Do It Fast, But Do It Right: Introdução à Programação Paralela

João Marcelo Uchôa de Alencar
Quixadá - UFC

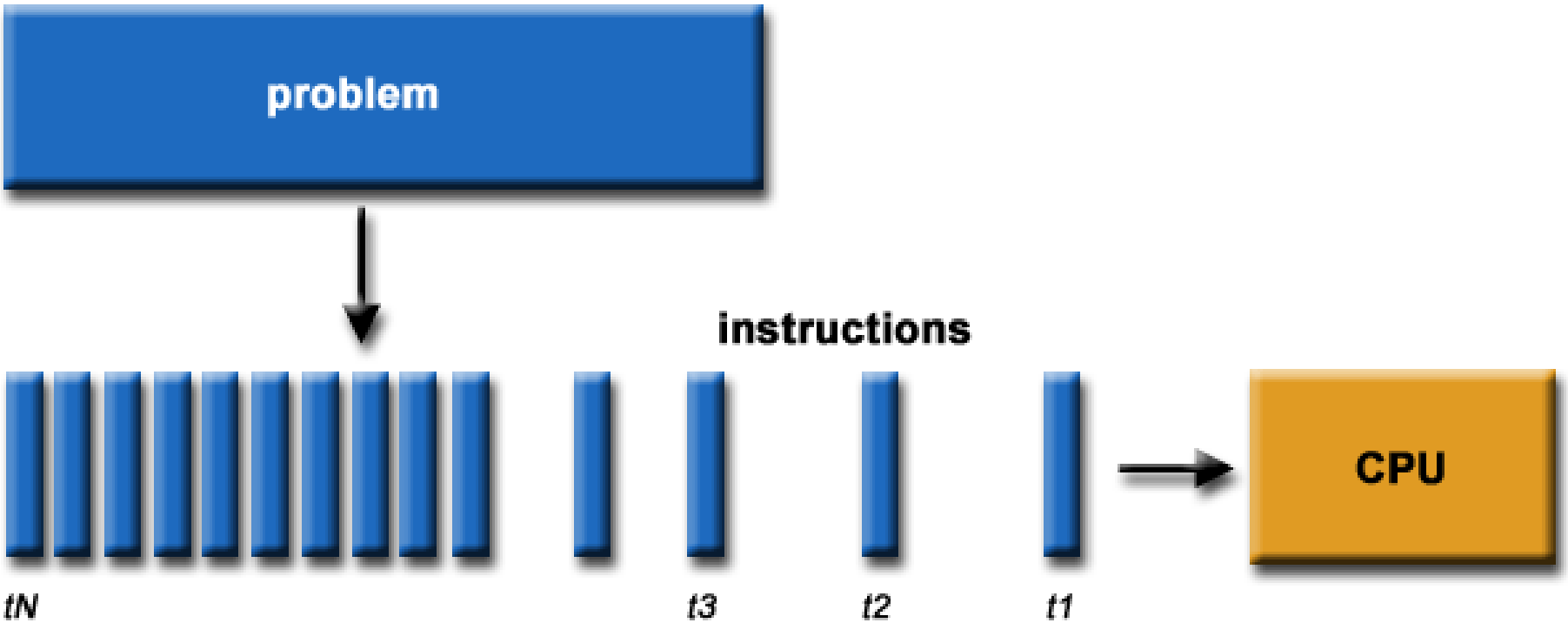
Agenda

Agenda

O que é Computação Paralela?

- Somos acostumados a programar pensando na execução serial:
  - Nosso código executa em uma CPU;
  - o compilador transforma nosso programa em um conjunto de instruções;
  - essas instruções são executadas uma após a outra;
  - apenas uma instrução executa por vez.
- Apesar de pensarmos serial, o computador moderno é paralelo por natureza:
  - Poucos computadores possuem apenas um núcleo;
  - o próprio compilador tenta paralelizar suas instruções;
  - as instruções podem executar fora de ordem devido aos *pipelines*;
  - dentro do *pipeline*, várias instruções podem estar em diferentes estágios.

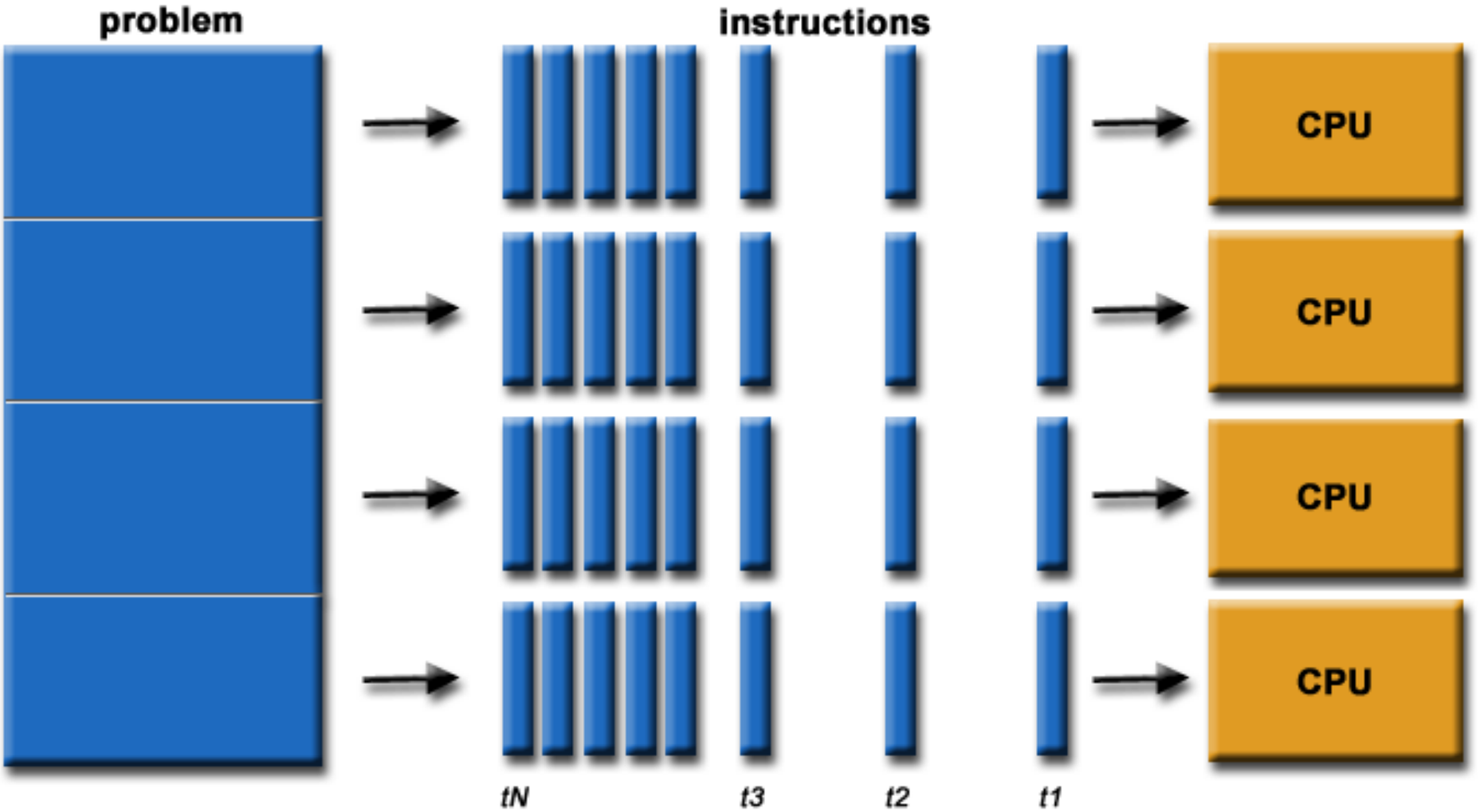
Execução Serial



Execução Paralela

- Podemos aproveitar melhor nossos computadores se, como eles, também pensarmos em paralelo:
  - Criar código para execução em vários processadores;
  - pensar no problema como tarefas individuais que podem ser resolvidas separadamente;
  - cada tarefas pode ser dividida em instruções que executam em processadores diferentes;
  - um mecanismo de coordenação de tarefas se faz necessário.

Execução Paralela



Nós vivemos em um Mundo Paralelo

- Em qualquer sistema da realidade, muitos agentes interagem entre si, com reações de causa e feito que se espalham pelo espaço no decorrer do tempo...
- Imagine o tráfego de uma grande cidade.
- Como os elementos naturais interagem para formar a percepção do clima?
- Como a molécula de uma vacina atua no organismo, incentivando a criação de anticorpos?

