Programação Paralela

Profa Mariana Kolberg e Prof. Luiz Gustavo Fernandes

Introdução

- Programação paralela é a divisão de um problema em partes, de maneira que essas partes possam ser executadas paralelamente por diferentes processadores.
- Os processadores devem cooperar entre si, utilizando primitivas de comunicação e sincronização.
- Objetivos
  - Principal: Ganho de desempenho
  - Secundário: Tolerância a falhas

Motivação

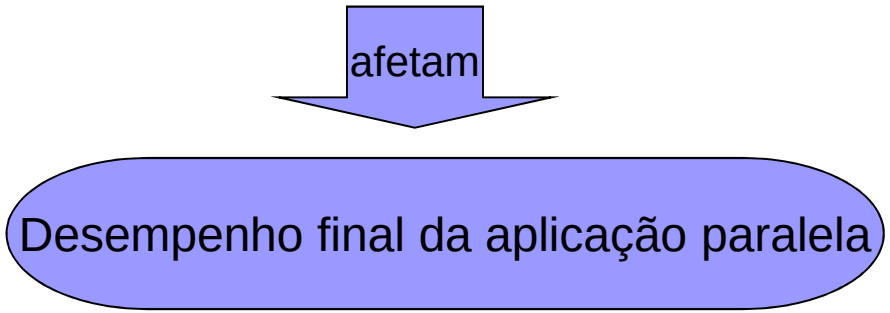
- Necessidade de poder computacional
  - Solução de aplicações complexas
    - Metereologia
    - Prospecção de petróleo
      - Análise de local para perfuração de poços
    - Simulações físicas
      - Aerodinâmica, energia nuclear
    - Matemática computacional
      - Análise de algoritmos para criptografia
    - Bioinformática
      - Simulação computacional da dinâmica molecular de proteínas

Conhecimento do Hardware

- Ideal seria não precisar conhecer detalhes da máquina
  - Portabilidade
- Em PP o conhecimento da máquina influencia diretamente o desempenho
  - Paradigma de comunicação
    - Memória compartilhada x distribuída
    - Preciso comunicar
  - Ferramenta de programação
    - Threads, openMP
    - MPI
  - Modelagem do problema

Modelagem do Problema

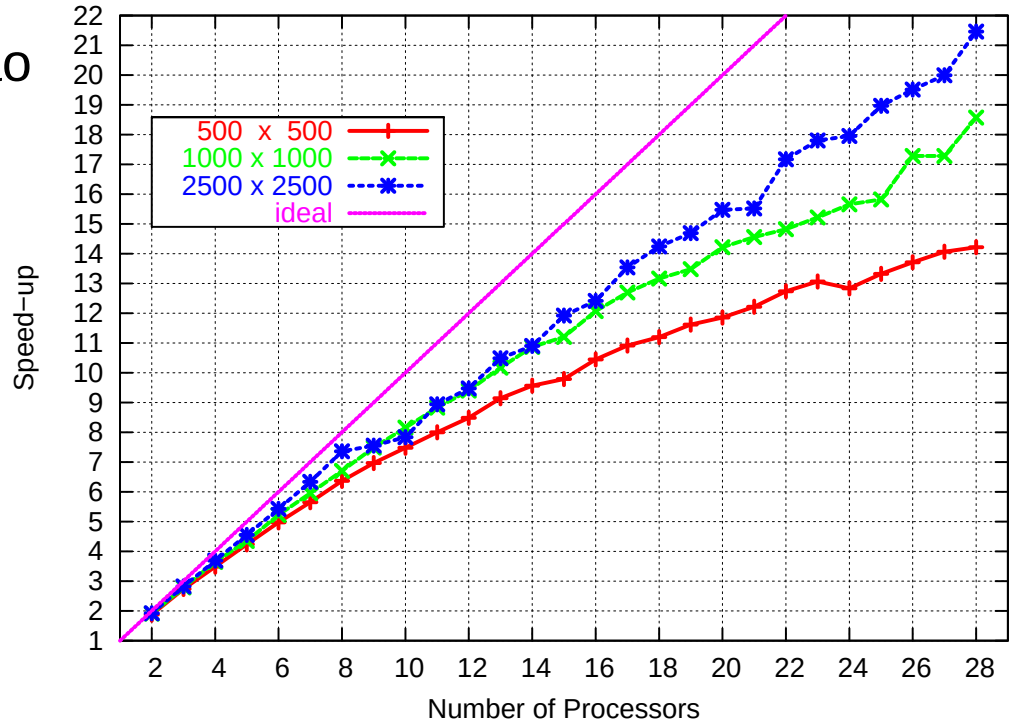
- Quantas processadores utilizar?
  - Quanto mais processadores mais rápido?
- Como distribuir o trabalho a ser feito (carga) entre os envolvidos?
- Quando comunicar/sincronizar?



