Programação Paralela

- Conteúdo:
  - Introdução
  - Motivação
  - Desafios
  - Modelagem
  - Programação Paralela – Memória Compartilhada
  - Pthreads
  - Programação Paralela – Troca de Mensagens
  - MPI
  - Métricas de desempenho

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

- artigo Wall Street Journal (1988) - Attack of the Killer Micros
  - troca de supercomputadores por diversos computadores comuns
  - PC - $3000 - 0.25 MFLOP/s
  - Supercomputador - $3 milhões - 100 MFLOP/s
  - 400 PC - $1.2 milhão - 100 MFLOp/s
- Top500 - http://www.top500.org

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

Desafios

- SpeedUp
  - fator de aceleração

$$SpeedUp = \frac{tempoSequencial}{tempoParalelo}$$

  - existe um limite para o número de processadores

  - 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

$$SpeedUp = \frac{1}{\frac{\wedge paralelo}{nroProcs} + \wedge sequencial}$$

Desafios

- Custo de coordenação (sincronização)
  - necessidade de troca de informação entre processos
- Divisão adequada da computação entre os recursos
  - decomposição do problema
- Complexidade de implementação
  - particionamento de código e dados
  - problema com sincronismo
  - dependência de operações
  - balanceamento de carga
  - deadlocks (comunicação)

Desafios

- Necessidade de conhecimento da máquina
  - código dedicado a máquina paralela
  - baixa portabilidade
  - influencia
    - paradigma utilizado para comunicação
    - modelagem do problema
- Dificuldade na conversão da aplicação sequencial em paralela
  - algumas aplicações não são paralelizáveis!
- Dificuldade de depuração

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 máquina

- Classificação de Flynn
  - SISD, SIMD, MISD, **MIMD**
- Arquiteturas com memória compartilhada
  - multiprocessador
  - SMP, NUMA
- Arquiteturas com memória distribuída
  - multicomputador
  - MPP, NOW, Cluster

Modelos de programação

- Distribuição do trabalho - granulosidade (ou granularidade)
  - 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), média, baixa (fina)
  - indica o tipo de arquitetura mais adequado para executar a aplicação: procs vetoriais (fina), SMP (média), clusters (grossa)

- **grossa**
  - menor custo processamento
  - dificulta balanceamento de carga
  - processamento > comunicação
  - menor custo de sincronização

- **fina**
  - maior frequencia de comunicação
  - facilita balanceamento de carga
  - processamento < comunicação
  - alto custo de sincronização

Modelos de programação

- paralelismo de dados vs paralelismo de tarefa
  - identifica como a concorrência da aplicação é caracterizada
  - **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

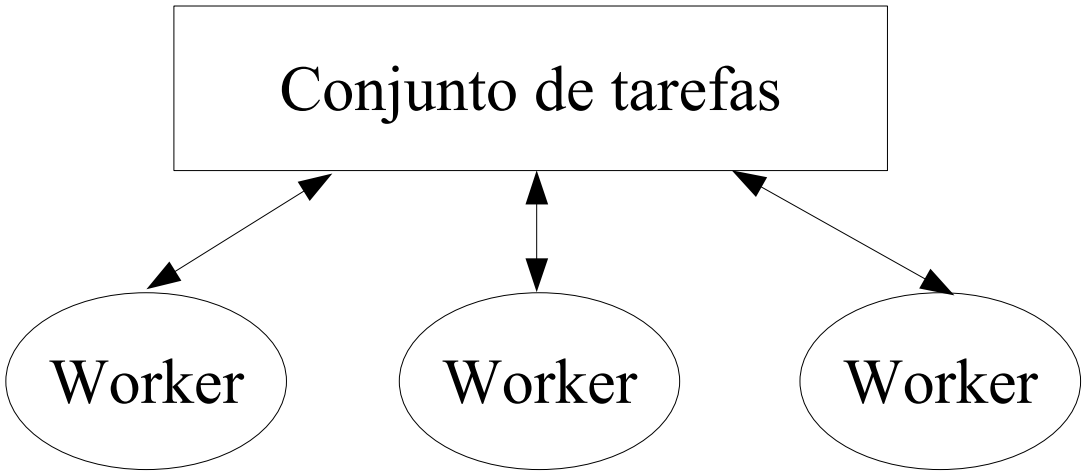
- memória compartilhada vs troca de mensagens
  - identifica como é realizado o compartilhamento de informações durante a execução
  - ligado diretamente ao tipo de arquitetura utilizado
  - **memória compartilhada**
    - as tarefas em execução compartilham um mesmo espaço de memória
    - comunicação 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 modelados 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, pipeline, divisão e conquista e fases paralelas

