Programação Paralela e Distribuída

ERAD-SP 2010

31 de julho de 2010

Liria Matsumoto Sato

liria.sato@poli.usp.br

Apresentação

- Introdução
- Arquiteturas Paralelas
- Programação Paralela para computadores com memória compartilhada
- Programação paralela para sistemas distribuídos
- Conclusão

Ferramentas: CPAR e MPI

Download CPAR e exemplos de programas:

http://regulus.pcs.usp.br/~lms/eradsp.html

Introdução

Visão do Mundo atual:

- Crescimento contínuo da demanda por processamento
- Necessidade de compartilhamento de Informação e de recursos
- Acesso a recursos computacionais de alto desempenho

Computação de alto desempenho: busca de soluções

Introdução

Aplicações envolvendo grande capacidade de processamento:

- meteorologia
- simulação de fenômenos físicos
- visualização científica
- modelagem das variações climáticas globais em longos períodos
- genoma humano
- dinâmica de fluídos
- Estrutura e Dinâmica molecular

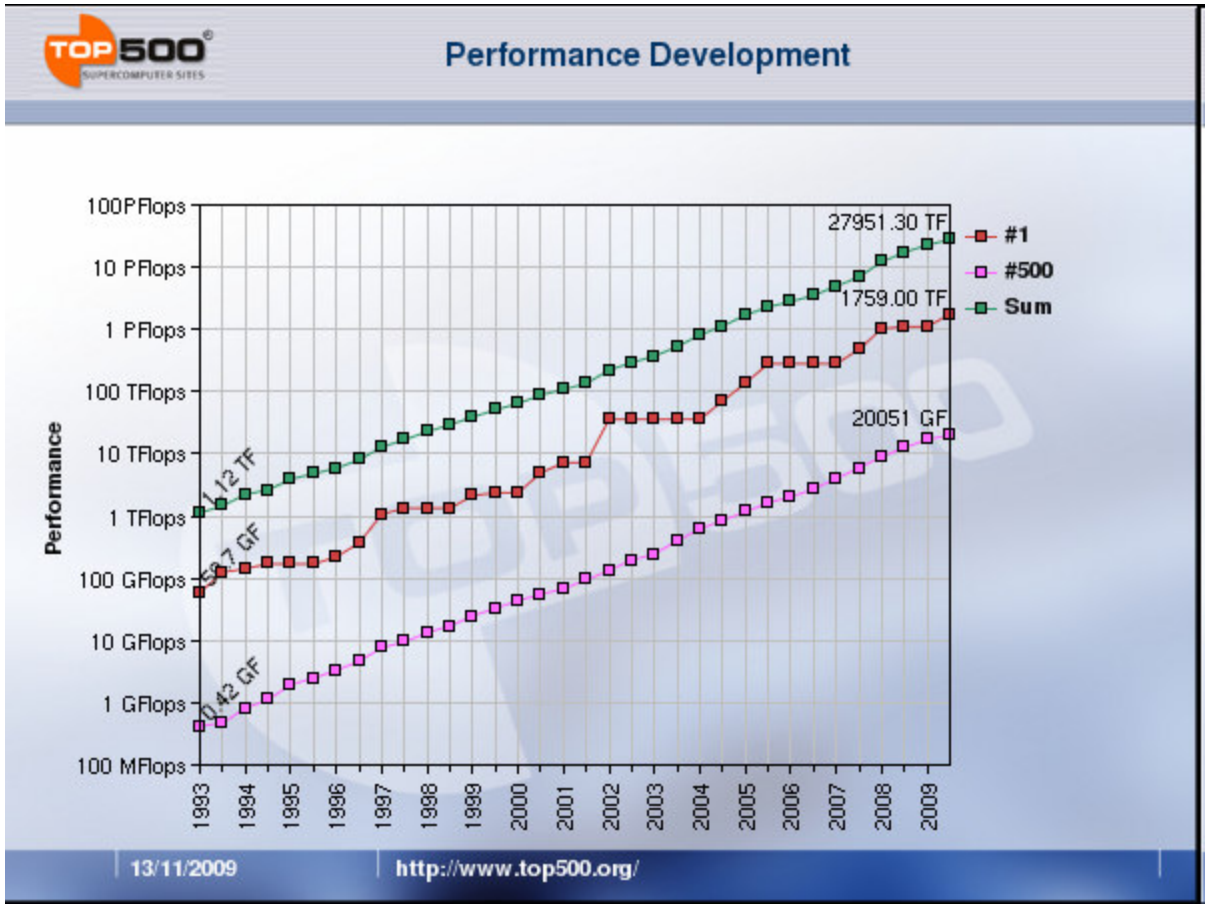


Figura extraída de www.top500.org.br (acesso em maio de 2010)

Demanda crescente →computadores mais potentes

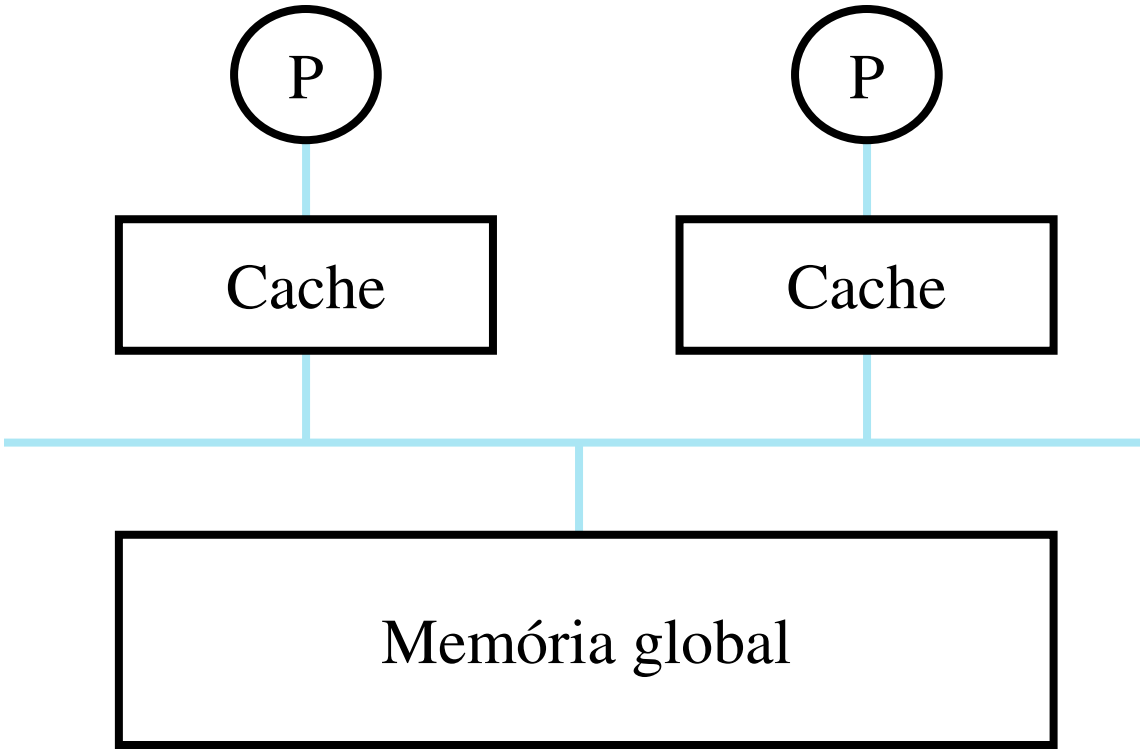
Como obter alto desempenho

- **Aumento do desempenho do processador**
  - aumento do clock $\Rightarrow$ aumento de temperatura
  - melhorias na arquitetura

- **Processamento Paralelo**:
  - Múltiplas unidades de processamento
  - Paralelização da aplicação

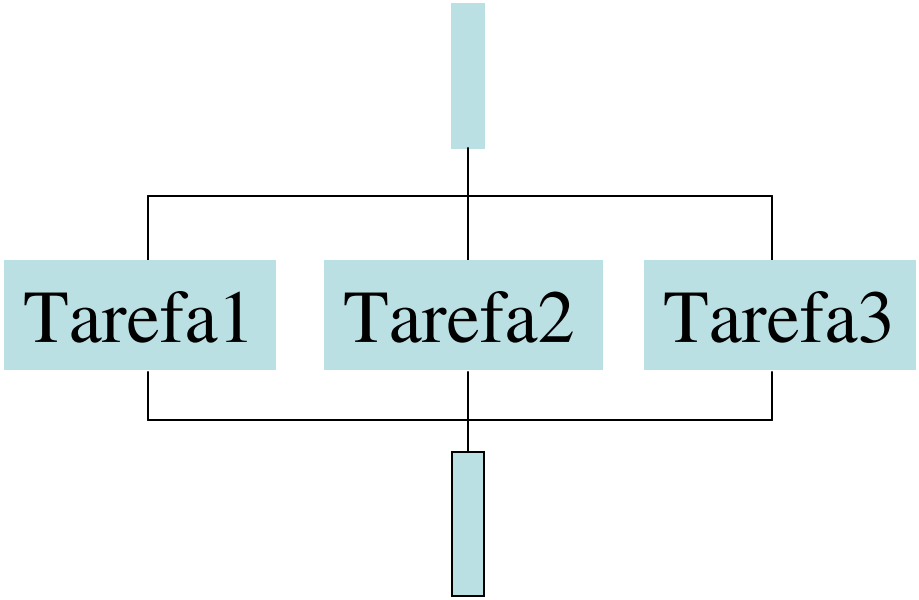
Processamento Paralelo

Utilização de múltiplos processadores



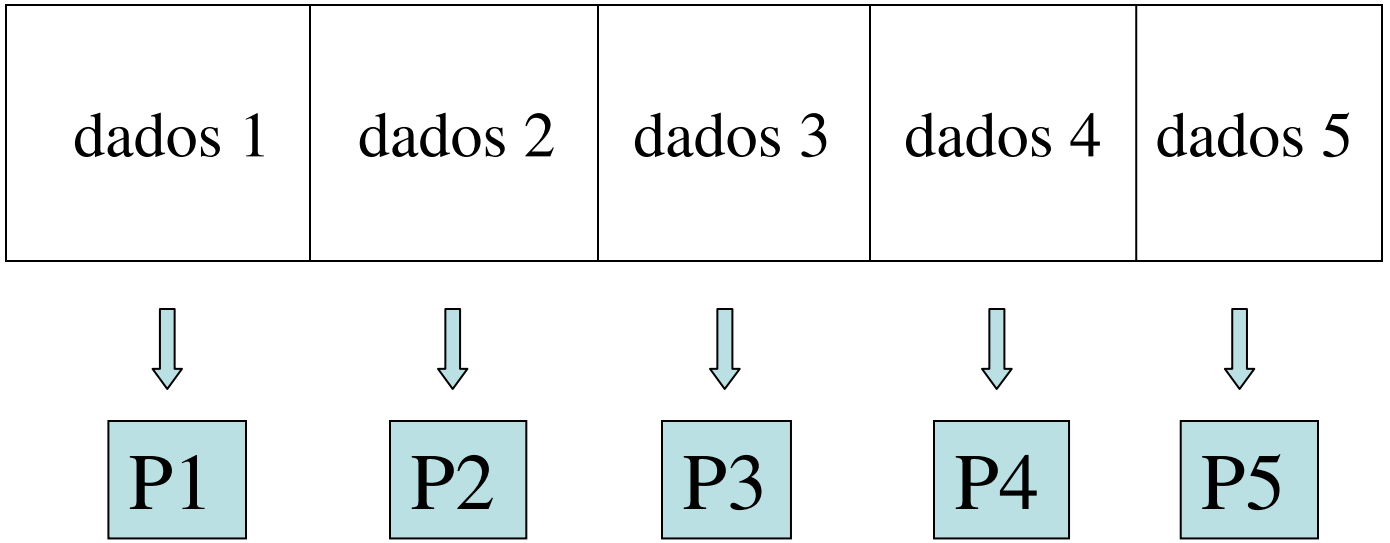
Processamento Paralelo

Paralelização da aplicação em múltiplas tarefas



Processamento Paralelo

Paralelismo de dados



Processamento Paralelo

Multi-core: multiplos núcleos no chip.