Análise de Desempenho

- Speedup
  - Fator de aceleração
  - Melhor tempo seqüencial
  - Normalmente sp<np
    - Overhead de comunicação
  - Ideal linear sp=np
  - Superlinear sp>np
    - efeito de cache

$$Sp_n = \frac{T_s}{T_n}$$



Análise de Desempenho

- Eficiência
  - Medida de utilização do processador $$E_n = \frac{Sp_n}{n}$$
- Lei de amdahl
  - Usado para encontrar o maior Speedup para uma aplicação onde apenas uma parte do sistema foi paralelizada. O Speedup é limitado pelo tempo necessário para execução da parte seqüencial.

$$MaxSp_n = \frac{1}{(1-P) + \frac{P}{n}}$$

P = Parte paralelizável

(1-P) = Parte seqüencial

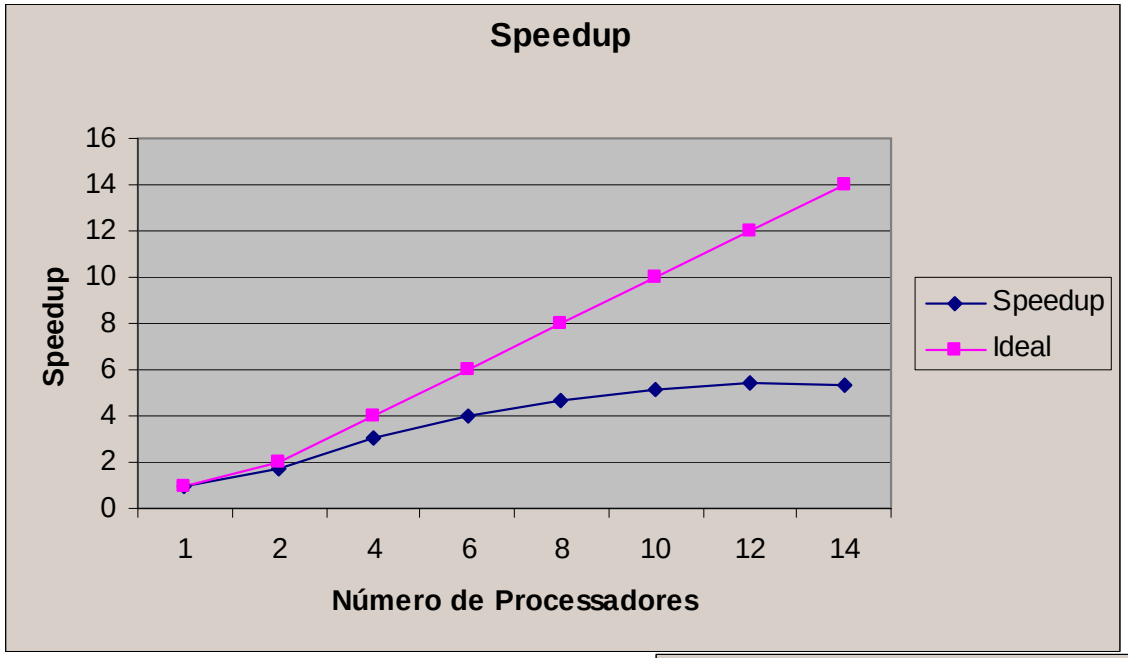
Análise de Desempenho

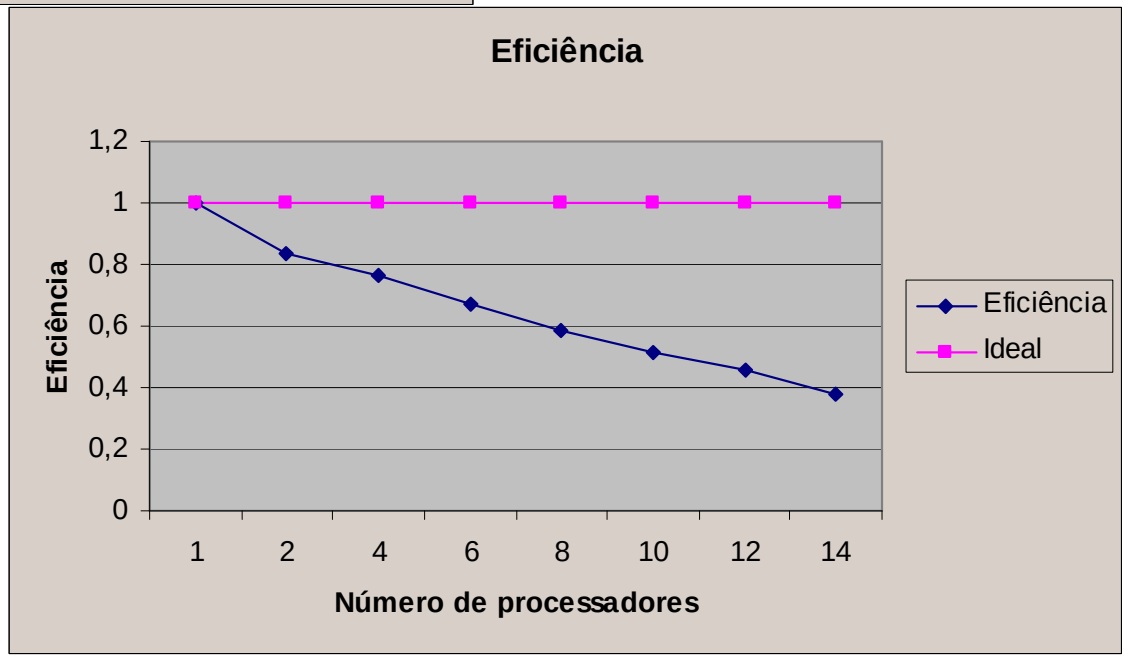
- Exercício: O tempo para execução em paralelo de um programa para multiplicação de duas matrizes de dimensão 5000 esta representado na tabela abaixo. Complete a tabela com o Speedup e a eficiência para cada número de processadores
- Tempos de execução feitos no cluster IC1 da universidade de Karlsruhe, Alemanha:
  - 200 nodos Intel Xeon com 8 cores e 16 GB de memória por nodo
  - InfiniBand 4X DDR Interconnect com ConnectX Dual Port DDR HCAs
  - Posição top500:
    - 104 novembro 2007
    - 265 junho 2008

Nr proc	T. Exec	speedup	Efic.
1	25,64	1	1
2	15,28		
4	8,37		
6	6,3483		
8	5,4437		
10	5,01		
12	4,69		
14	4,85		

Análise de Desempenho

número de processadores	Tempo de execução	Speedup	Eficiência
1	25,64	1	1
2	15,28	1,67801	0,839005
4	8,37	3,063321	0,76583
6	6,3483	4,038877	0,673146
8	5,4437	4,710032	0,588754
10	5,01	5,117764	0,511776
12	4,69	5,466951	0,455579
14	4,85	5,286598	0,377614


