meu_rank, np, tag=0;
      float a,b,c,d,x1,x2;
      MPI_Status status;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
      MPI_Comm_size(MPI_COMM_WORLD,&np);
      a = 1;
      b = 6;
      c = -10;
```

Programa Exemplo04

```
if (meu_rank == 0) {
        d = (b*b) - (4*a*c);
        if (d < 0 ) exit(0);
        else {
            MPI_Send(&d, 1, MPI_REAL, 1, tag, MPI_COMM_WORLD);
            MPI_Send(&d, 1, MPI_REAL, 2, tag, MPI_COMM_WORLD);

        }
    }
    else if(meu_rank == 1) {
        MPI_Recv(&d, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD, &status);
        x1 = ((-b) + sqrt(d))/(2*a);
        MPI_Send(&x1, 1, MPI_REAL, 3, tag, MPI_COMM_WORLD);
    }
```

Programa Exemplo04

```
else if(meu_rank == 2) {
        MPI_Recv(&d, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD, &status);
        x2 = ((-b) - sqrt(d))/(2*a);
        MPI_Send(&x2, 1, MPI_REAL, 3, tag, MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(&x1, 1, MPI_REAL, 1, tag, MPI_COMM_WORLD, &status);
        printf("O valor de x1 é %f\n", x1);
        MPI_Recv(&x2, 1, MPI_REAL, 2, tag, MPI_COMM_WORLD, &status);
        printf("O valor de x2 é %f\n", x2);
    }
    MPI_Finalize( );
}
```

Referências

- Valmir C. Barbosa, An Introduction to Distributed Algorithms, MIT Press, 1996
- http://www.ic.uff.br/~lucia/
- http://cs.ucsb.edu/~hnielsen/cs140/openmpi-install.html
- http://pt.wikipedia.org/wiki/Message_Passing_Interface