AMD, INTEL: quad-core, hexa-core processor

NEHALEN : quad-core, hexa-core

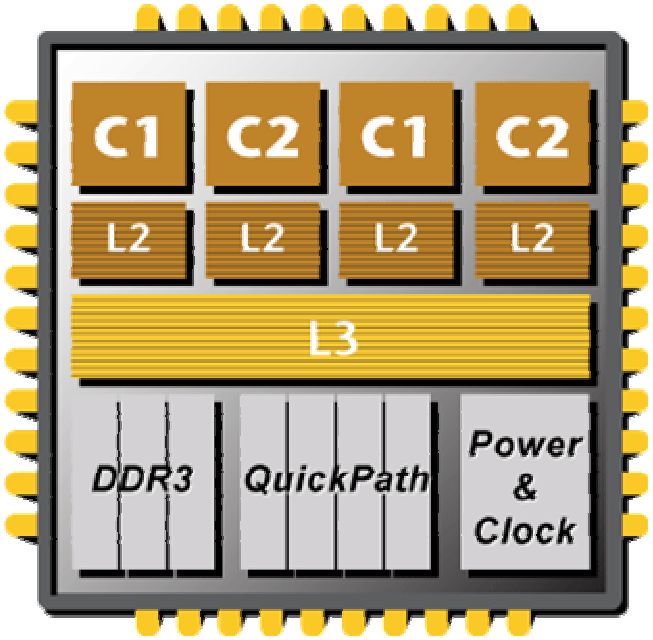
IBM: CELL

NVIDIA, ATI: GPUs (Graphics Processing Unit)

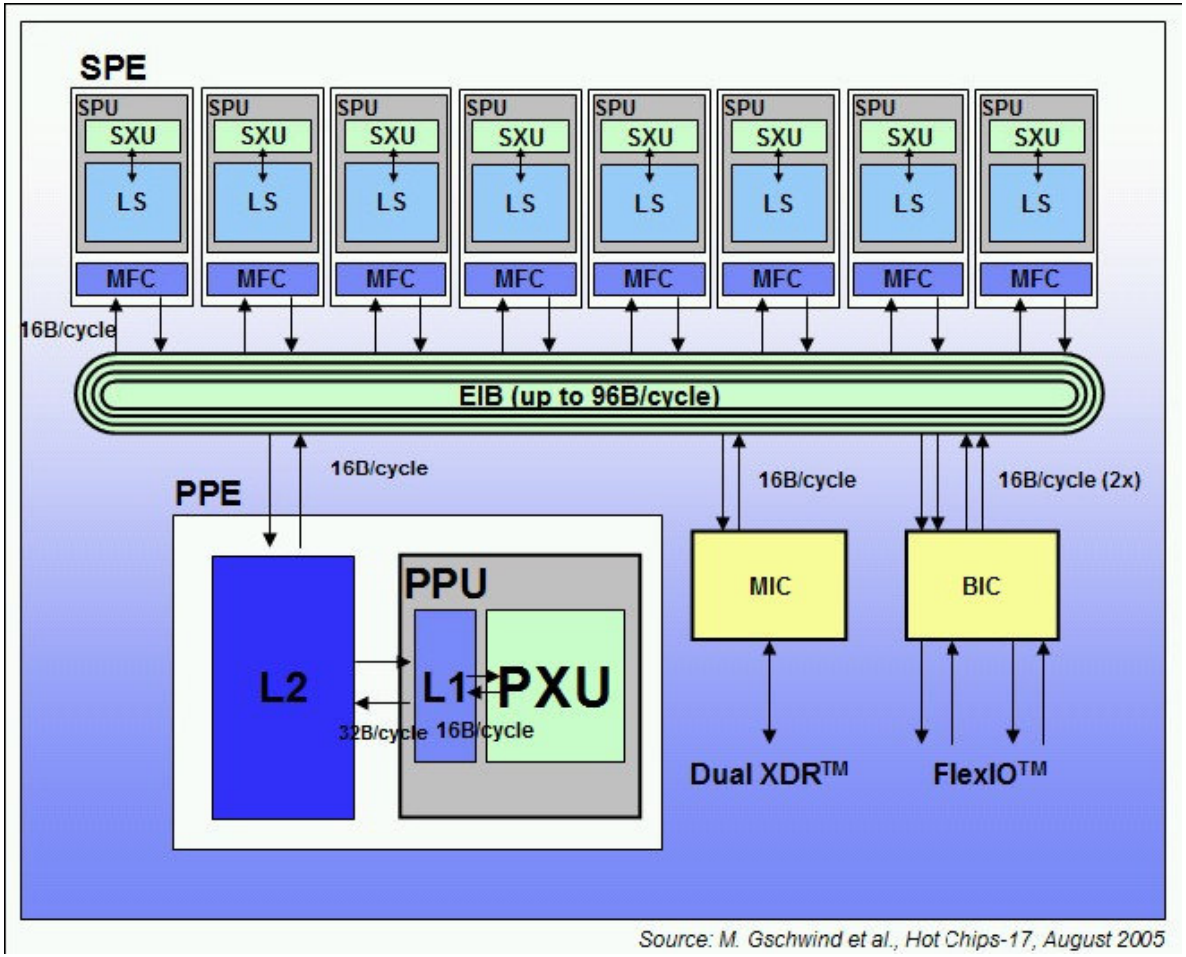
Computadores com arquitetura paralela

Multi-core

- NEHALEM

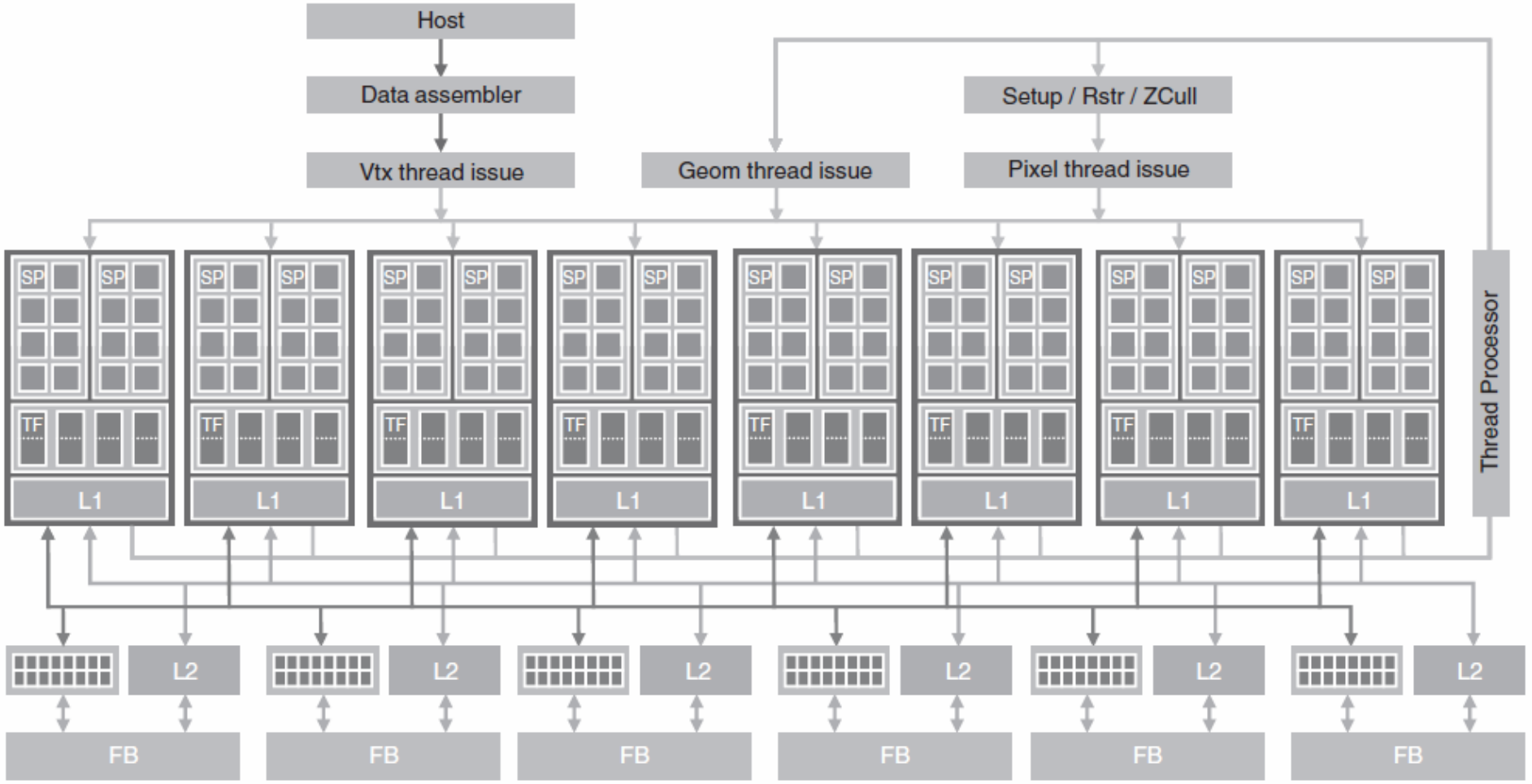


Arquitetura CELL



Extraído de http://www.research.ibm.com/cell/heterogeneousCMP.html

GPU



Geforce 8800 GTX [figura extraída de ¨Programming Massively Parallel Processors: a hands-on Aproach. David B. Kirk; Wen-mei Hwu. Elsevier Inc.