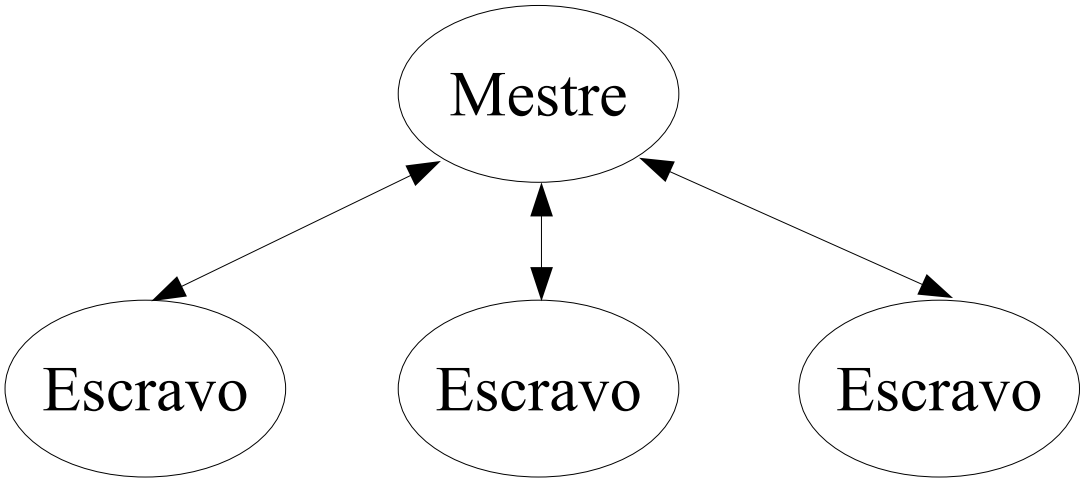
Modelos de aplicação

- Workpool
  - tarefas disponibilizadas em uma estrutura de dados global (memória compartilhada)
  - sincronização
  - balanceamento de carga



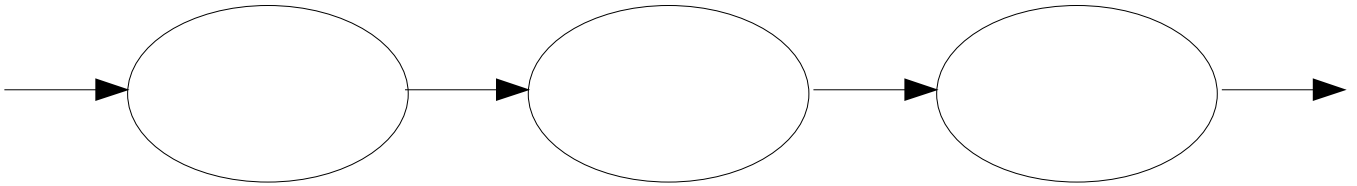
Modelos de aplicação

- Mestre / Escravo (Task farming)
  - mestre escalona tarefas entre processos escravos
  - escalonamento centralizado - gargalo
  - maior tolerância a falhas



