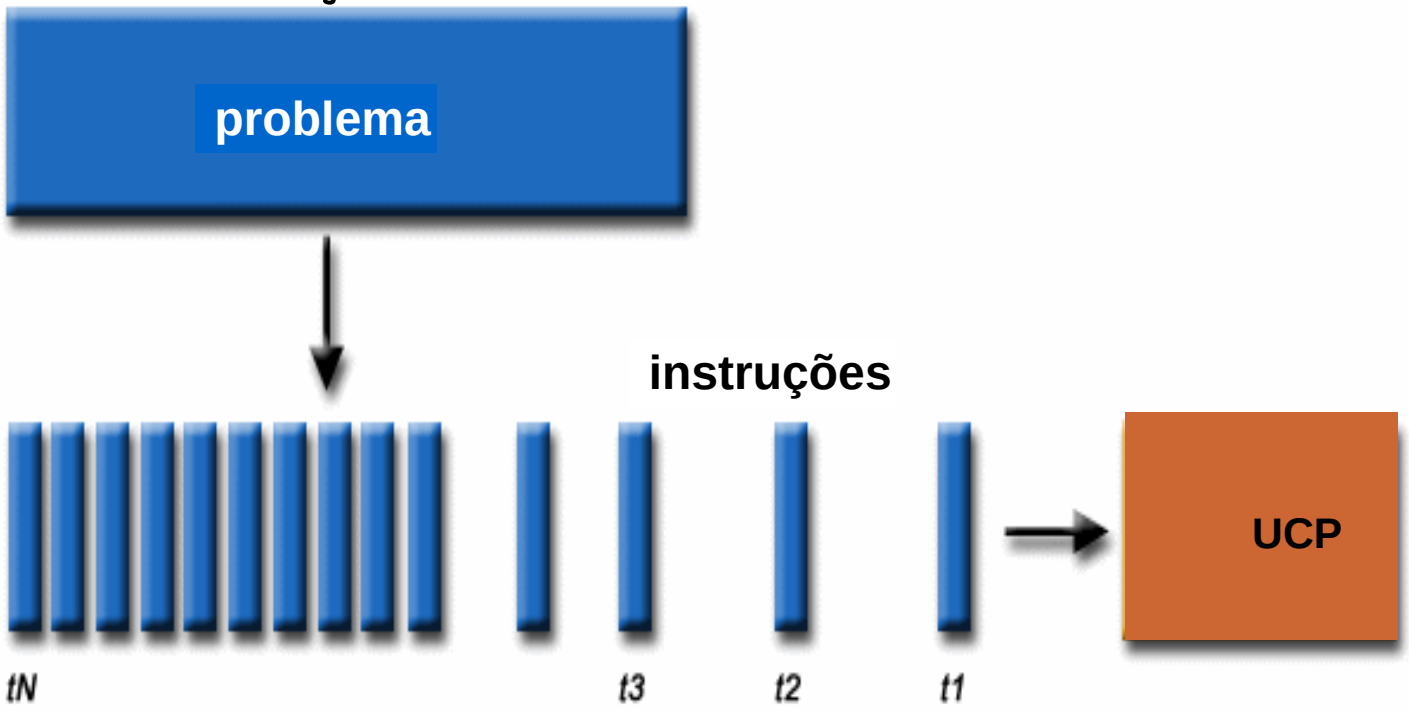
Programação Paralela

Simone de Lima Martins

Slides baseados no Tutorial Introduction to Parallel Computing
(https://computing.llnl.gov/tutorials/parallel_comp/)

