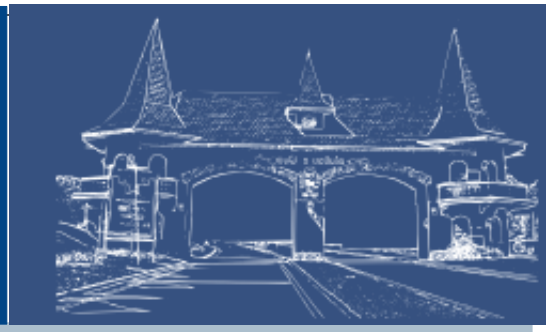
ERAD 2015
XV Escola Regional de Alto Desempenho
22 a 24 de abril de 2015
Gramado - RS



Programação Paralela em Memória Compartilhada e Distribuída

Prof. Claudio Schepke
claudioschepke@unipampa.edu.br

Prof. João V. F. Lima
jvlima@inf.ufsm.br

(baseado em material elaborado por professores de outras ERADs)

Apresentação

- Ministrante: Claudio Schepke
- Formação:
  - Graduação (UFSM - 2005), Mestrado (UFRGS - 2007) e Doutorado (UFRGS 2012) em Ciência da Computação
  - Doutorado sanduíche – TU-Berlin/Alemanha (2010/2011)
- Atividades:
  - Professor na Setrem - Três de Maio (2007-2008 e 2012)
  - Professor na Universidade Federal do Pampa (Unipampa) – Campus Alegrete (2012-atual)
    - Graduações em Ciência da Computação e Engenharia de Software + 5 cursos de Engenharia
- Áreas de atuação: Processamento Paralelo e Distribuído
  - Aplicações de Alto Desempenho
  - Programação Paralela

Apresentação

- Autor: João V. F. Lima
- Formação:
  - Graduação (UFSM - 2007), Mestrado (UFRGS - 2009) e Doutorado (UFRGS 2014) em Ciência da Computação com co-tutela (Universidade de Grenoble).
- Atividades:
  - Professor na Universidade Federal de Santa Maria UFSM – departamento de Linguagens e Sistemas de Computação (2014-atual)
  - Desenvolvedor da ferramenta Xkaapi.
- Áreas de atuação: Processamento de Alto Desempenho
  - Programação Paralela
  - Aplicações Paralelas

Sumário

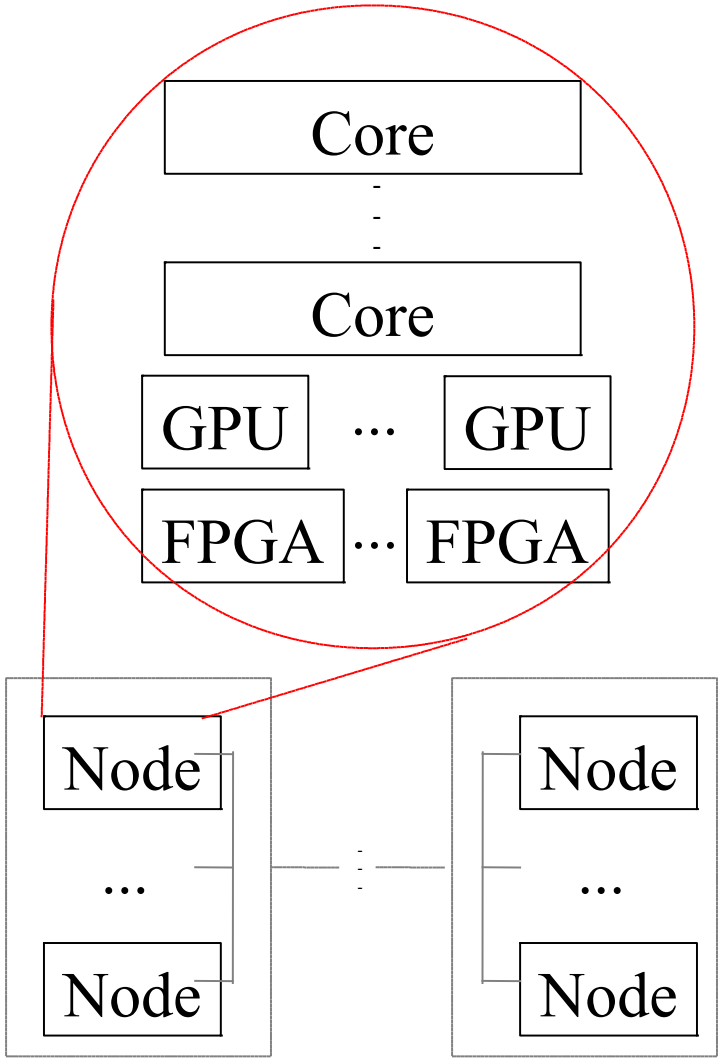
- Introdução
- Arquiteturas Paralelas
- Modelagem e Desenvolvimento de Aplicações Paralelas
- Programação em Memória Compartilhada com OpenMP
- Programação em Memória Distribuída com MPI
- Programação Híbrida com MPI e OpenMP
- Conclusão

Introdução

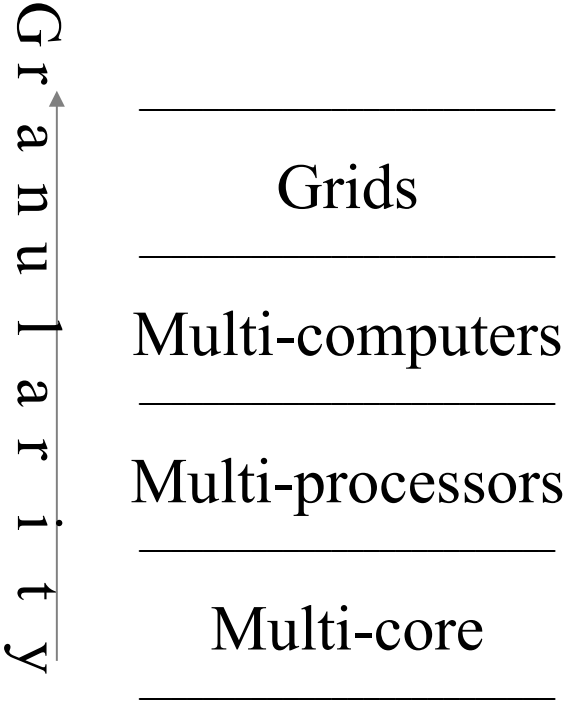
Introdução

- **Programação paralela** é a divisão de uma determinada aplicação em partes, de maneira que essas partes possam ser executadas simultaneamente, por vários elementos de processamento.
- Os elementos de processamento devem cooperar entre si utilizando primitivas de comunicação e sincronização, realizando a quebra do paradigma de execução seqüencial do fluxo de instruções.
- Objetivos
  - Alto Desempenho (Exploração Eficiente de Recursos)
  - Tolerância a falhas

Motivação



Diversos níveis de concorrência em hardware



Motivação

Solução de aplicações complexas (científicas, industriais e militares)

- Meteorologia
- Prospeção de petróleo
  - Análise de local para perfuração de poços de petróleo
- Simulações físicas
  - Aerodinâmica; energia nuclear
- Matemática computacional
  - Análise de algoritmos para criptografia
- Bioinformática
  - Simulação computacional da dinâmica molecular de proteínas

Top500 - http://www.top500.org

Desafios

- SpeedUp
  - Fator de aceleração
  - Existe um limite para o número de processadores