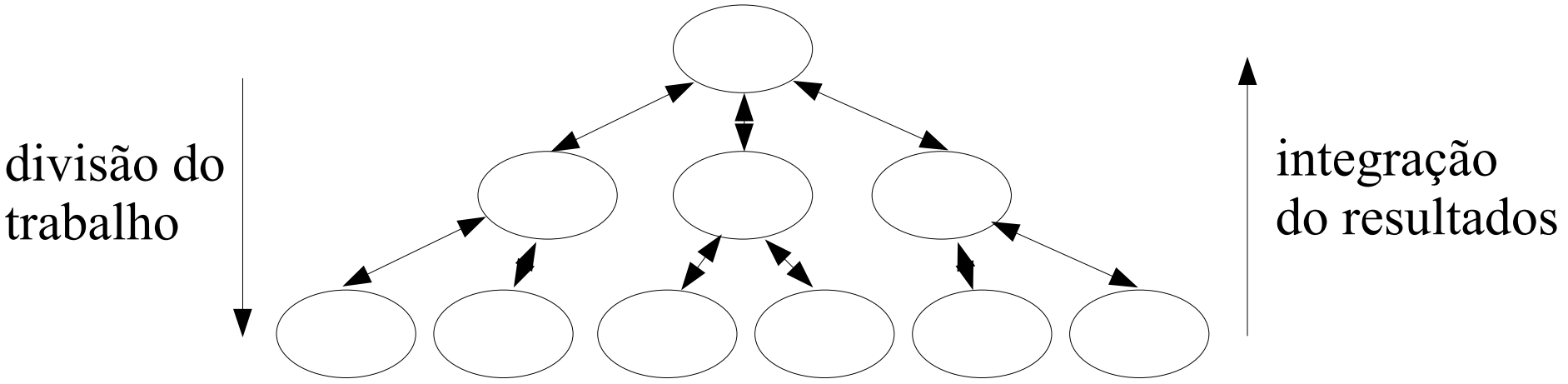
Modelos de aplicação

- Pipeline
  - pipeline virtual
  - fluxo contínuo de dados
  - sobreposição de comunicação e computação



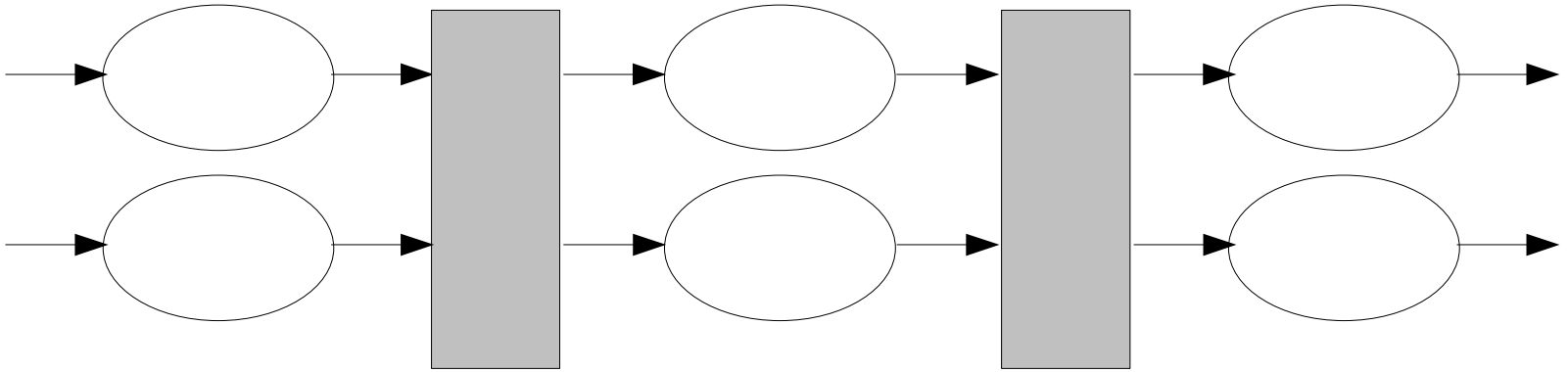
Modelos de aplicação

- 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
  - dificuldade de balanceamento de carga na divisão das tarefas



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

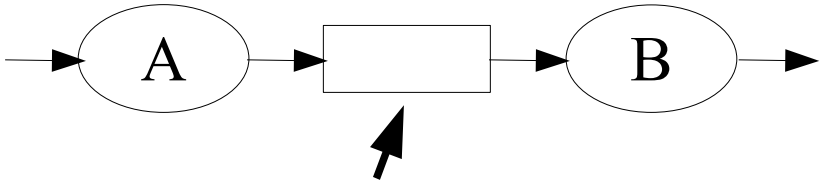


Programação Paralela
Memória compartilhada

- 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 primitvas de sincronização.

Programação Paralela
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
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 de 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
- Thread: também chamado de processo leve, em referência ao fato de que os recurso 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

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

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

POSIX threads (Pthreads)

- Distribuída com o SO (Linux) – implementação 1:1
- A thread é representada por uma função

  *void * func(void * args);*

- Criação de uma thread

  *int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine) (void *), void *arg)*