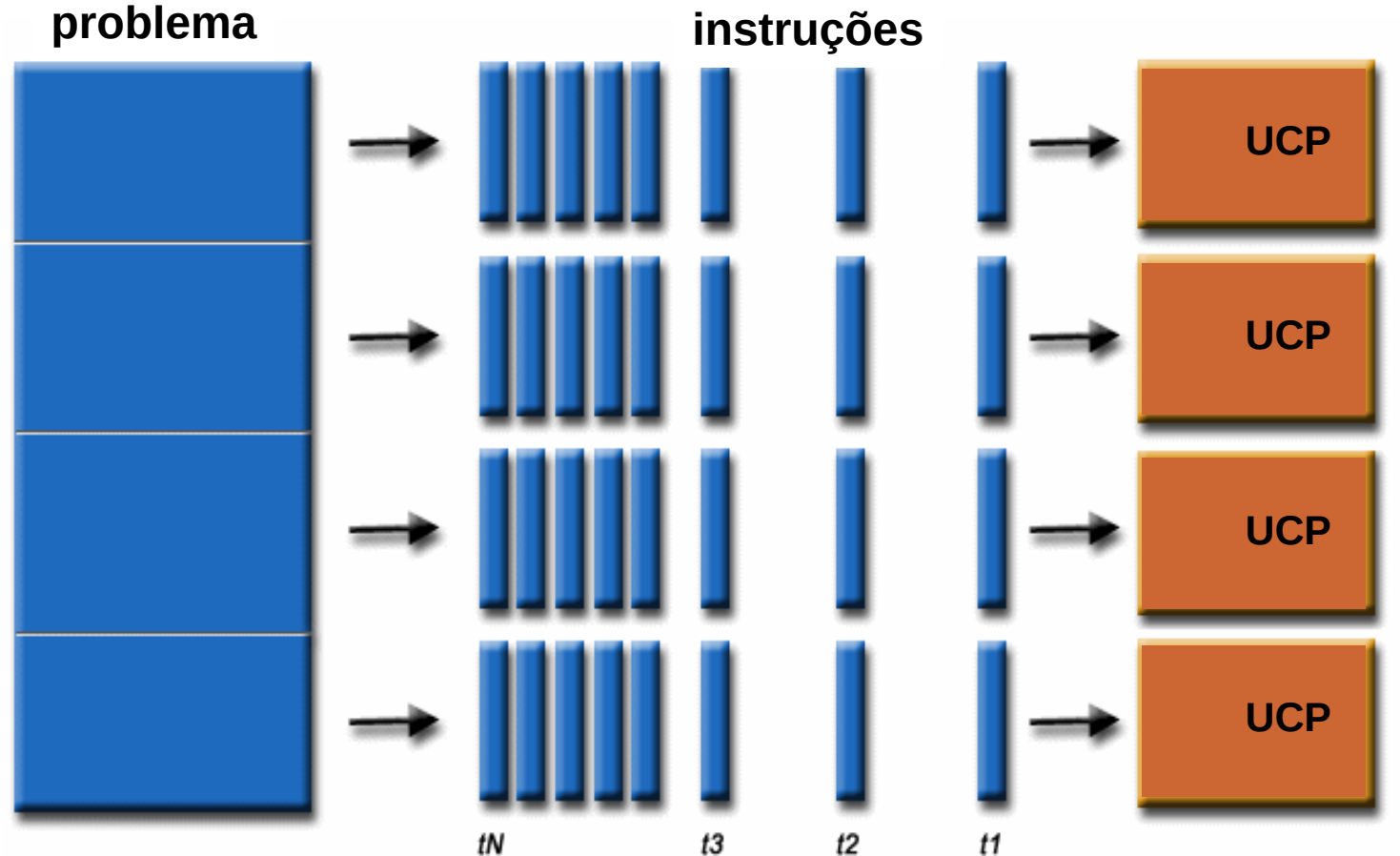
Programação sequencial

- O programa deve ser executado em um único computador com uma única Unidade Central de Processamento (UCP)
- Um programa é constituído de uma série de instruções que são executadas em sequência
- Somente uma instrução é executada de cada vez