SpeedUp = Tempo Sequencial / Tempo Paralelo

- Amdahl´s Law
  - Determina o potencial de aumento de velocidade a partir da porcentagem paralelizável do programa
  - Considera um programa como uma mistura de partes sequenciais e paralelas

$$SpeedUp = \frac{1}{\frac{\%paralelo}{numprocs} + \%sequencial}$$

Desafios

- Dificuldade na conversão da aplicação sequencial em paralela
- Divisão adequada da computação entre os recursos
  - Balanceamento de carga
- Complexidade de implementação
  - Particionamento de código e dados
- Custo de coordenação (sincronização)
  - Necessidade de troca de informação entre processos
- Dificuldade de depuração

Desafios

- Necessidade de conhecimento da máquina
  - Código dedicado a máquina paralela
  - Baixa portabilidade
  - Influência
    - Paradigma utilizado para comunicação
    - Modelagem do problema

Arquiteturas Paralelas
Modelagem e Desenvolvimento de Aplicações Paralelas

Modelagem

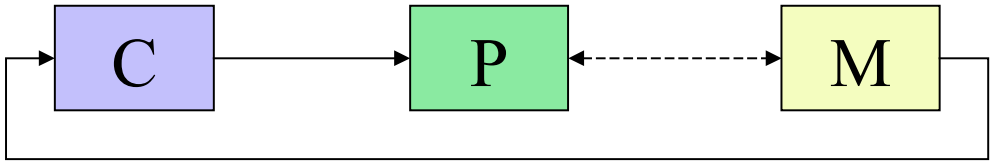
- Podemos dividir basicamente em
  - **Modelos de máquina:** descrevem as características das máquinas
  - **Modelos de programação:** permitem compreender aspectos ligados a implementação e desempenho de execução dos programas
  - **Modelos de aplicação:** representam o paralelismo de um algoritmo
- Vantagens
  - Permite compreender o impacto de diferentes aspectos da aplicação na implementação de um programa paralelo, tais como:
    - Quantidade de cálculo envolvido total e por atividade concorrente
    - Volume de dados manipulado
    - Dependência de informações entre as atividades em execução

Modelos de máquinas

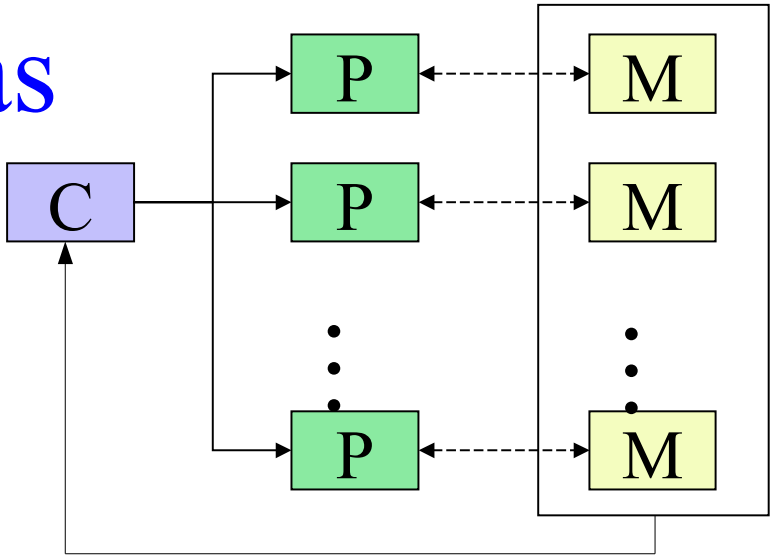
- Descreve as principais características de uma máquina;
- Primeira proposta feita por Flynn em 1966:
  - Classifica máquina de acordo com:
    - Fluxo de instruções;
    - Fluxo de dados.

	SD (Single)	**MD (Multiple Data)**
SI (Single Instruction	SISD Máquinas von Neumann convencionais	SIMD Máquinas *Array*
MI (Multiple Instruction)	MISD Sem representante	MIMD Multiprocessadores e Multicomputadores