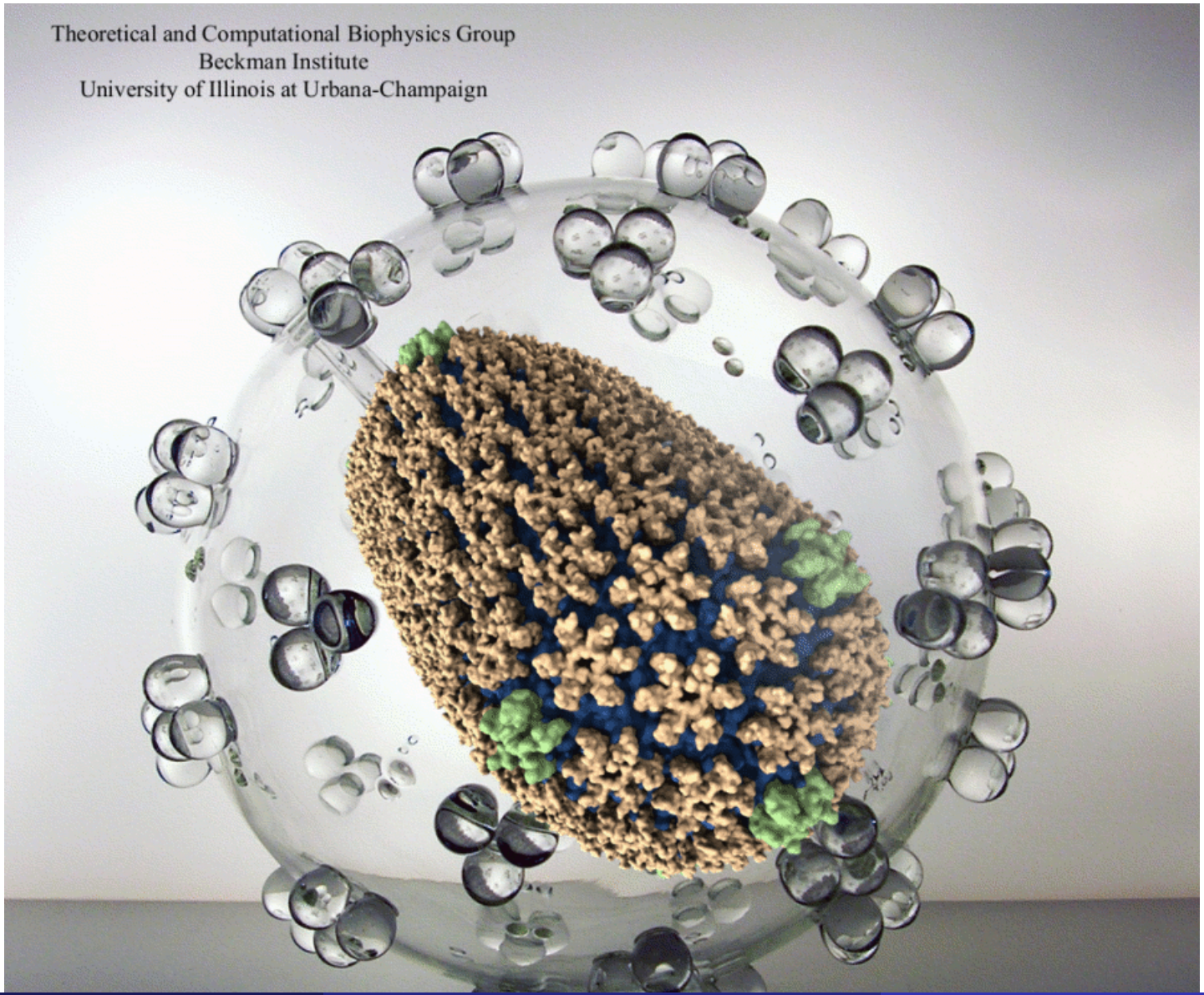
Conclusão

Para estudar ou simular o mundo real com o uso de computadores, é mais natural pensar em programas paralelos.

Exemplos - Dinâmica Molecular

- HIV1 - Variante do vírus da AIDS responsável pela maioria dos casos.
  - O vírus ataca as células infectando-as com uma molécula chamada capsídeo;
  - essa estrutura infecta a célula com o material genético do vírus, passando a controlá-la;
  - uma maneira de combater a doença é blindar a célula contra capsídeos.
- Dinâmica Molecular Computacional
  - Utilizando um supercomputador cientistas conseguiram simular o ataque do HIV1.
  - Blue Waters - 49.000 CPUs - `http://www.ncsa.illinois.edu/enabling/bluewaters`
  - O resultado foi a criação de uma estrutura que envolve o capsídeo, diminuindo a proliferação do HIV no corpo.

Exemplos - Dinâmica Molecular



Exemplos - Decodificação de Atividade Cerebral

- O cérebro é formado por bilhões de neurônios.
  - Nossos pensamentos, ações e memórias estão codificados nessas estruturas;
  - cada ação ou mudança de estado no nosso cérebro acarreta uma tempestade de correntes elétricas através das conexões dos neurônicos;
  - é possível capturar o estado de cada neurônio e decodificar o pensamento?
- Miguel Nicolelis - Pesquisador Brasileiro
  - Sensores inseridos no cérebro de macacos *rhesus*;
  - informações são coletadas em *cluster*;
  - pequenas ações e movimentos são decodificados.

Exemplos - Jogos Eletrônicos

- Jogos utilizam programação paralela massivamente:
  - Renderização de ambientes gráficos, incluindo sombras, texturas, etc;
  - simulação das leis da física, gravidade, impactos;
  - inteligência artificial para determinar o comportamento dos personagens;
  - controle de entrada e saída para comandos do jogador, conexão *multiplayer*;
  - um universo muito extenso.
- Gerações de Consoles
  - Cada geração costuma utilizar o que há de melhor em termos de arquitetura;
  - mas nem sempre é fácil desenvolver para essas plataformas;
  - os primeiros jogos do PS3 pareciam muito com os jogos do PS2;
  - no decorrer dos anos, programadores dominaram o *hardware* e desenvolveram jogos superiores.

Exemplos - Mecânica de Fluídos

- É ciência que estuda o comportamento dos fluídos (gases e líquidos).
  - Previsão do tempo;
  - projeto de prédios e edificações;
  - projeto de veículos;
  - desenvolvimento de motores;
  - simulação de turbina eólica.
- Fórmula 1
  - Na fase de projeto, as equipes fazem uso de CFD para criação dos carros;
  - os dados produzidos servem para criar moldes em menor escala que são validados no túnel de vento;
  - nos treinos, os engenheiros validam o modelo e realimentam o computador com novos dados;
  - o processo se repete, contribuindo para a evolução do carro no decorrer na temporada.

Exemplos - Diversos

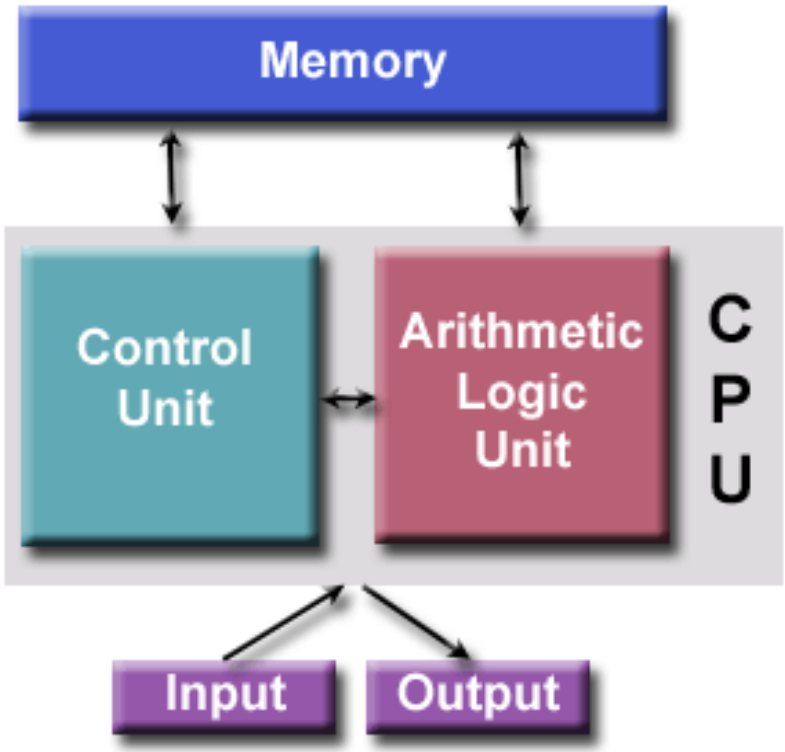
- Mineração de Dados;
- Exploração de Petróleo (também CFD);
- Diagnóstico Médico;
- Criação de Drogas (Dinâmica Molecular);
- Modelagem Financeira e Econômica;
- Realidade Virtual;
- e por aí vai...

E eu, pobre mortal?

- A maioria dos ambientes de desenvolvimento utiliza paralelismo direta ou indiretamente:
  - Interfaces gráficas;
  - conexões com fontes de dados;
  - comunicação em rede.
- No Brasil, várias empresas usa programação paralela:
  - Petrobras modela reservatórios de petróleo;
  - Banco Central executa modelos financeiros;
  - várias indústrias utilizam paralelismo no projeto de produtos e simulação de processos industriais;
  - `http://portais.fieb.org.br/portal_faculdades/apresentacao-mcti.html`.
- À medida que a economia do estado avança e se globaliza, esse conhecimento será cada vez mais valorizado.

Arquitetura Básica de uma CPU

- Arquitetura de von Neumann;
  - Memória de leitura e escrita, com acesso aleatório;
  - unidade de controle;
  - unidade de lógica aritmética;
  - entrada e saída.
- As instruções são códigos que orientam as unidades da CPU.