Processadores GPU

nVIDIA Tesla S1070: 4 GPUs

4 TFlops (ponto flutuante precisão simples)

345 Gflops (ponto flutuante precisão dupla)

4 GB memória dedicada

Arquiteturas Paralelas

Arquiteturas Paralelas: caracterizadas pela presença de múltiplos processadores que cooperam entre si na execução de um programa.

Década 1960 : ILLIAC IV (SIMD)

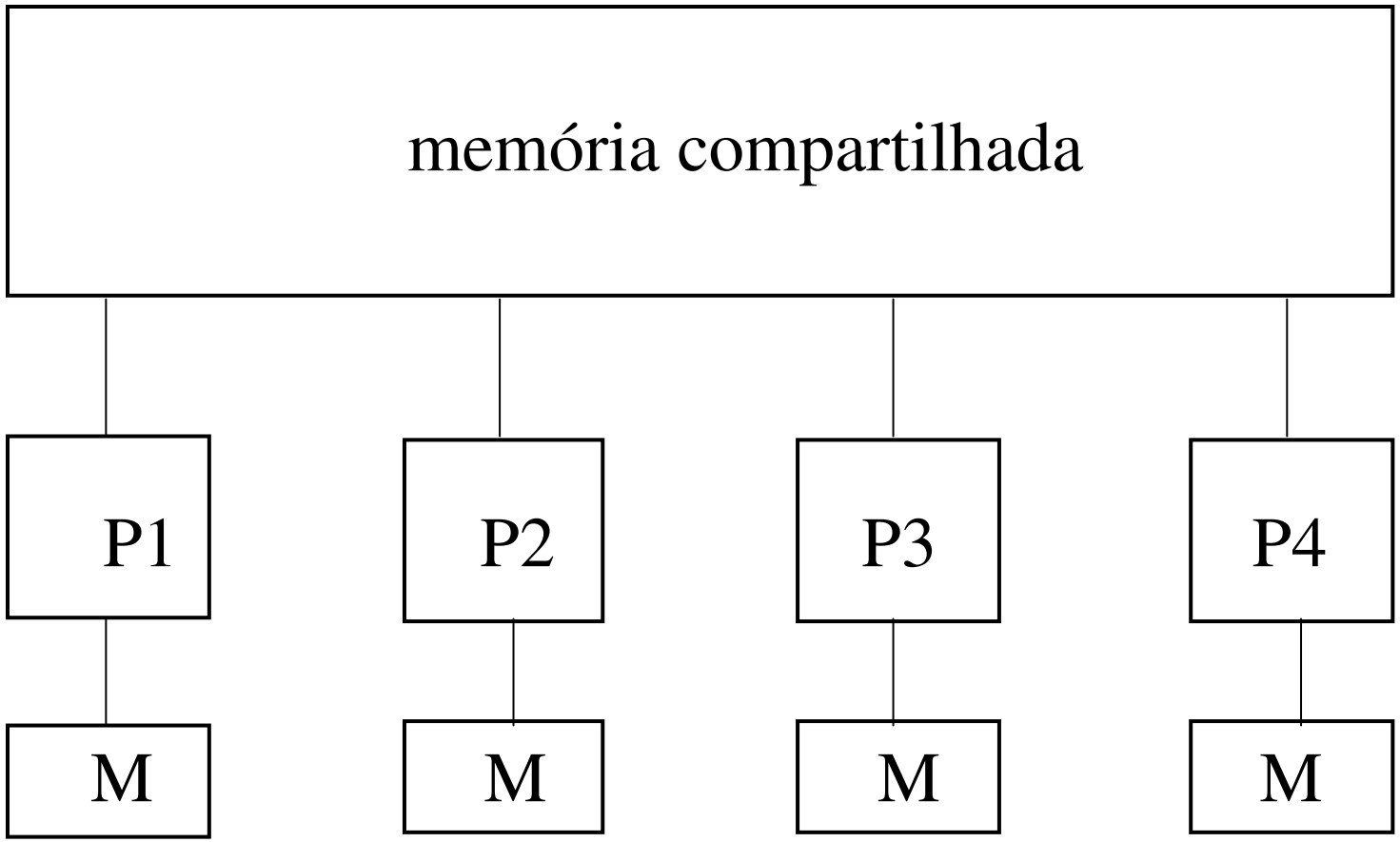
SIMD: todos os processadores executam a mesma instrução sobre dados distintos

MIMD: os processadores podem executar instruções distintas sobre dados distintos

- Multiprocessadores
- Multicomputadores: clusters

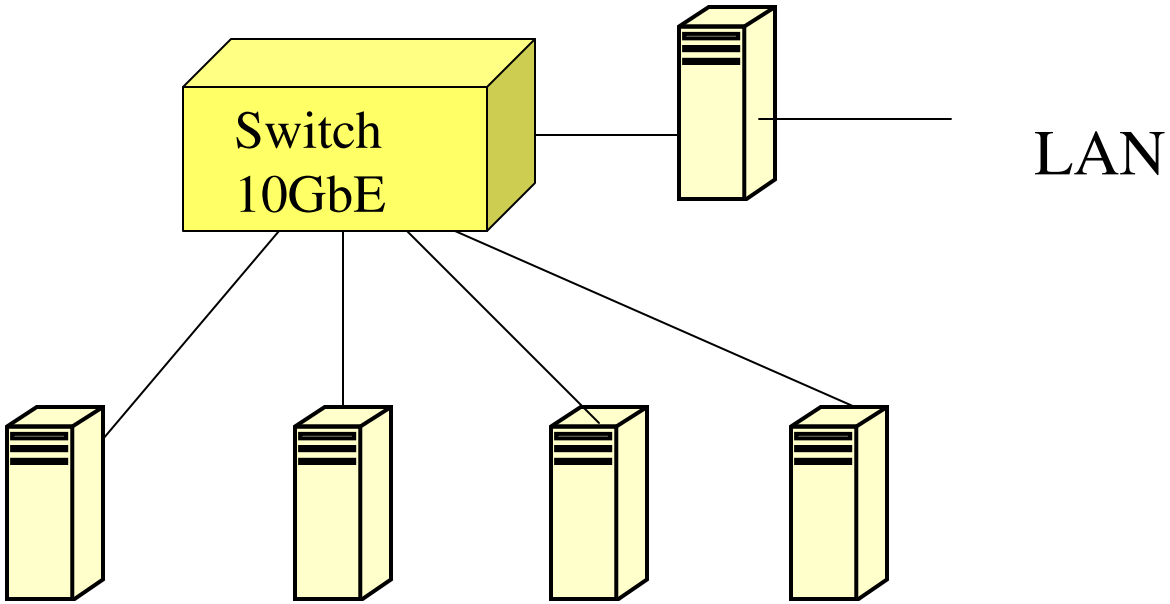
Arquiteturas Paralelas

Multiprocessadores



Arquiteturas Paralelas

Cluster



Aplicação

- Como paralelizar
- Como programar

Linguagens e compiladores específicos:

- multi-core, multiprocessadores: OpenMP, CPAR
  memória compartilhada
- clusters:
  memória distribuída:
  MPI (Message Passage Interface):
  - diversas implementações disponíveis
    MPICH, OPENMPI

Desenvolvimento de aplicações

Definir solução com processamento paralelo:

- Paralelização de sequências de instruções
- Paralelização de laços
- Algumas aplicações exigem soluções mais complexas:
  - Partes da aplicação envolvem cálculos dependentes de cálculos anteriores
  - Partes da aplicação exigem uma ordenação na geração de resultados

Exemplo com paralelização de laços

```
#pragma pfor iterate (i=0;
MAXRET; 1)
 for (i = 0; i < MAXRET; i++)
    {
    x = ((i-0.5) * largura);
          /* calcula x */
    local_pi = local_pi + (4.0 / (
1.0 + x * x));
    }
  local_pi = local_pi * largura;
#pragma critical
  {
  total_pi = total_pi + local_pi;
  }
```

```
forall i = 0 to MAXRET    {
  x = ((i-0.5) * largura);
          /* calcula x */

local_pi=local_pi+(4.0/(1.0+
x* x));
  local_pi = local_pi *
largura;
  lock (&semaforo);
  total_pi = total_pi +
local_pi;
  unlock (&semaforo);
  }
}
```

Programação paralela para sistemas com memória compartilhada

Sistemas com memória compartilhada:

- computadores multiprocessadores (processadores multicore ou não)
- Computadores com processador multicore.

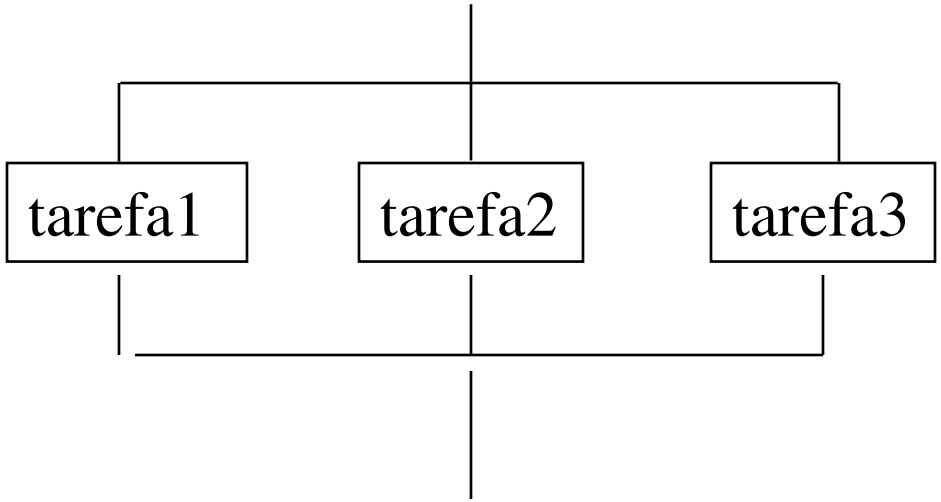
Ferramenta didática: linguagem CPAR

  - Extensão da linguagem C

Paralelização de seqüências de instruções

- Múltiplas tarefas simultâneas: macrotarefas
- Paralelização do programa em blocos paralelos
- Laços e blocos paralelos
- CPAR:
  - Múltiplas tarefas simultâneas
  - laços e blocos paralelos dentro de uma tarefa
  - Blocos paralelos na função main

Múltiplas tarefas simultâneas



Macrotarefas: são procedimentos especiais

Tempo execução seqüencial=$t_{tarefa1} + t_{tarefa2} + t_{tarefa3}$

Tempo execução paralela=max($t_{tarefa1}$, $t_{tarefa2}$, $t_{tarefa3}$)