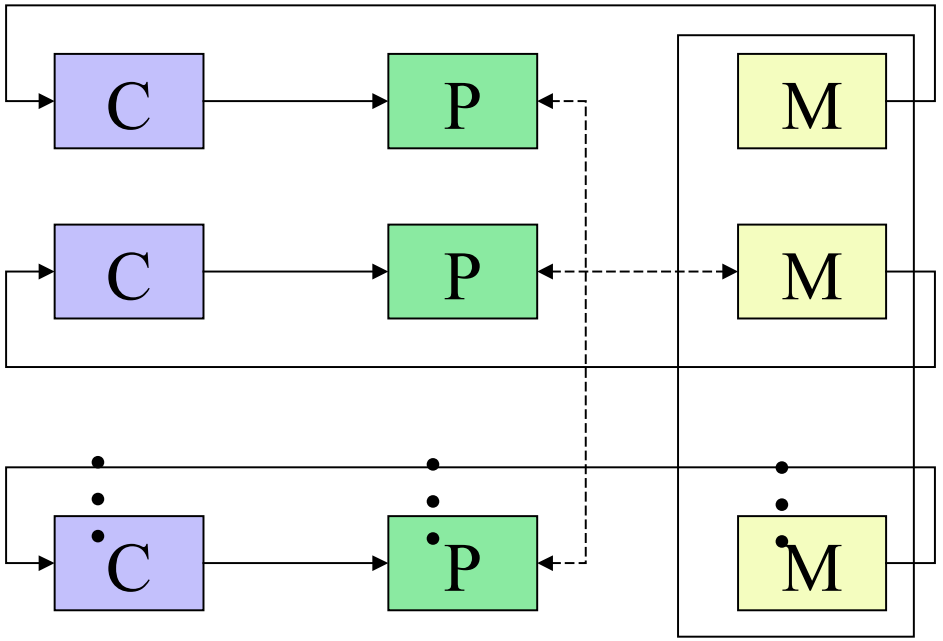
Modelos de máquinas



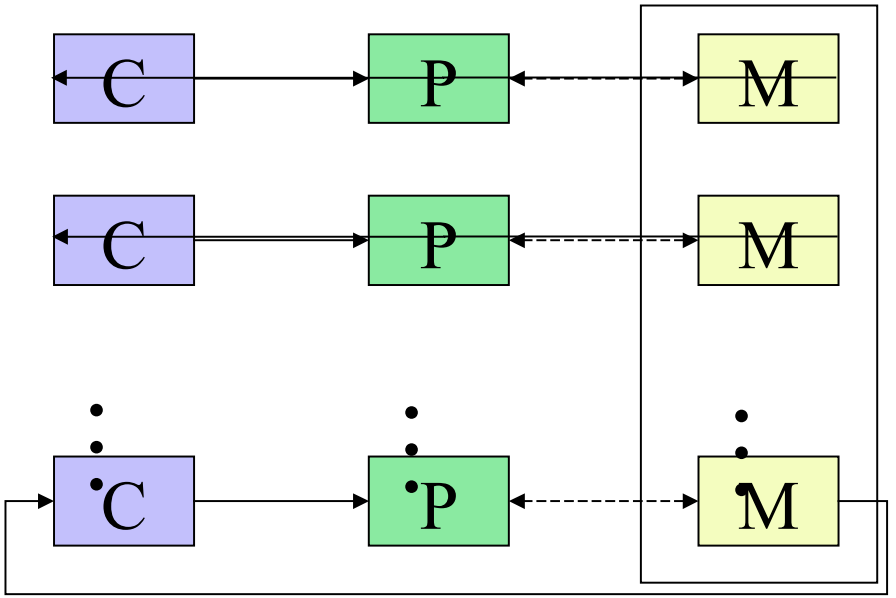
SISD



SIMD



MISD

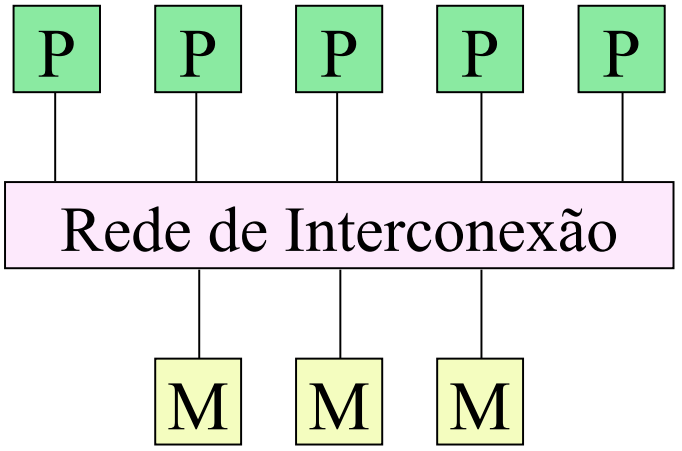


MIMD

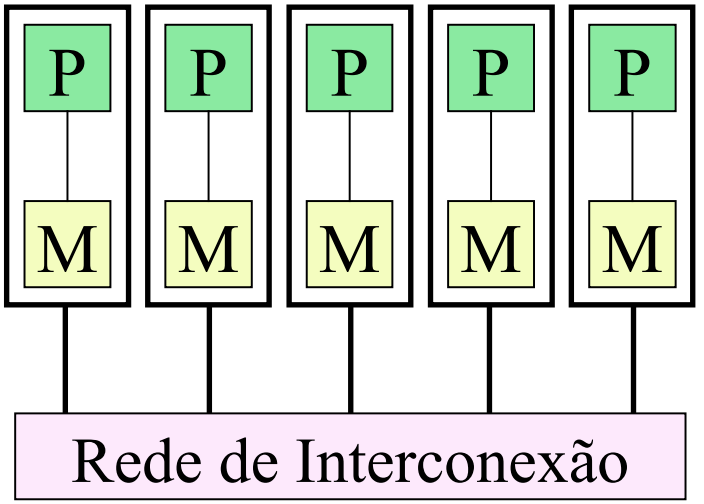
(Figuras do curso Arquiteturas Paralelas – ERADs - do Prof. Cesar De Rose)

Modelos de máquinas

- Classificação segundo o compartilhamento de memória:
  - Multiprocessadores;
  - Multicomputadores.



Multiprocessador

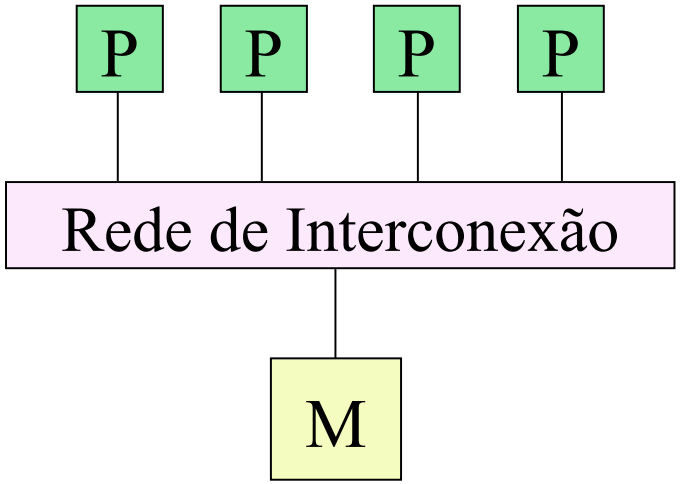


Multicomputador

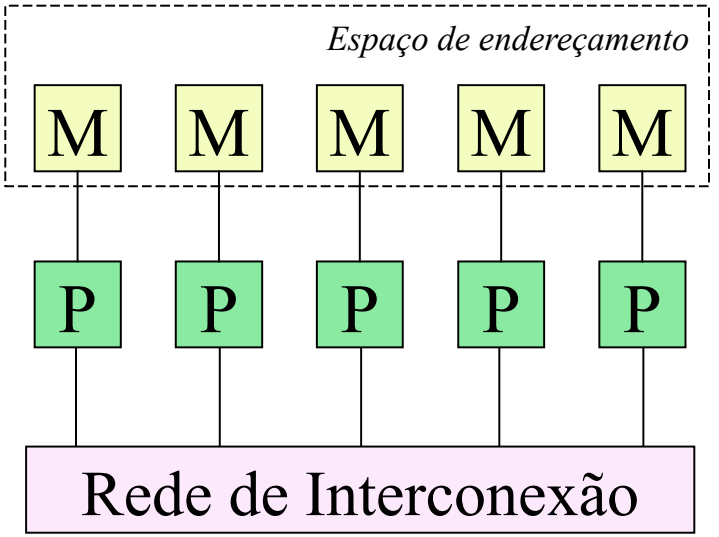
(Figuras do curso Arquiteturas Paralelas – ERADs - do Prof. Cesar De Rose)

Modelos de máquinas

- Multiprocessadores (tipo de acesso à memória):
  - UMA;
  - NUMA.



UMA



NUMA

(Figuras do curso Arquiteturas Paralelas – ERADs - do Prof. Cesar De Rose)

Modelos de programação

- Granulosidade (ou granularidade)
  - Relação entre o tamanho de cada tarefa e o tamanho total do programa (ou a razão entre computação e comunicação)
  - Pode ser alta (grossa), média, baixa (fina)
  - Indica o tipo de arquitetura mais adequado para executar a aplicação: procs vetoriais (fina), SMP (média), clusters (grossa)

- **Grossa**
  - processamento > comunicação
  - menor custo de sincronização

- **Fina**
  - processamento < comunicação
  - maior frequencia de comunicação

Modelos de programação

- Paralelismo de dados
  - Execução de uma mesma atividade sobre diferentes partes de um conjunto de dados
  - Os dados determinam a concorrência da aplicação e a forma como o cálculo deve ser distribuído na arquitetura
- Paralelismo de tarefa
  - Execução paralela de diferentes atividades sobre conjuntos distintos de dados
  - Identificação das atividades concorrentes da aplicação e como essas atividades são distribuídas pelos recursos disponíveis

Modelos de programação

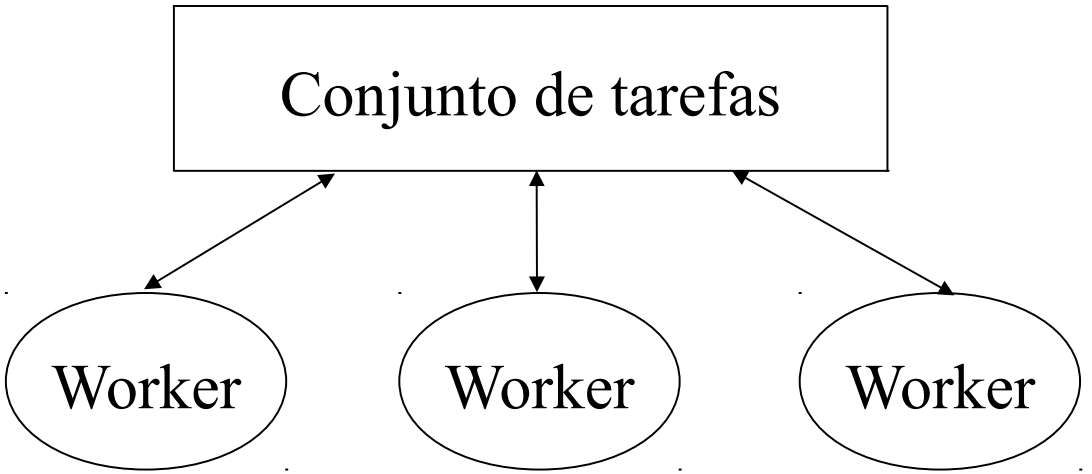
- Memória compartilhada
  - As tarefas em execução compartilham um mesmo espaço de memória
  - Comunicação através do acesso a uma área compartilhada
- Troca de mensagens
  - Não existe um espaço de endereçamento comum
  - Comunicação através de troca de mensagens usando a rede de interconexão.