- Término de uma thread

  *void pthread_exit (void *status)*

- Sincronização entre threads

  *int pthread_join (pthread_t thread, void **status)*

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

void * helloWorld( void *str ) {
  printf("%s", (char *) str);
  printf("I am thread %d\n",  (int) pthread_self());
}

void main() {
  pthread_t thid;
  char *str = "Hello World!";

  if ( pthread_create (&thid, NULL, helloWorld, NULL) != 0) {
    printf("Error!\n");
    exit(0);
  }

  printf("Criada thread %d\n", (int) thid);
  pthread_join(thid, NULL);
  printf("A thread %d já terminou\n", (int ) thid);
}
```

POSIX threads (Pthreads)

- Sincronização
- Modelos de sincronização: mutex locks, condition variables e semáforos
  - Mutex locks - garante que somente uma thread por vez irá acessar uma seção de código específica ou acessar dados compartilhados – serializa a execução de threads
  - Variáveis de condição - bloqueia threads até que uma condição seja satisfeita
  - Semáforos - coordenam acessa a recursos, o semáforo indica o limite de threads que podem ter acesso concorrente a um recurso

POSIX threads (Pthreads)

- Mutex: utilizado para proteger estruturas de dados compartilhadas em modificações concorrentes e implementar seções críticas e monitores; pode possuir dois estados: unlocked e locked.
- Uma thread que deseja fazer lock em mutex que já está com lock por outra thread é suspenso até que a thread faça o unlock do mutex primeiro

POSIX threads (Pthreads)

- Mutex (operações)

*int pthread_mutex_init(pthread_mutex_t *mutex,*
*const pthread_mutex-attr_t *mutexattr);*

*int pthread_mutex_lock(pthread_mutex_t *mutex);*

*int pthread_mutex_trylock(pthread_mutex_t *mutex);*

*int pthread_mutex_unlock(pthread_mutex_t *mutex);*

*int pthread_mutex_destroy(pthread_mutex_t *mutex);*

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int saldo = 100;
pthread_mutex_t m;
void * deposita( void *str ) {
  int i, a;
  for(i=0; i<100; i++) {
    pthread_mutex_lock(&m);
    a = saldo; a = a + 1; saldo = a;
    pthread_mutex_unlock(&m);
  }
}

void * retira( void *str ) {
  int i, b;
  for (i=0; i<100; i++) {
    pthread_mutex_lock(&m);
    b = saldo;
    b = b - 1;
    saldo = b;
    pthread_mutex_unlock(&m);
  }
}

void main() {
  pthread_t thid1, thid2;
  pthread_mutex_init(&m, NULL);
  if ( pthread_create (&thid1, NULL, deposita, NULL) != 0) {
    printf("Error!\n");
    exit(0);
  }
  if ( pthread_create (&thid2, NULL, retira, NULL) != 0) {
    printf("Error!\n");
    exit(0);
  }
  pthread_join(thid1, NULL);
  pthread_join(thid2, NULL);
  printf("SALDO ATUAL = %d\n", saldo);
}
```

POSIX threads (Pthreads)

- Variáveis de condição
  - permite controlar o avanço de threads de acordo com a ocorrência de condições
  - não controla acesso a seções críticas, mas permite coordenar a evolução da execução das threads
  - devem ser associadas a um mutex para evitar condição de corrida
    - thread t1 prepara para esperar (wait) em uma variável de condição e outra thread t2 sinaliza (signal) a condição logo antes da thread t1 realmente esperar nesta variável

POSIX threads (Pthreads)

- Variáveis de condição (principais operações)
  - inicializa uma variável de condição

  *pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);*

  - desbloqueia uma thread que esteja bloqueada pela variável de condição

  *pthread_cond_signal(pthread_cond_t *cond);*

  - libera todas as threads bloqueadas

  *int pthread_cond_broadcast(pthread_cond_t *cond);*

  - libera o mutex e fica bloqueado na variável de condição

  *int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);*

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

int buffer = 0;
pthread_mutex_t m;
pthread_cond_t c;
void *produtor(void *str) {
  int i, a;
  for(i=0; i<100; i++) {
    pthread_mutex_lock(&m);
    buffer++;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
  }
}
void *consumidor(void *str) {
  int i, b;
  for (i=0; i<100; i++) {
    pthread_mutex_lock(&m);
    while(buffer <= 0)
      pthread_cond_wait(&c, &m);
    buffer--;
    pthread_mutex_unlock(&m);
  }
}
void main() {
  pthread_t thid1, thid2;