Macrotarefas

```
#include <stdio
task spec tarefa1();
task spec tarefa2();
task spec tarefa3();
task body tarefa1()
{ int i;
   for (i=0;i<199;i++)
      { printf("%c",'a');
        fflush(stdout);
      }
}
```

```
task body tarefa2()
{ int i;
   for (i=0;i<199;i++)
      { printf("%c",'b');
        fflush(stdout);}
}
task body tarefa3()
{ int i;
   for (i=0;i<199;i++)
      {printf("%c",'c');
        fflush(stdout);}
```

Macrotarefas

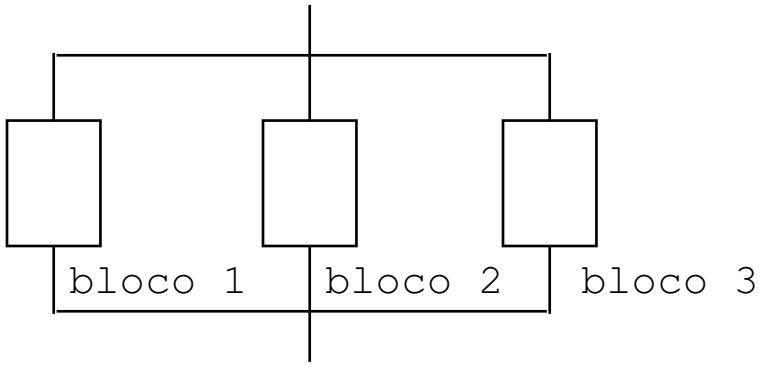
```
main()
{ printf(“INICIO TESTE\n”);
   alloc_proc(4);
   create 1,tarefa1();
   create  1,tarefa2();
   create  1,tarefa3();
 }
```

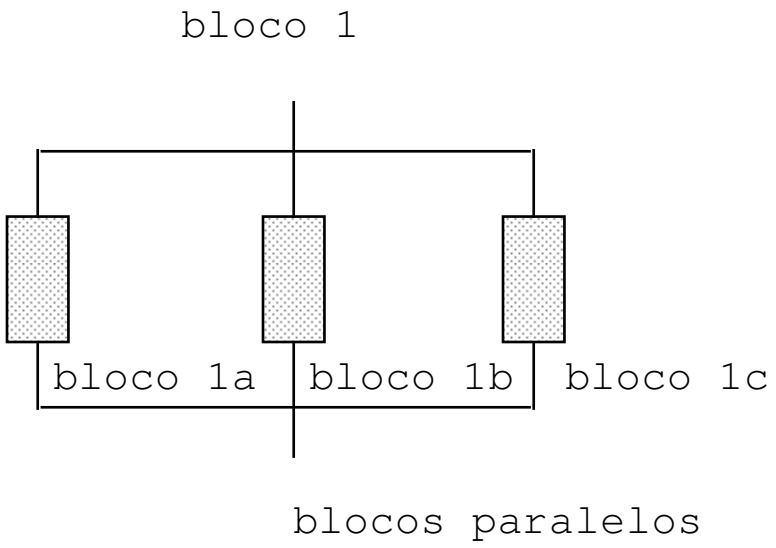
Paralelização da função principal

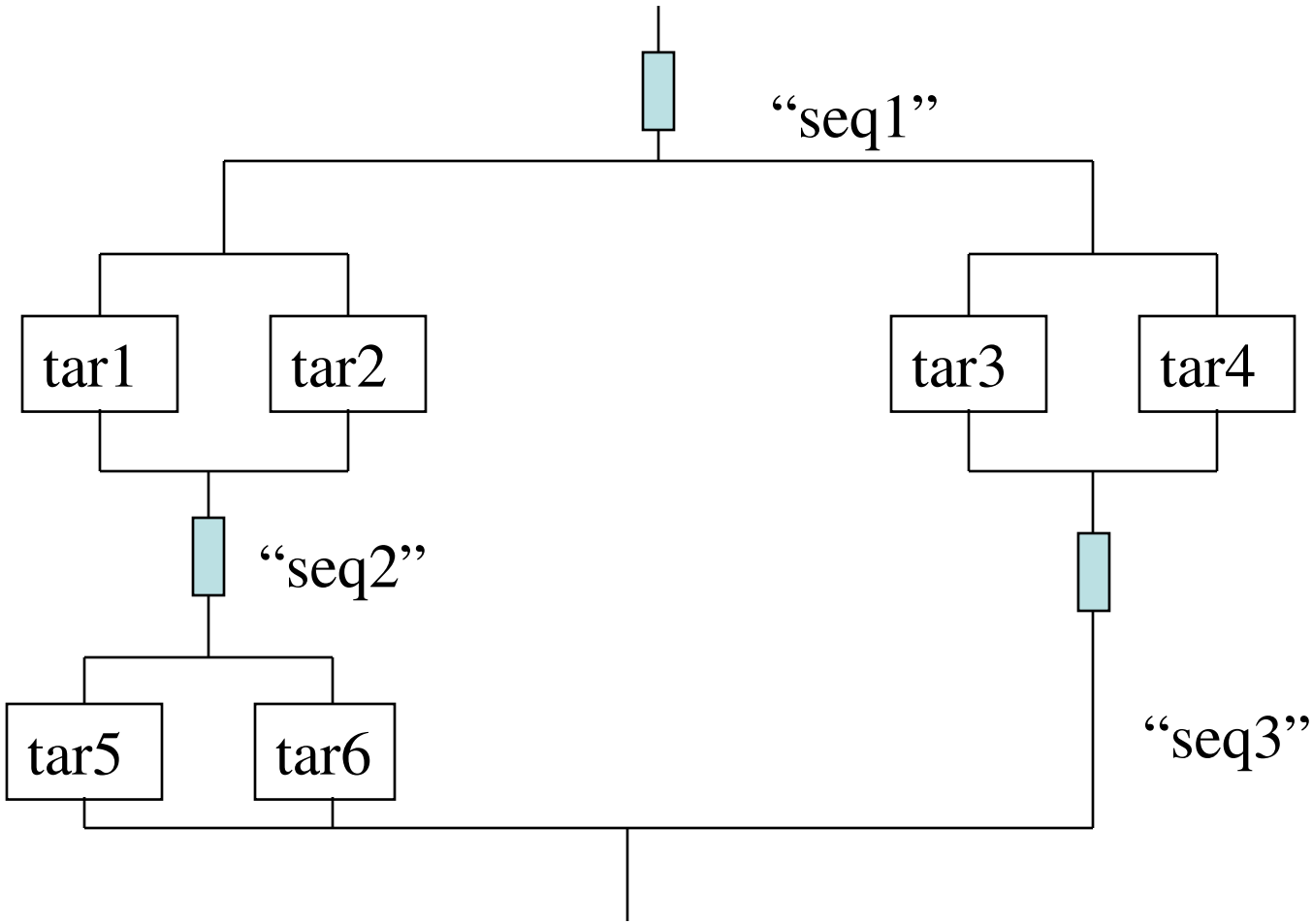
blocos paralelos na função principal

cada bloco:

- Blocos
- Tarefas
- Elementos Seqüenciais






“seq1”
tar1
tar2
tar3
tar4
“seq2”
tar5
tar6
“seq3”

Blocos Paralelos (CPAR)

- comando cobegin ... also ... coend
- restrição: presente apenas na função main()

```
main()
  {  ...
     cobegin
        .....
        also
        ......
        also
        .......
     coend
      ......
  }
```

sincronização de tarefas

- espera pelo término de uma tarefa

  wait_task(nome_tarefa);

  exemplo: wait_proc(tarefa1);

- espera pelo término de todas tarefas em execução

  wait_all()

Laços Paralelos: Paralelismo homogêneo

```
forall i=1 to max{
  a[i]=b[i]+1;
  c[i]=c[i]+a[i];
}
```

CPAR: laços paralelos internos a uma macrotarefa

Exemplo

```
#include <stdio.h>
task spec tarefa1();
task spec tarefa2();

task body tarefa1()
{int i;
 for(i=0;i<10;i++)
 {printf("%c",'a');
  fflush(stdout);
 }
}
```

```
task body tarefa2()
{int i;
 for(i=0;i<10;i++)
 {printf("%c",'b');
  fflush(stdout);
 }
}
main()
{printf("INICIO TESTE\n");
 alloc_proc(2);
 create 1,tarefa1();
 wait_task(tarefa1);
 create 1,tarefa2();
}
```

Paralelismo heterogêneo

```
parbegin
  a=a+sqr(b);
  x=x*2;
also
 c=c+sqr(c);
 y=y+1;
parend
```

CPAR: blocos paralelos dentro de uma macrotarefa

memória compartilhada

Cada processo pode ter uma porção de memória privada.

Algumas locações de memória podem ser:

- acessadas por 2 ou mais processos, referenciadas no programa através de variáveis compartilhadas, promovendo a comunicação entre os processos.

memória compartilhada

O sistema CPAR permite o uso de:

· variáveis privadas (locais)

· variáveis compartilhadas:

shared int A;

- Variáveis compartilhadas entre macrotarefas

declaração na área de declarações de variáveis globais

- Variáveis compartilhadas pelas microtarefas de uma macrotarefa

declaração na área de declarações de variáveis da macrotarefa

Laço Paralelo: exemplo

iniciação de uma matriz

```
#include <stdio.h>
shared float a[1000][1000];
task spec inic_mat();
task body inic_mat()
{int i,j;
  forall i=0 to 999
    {
      for(j=0;j<1000;j++)
          a[i][j]=i+2*j;
    }
 }
```

```
main()
{ alloc_proc(4);
   printf("inicia matriz\n"):
   create   4,inic_mat();
   wait_proc(inic_mat);
   printf("iniciacao
    efetuada\n");
 }
```

Laço Paralelo

- CPAR: iterações são particionadas uniformemente (pré-scheduling)
- pré-compilador: gera código para tratar o particionamento
- em outros sistemas: self-scheduling (distribuiçao dinâmica de blocos de tamanho definido), guided self-scheduling (similar ao self-scheduling, com calculo dinâmico do tamanho dos blocos)

Exclusão Mútua

Processamento exclusivo de processos é necessário para que recursos possam ser compartilhados sem interferências mútuas.

Exemplo: Em um programa um contador (COUNT) é compartilhado e incrementado por mais de um processo.

COUNT = COUNT + 1

LD COUNT
ADD 1
STO COUNT

Exclusão mútua

Se 2 processos executarem esta seqüência, pode ocorrer:

```
LD COUNT      {processo 1}
LD COUNT      {processo 2}
ADD 1         {processo 2}
STO COUNT     {processo 2}
ADD 1         {processo 1}
STO COUNT     {processo 1}
```

COUNT: incrementado apenas de 1.