Modelos de aplicação

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

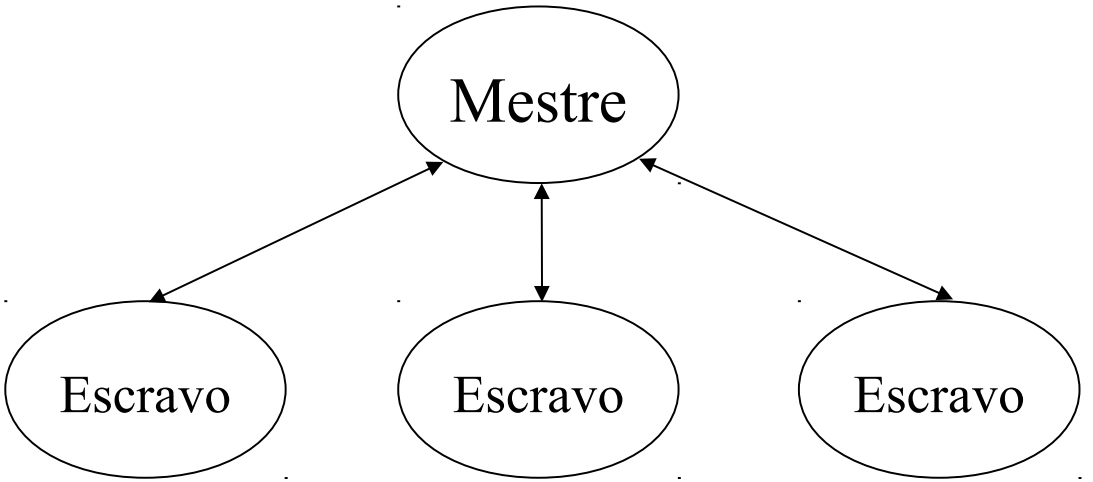
Modelos de aplicação

- Workpool
  - Tarefas disponibilizadas em uma estrutura de dados global (memória compartilhada)
  - Sincronização no acesso à área compartilhada
  - Balanceamento de carga



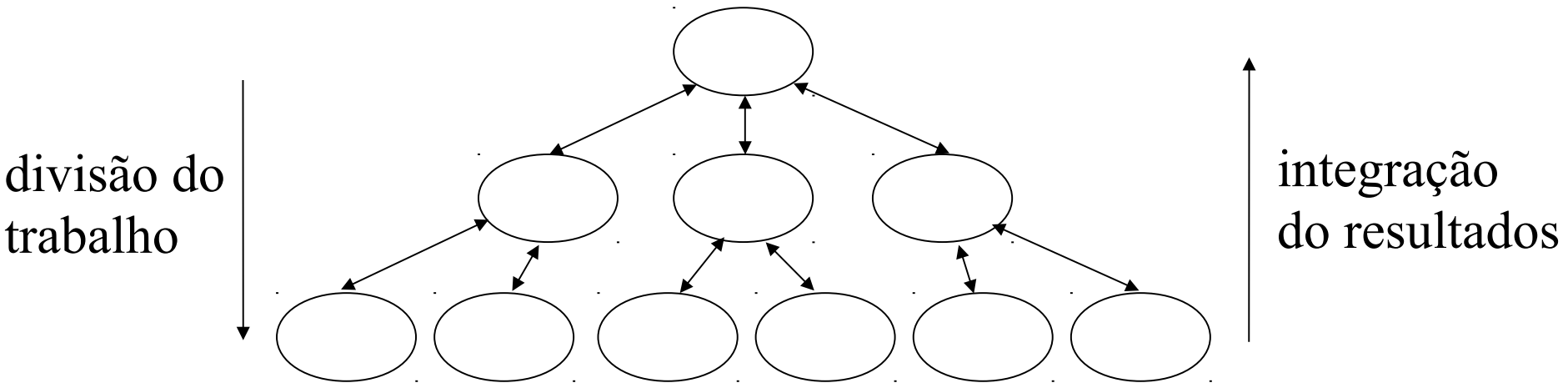
Modelos de aplicação

- Mestre / Escravo
  - Mestre escalona tarefas entre processos escravos
  - Escalonamento centralizado – gargalo
  - Maior tolerância a falhas



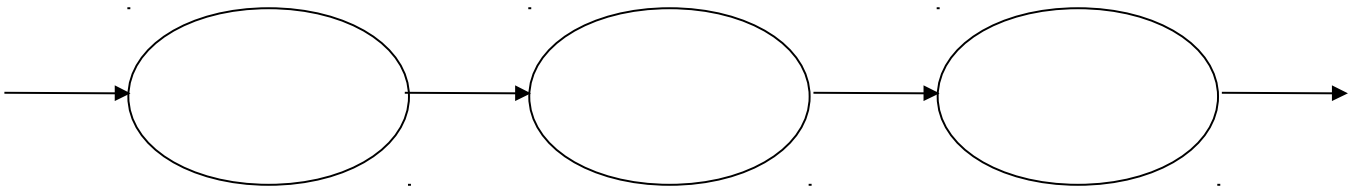
Modelos de aplicação

- Divisão e conquista (Divide and Conquer)
  - Processos organizados em uma hierarquia (pai e filhos)
  - Processo pai divide trabalho e repassa uma fração deste aos seus filhos
  - Integração dos resultados de forma recursiva
  - Distribuição do controle de execução das tarefas (processos pai)



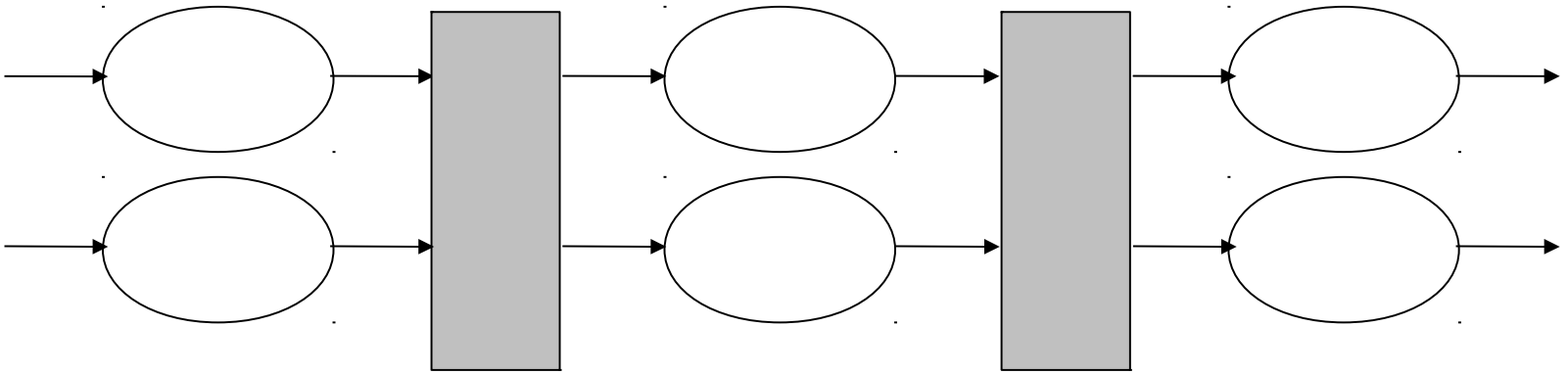
Modelos de aplicação

- Pipeline
  - Pipeline virtual
  - Fluxo contínuo de dados
  - Sobreposição de comunicação e computação



Modelos de aplicação

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



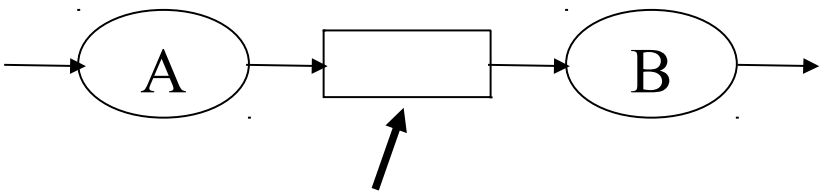
Programação em Memória Compartilhada com OpenMP

Programação paralela usando memória compartilhada

- A comunicação entre os processos é realizada através de acessos do tipo load e store a uma área de endereçamento comum.
- Para utilização correta da área de memória compartilhada é necessário que os processos coordenem seus acesso utilizando primitivas de sincronização.