Speedup
16
14
12
10
8
6
4
2
0
Speedup
1
2
4
6
8
10
12
14
Número de Processadores
Speedup
Ideal


Eficiência
1,2
1
0,8
0,6
0,4
0,2
0
Eficiência
1
2
4
6
8
10
12
14
Número de processadores
Eficiência
Ideal

Desafios na Modelagem do Problema

- Conversão da aplicação seqüencial em paralela
  - Algumas aplicações não são paralelizáveis
- Particionamento de código e dados
  - Modelos de distribuição de dados
  - Dependência das operações
- Custo de sincronização
  - Necessidade de troca de informação entre processos
- Balanceamento de carga
  - Divisão adequada da computação entre os recursos
- Deadlocks (comunicação)
  - Faz send e não receive
- Dificuldade de depuração

Modelagem

- **Modelos de máquina**: descrevem as características das máquinas
- **Modelos de programação**: aspectos ligados a implementação e desempenho
- **Modelos de aplicação**: paralelismo de um algoritmo
- Vantagens:
  - Compreender diferentes aspectos da aplicação na implementação de um programa paralelo
    - Quantidade de cálculo envolvido total e por atividade
    - Volume de dados manipulado
    - Dependência de informações entre atividades em execução

Modelos de Máquina

- Classificação de Flynn
  - SISD (pc), SIMD (maq vetorial), MISD, **MIMD** (maq par e dist)
- Arquiteturas com memória compartilhada
  - Multiprocessador
  - SMP, NUMA
  - Comunicação realizada através de escritas e leituras na memória compartilhada
- Arquiteturas com memória distribuída
  - Multicomputador
  - MPP,NOW,Cluster
  - Comunicação realizada por troca de mensagens em uma rede de interconexão

Modelos de Programação

- Distribuição do trabalho – granulosidade
  - Relação entre o tamanho de cada tarefa e o tamanho total do programa, ou seja, é a razão entre computação e comunicação. Pode ser alta (grossa), media ou baixa (fina)
    - O tamanho de cada tarefa depende do poder de processamento de cada nó
    - A comunicação depende da latência e da vazão da rede de interconexão

Modelos de Programação

- Granulosidade grossa
  - Maior custo processamento
  - Dificulta balanceamento de carga
  - Muito processamento, pouca comunicação
  - Menor custo de sincronização
- Granulosidade fina
  - Maior freqüência de comunicação
  - Facilita balanceamento de carga
  - Pouco processamento, muita comunicação
  - Alto custo de sincronização
- Indica o tipo de arquitetura mais adequado: procs vetoriais (fina), SMP (média), cluster (grossa)

Modelos de Programação

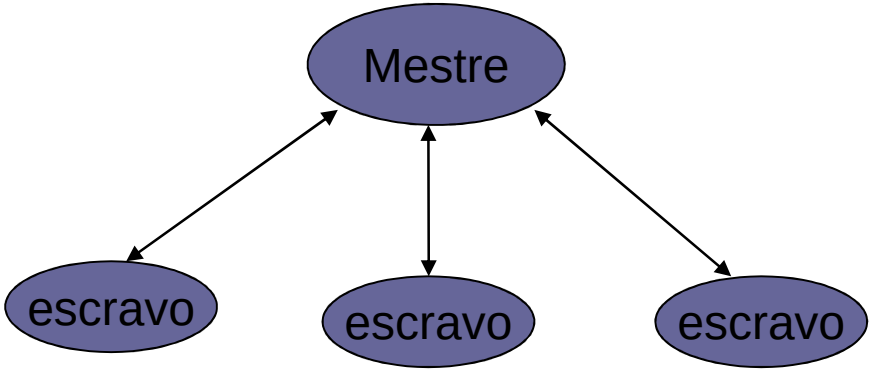
- Como o paralelismo da aplicação é caracterizado
  - **SPMD (Single Program Multiple Data) - Paralelismo de dados -** execução de uma mesma atividade sobre diferentes partes do conjunto de dados. Dados determinam o paralelismo e a forma como os cálculos devem ser distribuídos
    - Réplica, mas precisa atenção com comunicação entre processos
    - Mais fácil balanceamento por ser a mesma atividade
  - **MPMD (Multiple Program Multiple Data) - Paralelismo de tarefa -** execução paralela de diferentes atividades sobre conjuntos distintos de dados. Identificação das atividades paralelas e como são distribuídas pelos recursos disponíveis
    - Balanceamento de carga é mais difícil do que em SPMD
    - Processos comunicam menos durante as operações