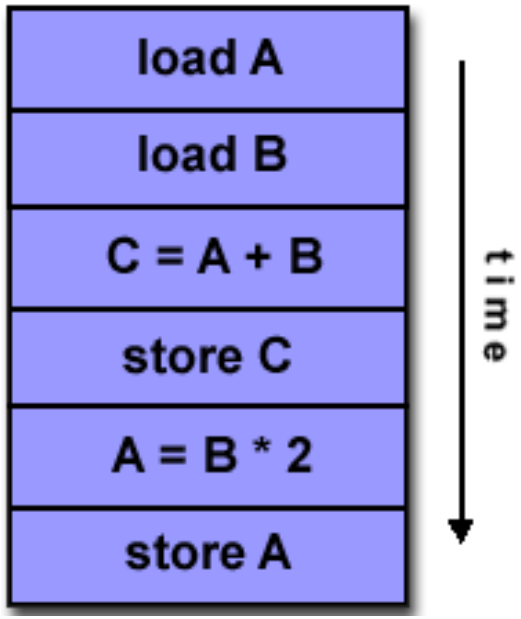


Taxonomia de Arquiteturas Paralelas

- Esta classificação leva em consideração os fluxos de instrução em execução:
  - SISD - *Single Instruction, Single Data*
  - SIMD - *Single Instruction, Multiple Data*
  - MIMD - *Multiple Instruction, Multiple Data*
  - MISD - *Multiple Instruction, Single Data*
- Detalhes de como o *hardware* é organizado são desconsiderados no momento.

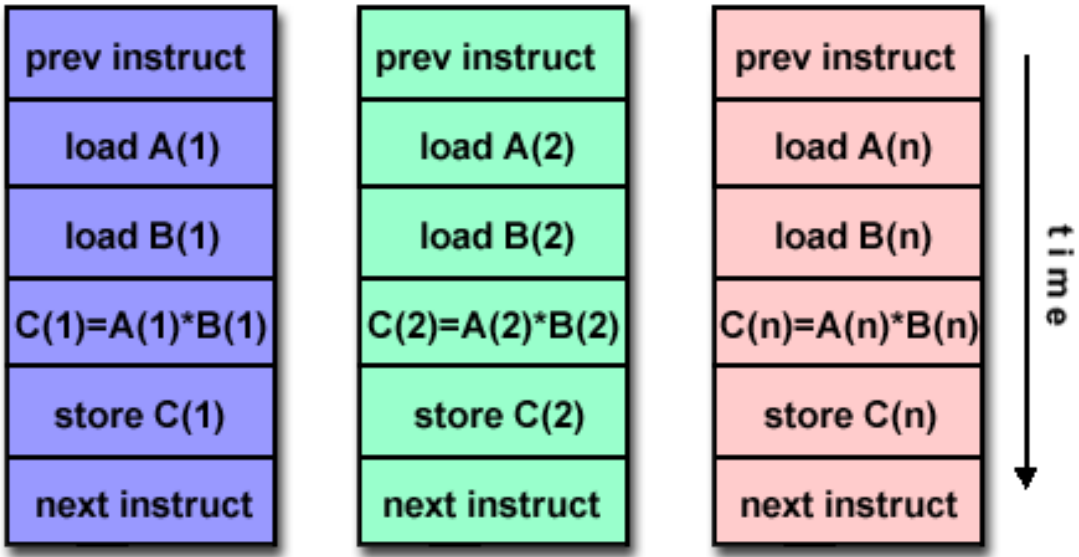
Single Instruction, Single Data

- Um computador serial, comum há alguns anos;
- **Single Instruction**: em um ciclo de *clock*, apenas um fluxo de instruções na CPU;
- **Single Data**: apenas um fluxo de dados;
- execução determinística: não importa quantas vezes você execute o programa, as instruções serão executadas sempre na mesma ordem.



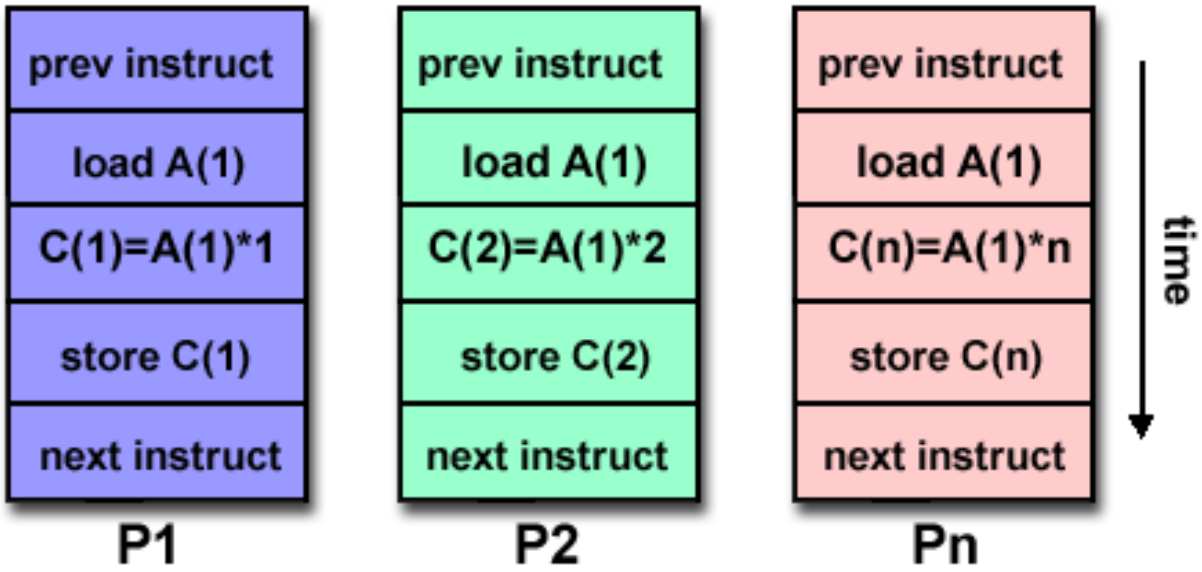
Single Instruction, Multiple Data

- Arquitetura comum de computadores paralelos;
- **Single Instruction**: em um ciclo de *clock*, apenas um fluxo de instruções executando em várias unidades de processamento;
- **Multiple Data**: para cada unidade de processamento, um fluxo de dados diferente;
- ótima opção para aplicações regulares, como processamento de imagens;
- um bom exemplo são as placas gráficas da NVIDIA e AMD/ATI.



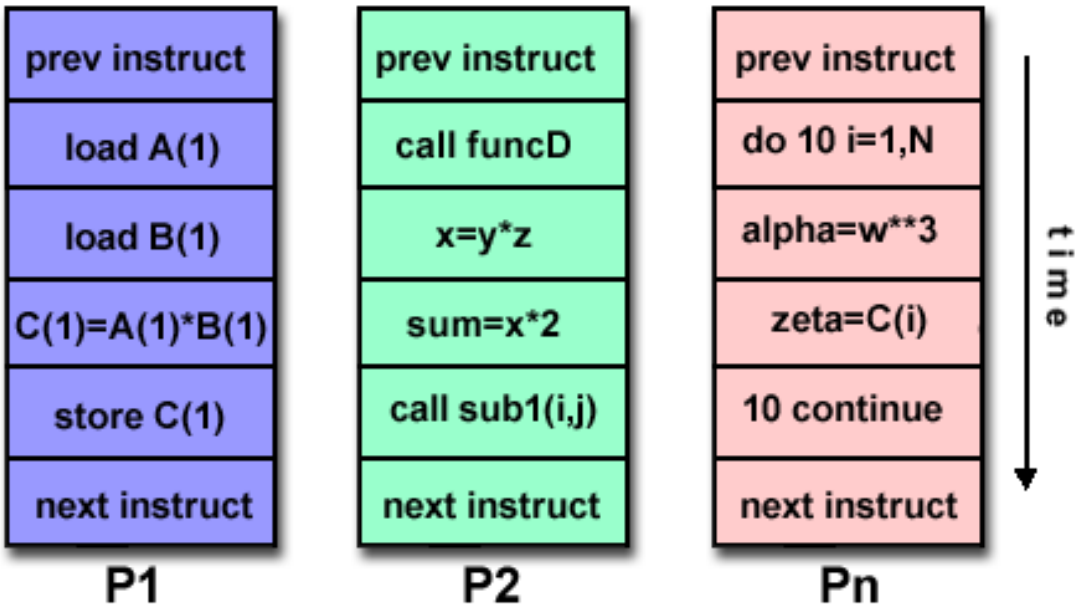
Multiple Instruction, Single Data

- Exemplo raro;
- **Multiple Instruction**: cada unidade de processamento executa um fluxo diferente de instruções no mesmo fluxo de dados;
- **Single Data**: apenas um fluxo de dados;
- exemplo: vários algoritmos de criptografia diferentes tentando decodificar a mesma mensagem.



Multiple Instruction, Multiple Data

- Arquitetura mais geral;
- **Multiple Instruction**: cada unidade de processamento executa um fluxo diferente de instruções no mesmo fluxo de dados;
- **Muiltiple Data**: para cada unidade de processamento, um fluxo de dados diferente;
- exemplos: computadores *multicore*.