Programação paralela usando memória compartilhada

- **Execução sequencial de tarefas**
  - O resultado de uma tarefa é comunicado a outra tarefa através da escrita em uma posição de memória compartilhada
  - Sincronização implícita, isto é, uma tarefa só é executada após o término da tarefa que a precede



área de memória compartilhada

Programação paralela usando memória compartilhada

- **Execução concorrente de tarefas**
  - Não existe sincronismo implícito!
  - Seção crítica: conjunto de instruções de acesso a uma área de memória compartilhada acessada por diversos fluxos de execução
  - É responsabilidade do programador fazer uso dos mecanismos de sincronização para garantir a correta utilização da área compartilhada para comunicação entre os fluxos de execução distintos
  - Mecanismos mais utilizados para exclusão mútua no acesso a memória:
    - Mutex
    - Operações de criação e bloqueio dos fluxos de execução (create e join)

Multiprogramação leve

- Multithreading: permite a criação de vários **fluxos de execução (threads)** no interior de um processo
- Os recursos de processamento alocados a um processo são compartilhados por todas suas threads ativas

As threads compartilham dados e se comunicam através da memória alocada ao processo

Implementação de threads

- 1:1 (one-to-one)
  - Threads sistema (ou kernel)
  - O recurso de threads é suportado pelo SO
  - As threads possuem o mesmo direito que processos no escalonamento do processador
  - **Vantagens**
    - Melhor desempenho em arquitetura multiprocessada - cada thread de uma aplicação pode ser escalonada para um processador diferente (maior paralelismo)
    - Bloqueio de uma thread (E/S) não implica no bloqueio de todas as threads do processo
  - **Desvantagens**
    - Maior overhead de gerência pelo SO
    - Distribuição de CPU desigual entre processos – processo com mais threads recebe mais CPU

Implementação de threads

- N:1 (many-to-one)
  - Threads em nível de usuário (threads usuário)
  - O recurso de threads é viabilizado através de bibliotecas quando não fornecido pelo SO
  - Escalonamento das threads é realizado dentro do processo quando este tiver acesso ao processador
  - **Vantagens**
    - Baixo overhead de manipulação
    - Permite a criação de um número maior de threads
  - **Desvantagens**
    - Se uma thread fica bloqueada, todas as outras threads do mesmo processo também serão bloqueadas
    - Ineficiente em máquinas multiprocessadas – processo pesado é escalonado para somente um processador

Implementação de threads

- M:N (many-to-many)
  - Combinação de threads sistema e threads usuário
  - Cada processo pode conter M threads sistema, cada uma com N threads usuário
  - SO escalona as M threads sistema, e a biblioteca de threads escalona as N threads usuário internamente
  - **Vantagens**
    - Benefício da estrutura mais leve das threads usuário
    - Benefício do maior paralelismo das threads sistema
  - **Desvantagem**
    - Complexidade de implementação

OpenMP

OpenMP

- Multi-plataformas com memória compartilhada
- Linguagens de programação C, C++ e Fortran
- Portável e modelo escalável
- Acessos concorrentes implícitos

OpenMP

- Estruturas de controle paralelo
  - Criação de threads através da diretiva `parallel`

```
#include <stdio.h>
int main(void)
{
        #pragma omp parallel
        printf("Hello, world.\n");
        return 0;
}
```

OpenMP

- Variáveis de ambiente
  - `OMP_SCHEDULE`
  - `OMP_NUM_THREADS`
    - `export OMP_NUM_THREADS=2`
- Funções em tempo de execução

```
#include <stdio.h>
int main(void)
{
        #pragma omp parallel
        printf("%d Hello, world.\n",
omp_get_thread_num());

        return 0;
}
```

OpenMP

- Variáveis de ambiente
- Funções em tempo de execução
  - `omp_get_thread_num()`
  - `omp_set_num_threads(int n)`
  - `omp_get_num_threads()`

```
#include <stdio.h>
int main(void)
{
        omp_set_num_threads(7);
        #pragma omp parallel
        printf("%d Hello, world.\n",
omp_get_thread_num());
        return 0;
}
```

OpenMP

- Exemplo do uso de funções

```
#include <stdio.h>
int main(void) {
        printf("Numero de threads = %d\n",
                        omp_get_num_threads());
        omp_set_num_threads(7);
        printf("Numero de threads = %d\n",
                        omp_get_num_threads());
        #pragma omp parallel
        printf("%d Hello, world.\n",
                        omp_get_thread_num());
        #pragma omp parallel
        if(omp_get_thread_num() == 0)
                printf("Numero de threads: %d\n",
                   omp_get_num_threads());
        return 0;
}
```

OpenMP

- Compartilhamento de trabalho
  - Laços paralelos – `parallel for`
  - `do, parallel do` e `section`

```
#include <stdio.h>
#define N 100000
int main(int argc, char *argv[]) {
        int i, v[N];
        #pragma omp parallel for
        for (i = 0; i < N; i++) {
                v[i] = 2 * i;
                printf("%d ", v[i]);
        }
        return 0;
}
```