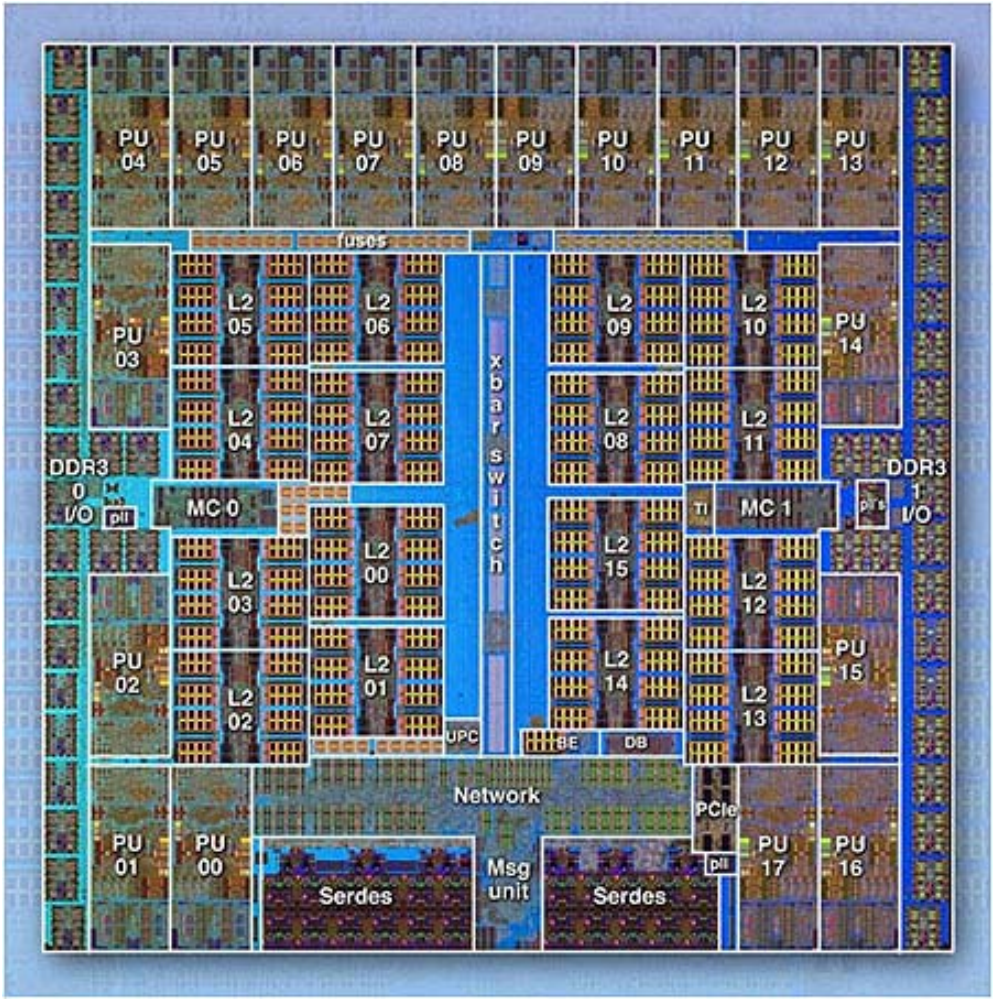
Programação paralela

- O problema é decomposto em partes que podem ser executadas concorrentemente
- Cada parte é constituída de uma sequência de instruções que são executadas por UCPs diferentes simultaneamente



Computadores paralelos

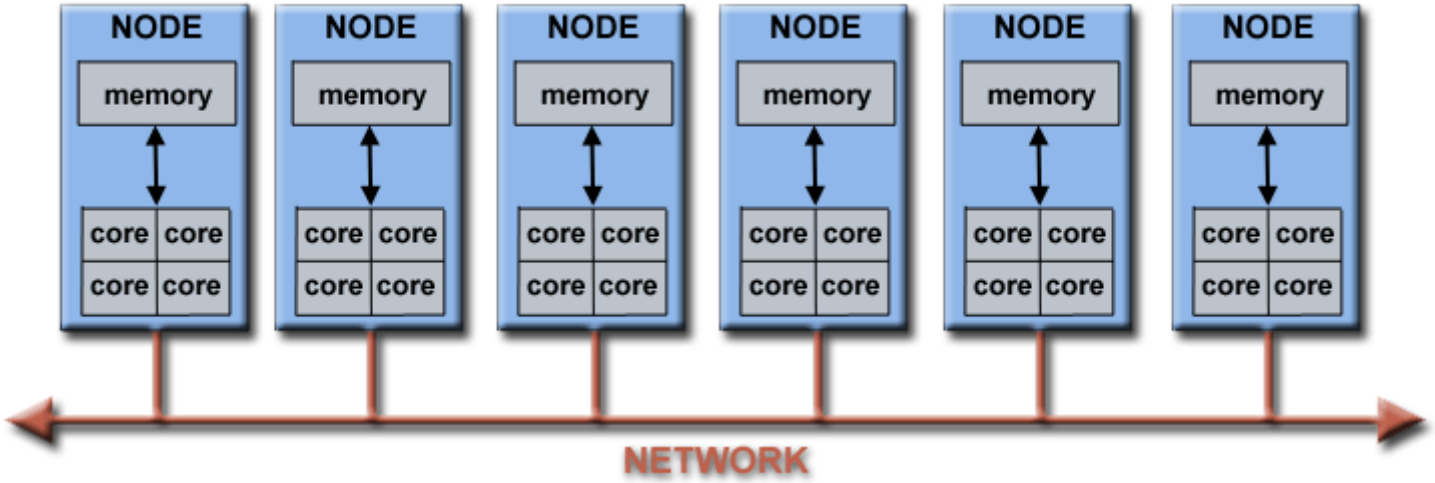
- Os computadores hoje em dia são inerentemente paralelos:
  - Múltiplas unidades funcionais (cache L1, cache L2, branch, prefetch, decode, floating-point, graphics processing (GPU), integer, etc.)
  - Múltiplas unidades de execução/cores
  - Múltiplas threads de hardware