Organização da Memória de Computadores Paralelos

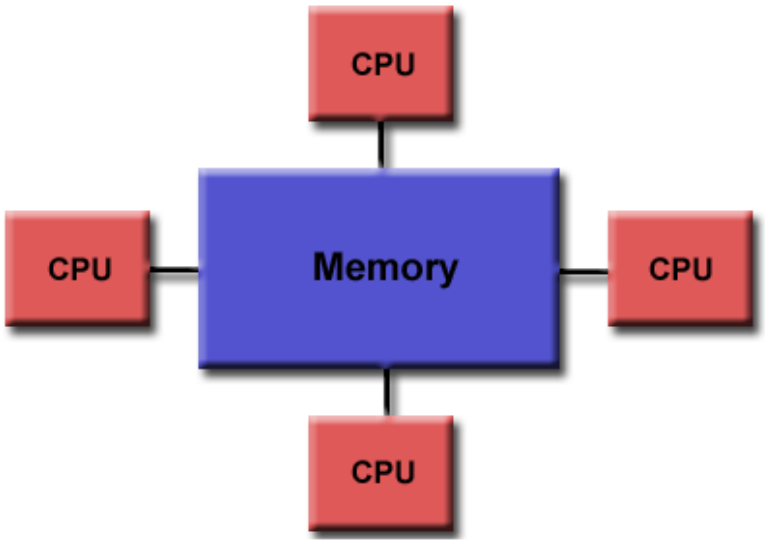
- Esta classificação leva em consideração como a memória é organizada:
  - Memória compartilhada;
  - memória distribuída;
  - sistemas híbridos.
- Nessa classificação, a estrutura interna dos núcleos é considerada.

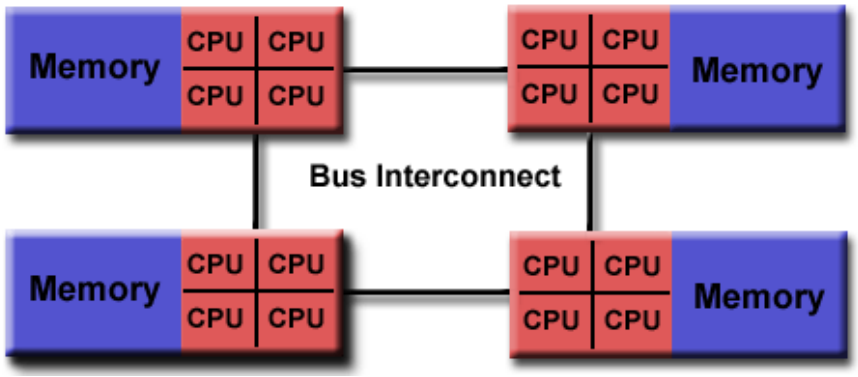
Memória Compartilhada

- Todos as unidades de processamento podem acessar qualquer endereço de memória. O endereçamento é global;
- os processadores podem executar fluxos de instruções diferentes, mas compartilham a mesma memória;
- qualquer mudança feita por um processador na memória é visível para todos;
- UMA - *Uniform Memory Access*:
  - Máquinas SMP - *Symmetric Multiprocessor*;
  - processadores idênticos;
  - tempos de acesso igual à qualquer posição da memória;
- NUMA - *Non-Uniform Memory Access*:
  - Geralmente, é feita pela interconexão de dois SMP;
  - nem todos os processadores têm tempo de acesso igual à memória;
  - acesso de memória através da interconexão é mais lento.

Memória Compartilhada

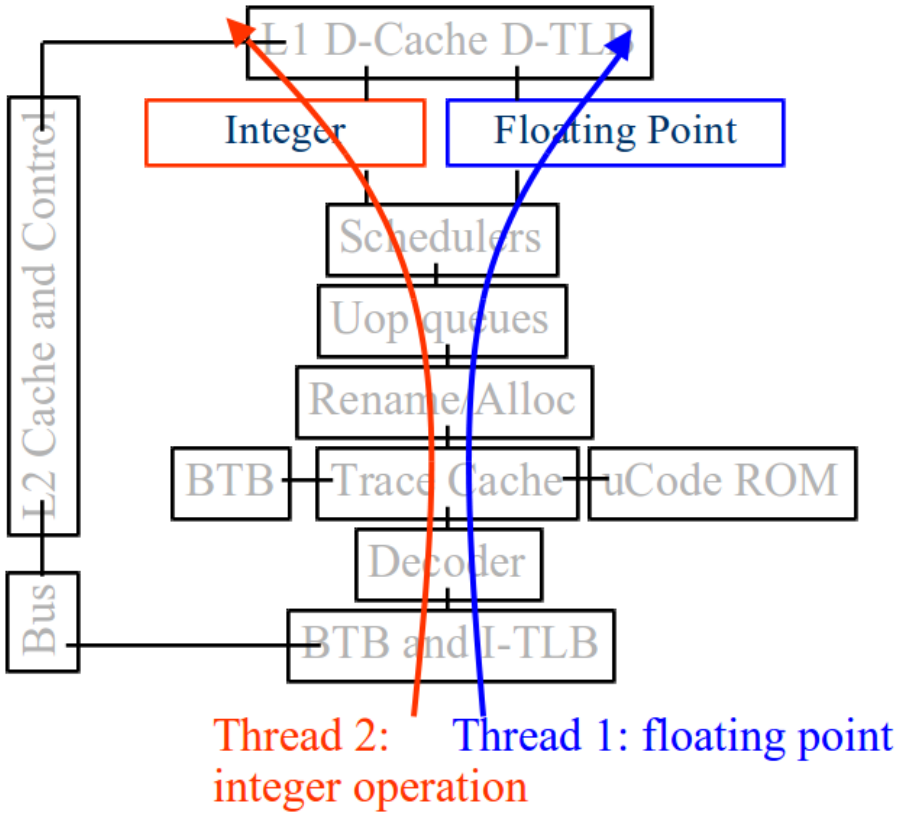
- Vantagens:
  - Mais fácil de programar;
  - compartilhamento de dados é rápido.
- Desvantagens:
  - Baixa escalabilidade;
  - o programador deve tratar a sincronização;
  - é mais caro.





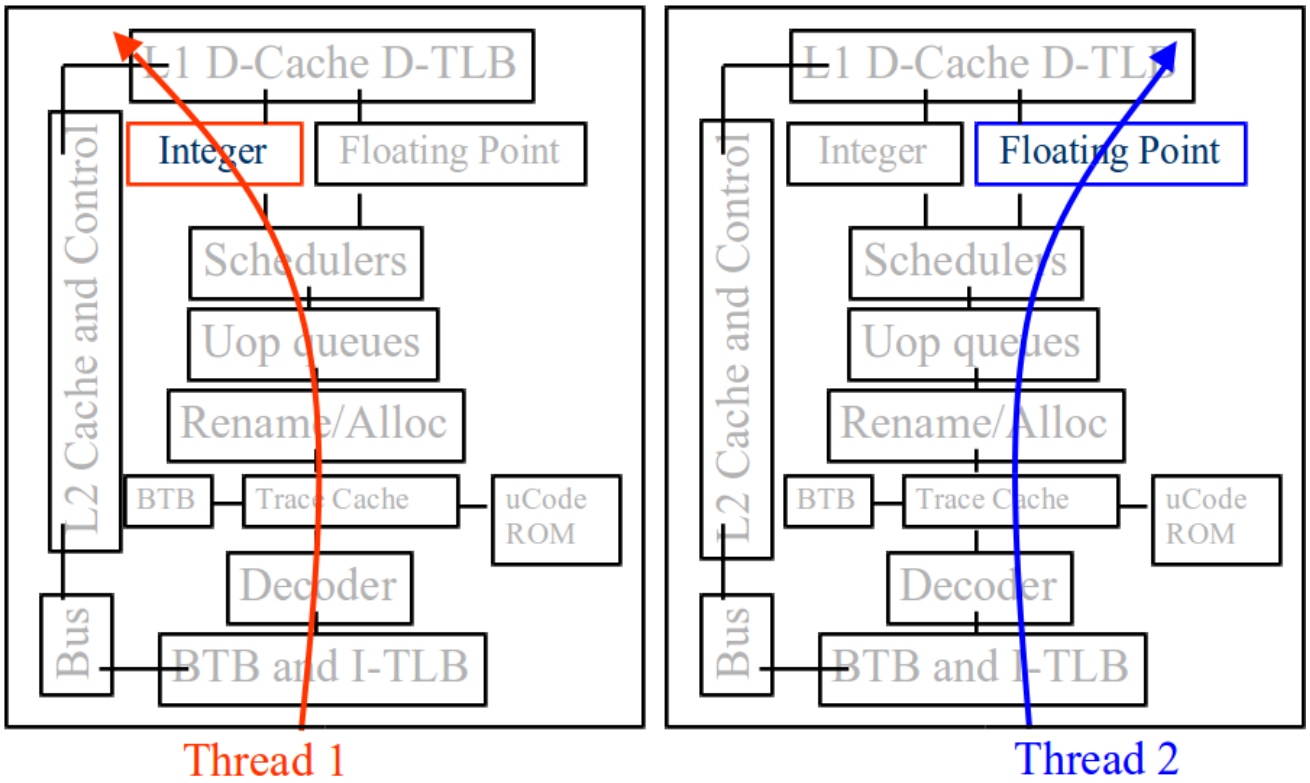
Memória Compartilhada - Exemplos

- *Simultaneous Multithreading - SMT*
  - Ainda é apenas um núcleo, mas algumas estruturas internas duplicadas;
  - pode executar em paralelo uma operação inteira e outra em ponto flutuante;
  - é o famoso *Hyper Threading* da Intel.