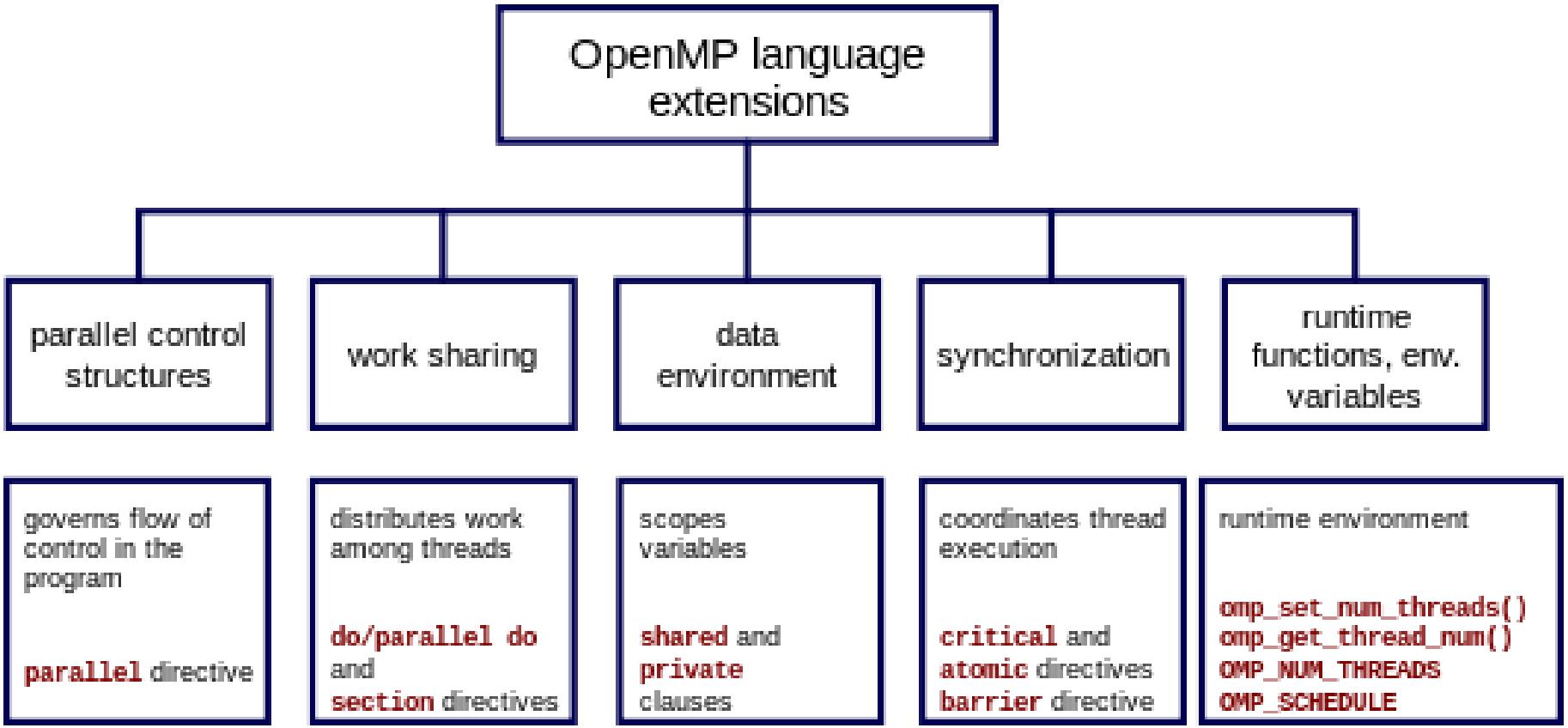
OpenMP

- Ambiente de dados: variáveis e escopo
  - `shared` e `private`
- Sincronização
  - `critical/atomic` e `barrier`

OpenMP

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
    int id, nthreads;
    #pragma omp parallel private(id)
    {
        id = omp_get_thread_num();
        printf("Hello World from thread %d\n", id);
        #pragma omp barrier
        if (id == 0) {
            nthreads = omp_get_num_threads();
            printf("There are %d threads\n",nthreads);
        }
    }
    return 0;
}
```

OpenMP


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and
section directives
scopes variables
shared and
private
clauses
coordinates thread execution
critical and
atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

Programação em Memória Distribuída com MPI

Programação Paralela
Troca de Mensagens

- Opções de programação
  - Linguagem de programação paralela (específica)
    - Occam (Transputer)
  - Extensão de linguagens de programação existentes
    - CC++ (extensão de C++)
    - Fortran M
    - Geração automática usando anotações em código e compilação (FORTRAN)
  - Linguagem padrão com biblioteca para troca de mensagens
    - MPI (Message Passing Interface)
    - PVM (Parallel Virtual Machine)

Linguagem padrão com biblioteca para troca de mensagens

- Descrição explícita do paralelismo e troca de mensagens entre processos
- Métodos principais
  - Criação de processos para execução em diferentes computadores
  - Troca de mensagens (envio e recebimento) entre processos
  - Sincronização entre processos

Criação de processos

- Mapeamento de um processo por processador
- Criação estática de processos
  - Processos especificados antes da execução
  - Número fixo de processos
  - Mais comum com o modelo SPMD
- Criação dinâmica de processos
  - Processos criados durante a execução da aplicação (spawn)
  - Destruição também é dinâmica
  - Número de processos variável durante execução
  - Mais comum com o modelo MPMD

SPMD e MPMD

- SPMD (*Single Program Multiple Data*)
  - Existe somente um programa
  - O mesmo programa é executado em diversas máquinas sobre um conjunto de dados distinto

- MPMD (*Multiple Program Multiple Data*)
  - Existem diversos programas
  - Programas diferentes são executados em máquinas distintas
  - Cada máquina possui um programa e conjunto de dados distinto

Troca de mensagens

- Primitivas send e receive
  - Comunicação síncrona (bloqueante)
    - Send bloqueia emissor até receptor executar receive
    - Receive bloqueia receptor até emissor enviar mensagem
  - Comunicação assíncrona (não bloqueante)
    - Send não bloqueia emissor
    - Receive pode ser realizado durante execução
      - Chamada é realizada antes da necessidade da mensagem a ser recebida, quando o processo precisa da mensagem, ele verifica se já foi armazenada no buffer local indicado

Troca de mensagens

- Seleção de mensagens
  - Filtro para receber uma mensagem de um determinado tipo (message tag), ou ainda de um emissor específico
- Comunicação em grupo
  - **Broadcast**
    - Envio de mensagem a todos os processos do grupo
  - **Gather/scatter**
    - Envio de partes de uma mensagem de um processo para diferentes processos de um grupo (distribuir), e recebimento de partes de mensagens de diversos processos de um grupo por um processo (coletar)

Sincronização

- Barreiras
  - Permite especificar um ponto de sincronismo entre diversos processos
  - Um processo que chega a uma barreira só continua quando todos os outros processos do seu grupo também chegam a barreira
  - O último processo libera todos os demais bloqueados

Biblioteca MPI
(*Message Passing Interface*)

MPI

- MPI – *Message Passing Interface*
- Padrão definido pelo MPI Fórum para criação de programas paralelos (versão 1.0 em 1994).
- Atualmente está na versão 3.0 chamada de MPI-3.
- Possui diversas implementações (MPICH, OpenMPI).

MPI

- Modelo de programação SPMD (Single Program - Multiple Data).
- Modelo de comunicação usando troca de mensagens.
- Execução a partir de um único nó (hospedeiro).
- Utiliza uma lista de máquinas para disparar o programa.

Características

- Possui cerca de 125 funções para programação.
- Implementado atualmente para as seguintes linguagens: C, C++ e Fortran.
- **Bibliotecas matemáticas**: BLACS, SCALAPACK, etc.
- **Biblioteca gráfica**: MPE_Graphics.

Comandos

- mpicc, mpiCC, mpif77
  - compiladores MPI para linguagens C, C++ e FORTRAN77
- mpirun
  - dispatcher de programas paralelos em MPI

- Exemplo de sessão usando MPI

```
$ mpicc -o hello hello.c
$ mpirun -np 4 ./hello
$ mpirun -hostfile hostfile.txt -np
  4 ./hello
```

Diretivas Básicas

```
int MPI_Init(int *argc, char *argv[])
```

- Inicializa um processo MPI, e estabelece o ambiente necessário para sua execução. Sincroniza os processos para o início da aplicação paralela.

```
int MPI_Finalize()
```

- Finaliza um processo MPI. Sincroniza os processos para o término da aplicação paralela.

```c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    printf("Hello World!\n");

    MPI_Finalize();
    return 0;
}
```

Diretivas Básicas

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

- Retorna o número de processos dentro do grupo.

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

- Identifica um processo MPI dentro de um determinado grupo. O valor de retorno está compreendido entre 0 e (número de processos)-1.