  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&c, NULL);

  if (pthread_create (&thid1,
NULL, produtor, NULL) != 0) {
    printf("Error!\n");
    exit(0);
  }
  if (pthread_create (&thid2,
NULL, consumidor, NULL) != 0) {
    printf("Error!\n");
    exit(0);
  }
  pthread_join(thid1, NULL);
  pthread_join(thid2, NULL);
}
```

POSIX threads (Pthreads)

- Semáforo
  - permite controlar o fluxo de controle do programa e acessar a áreas de dados compartilhadas por threads concorrentes
  - permite especificar um número determinado de threads que podem entrar na seção crítica
  - Exemplo: fila para pegar passagem no aeroporto com 5 caixas, quando uma caixa fica livre outro cliente pode ser tratado, mas somente 5 clientes podem ser tratados concorrentemente

POSIX threads (Pthreads)

- Semáforo (principais operações)

*int sem_init(sem_t *sem, int pshared, unsigned int value);*

*int sem_wait(sem_t * sem);*

*int sem_trywait(sem_t * sem);*

*int sem_post(sem_t * sem);*

*int sem_getvalue(sem_t * sem, int * sval);*

*int sem_destroy(sem_t * sem);*

```c
#include <pthread.h>
#include <semaphore.h>

sem_t s0, s1;
int buffer;

void *produtor() {
  int i;
  for(i=0; i<100; i++) {
    sem_wait(&s0);
    buffer = i;
    sem_post(&s1);
  }
}

void *consumidor() {
  int i, k;
  for(i=0; i<100; i++) {
    sem_wait(&s1);
    k = buffer;
    sem_post(&s0);
    printf("Valor consumido: %d\n", k);
  }
}

main() {
  pthread_t tid1, tid2;
  sem_init(&s0, 0, 1);
  sem_init(&s1, 0, 0);
  pthread_create(&tid1,
NULL, produtor, NULL);
  pthread_create(&tid2,
NULL, consumidor, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
}
```

```
#include <pthread.h>
#include <semaphore.h>
#define N 10
sem_t full, empty, mutex;
int buffer[N];
int i=0, j=0;
void *produtor() {
  int i=0;
  for(;;) {
    sem_wait(&empty);
    sem_wait(&mutex);
    buffer[i] = 50;
    i = (i+1)%N;
    sem_post(&mutex);
    sem_post(&full);
  }
}
void *consumidor() {
  int j=0, c;
  for(;;) {
    sem_wait(&full);
    sem_wait(&mutex);
    c = buffer[j];
    j = (j+1)%N;
    sem_post(&mutex);
    sem_post(&empty);
  }
}
main() {
  sem_init(&mutex, 0, 1);
  sem_init(&full, 0, 0);
  sem_init(&empty, 0, N);
  pthread_create(&tid1, NULL, produtor, NULL);
  pthread_create(&tid2, NULL, consumidor, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
}
```

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)

Programação Paralela
Troca de Mensagens

- 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

- Troca de mensagens síncrona
- 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

MPI

- MPI – *Message Passing Interface*
- Padrão definido pelo MPI Fórum para criação de programas paralelos (versão 1.0 em 1994).
- Atualmente está na versão 1.2 chamada de MPI-2.
- Possui diversas implementações (MPICH, LAMMPI, Java-MPI).

MPI

- Modelo de programação SPMD (Single Program - Multiple Data).
- Modelo de comunicação Message-Passing (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

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.

```
#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: nro. de dados a serem recebidos

datatype: tipo dos dados

source: processo que envio mensagem

tag: identificador

comm: comunicador

status: informação de controle

```
#include "mpi.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("Message received: %s\n",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: nro 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: nro 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 bloqueantes

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

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