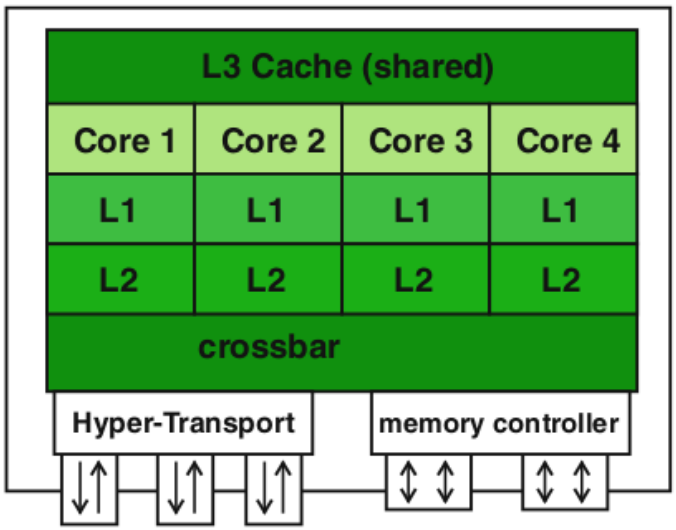
Memória Compartilhada - Exemplos

- *Symmetric multiprocessing - SMP*
  - Mais de um núcleo, todos iguais;
  - cada núcleo pode executar um fluxo diferente de instruções e dados;
  - faz uso de um barramento ou rede de interconexão;
  - maioria dos processadores atuais se encaixa.

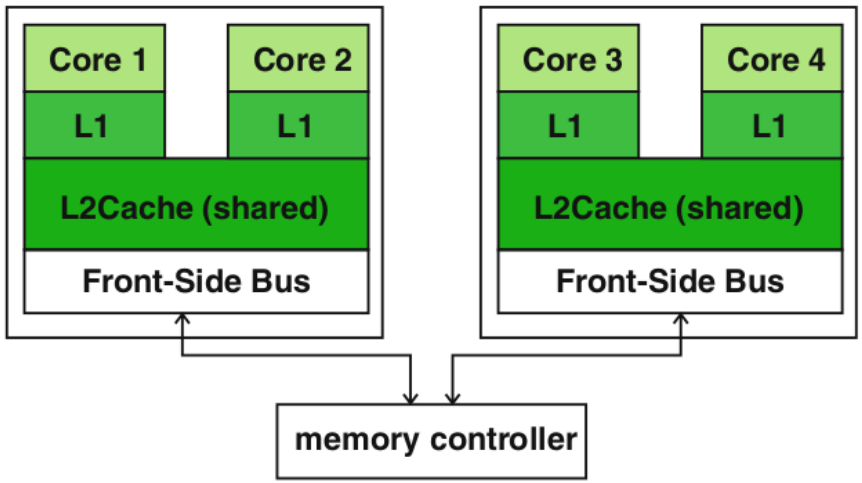


Memória Compartilhada - Exemplos

AMD Opteron



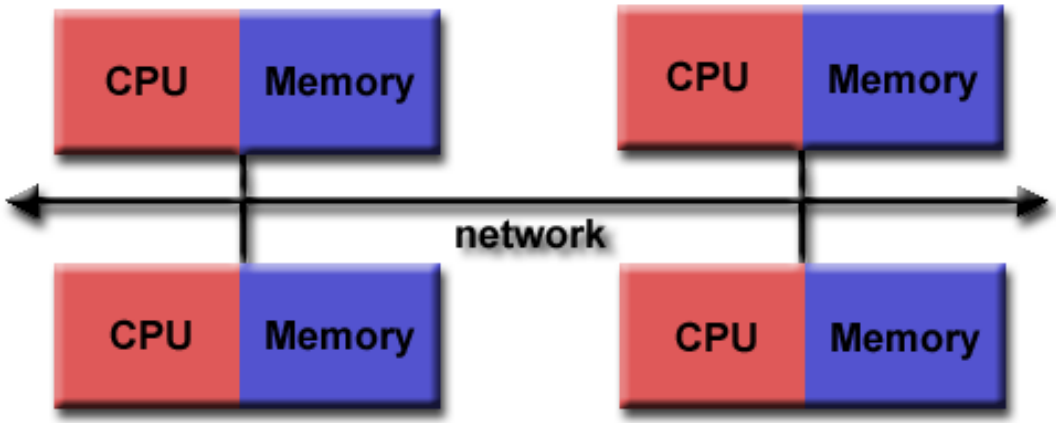
Intel Xeon



Memória Distribuída

- É necessária troca de mensagens para acessar a memória de um processador distinto;
- cada processador tem sua memória local. Entretanto, não há um endereçamento global envolvendo todas as memórias locais;
- é papel do programador invocar e sincronizar as rotinas de troca de mensagens;
- Vantagens:
  - Maior escalabilidade;
  - sistemas podem ser construídos com máquinas *desktop* e rede *ethernet*.
- Desvantagens:
  - Lidar com a comunicação é um trabalho árduo para o programador;
  - redefinição de estruturas de dados;
  - acessar dados de outro processador pode ser lento.

Memória Distribuída - Exemplos

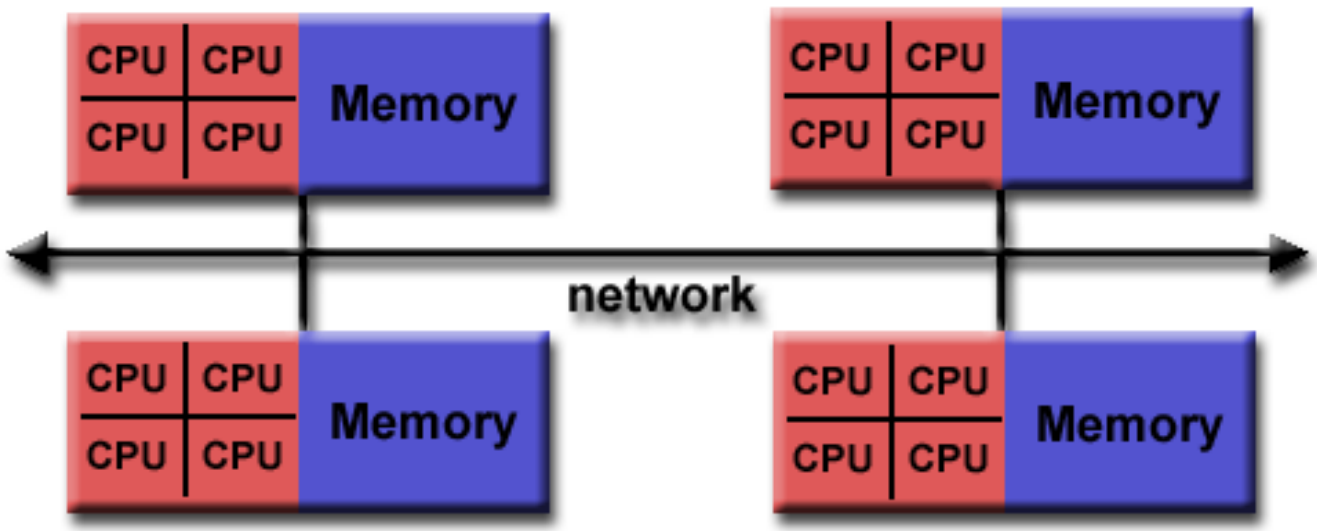


Cluster Beowulf

Cluster DELL

Hierarquia de Memória Híbrida

- É a opção mais comum da atualidade;
- existe memória compartilhada em servidores com CPUs ou GPUs;
- a comunicação entre os servidores é através de troca de mensagens, ou seja, memória distribuída;
- Vantagens:
  - Maior escalabilidade.
- Desvantagens:
  - Programação complexa.