Solução: encerrar a seqüência de instruções em uma seção crítica.

Exclusão mútua

Seção crítica: seqüência de códigos executada ininterruptamente, garantindo que estados inconsistentes de um dado processo não sejam visíveis aos restantes. Isto é realizado utilizando mecanismos de exclusão mútua.

exclusão mútua: implementação com semáforos binários

Semáforos

Em CPAR:

declaração de semáforo: global ou local a macrotarefa

shared Semaph nome_semaforo;

Ex: shared Semaph X;

criação : create_sem(&nome_semafor,valor_inicial);

Ex: create_sem(&X,1);

remoção: rem_sem(&nome_semaforo);

Ex: rem_sem(&X);

operação P: lock(&nome_semaforo);

Ex: lock(&X);

operação V: unlock(&nome_semaforo);

Ex: unlock(&X);

Semáforo Binário

```
shared Semaph X;

create_sem(&X,1);

lock(&X);

unlock(&X);

rem_sem(&X);
```

Exemplo: seção crítica

```
task body tarefa()
{shared Semaph A;
  int i;
  create_sem(&A,1);
  forall i=0 to 99
    { ....
       lock(&A);
       COUNT=COUNT+1;
       unlock(&A);
       .....
      }
    ......
    rem_sem(&A);
   }
```

Threads

- thread: é um fluxo de controle seqüencial em um programa.
- programação multi-threaded: uma forma de programação paralela onde vários threads são executados concorrentemente em um programa. Todos threads executam em um mesmo espaço de memória, podendo trabalhar concorrentemente sobre dados compartilhados.

multi-threaded X multi-processing

- multi-threaded programming: todos os threads compartilham o mesmo espaço de memória e alguns outros recursos, como os descritores de arquivos.
- multi-processing: processos executam sobre seu próprio espaço de memória. O compartilhamento de dados se obtém através de funções que fazem com que endereços lógicos apontem para o mesmo endereço físico.

Pthreads

- POSIX : Pthreads
- Exemplo: pi-pthreads.c
- Compilação:

gcc –o pi-pthreads pi-pthreads.c -lpthread

OpenMP

- spec25.pdf : www.openmp.org/mp-documents/spec25.pdf
- Diretivas

#pragma omp *directive-name [clause[[,] clause]...] new-line*

Região paralela:

#pragma omp parallel

Compilação:

gcc –o pi-openmp pi-openmp.c -fopenmp

icc –o pi-openmp pi-openmp.c -openmp

Exemplos: pi-openmp.c

Seções: omp-section.c

Redução: pi-openmp-reduction.c

pi - openmp

```
int main()
{
.....
omp_set_num_threads(4);
#pragma omp parallel shared (total_pi,largura) private(i,x,local_pi)
{
#pragma omp for
 for (i = 0; i < MAXRET; i++)
   { x = ((i-0.5) * largura);              /* calcula x */
    local_pi = local_pi + (4.0 / ( 1.0 + x * x));
    }
  local_pi = local_pi * largura;
printf("LOCAL-PI %lf\n",local_pi);
#pragma omp critical
  {
  total_pi = total_pi + local_pi;
.....
}
```

pi-openmp-reduction

```
omp_set_num_threads(4);
#pragma omp parallel shared (largura) private(i,x) reduction(+:y)
{
#pragma omp for
 for (i = 0; i < MAXRET; i++)
   { x = ((i-0.5) * largura);              /* calcula x */
    y = y + (4.0 / ( 1.0 + x * x));
    }

}
total_pi = y * largura;
  printf("TOTAL-PI %lf\n",total_pi);
}
```

omp-section

```
 omp_set_num_threads(2);
#pragma omp parallel sections
  {
#pragma omp section
  { printf("secao 1\n");
    for (i=0;i<20000;i++)
     {printf("a");
      fflush(stdout);
     }
  }
#pragma omp section
  { printf("secao 2\n");
      for (i=0;i<20000;i++)
     {printf("b");
      fflush(stdout);
     }
  }
}
```

Sistemas Distribuídos

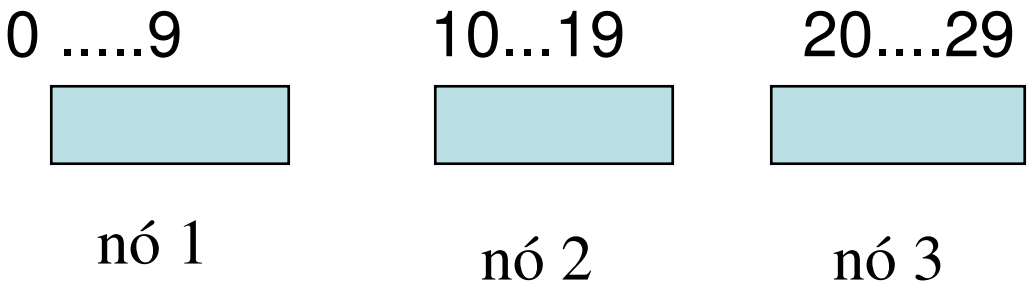
- Não possui memória compartilhada
- Bibliotecas: PVM, MPI
- Sistemas DSM (distributed shared memory): simula memória compartilhada, permitindo a programação no paradigma de variáveis compartilhadas

Sistemas Distribuídos: Programação com bibliotecas

- PVM (parallel virtual machine)
- MPI (message passage interface)
- programação utilizando passagem de mensagem

Modelo de computação: spmd (single program multiple data)

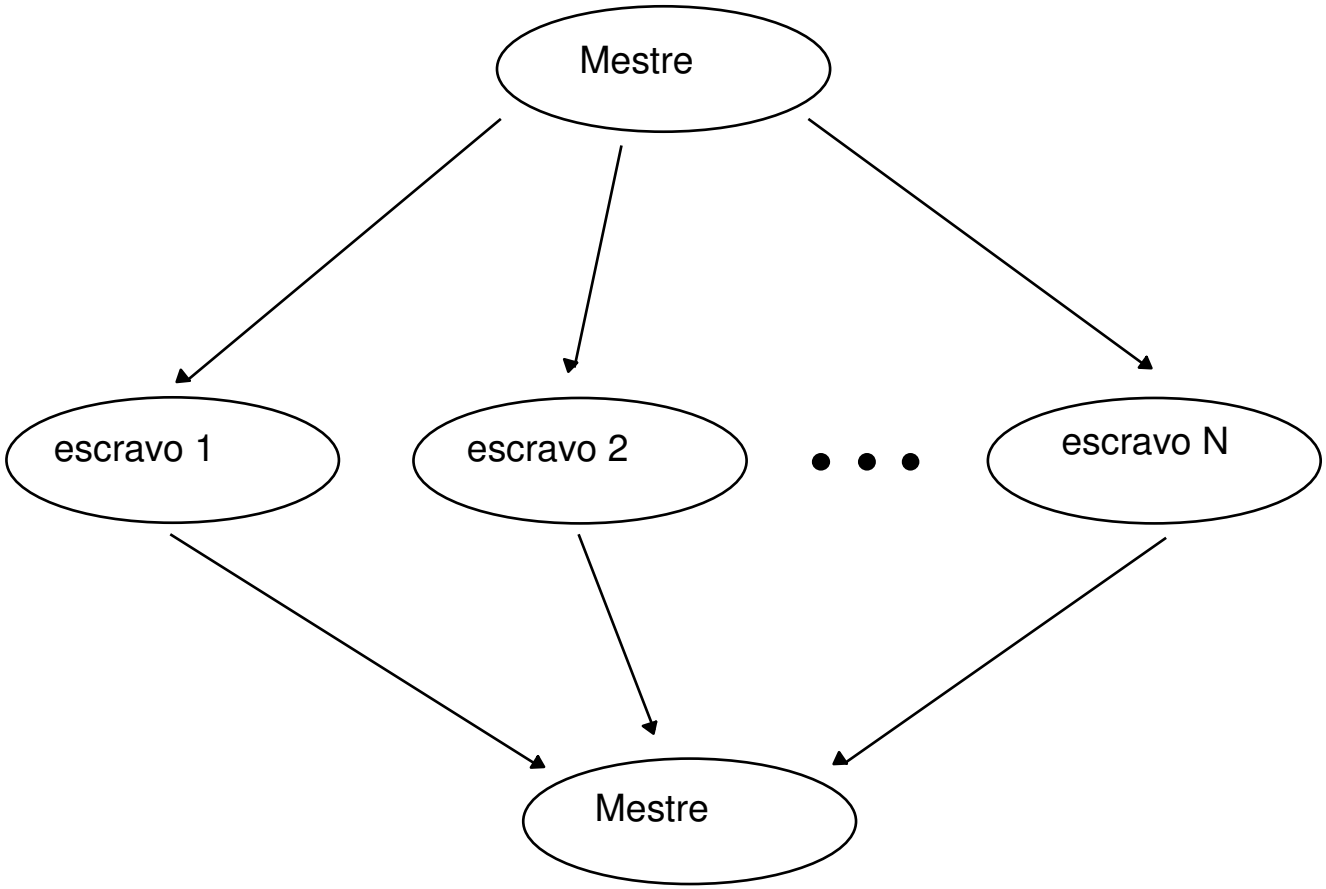
- todos os nós executam o mesmo programa sobre dados múltiplos