```
#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)*

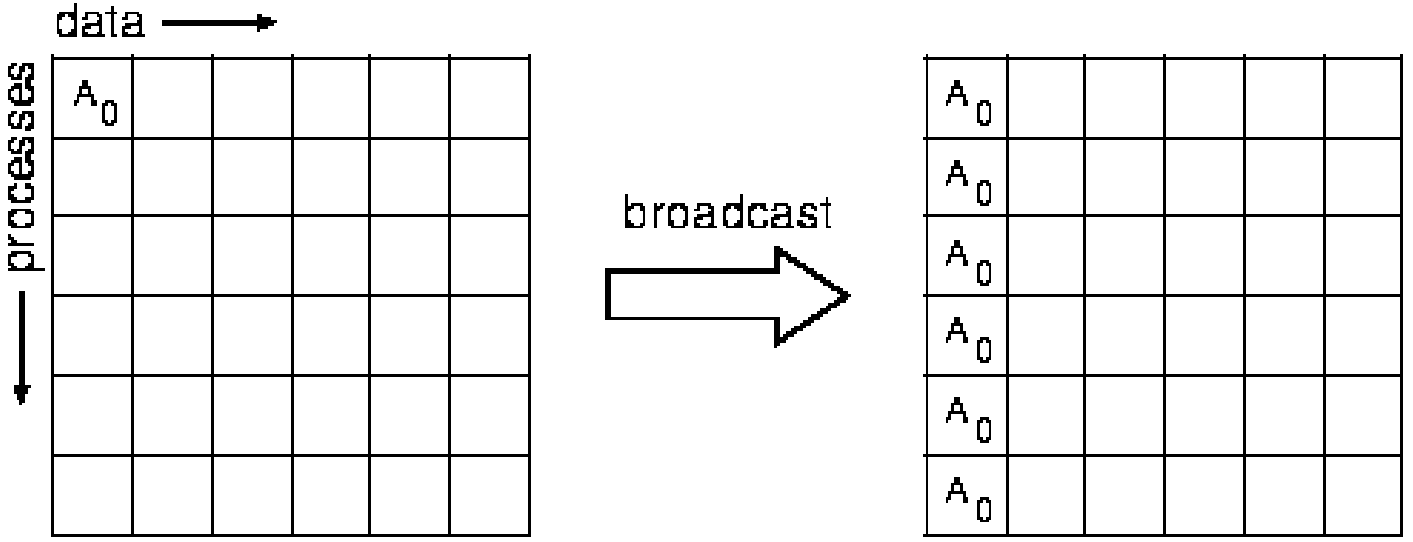
buffer: área de memória

count: nro de dados

datatype: tipo dos dados

root: rank do processo mestre

com: comunicador



```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
  char message[30];
  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)
    strcpy(message, “Hello World!”);
  MPI_Bcast(message, strlen(message)+1, 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: nro 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

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: nro de dados a serem enviados

sendtype: tipo dos dados

recvbuf: buffer para recebimento

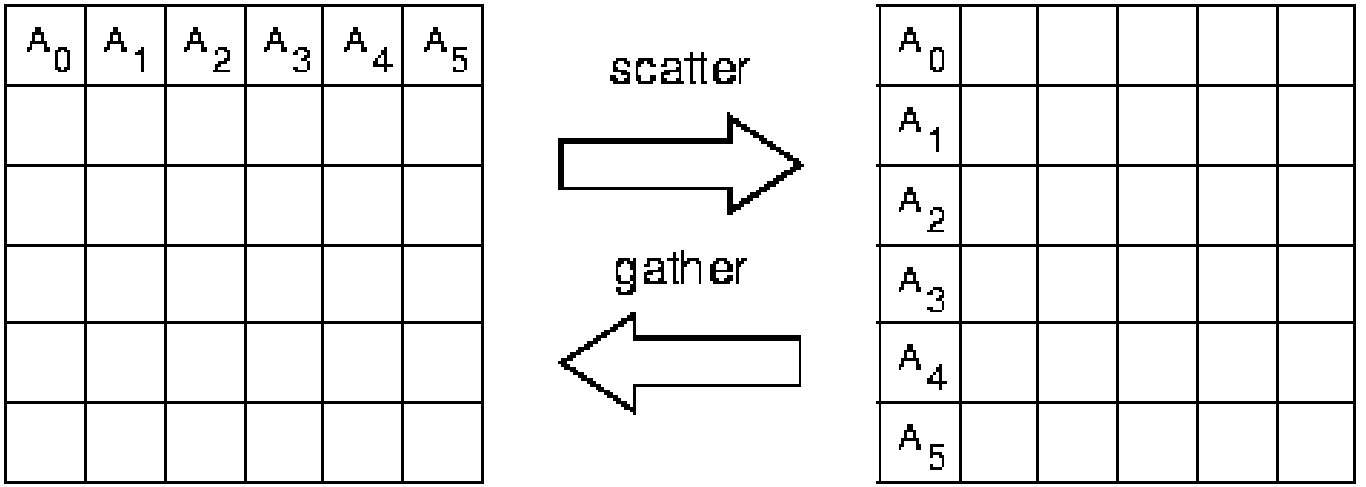
recvcount: nro de dados para recebimento

recvtype: tipo dos dados para recebimento

root: processo mestre

com: comunicador

Comunicação em grupo