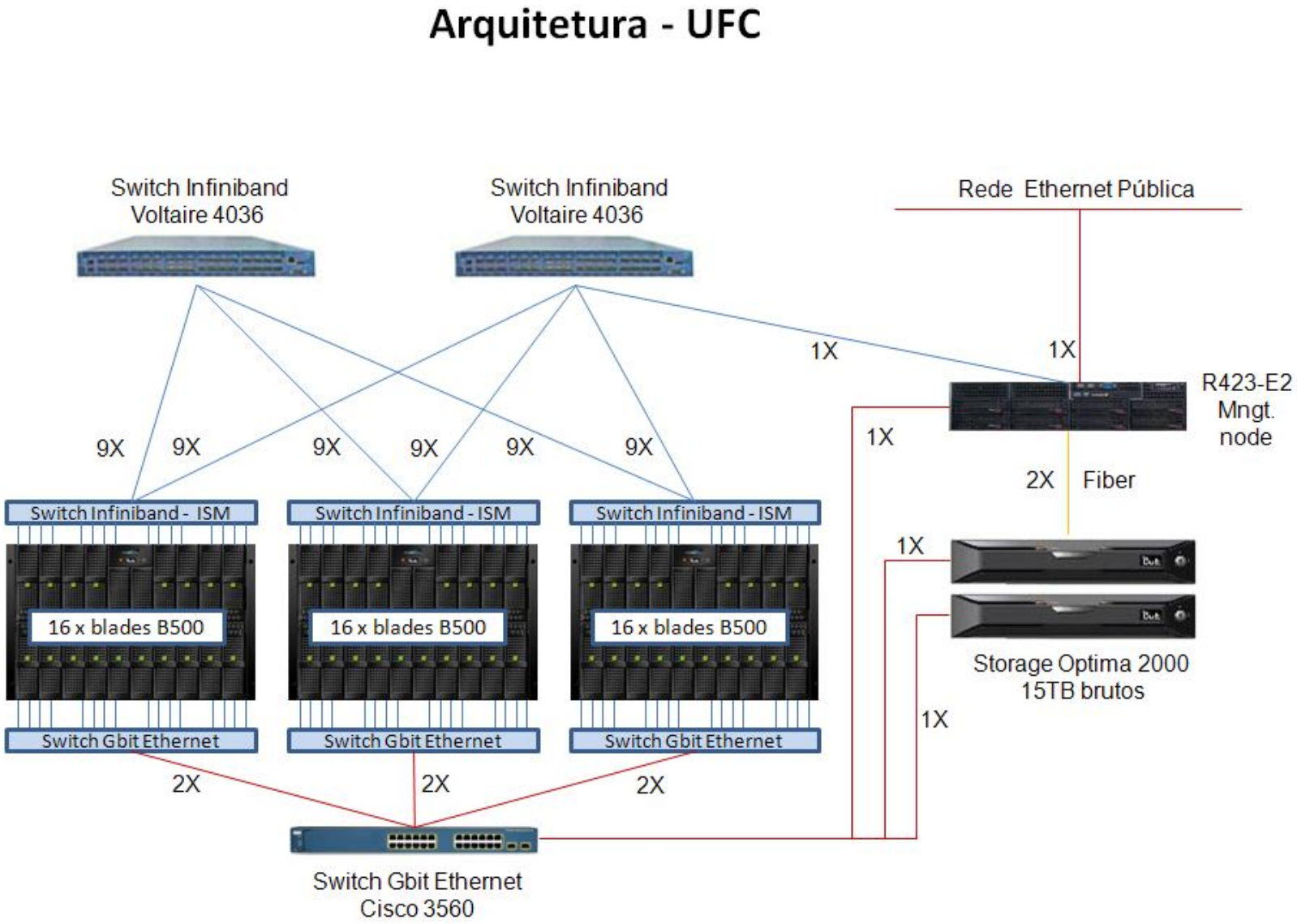


Memória Híbrida - Exemplos

- *Job* ou tarefa: programa paralela submetido para execução no *cluster*;
- Nó ou *blade*: um servidor;
- Nós de processamento: servidores que executam aplicações paralelas;
- Nó de gerência: servidor que controla o *cluster*, onde os *jobs* são submetidos e atividades administrativas são realizadas;
- *Storage*: nó que exporta um sistema de arquivos para todo o *cluster*.

Cluster CENAPAD-UFC

Memória Híbrida - Exemplos



Interlúdio Programático

- Uma pausa para assimilar os conceitos e pensar em outras coisas;
- Pergunta básica 1: sabe compilar um programa em C no Linux?
- Pergunta básica 2: como trabalhar com matrizes alocadas dinamicamente em C?

Compilação de Código C no Linux

```
jmhal@earth:~ gcc -pthread teste.c -o teste
```

-pthread: indica que vamos usar a biblioteca PThread em teste.c;
teste.c: código fonte;
-o teste: binário gerado;
existe muitos mais detalhes, mas por enquanto é só.

Trabalhar com matrizes em C

Matriz MxN (M linhas por N colunas)

```
int N = 100;
int M = 100;
int **matrix;
matrix = (int **) malloc(M * sizeof(int*));
int i;
for (i = 0; i < M; i++)
     matrix[i] = (int *) malloc(N * sizeof(int));

// Acessar o elemento (50,50) da Matriz
int i = j = 50;
matrix[i][j] = 100;
```

É essa a melhor maneira? Quantas invocações de *malloc*? Quantas conversões de ponteiros?

Trabalhar com matrizes em C

Matriz MxN (M linhas por N colunas)

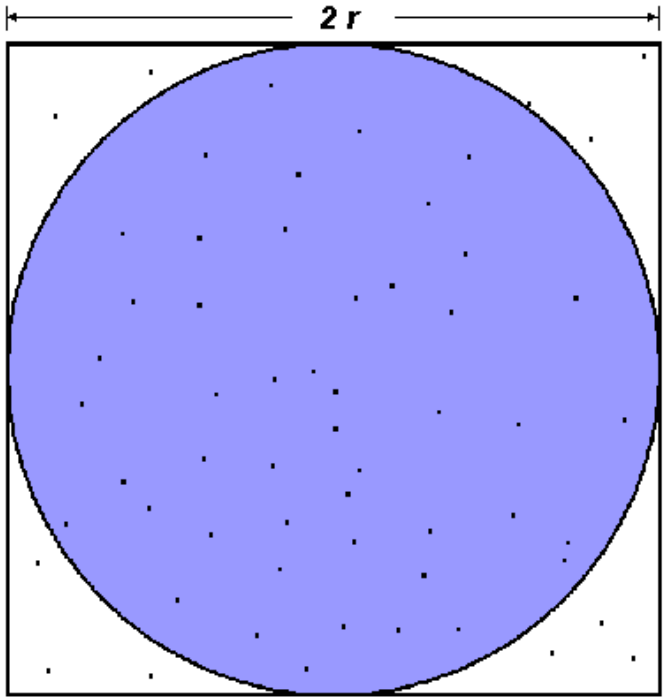
```
int N = 100;
int M = 100;
int *matrix;
matrix = (int *) malloc(M * N * sizeof(int));

// Acessar o elemento (50,50) da Matriz
int i = j = 50;
matrix[i * N + j] = 100;
```

Somente uma *malloc*. Alocação contígua, mais rápida!

Um problema simples...

- O Cálculo do número π pode ser feito de várias maneiras. Considere o algoritmo abaixo:
  - Inscreva um círculo em um quadrado (é isso mesmo que você leu ;));
  - crie pontos (x,y) aleatórios dentro da área do quadrado;
  - conte quantos números dentro do quadrado também estão dentro do círculo;
  - seja r o número de pontos no círculo dividido pelo número de pontos no quadrado;
  - $\pi \approx 4 * r$;
  - quanto mais pontos você gerar, melhor a aproximação;
  - não me pergunte porque é verdade, mas é!!!

$\pi - \pi - \pi$-radinha!!!



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

Entendeu o problema? Quem consegue fazer uma versão serial agora?

Agenda

Modelos de Programação Paralela

- Os modelos de programação paralela fornecem uma abstração para o *hardware* e a arquitetura de memória;
- modelo memória compartilhada com *threads*;
- modelo memória distribuída com troca de mensagens;
- Apesar dos nomes iguais, os modelos não estão presos a uma determinada arquitetura de memória;
- é possível, por exemplo, fornecer uma biblioteca de programação com memória compartilhada que execute em um *cluster* de memória distribuída;
- da mesma forma, é possível em uma máquina de memória compartilhada, programar utilizando troca de mensagens entre os *threads* ou processos;
- entretanto, o caso comum é casar o modelo de programação e a arquitetura de memória.

Memória Compartilhada com *Threads*

- Neste modelo, um processo poder ter vários *threads*
- Exemplo:
  - O sistema operacional nativo carrega um processo inicial;
  - esse processo organiza as estruturas de dados e cria um número de *threads* que são escalonadas pelo sistema operacional para executar concorrentemente;
  - cada *thread* tem dados locais, mas também acessa dos dados do processo inicial;
  - a comunicação entre as *threads* requer diretivas de sincronização.
- Implementações:
  - POSIX Threads: uma API padronizada;
  - OpenMP: diretivas de compilação.

Memória Distribuída com Troca de Mensagens

- Um conjunto de *jobs* utilizam sua própria memória local;
- a comunicação é feita através da troca de mensagens;
- em geral, a troca de mensagens existe alguma sincronização. Por exemplo, um processo chama a função de envio enquanto o outro chama a função de recebimento.
- Implementações:
  - Várias implementações proprietárias foram criadas com o decorrer do tempo;
  - hoje o padrão é o MPI, uma biblioteca de rotinas.

Pthreads - Básico

Cabeçalho e formato das funções.

```
include <pthread.h>

pthread[_object_]_<operation>();
```

Por exemplo, **pthread_mutex_init** é uma função que inicia (*init*) um objeto *mutex*.
No caso do objeto ser o próprio *thread*, ele é omitido no nome da função.

Pthreads - Tipos de Dados

Tipo de Dados	**Significado**
pthread_t	ID do *thread*
pthread_mutex_t	Variável *mutex*
pthread_cond_t	Variável de condição
pthread_key_t	Chave de acesso
pthread_attr_t	Objeto de atributos do *thread*
pthread_mutexattr_t	Objeto de atributos do *mutex*
pthread_condattr_t	Objetivo de atributos das condições
pthread_once_t	Controle de contexto de inicialização

Pthreads - Escalonamento e Criação

- Cabe ao programador dividir o problema em *threads*;
- a biblioteca Pthreads, em espaço do usuário, mapeia os *threads* do usuário para *threads* do sistema;
- em geral, o usuário não tem controle em qual núcleo ou processador cada *thread* executará.