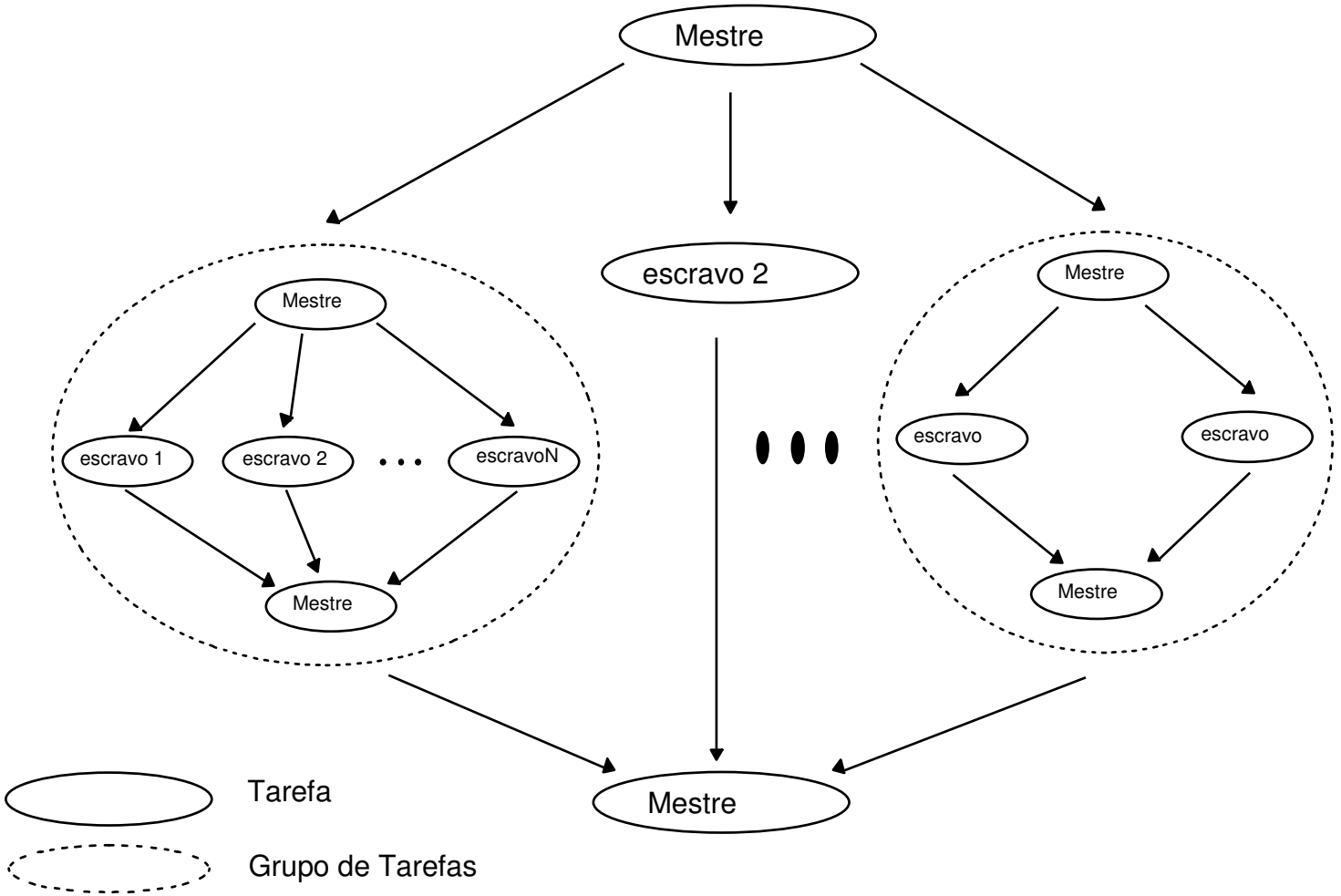
modelo de computação: esquema mestre escravo com spmd

- esquema mestre-escravo utilizando spmd: todos executam o mesmo programa, sendo que através de um controle interno ao programa um nó executa a função mestre eos demais são escravos.

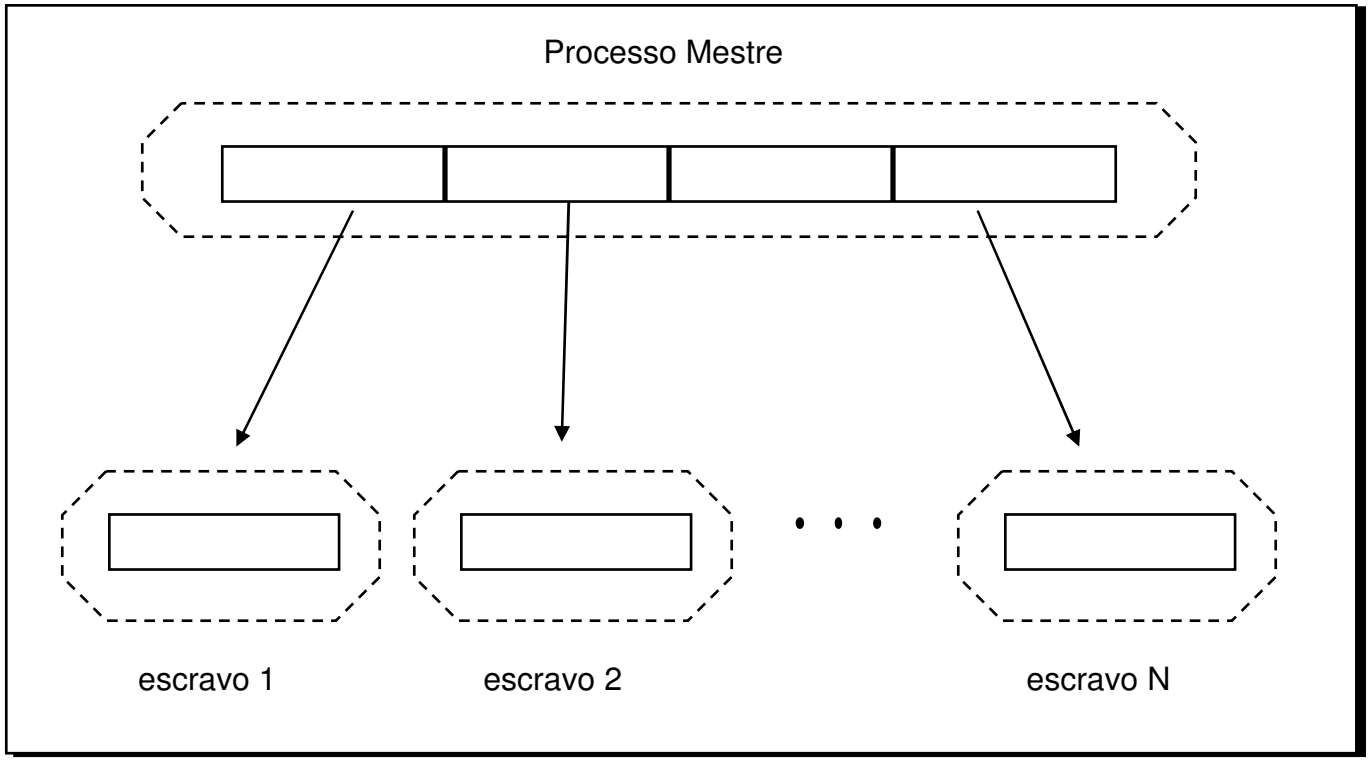
Modelo de computação



Grupos de tarefas



Exemplo : Soma dos elementos de um vetor



Para obter um bom desempenho

- Use granularidade grossa
- minimize o # de mensagens
- maximize o tamanho de cada mensagem
- use alguma forma de Balanceamento de carga

MPI

OPEN MPI

http://www.open-mpi.org/

MPI

- Processos: são representados por um único “rank”(inteiro) e ranks são numerados 0, 1, 2 ..., N-1. (N = total de processos)
- Enter e Exit

```
MPI_Init(int *argc,char *argv);
MPI_Finalize(void);
```

- Quem eu sou?

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

- informa total de processos

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

Programa MPI - SPMD

```
#include <mpi.h>
.........

main(argc, argv)
int                 argc;
char                        *argv[];
{
    int             size, rank;
    MPI_Status      status;
     ........

 // Initialize MPI.
    MPI_Init(&argc, &argv);
     ..........
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     ......
```

Programa MPI - SPMD

```
if (0 == rank)              //  rank = 0
     {
          ..........

     }
    else                    // rank > 0
    {
          ..........

    }

    MPI_Finalize();
    return(0);
}
```

MPI

- Enviando Mensagens

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

- Recebendo Mensagens

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

status: status.MPI_TAG

status.MPI_SOURCE

Exemplo: enviando mensagens

```
//  rank 0, send a message to rank 1.
    if (0 == rank) {
          for (i=0;i<64;i++)
            buf[i]=i;
       MPI_Send(buf, BUFSIZE, MPI_INT, 1, 11, MPI_COMM_WORLD);
    }

  //  rank 1, receive a message from rank 0.
    else {
          MPI_Recv(buf, BUFSIZE, MPI_INT, 0, 11, MPI_COMM_WORLD,
                                  &status);
           for(i=0;i<64;i++)
          {   printf("%d",buf[i]);
          fflush(stdout);}
           printf("\n");
    }
```

MPI – compilação e execução

- Compilação (openmpi)

mpicc -o trivial trivial.c

mpiCC –o trivial trivial.cpp

execução: mpirun -np 2 –hosts trivial

hosts: arquivo contendo os IPs ou nomes dos nós

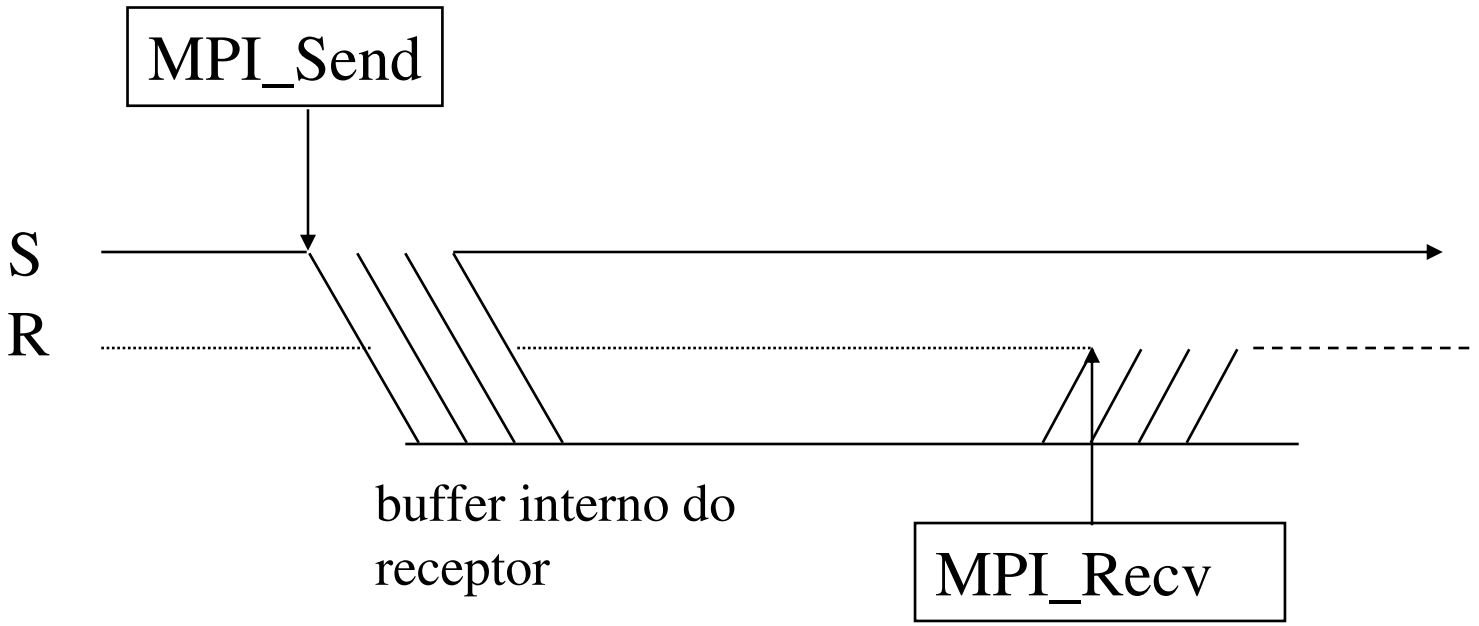
Comunicação

- Comunicação ponto a ponto: uma origem e um destino
  - Envio bloqueante
  - Recepção bloqueante
  - Envio não bloqueante
  - Recepção não bloqueante