Modelos de Aplicação

- As aplicações são modeladas usando um grafo que relaciona as tarefas e trocas de dados
  - Nós: tarefas
  - Arestas: trocas de dados (comunicação e/ou sincronização)
- Modelos básicos
  - Mestre/escravo
  - Pipeline
  - Divisão e conquista
  - Fases paralelas

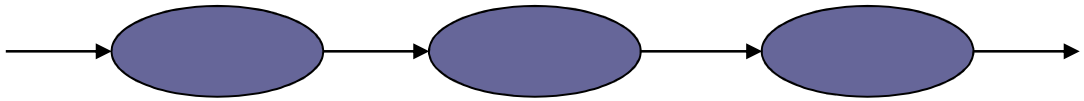
Modelos de Aplicação

- Mestre/escravo
  - Mestre executa as tarefas essenciais do programa e divide o resto entre processos escravos
  - Escalonamento centralizado – gargalo 1-N
  - Maior tolerância a falhas: um escravo para, não todo processamento



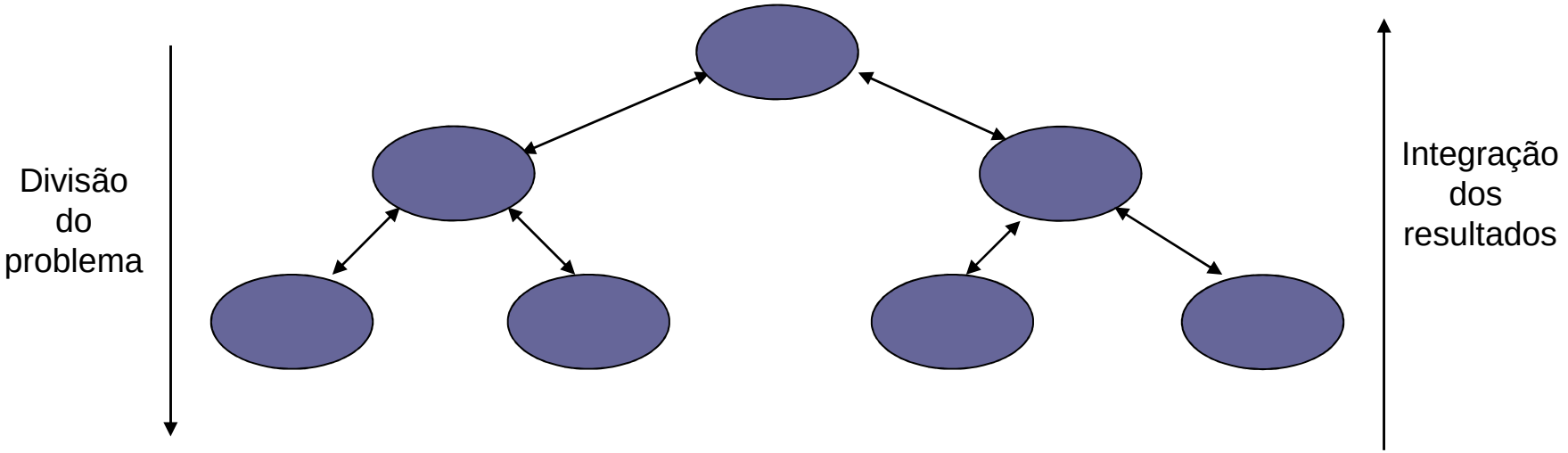
Modelos de Aplicação

- Pipeline
  - Encadeamento de processos
  - Pipeline virtual (rede não muda)
  - Fluxo contínuo de dados
  - Sobreposição de comunicação e computação
  - Não tolerante a falhas – 1 para, todos param