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

Pthreads - Olá Mundo

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4

void *PrintHello(void *threadid) {
   long tid;
   tid = (long) threadid;
   printf("Ola Mundo! Sou eu, thread #%ld!\n", tid);
   pthread_exit(NULL);
}
```

Pthreads - Olá Mundo

```
int main(int argc, char *argv[]) {
             pthread_t threads[NUM_THREADS];
int rc;
long t;

for (t = 0; t < NUM_THREADS; t++){
printf("Na main: criando thread %ld\n", t);
rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
if (rc) {
  printf("ERROR; return code from pthread_create() is %d\n", rc);
   exit(-1);
  }
}

pthread_exit(NULL);
}
```

Pthreads - Mais algumas funções...

pthread_t pthread_self()

Retorna o identificar do *thread* que a invoca.

int pthread_equal(pthread_t t1, pthread_t t2)

Retorna 0 se t1 e t2 não forem o mesmo *thread*.

void pthread_exit(void *valuep)

A *thread* pode terminar chamando **return** ou invocando pthread_exit. O argumento valuep será retornado para quem chamar pthread_join nesse *thread* (ver abaixo).

int pthread_join(pthread_t threadID, void **valuep)

O *thread* que a invoca aguarda pelo termino do *thread* threadID. O argumento valuep indica o endereço de memória no qual o valor de retorno deve ser armazenado.

Pthreads - Variáveis Mutex

- pthread_mutex_t;
- útil para controlar o acesso a estruturas de dados compartilhadas;
- possui dois estados: bloqueado e desbloqueado;
  - Antes de acessar dados compartilhados, o *thread* tenta bloquear a *mutex*;
  - após acessar dados compartilhados, o *thread* desbloqueia a *mutex*;
  - quando um *thread* A tenta bloquear uma variável *mutex* que já pertence a outro *thread* B, *thread* A é bloqueado até que *thread* B desbloqueie a variável *mutex*.

Pthreads - Variáveis Mutex

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)

Inicializa um *mutex*.

int pthread_mutex_destroy(pthread_mutex_t *mutex)

Destroi um *mutex*.

int pthread_mutex_lock (pthread_mutex_t *mutex)

Bloqueia um *mutex*.

int pthread_mutex_unlock (pthread_mutex_t *mutex)

Desbloqueia um *mutex*.

pthread_mutex_trylock(pthread_mutex_t *mutex)

Tenta bloquear um *mutex*.

Pthreads - Variáveis de Condição

- Servem para evitar que um *thread* verifique repetidamente se determinada condição foi atingida (espera ocupada);
- quando um *thread* espera em uma condição, ele só é desbloqueado quando outro *thread* sinaliza essa condição;
- um *mutex* é usado em conjunto para controlar o acesso à variável de condição;
- pense no problema dos produtores e consumidores.

```
pthread_mutex_lock(&mutex);
while (!condition())
    pthread_cond_wait(&cond, &mutex);
faca_alguma_coisa();
pthread_mutex_unlock(&mutex);
```

Pthreads - Variáveis de Condição

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)

Inicializa uma variável de condição.

int pthread_cond_destroy(pthread_cond_t *cond)

Destrói um *mutex*.

int pthread_cond_wait(pthread_cond_t *cond, pthreat_mutex_t *mutex)

Bloqueia enquanto uma variável de condição não é sinalizada.

int pthread_cond_signal (pthread_cond_t *cond)

Sinaliza uma variável de condição. Apenas um *thread* é "acordado".

pthread_cond_broadcast(pthread_cond_t *cond)

Acorda todos os *threads* em espera da condição.

MPI - Message Passing Interface

- Assim a Pthreads, MPI é um padrão. Porém, existe várias implementações;
- MPICH, OpenMPI, LAM/MPI, etc;
- em geral, são compatíveis;
- diferente da Pthreads, não usamos o gcc diretamente, mas sim um *script* que configura o ambiente e depois chama o gcc. Também chamamos um *script* para execução.

```
#include "mpi.h"
```

```
jmhal@earth:~/parallel/mpi$ mpicc HelloWorld.c -o HelloWorld
jmhal@earth:~/parallel/mpi$ mpirun -np 4 HelloWorld
```

MPI - Olá Mundo

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

int main (argc, 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 from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

MPI - O papel do Buffer

- A comunicação pela rede pode ser feita de duas maneiras:
  - Sem *buffer*: a comunicação é feita diretamente entre os processos MPI;
  - Com *buffer*: a biblioteca MPI mantém um *buffer* para cada máquina. Quando um processo envia uma mensagem, primeiro ela é copiada para o *buffer* remetente, depois para o *buffer* da máquina destino, e só depois para o processo receptor.
- esse é um dos pontos onde as implementações podem variar.

MPI - Tipos de Comunicação

- Bloqueio:
  - Operação bloqueante: o processo que invoca a operação fica bloqueado até todos os recursos, *buffer* inclusive, puderem ser utilizados em outra operação ;
  - Operação não bloqueante: o processo não bloqueia, retornando o controle assim que a operação for inicializada. Entretanto, como o *buffer* ainda pode estar sendo usado, não é seguro fazer outra operação logo em seguida.
- Sincronização:
  - Comunicação Síncrona: A comunicação entre os processos só ocorre quando ambos iniciarem suas operações de comunicação;
  - Comunicação Assíncrona: o processo emissor pode executar sua operação sem que o receptor tenha iniciado a sua.
- importante é notar que se não houver *buffers*, a operação bloqueante é síncrona.

MPI - Comunicação Ponto-a-ponto

As operações básicas de comunicação no MPI são assíncronas bloqueantes. O conceito de comunicador é usado para agrupar processos.

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

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

MPI - Tipos de Dados

Tipo de Dados	**Significado**
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	um *byte*

MPI - Comunicação Ponto-a-ponto

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

int main (int argc, char *argv[]) {
   int rank, size;
   int value = 0;
   MPI_Status status;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   if (rank == 0) {
       value++;
       MPI_Send(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
       printf("Processo %d: %d.\n", rank, value);
   } else if (rank == 1) {
       MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
       value++;
       printf("Processo %d: %d.\n", rank, value);
   }
         MPI_Finalize();
   return 0;
}
```

MPI - Deadlocks

```c
/* sempre causa deadlock */
if (my_rank == 0) {
    MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status);
    MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
} else if (my_rank == 1){
    MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status);
    MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);
}
/* pode causar, depende se usa buffers */
if (my_rank == 0) {
    MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
    MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status);
} else if (my_rank == 1){
    MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);
    MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status);
}
/* livre */
if (my_rank == 0) {
    MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
    MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status);
} else if (my_rank == 1){
    MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status);
    MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);
}
```

MPI - Enviar e Receber

Caso você tenha que enviar e receber ao mesmo tempo, existe uma chamada para isso.

```
int MPI_Sendrecv( void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  int dest,
                  int sendtag,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  int source,
                  int recvtag,
                  MPI_Comm comm,
                  MPI_Status *status)
```

MPI - Chamadas não-bloqueantes

É preciso verificar se a comunicação de fato ocorreu.

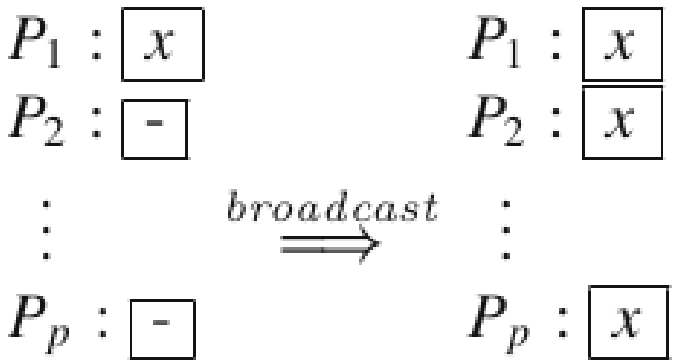
```
int MPI_Isend(void *smessage,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request )

int MPI_Recv(void *rmessage,
              int count,
              MPI_Datatype datatype,
              int source,
              int tag,
              MPI_Comm comm,
              MPI_Request *request )

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

MPI - Chamadas não-bloqueantes

```
if (rank == 0) {
    value++;
    MPI_Isend(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

    int buffer[100];
    for (i = 0; i < 100; i++) buffer[i] = i*i;

    MPI_Wait(&request, &status);
    printf("Processo %d: %d.\n", rank, value);
} else if (rank == 1) {
    MPI_Irecv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);

    int buffer[100];
    for (i = 0; i < 100; i++) buffer[i] = i*i;

    MPI_Wait(&request, &status);
    value++;
    printf("Processo %d: %d.\n", rank, value);
}
```

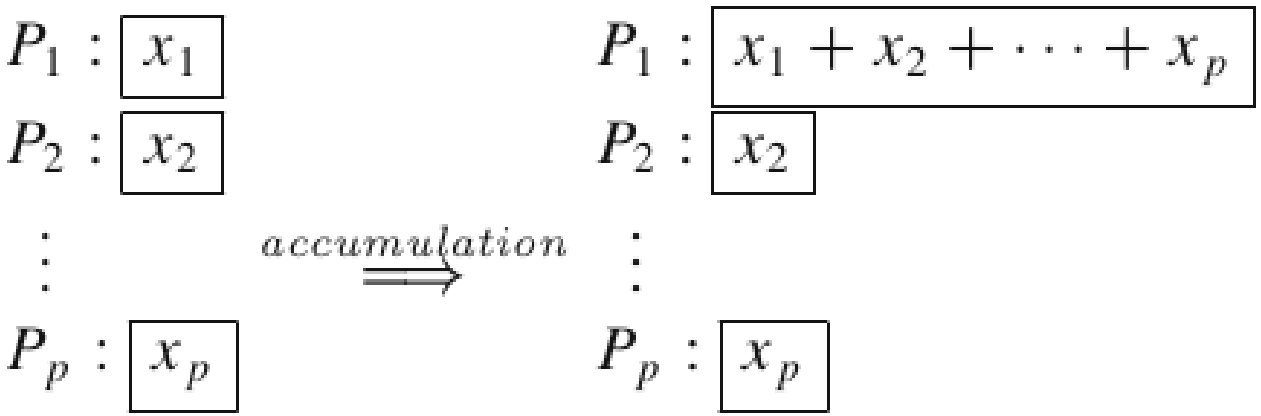
MPI - Comunicação Coletiva - Broadcast