- Comunicação Coletiva

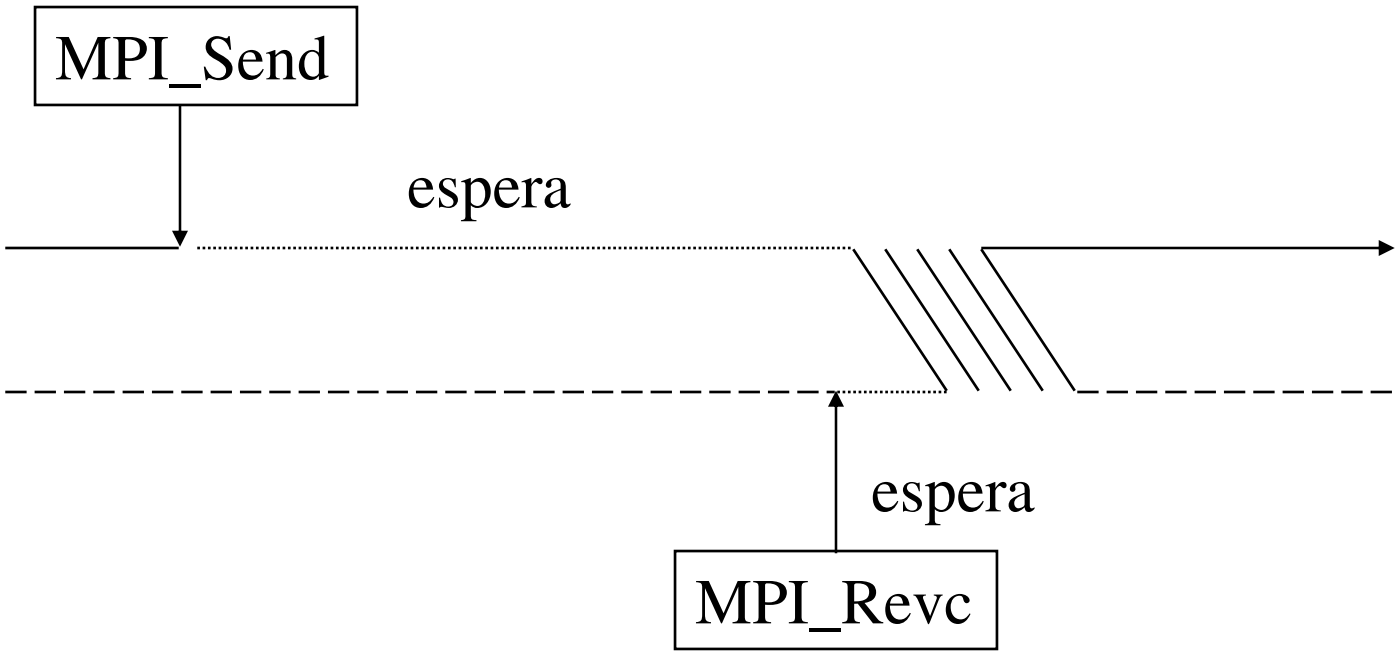
Envio Padrão Bloqueante

- O comportamento do sistema depende do tamanho da mensagem, se é menor ou igual ou maior do que um limite. Este limite é definido pela implementação do sistema e do número de tarefas na aplicação.
- mensagem<= limite



Envio Padrão Bloqueante

- mensagem>limite



Programação: modelo mestre-escravo em SPMD

```
void main()
int       argc;
char    *argv[];
{ int myrank;
   MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank==0) {master();}
 else { slave();}
 MPI_Finalize();
 return(0);
}
```

Exemplo: soma de elementos de um vetor (MPI_Send)

```
k=n/n_nos;
      inicio=rank*k;
      fim=inicio+k;
      if (rank==0) {
        for(i=inicio;i<n;i++)
          vetor[i]=1;
        for (i=1;i<n_nos;i++)
           MPI_Send(vetor+k*i,k,MPI_INT,i,10,MPI_COMM_WORLD);
      }
      else {

   MPI_Recv(vetor,k,MPI_INT,0,10,MPI_COMM_WORLD,&status);
      }
      soma_parcial=0;
      for(i=0;(i<k);i++)
        soma_parcial+=vetor[i];
```

Exemplo: soma de elementos de um vetor (MPI_Send)

```
if (rank==0) {
        soma_total=soma_parcial;
        for(i=1;i<n_nos;i++){
          MPI_Recv(&soma,1,MPI_INT,MPI_ANY_SOURCE,11,
                              MPI_COMM_WORLD,&status);
          soma_total+=soma;}
        }
      else {
        MPI_Send(&soma_parcial,1,MPI_INT,0,11,MPI_COMM_WORLD);
          }
   MPI_Finalize();
   return(0);
```

Envio Bloqueante

- **Padrão:**

MPI_Send(&buf,n_elementos,MPI_INT,dest, tag, MPI_COMM_WORLD)

- **Blocking Syncronous Send :**

MPI_Ssend (&buf,n_elementos,MPI_INT,
dest, tag, MPI_COMM_WORLD)

- tarefa transmissora: envia para a tarefa receptora uma mensagem "**ready to send**".
- tarefa receptora: ao executar uma chamada receive, envia para a tarefa transmissora uma mensagem de "**ready to receive**".
- Os dados são transferidos.

Envio Bloqueante

- **Blocking Ready Send**

MPI_Rsend (&buf,n_elementos,MPI_INT,
dest, tag, MPI_COMM_WORLD)

- envia a mensagem na rede.
- Requer que uma notificação de "**ready to receive**" tenha chegado.
- Notificação não recebida: erro.

Envio Bloqueante

- **Blocking Buffered Send**

MPI_Bsend(&buf,n_elementos,MPI_INT, dest, tag, MPI_COMM_WORLD)

- Aplicação deve prover o **buffer**: array alocado estaticamente ou dinamicamente com malloc. Deve ser incluido bytes do header.
- **MPI_Buffer_attach**(...)
- Dados do buffer de mensagem copiados para buffer do usuário. Execução retorna.
- O dado será copiado do buffer do usuário sobre a rede quando uma notificação de “ready to receive”tiver chegado
- **MPI_Buffer_detach**(...)

Recepção Bloqueante

- **MPI_Recv(&a,10,MPI_INT,origem,tag,MPI_COMM_WORLD,&status)**

  bloqueante: receptor fica bloqueado até receber a mensagem.

  tag: pode ser MPI_ANY_TAG (qualquer tag)

  origem: pode ser MPI_ANY_SOURCE (qualquer origem)

  status: 2 campos

  status.source: origem da mensagem

  status.tag: tag da mensagem

Recepção: outras funções

- **MPI_Probe(in source,int tag,MPI_Comm comm,MPI_Status *status)**

  Sincroniza uma mensagem e retorna informações. Não retorna até que uma mensagem seja sincronizada.
- **MPI_Get_count(MPI_Status *status,MPI_Data type dtype,int *count)**

  Retorna o número de elementos da mensagem recebida.

  uso: quando não se conhece o tamanho da mensagem a ser recebida. Mensagem sincronizada com MPI_Probe.

Funções de comunicação não bloqueantes

- chamadas não bloqueantes retornam imediatamente após o início da comunicação. O programador não sabe se o dado enviado já saiu do buffer de envio ou se o dado a ser recebido já chegou. Então, o programador deve verificar o seu estado antes de usar o buffer.

Comunicação não bloqueante

- **envio**: retorna após colocar o dado no buffer de envio.

MPI_Isend(void *buf,int count,MPI_Datatype dtype,int dest,int tag,MPI_Comm comm,MPI_Request *req)

req: objeto que contém informações sobre a mensagem, por exemplo o estado da mensagem.

Comunicação não bloqueante

- **Recepção**: é dado o início à operação de recepção e retorna.

MPI_Irecv(void *buf, count,MPI_INT, origem, tag, MPI_Comm comm, MPI_Request *req)

- Dados: não devem ser lidos enquanto não estiverem disponíveis

  Verificação: **MPI_Wait** ou **MPI_Test**.

Comunicação não bloqueante

MPI_Wait(MPI_Request *req,MPI_Status *status)

Espera completar a transmissão ou a recepção.

MPI_Test(MPI_Request *req,int *flag,MPI_Status *status)

Retorna em flag a indicação se a transmissão ou recepção foi completada. Se true, o argumento status está preenchido com informação

MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag,MPI_Status *status)

Seta flag, indicando a presença do casamento da mensagem.

Comunicação não bloqueante

```
if (rank==0) {
 ..............

MPI_Isend(&a[0][0]+(k*n*i),k*n,MPI_DOUBLE,i,10,MPI_COMM_WO
RLD,&req);
printf("rank 0 apos Send\n");
fflush(stdout);
.......

}
else {

MPI_Recv(a,k*n,MPI_DOUBLE,0,10,MPI_COMM_WORLD,&status);

.............

}
```

Comunicação coletiva

MPI_Bcast(void *buf,int count,MPI_Datatype dtype,int root,MPI_Comm comm);

- todos processos executam a mesma chamada de função.
- Após a execução da chamada todos os buffers contêm os dados do buffer do processo root.

Comunicação coletiva

MPI_Scatter(void *sendbuf,int sendcount,

MPI_Datatype sendtype,void *recvbuf,

int recvcount,MPI_Datatype recvtype,int root,

MPI_Comm comm);

- todos os N processos do comunicador especificam o mesmo count (número de elementos a serem recebidos).
- O buffer de root: contém sendcount*N elementos do datatype tipo especificado.
- processo root: distribuirá os dados para os processos, incluindo ele próprio.