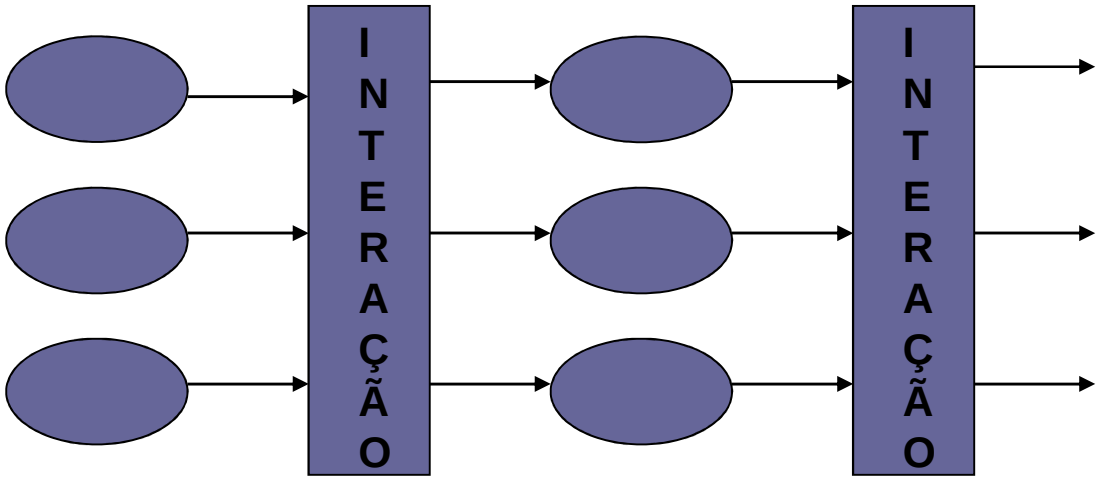
Modelos de Aplicação

- Divisão e conquista
  - Processos organizados em uma hierarquia (pai e filhos)
  - Processo pai divide trabalho e repassa uma fração deste ao seus filhos
  - Integração dos resultados de forma recursiva
  - Dificuldade de balanceamento de carga na divisão das tarefas



Modelos de Aplicação

- Fases paralelas
  - Etapas de computação e sincronização
  - Problema de balanceamento devido ao sincronismo
    - Processos que acabam antes
  - Overhead de comunicação
    - Toda comunicação é realizada ao mesmo tempo



Problema

- Como paralelizar a ordenação de um grande vetor?
  - Mestre/escravo
  - Pipeline
  - Fases paralelas
  - Divisão e conquista

Mestre/Escravo

- O mestre envia partes do vetor para cada escravo, que retorna este ordenado e espera por um novo pedaço do vetor
  - Muita comunicação
  - Não é um bom modelo para o caso

Pipeline

- Cada processador faz uma troca simples, sendo necessário para um vetor de tamanho n, n-1 processadores.
  - Só 1 vetor, só uma operação
  - Não é o modelo ideal

Fases Paralelas

- Cada processo inicia com uma parte do vetor, ordenam e interagem com os outros processos a fim de realizar trocas entre os limites de cada pedaço do vetor
  - Faz ordenação local, e mando o meu maior e recebo o menor do vizinho (% para vetores grandes)
  - Condição parada:
    - meu maior é menor do que o menor do meu vizinho para todos processadores?