```
int MPI_Bcast(void *message,
 int count,
 MPI_Datatype type,
 int root,
 MPIC_Comm comm)
```



MPI - Comunicação Coletiva - Reduction

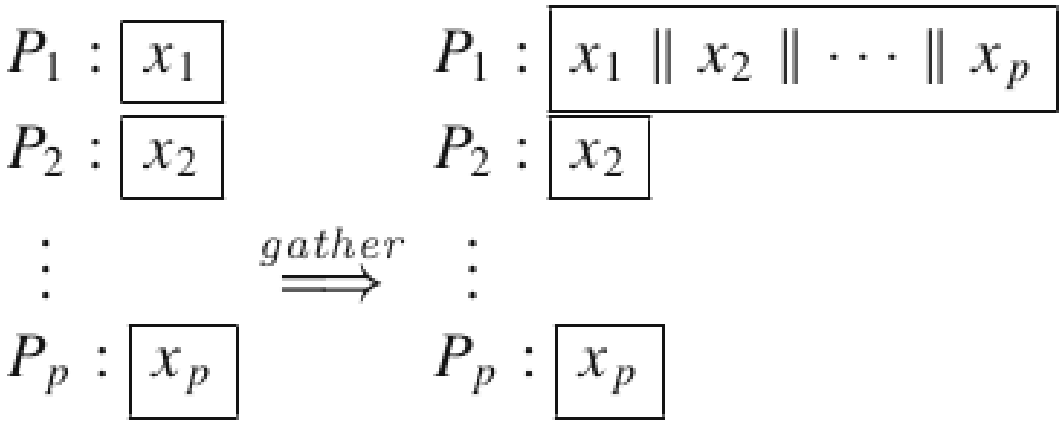
```
int MPI_Reduce(void *sendbuf,
void *recvbuf,
int count,
MPI_Datatype type,
MPI_Op op,
int root,
MPI_Comm comm)
```



Operações: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, etc

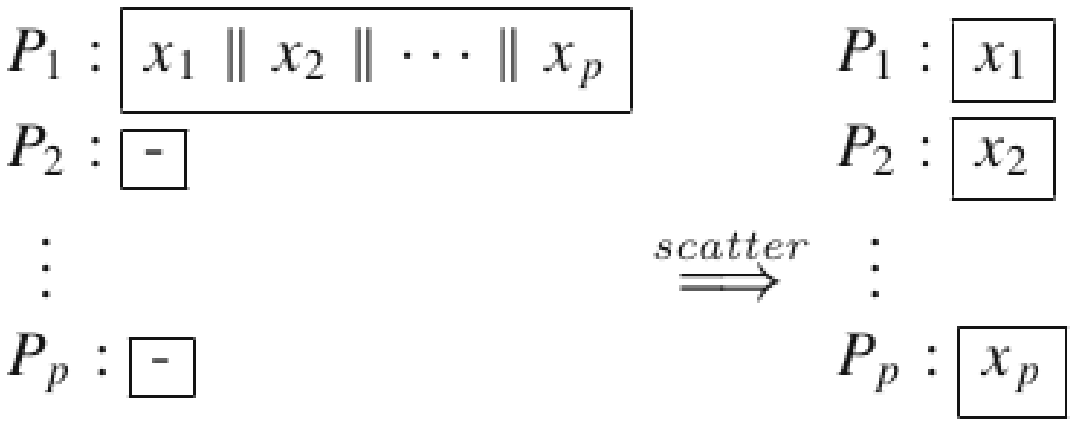
MPI - Comunicação Coletiva - Gather

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



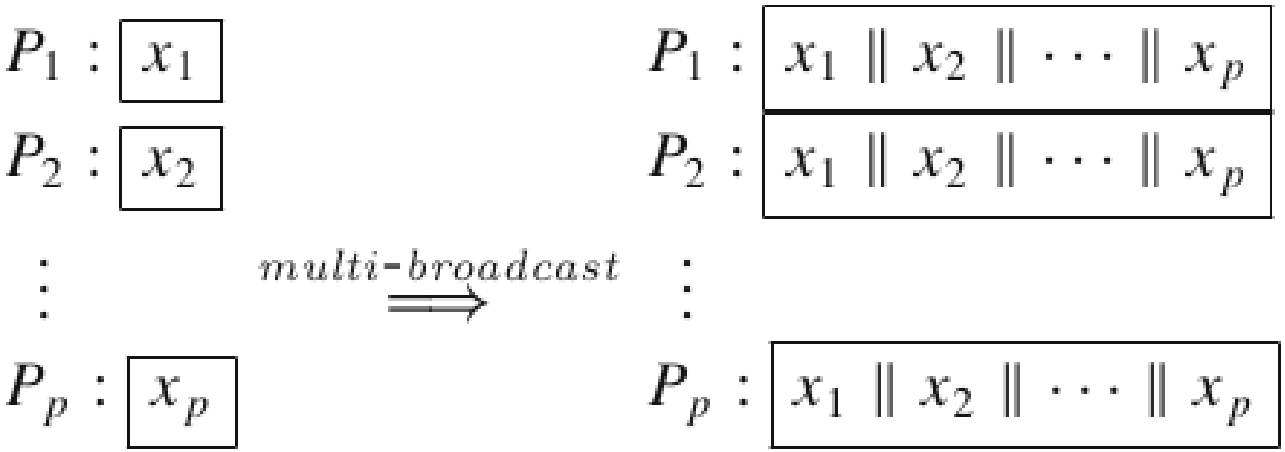
MPI - Comunicação Coletiva - Scatter

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



MPI - Comunicação Coletiva - AllGather

```
int MPI_Allgather(void *sendbuf, int sendcount,
MPI_Datatype sendtype,
void *recvbuf, int recvcount,
MPI_Datatype recvtype,
MPI_Comm comm)
```



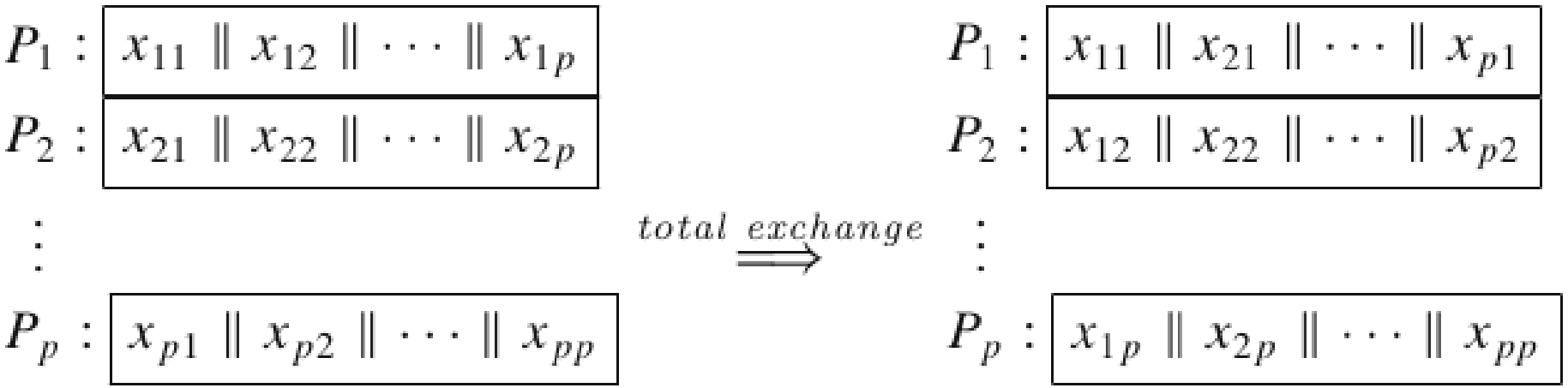
MPI - Comunicação Coletiva - AllReduce

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
int count,
MPI_Datatype type,
MPI_Op op,
MPI_Comm comm)
```

$$
\begin{array}{ll}
P_0 : x_0 & P_0 : x_0 + x_1 + \cdots + x_{p-1} \\
P_1 : x_1 & P_1 : x_0 + x_1 + \cdots + x_{p-1} \\
\vdots \quad \overset{MPI-accumulation(+)}{\Longrightarrow} & \vdots \\
P_{p-1} : x_n & P_{p-1} : x_0 + x_1 + \cdots + x_{p-1}
\end{array}
$$

MPI - Comunicação Coletiva - Total Exchange

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



Agenda

Referências

- https://computing.llnl.gov/tutorials/parallel_comp
- http://www.ks.uiuc.edu/Research/namd/
- http://en.wikipedia.org/wiki/Symmetric_multiprocessing
- http://en.wikipedia.org/wiki/Hyper-threading
- Parallel Programming for Multicore and Cluster Systems http://www.amazon.com/Parallel-Programming-Multicore-Cluster-Systems/dp/364204817X