```c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char **argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello World! I'm %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Comunicação

- A comunicação pode ser bloqueante ou não bloqueante.
- Funções bloqueantes:
  - MPI_Send(), MPI_Recv()
- Funções não bloqueantes;
  - MPI_Isend(), MPI_Irecv(), MPI_Wait(), MPI_Test()

Comunicação

```
int MPI_Send(void *sndbuf, int count,
MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm)
```

- sndbuf: dados a serem enviados
- count: número de dados
- datatype: tipo dos dados
- dest: rank do processo destino
- tag: identificador
- comm: comunicador

Comunicação

```
int MPI_Recv(void *recvbuf, int count,
MPI_Datatype datatype, int source, int
tag, MPI_Comm comm, MPI_Status status)
```

- recvbuf: área de memória para receber dados
- count: número de dados a serem recebidos
- datatype: tipo dos dados
- source: processo que envio mensagem
- tag: identificador
- comm: comunicador
- status: informação de controle

```c
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv){
      int rank, size, tag, i;
      MPI_Status status;
      char msg[20];
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      if(rank == 0) {
            strcpy(msg, "Hello World!\n");
            for(i=1; i < size; i++)
                  MPI_Send(msg, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
      } else {
            MPI_Recv(msg, 20, MPI_CHAR, 0,tag, MPI_COMM_WORLD, &status);
            printf("Process %d: Message received: %s\n", rank, msg);
      }
      MPI_Finalize();
      return 0;
}
```

Comunicação

```
int MPI_Isend(void *buf, int count,
MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm, MPI_Request
*request)
```

- buf: dados a serem enviados
- count: número de dados
- datatype: tipo dos dados
- dest: rank do processo destino
- tag: identificador
- comm: comunicador
- request: identificador da transmissão

Comunicação

```
int MPI_Irecv(void *buf, int count,
MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request
*request)
```

- buf: área de dados para receber
- count: número de dados
- datatype: tipo dos dados
- source: rank do processo que enviou
- tag: identificador
- comm: comunicador
- request: identificador da transmissão

Comunicação

```
int MPI_Wait(MPI_Request *request,
MPI_Status *status)
```

- request: identificador da transmissão
- status: informação de controle

```
int MPI_Test(MPI_Request *request,
int *flag, MPI_Status *status)
```

- request: identificador da transmissão
- flag: resultado do teste
- status: informação de controle

Sincronização

int MPI_Barrier(MPI_Comm comm)

- comm: comunicador

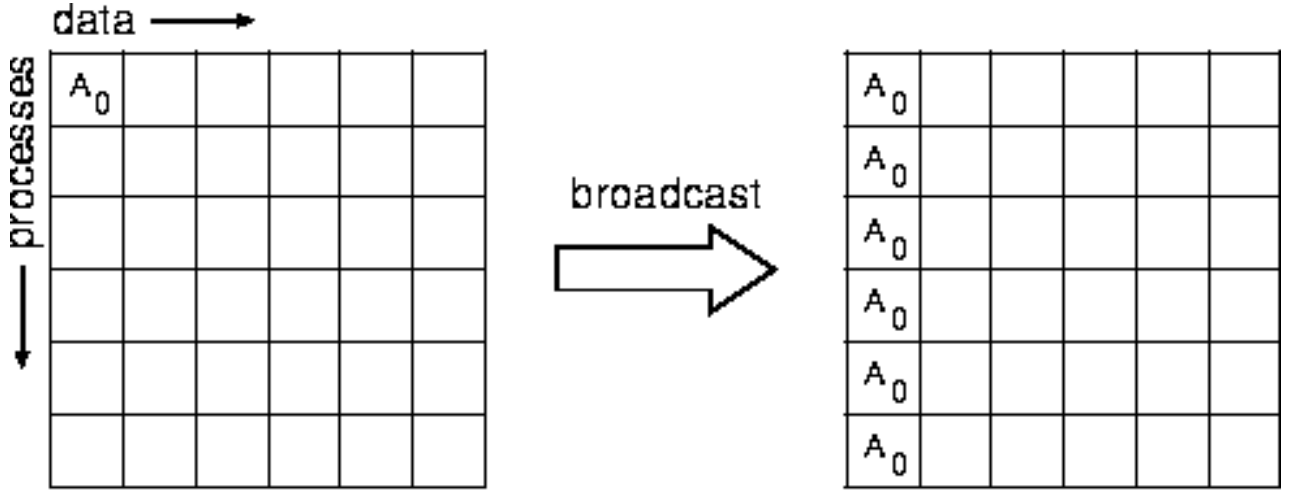
- Outras formas de sincronização:
  - Utilizando as funções de troca de mensagens bloqueantes

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv) {
	int rank, size;

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);

	printf("I'm %d of %d\n", rank, size);

	if(rank == 0) {
		printf("(%d) -> Primeiro a escrever!\n", rank);
		MPI_Barrier(MPI_COMM_WORLD);
	} else {
		MPI_Barrier(MPI_COMM_WORLD);
		printf("(%d) -> Agora posso escrever!\n", rank);
	}

	MPI_Finalize();
	return 0;
}
```

Comunicação em grupo

```
int MPI_BCast(void *buffer, int count,
MPI_Datatype datatype, int root,
MPI_Comm com)
```

- buffer: área de memória
- count: número de dados
- datatype: tipo dos dados
- root: rank do processo mestre
- com: comunicador



```c
#include "mpi.h"
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
	char message[30];
	int rank, size, length;

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);
	printf("I'm %d of %d\n",rank,size);
	if(rank == 0) {
		strcpy(message, "Hello World!");
		length = strlen(message);
		printf("%d\n", length);
	}
	MPI_Bcast(&length, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(message, length, MPI_CHAR, 0, MPI_COMM_WORLD);
	if(rank != 0)
		printf("(%d) – Received %s\n", rank, message);

	MPI_Finalize();
	return 0;
}
```

Comunicação em grupo

```
int MPI_Gather(void *sendbuf, int
sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm com)
```

- sendbuf: buffer para envio
- sendcount: número de dados a serem enviados
- sendtype: tipo dos dados
- recvbuf: buffer para recebimento
- recvcount: nro de dados para recebimento
- recvtype: tipo dos dados para recebimento
- root: processo mestre
- com: comunicador

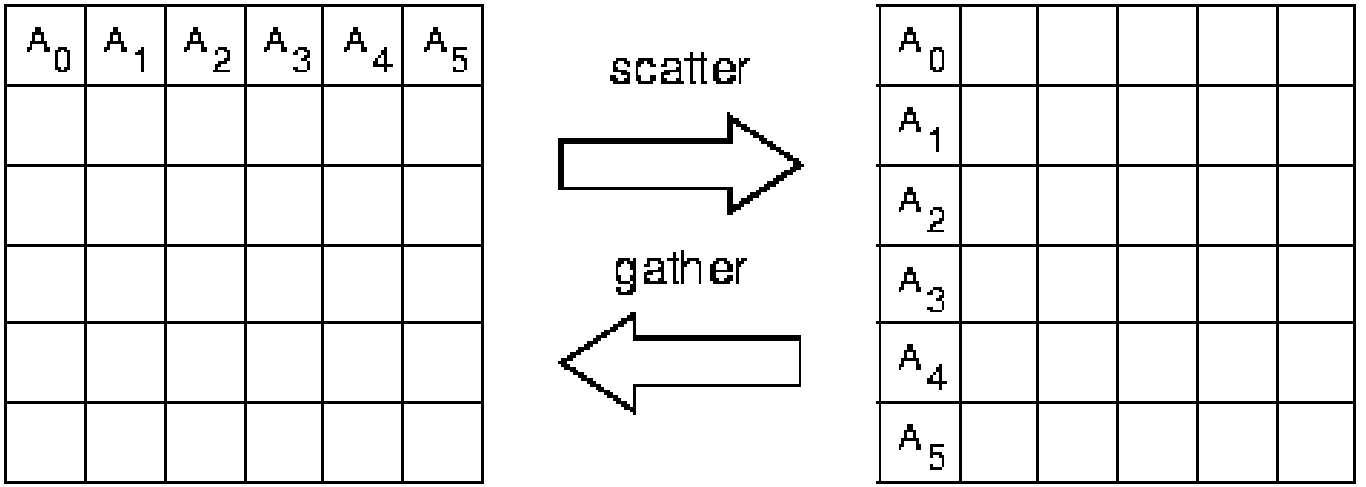
```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
      int sndbuffer, *recvbuffer;
      int rank, size, i;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);

      recvbuffer = (int *)malloc(size*sizeof(int));
      sndbuffer = rank*rank;