Divisão e Conquista

- Subdivisão do vetor em partes menores a serem ordenadas por outros processos
  - Vai subindo na árvore intercalando o vetor preordenado dos filhos

MPI

- Message Passing Interface
- Surgiu em 1992 para padronizar as bibliotecas de troca de mensagens
  - Estável
  - Eficiente
  - Portável
- Biblioteca de funções para C, Fortran e C++ executando em máquinas MIMD com organização de memória NORMA
- 129 funções com vários parâmetros e variantes
- Conjunto fixo de processos é criado na inicialização do programa (mpi 2 dinâmico)
- Modelo de programação usual é SPMD

Funções básicas - Inicialização

- MPI_Init(&argc, &argv)

  Inicializa uma execução MPI, é responsável por copiar o código em todos processadores
- MPI_Finalize()

  Termina uma execução MPI
- Exemplo

```
#include <stdio.h>
#include “mpi.h“
Main(int argc, char** argv)
{
MPI_Init(&argc, &argv);
...
MPI_Finalize();
}
```

Funções Básicas - Inicialização

- Paralelismo se da de forma implícita
- Não existem funções para criação e terminação de processos
- Identificação dos processos é fundamental para comunicação
- Communicators viabilizam a comunicação de grupo de processos e a programação modular

Funções básicas - Inicialização

- MPI_Comm_rank(comm, pid);
  - Determina o identificador do processo corrente
  - IN: comm communicator(handle)
  - Out: pid: identificador do processo no grupo comm (int)
- MPI_Comm_size(comm, size);
  - Determina o número de processos em uma execução
  - IN: comm communicator(handle)
  - OUT: size: número de processos no grupo comm (int)

Funções básicas - Inicialização

```
#include <stdio.h>
#include “mpi.h“
Main(int argc, char** argv)
{
int mypid;
int np;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &mypid);
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Finalize();
}
```

Funções básicas - Comunicação

- Comunicação ponto-a-ponto
  - Funções MPI_Send e MPI_Rcv
  - Bloqueante ou não bloqueante
- Comunicação coletiva
  - Outras funções de broadcast e concentração de informações

Funções básicas - Comunicação

- MPI_Send(buf, count, datatype, dest, tag, comm)

  IN buf: endereço do buffer de envio
  IN count: número de elementos a enviar
  IN datatype: tipo dos elementos a enviar
  IN dest: identificador do processo de destino
  IN tag: (etiqueta) da mensagem
  IN comm: communicator (handle)

Funções básicas - Comunicação

- MPI_Recv(buf, count, datatype, source, tag, comm,status)

  OUT buf: endereço do buffer de recebimento
  IN count: número de elementos a receber
  IN datatype: tipo dos elementos a receber
  IN dest: identificador do processo de origem
  IN tag: (etiqueta) da mensagem
  IN comm: communicator (handle)

Funções básicas - Comunicação

```
#include <stdio.h>
#include “mpi.h“
main(int argc, char** argv)
{
      int mypid;
      int np;
      int source;
      int dest;
      int tag=50;
      char message[100];
      MPI_Status status;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &mypid);
      MPI_Comm_size(MPI_COMM_WORLD, &np);
```

Funcoes básicas - Comunicacao

```
If (mypid!=0)
{
        sprintf(message, “greetings from process %d!“, mypid);
       dest=0;
       MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
else
{
       for (source=1; source< np; source++)
       {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
       printf(“%s\n“, message);
       }
}
MPI_Finalize();
}
```

Funções básicas – sincronização

- Sincronização implícita
  - Através de comunicação bloqueante
- Sincronização explícita e global
  - Uso do isend e irecv
  - Através de barreiras
  - MPI_Barrier(comm)

    IN comm: communicator (handle)

MPI - Conclusões

- Existem diversas implementações (MPICH, LAM)
  - Geralmente com compilador próprio
  - Ferramenta de execução própria
- Ferramenta de depuração e editores
  - Debuggers
  - Profilers
  - Sistema hospedeiro
- É atualmente o mais utilizado em programação paralela (existem outros como o PVM) e vem diminuindo com o aumento do uso de OpenMP (memória compartilhada)
  - Aumento de maquinas multicore