```
#include <mpi.h>
#include <stdio.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 %s\n", rank, message);
  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: nro de dados a serem enviados

sendtype: tipo dos dados

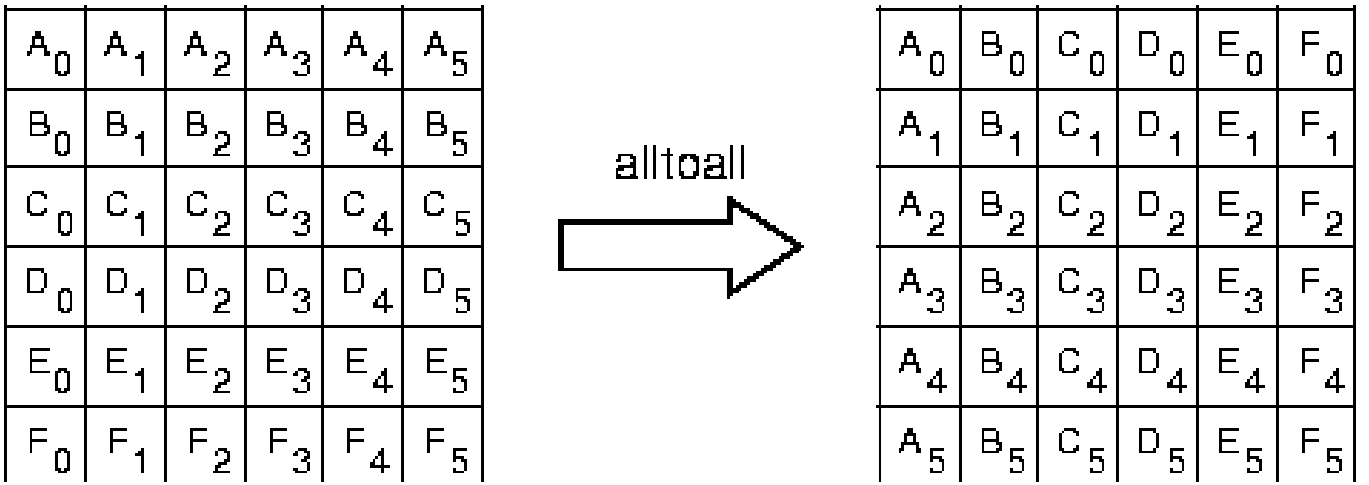
recvbuf: buffer para recebimento

recvcount: nro de dados para recebimento

recvtype: tipo dos dados para recebimento

com: comunicador

Comunicação em grupo



```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
  int *sndbuffer, *recvbuffer;
  int rank, 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,
    MPI_INT, MPI_COMM_WORLD);
  printvector(rank, sndbuffer);
  MPI_Finalize();
  return 0;
}
```

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

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
  int result;
  int rank, size, i;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Reduce(&rank, &result, 1, MPI_INT, MPI_SUM, 0,
    MPI_COMM_WORLD);
  if(rank == 0)
    printf("Total is %d\n", result);
  MPI_Finalize();
  return 0;
}
```

Considerações

- Amplamente utilizado no meio acadêmico e comercial.
- Opção atual para programação paralela.
- Não é a melhor solução desejada, pelas seguintes dificuldades:
  - paralelismo explícito;
  - particionamento manual;
  - necessita o conhecimento total da máquina.

Referências MPI

- MPI Fórum - http://www.mpi-forum.org
- MPICH - http://www-unix.mcs.anl.gov/mpi/mpich/
- LAMMPI - http://www.mpi.nd.edu/lam/

Métricas de desempenho

- Speed-up
  - fator de aceleração
  - aumento de velocidade observado em um computador paralelo com p elementos de processamento, em relação a um computador seqüencial [Hwang; Xu 1998]
  - SpeedUp = Tsequencial/Tparalelo
  - anomalia de speedup ou speedup super linear
    - speedup obtido pode ser maior que o previsto (Sp > p)
    - características do algoritmo paralelo ou da plataforma de hardware
      - (e.g.utilização da memória cache dos elementos de processamento)

Métricas de desempenho

- Eficiência
  - porcentagem dos elementos de processamento que estão sendo efetivamente utilizados
  - Eficiência = Tsequencial/Tparalelo*N
- Custo
  - considerado ótimo para um determinado algoritmo paralelo quando o custo da execução paralela é proporcional ao custo da execução seqüencial
  - custo = T*N = Ts/Eficiência