Comunicação coletiva

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

- operação gather: reverso da operação scatter (dados de buffers de todos processos para processo root)

Exemplo: MPI_Scatter

```
if (rank==0) {
        for(i=0;i<n;i++)
          for(j=0;j<n;j++)
            a[i][j]=1;
      }
   MPI_Scatter(a,k*n,MPI_DOUBLE,a,k*n,MPI_DOUBLE,0
   ,MPI_COMM_WORLD);            //  rank 0: root
      parcial=0;
      for(i=0;(i<k);i++)
       for(j=0;j<n;j++)
        parcial+=a[i][j];
```

Comunicação coletiva

- MPI_Reduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype dtype,MPI_Op op,int root,MPI_Comm comm);
- Elementos dos buffers de envio: combinados par a par para um único elemento correspondente no buffer de recepção do root.
- Operaçoes de redução:
- MPI_MAX (maximum), MPI_MIN (minimum), MPI_SUM (sum), MPI_PROD (product), MPI_LAND (logical and), MPI_BAND (bitwise and) MPI_LOR (logical or) MPI_BOR (bitwise or) MPI_LXOR (logical exclusive or) MPI_BXOR (bitwise exclusive or)

Exemplo: MPI_Reduce

```
k=n/n_nos;
MPI_Scatter(a,k*n,MPI_DOUBLE,a,k*n,MPI_DOUBLE,0,M
    PI_COMM_WORLD);  // rank 0: root
parcial=0;
fim=k;
for (i=0;i<fim;i++)
          for (j=0;j<n;j++)
             parcial=parcial+a[i][j];
MPI_Reduce(&parcial,&soma,1,MPI_DOUBLE,MPI_SUM
    ,0,MPI_COMM_WORLD);   // rank 0:  root
```

Aplicação

Projeto OpenModeller

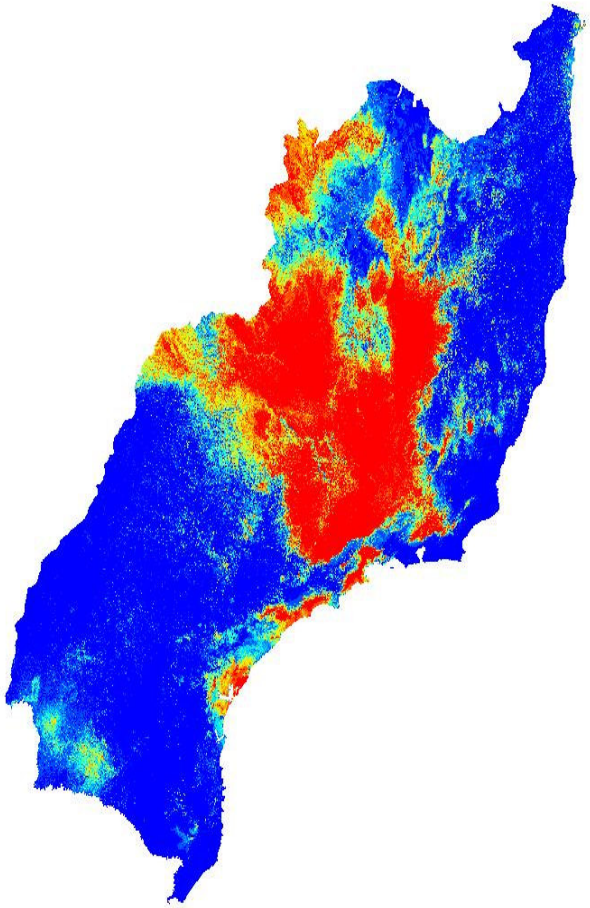
Objetivo: construir uma versão eficiente da ferramenta OpenModeller para predizer a distribuição de espécies em uma área geográfica a partir de informações de presença em um conjunto de locais e as condições topográficas e climáticas.

Instituções: CRIA, EPUSP, INPE

Participação do LAHPC: paralelização da aplicação

Biodiversidade

- Mapa de distribuição da espécie



OpenModeller

Duas etapas:

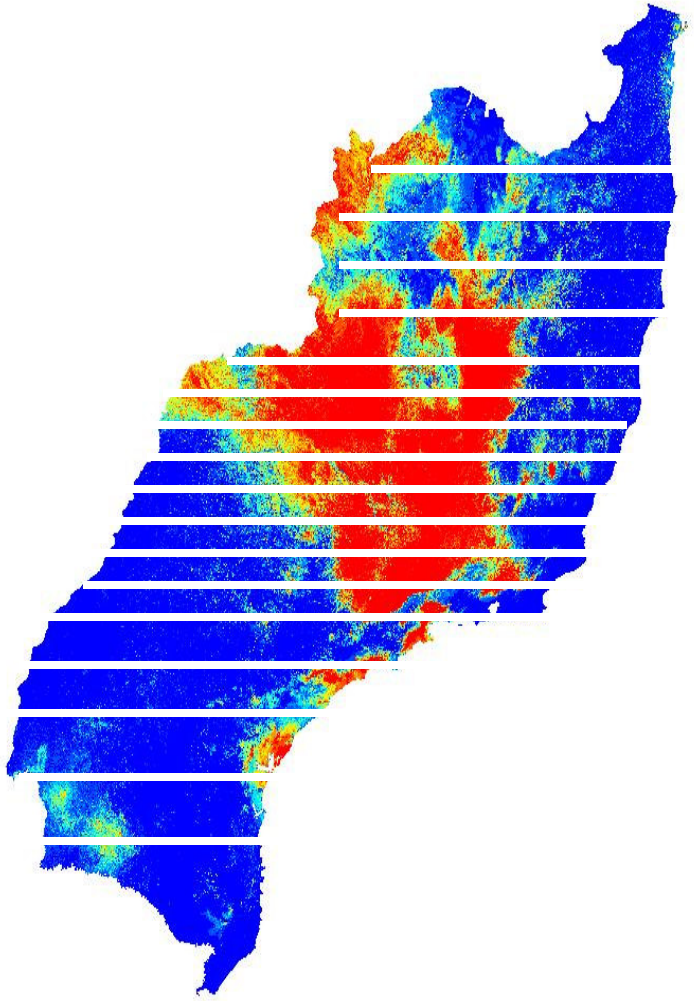
- Modelagem: modelagem da presença da espécie considerando condições climáticas e geográficas (layers)
  - dados de presença coletados em campo
- Projeção: prediz a presença da espécie em pontos de uma área

Biodiversidade

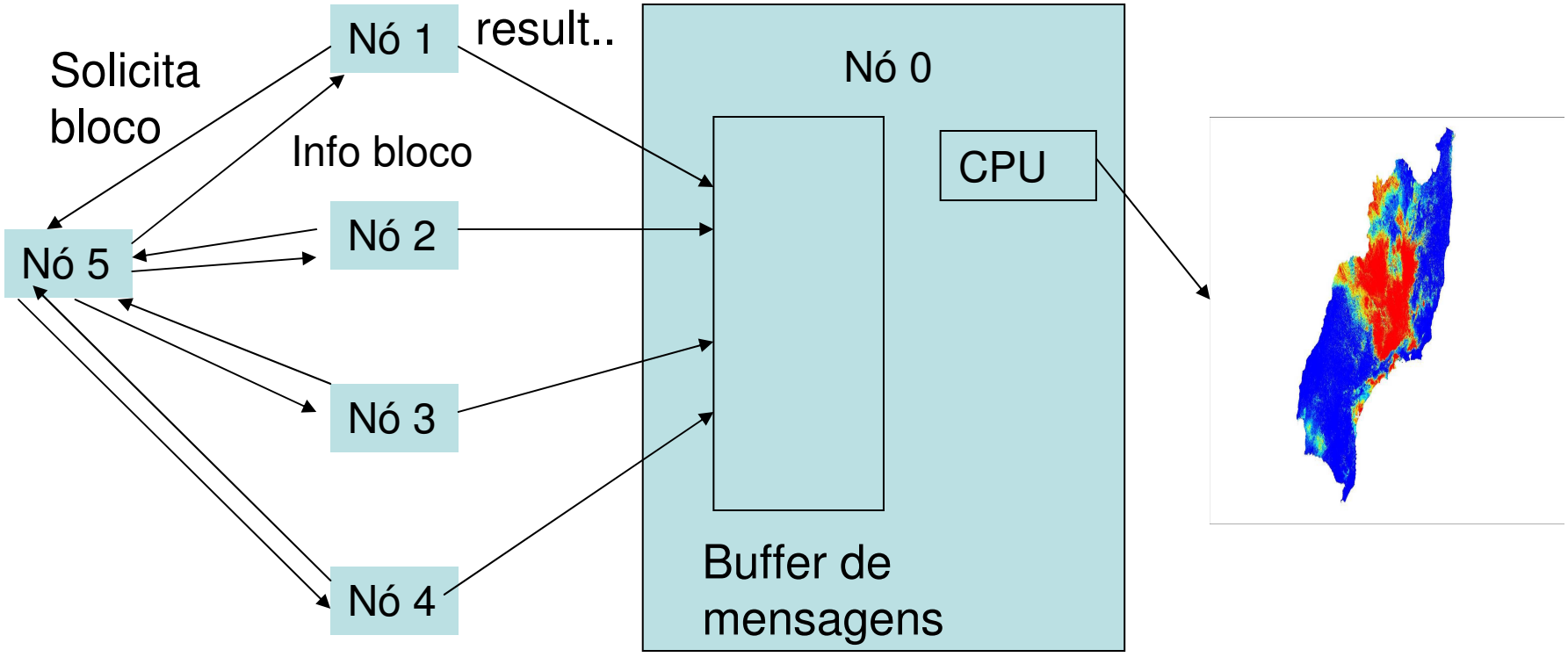
- Experimento:
  - Predição da distribuição de uma espécie para um conjunto de camadas ambientais cortados para a região sudeste com alta resolução (250m).
  - Paralelização do passo de projeção.
- Plataforma de execução:cluster

  Programação utilizando MPI

experimento



Estratégia de paralelização



Biodiversidade

- Tempo de execução sequencial: 66 min.
- Tempo de execução da projeção na aplicação openModeller versão paralelizada:

Conclusão

Impacto da computação de alto desempenho nas áreas da computação

- Novas ferramentas de programação
- Metodologias e Ferramentas de desenvolvimento
- Novos algoritmos

Desafios

- Facilitar a programação
- Facilitar o uso das plataformas (multi-core, clusters, grids)
- Disseminar como utilizar e explorar o uso dos recursos destas plataformas

Conclusão

Pesquisas

- construção de grids para processamento paralelo
  - MPI utilizando múltiplos clusters e servidores
- ambientes de programação paralela
  - ideal: mesma linguagens atendendo estas plataformas
- clusters heterogêneos
- uso de outras arquiteturas (GPUs)
- paralelização de aplicações

Bibliografia

1. Ben-Ari, M. "Principles of Concurrent and Distributed Programming", Addison-Wesley, Second Edition, 2006.
2. Akthter, S.; Roberts, J. " Multi-Core Programming"; Intel Press, 2006.
3. Grama, A.; Gupta,A.; Karypis, G.; Kumar, V. "Introduction to Parallel Computing"; Addison-Wesley, Second Edition, 2003.
4. Culler, D.E.;Singh, J.P.;Gupta, A. "Parallel Computer Architecture: a hardware/software approach"; Morgan Kaufmann Publishers, Inc. , 1999.
5. Quinn, M.J. “Parallel Programming in C with MPI and OpenMP”; McGraw-Hill – Higher Education, 2004.