      MPI_Gather(&sndbuffer, 1, MPI_INT, recvbuffer, 1, MPI_INT, 0,
                                                        MPI_COMM_WORLD);
      if(rank == 0) {
            printf("(%d) – Recebi vetor: ");
            for(i=0; i<size; i++) printf("%d ", recvbuffer[i]);
            printf("\n");
      }
      MPI_Finalize();
      return 0;
}
```

Comunicação em grupo

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm com)
```

- sendbuf: buffer para envio
- sendcount: número de dados a serem enviados
- sendtype: tipo dos dados
- recvbuf: buffer para recebimento
- recvcount: número de dados para recebimento
- recvtype: tipo dos dados para recebimento
- root: processo mestre
- com: comunicador

Comunicação em grupo



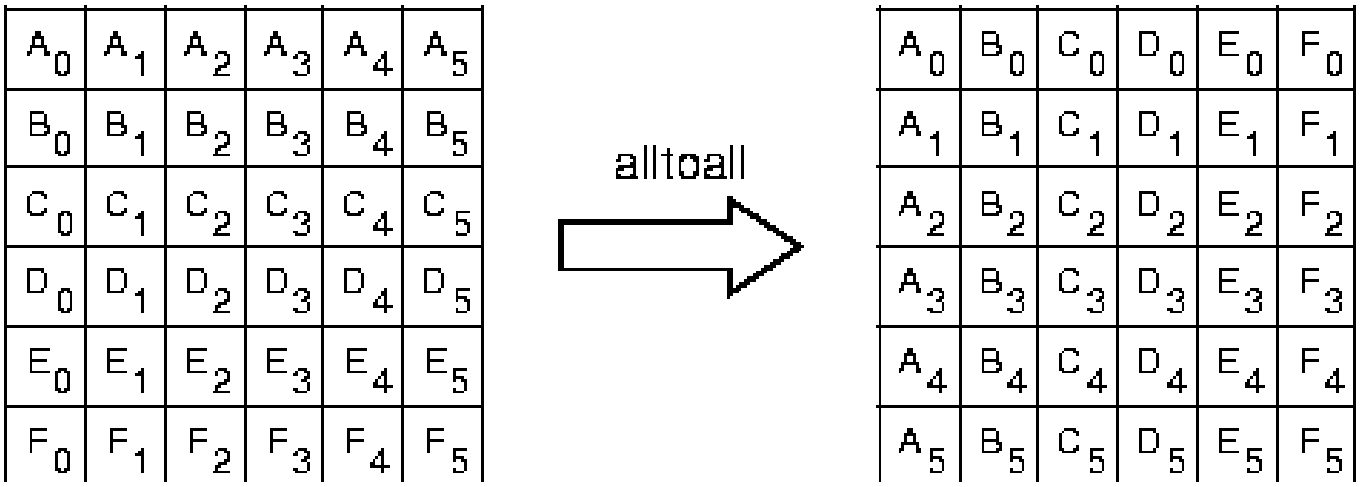
```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
      int *sndbuffer, recvbuffer;
      int rank, size, i;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);

      sndbuffer = (int *)malloc(size*sizeof(int));

      if(rank == 0)
            for(i=0; i<size; i++) sndbuffer[i] = i*i;

      MPI_Scatter(sndbuffer, 1, MPI_INT, &recvbuffer, 1,
                        MPI_INT, 0, MPI_COMM_WORLD);
      if(rank != 0)
            printf("(%d) – Received %d\n", rank, recvbuffer);

      MPI_Finalize();
      return 0;
}
```

Comunicação em grupo

```
MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm com)
```

- sendbuf: buffer para envio
- sendcount: número de dados a serem enviados
- sendtype: tipo dos dados
- recvbuf: buffer para recebimento
- recvcount: número de dados para recebimento
- recvtype: tipo dos dados para recebimento
- com: comunicador

Comunicação em grupo



```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
      int *sndbuffer, *recvbuffer;
      int rank, size, i;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);

      sndbuffer = (int *)malloc(size*sizeof(int));
      recvbuffer = (int *)malloc(size*sizeof(int));

      for(i=0; i<size; i++) sndbuffer[i] = i*i+rank;
      printvector(rank, sndbuffer);

      MPI_Alltoall(sndbuffer, 1, MPI_INT, recvbuffer, 1, MPI_INT,
                                                      MPI_COMM_WORLD);
      printvector(rank, recvbuffer);

      MPI_Finalize();
      return 0;
}
```

```
void printvector(int rank, int *buffer) {
        int size, i;
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        for (i = 0; i < size; i++)
                printf("rank %d - %d\n", rank, buffer[i]);
}
```

Comunicação em grupo

```
MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm com)
```

- Operações
  - MPI_MAX, MPI_MIN
  - MPI_SUM, MPI_PROD
  - MPI_LAND, MPI_BAND
  - MPI_LOR, MPI_BOR
  - MPI_LXOR, MPI_BXOR
    - etc

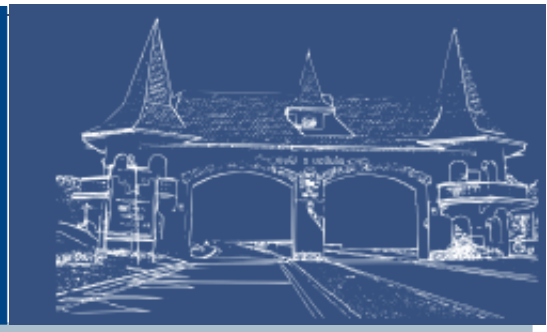
```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define TAM 100

int main( int argc, char** argv ) {
	int myrank, size;
	int i, somalocal = 0, somatotal;
	int vet[TAM];
	MPI_Init( &argc, &argv );

	MPI_Comm_rank( MPI_COMM_WORLD, &myrank ); // Quem sou ?
	MPI_Comm_size( MPI_COMM_WORLD, &size ); // Quantos somos ?
	if(myrank == 0)
		for(i = 0; i < TAM; i++)
			vet[i] = 1;
	MPI_Bcast( vet, MPI_INT, TAM, 0, MPI_COMM_WORLD );
	for( i = (TAM/size)*myrank ; i < (TAM/size) *(myrank+1) ; i++)
		somalocal += vet[i]; // Realiza as somas parciais

	MPI_Reduce(&somalocal, &somatotal, MPI_INT, 0,MPI_COMM_WORLD);
	if(myrank == 0)
		printf("Soma = %d\n", somatotal);
	return 0;
}
```

Conclusões

Conclusões

- OpenMP e MPI são amplamente utilizados no meio acadêmico e comercial
  - Permite realizar ajuste fino para obter alto desempenho
  - Opção atual para programação paralela em multiprocessadores e multicomputadores
- Tendências e desafios
  - Multicore
  - GPUs
  - Cloud Computing

ERAD 2015
XV Escola Regional de Alto Desempenho
22 a 24 de abril de 2015
Gramado - RS



Programação Paralela em Memória Compartilhada e Distribuída

Prof. Claudio Schepke
claudioschepke@unipampa.edu.br

Prof. João V. F. Lima
jvlima@inf.ufsm.br

(baseado em material elaborado por professores de outras ERADs)