CPU com 18 cores e 16 unidades cache L2

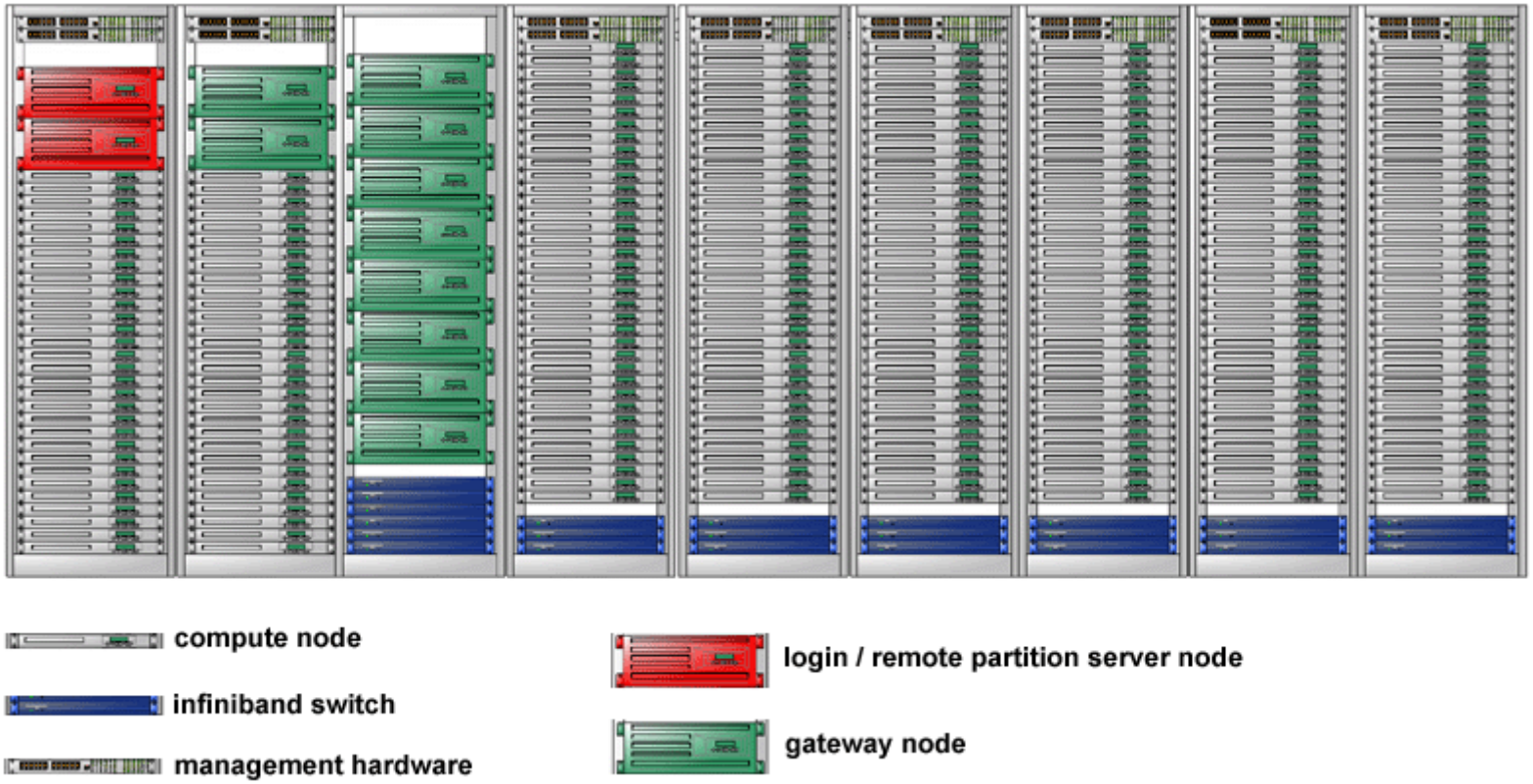
Computadores paralelos

- Redes conectam computadores formando clusters



Computadores paralelos

- Computadores multiprocessados conectados por rede de comunicação

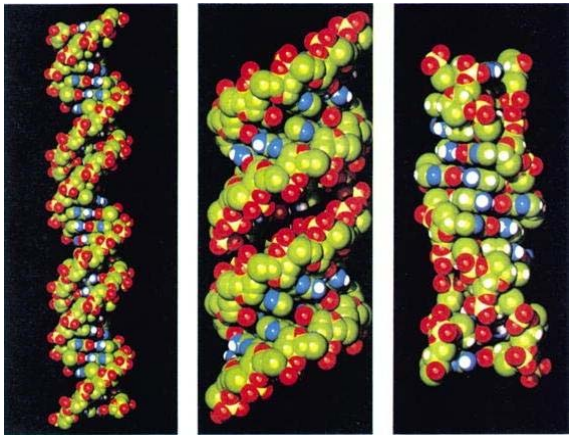


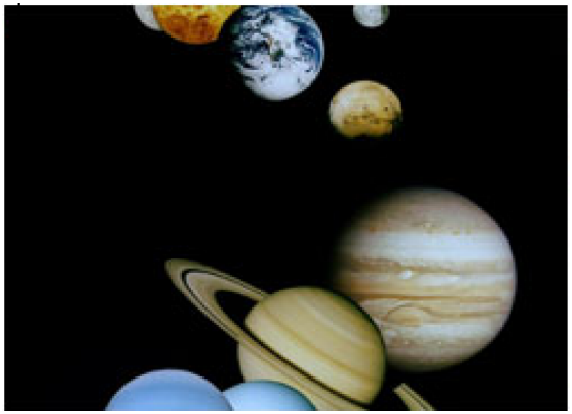
Programação paralela

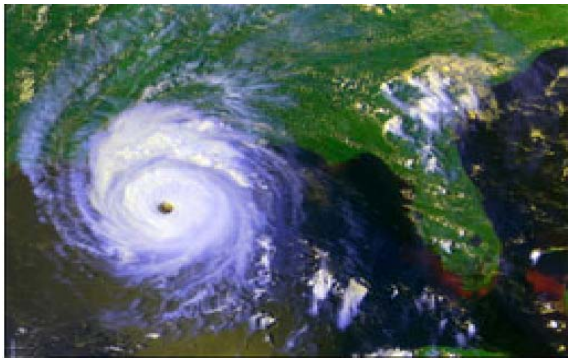
- Características comuns dos problemas computacionais que podem ser resolvidos utilizando programação paralela:
  - Podem ser decompostos em partes que podem ser resolvidas de forma simultânea
  - Necessitam executar muitas instruções ao mesmo tempo
  - São resolvidos em menor tempo utilizando múltiplas UCPs do que somente uma

Programação paralela

- Muitos problemas reais tem natureza intrínsicamente paralela
  - Movimento de corpos celestes
  - Linha de montagem de automóveis
  - Caracterização de genomas
  - Fenômenos meteorológicos







Utilização de programação paralela

- Utilizada inicialmente para resolver problemas difíceis das área de ciência e engenharia:
  - Meteorologia
  - Física
  - Biociências
  - Geologia
  - Microeletrônica
  - Ciência da Computação

Utilização de programação paralela

- Atualmente utilizada em aplicações comerciais para manipular grande quantidade de dados:
  - Banco de dados, Mineração de dados
  - Exploração de petróleo
  - Busca na Web
  - Computação gráfica e Realidade virtual
  - Ambientes de trabalho cooperativos
  - Tecnologia multimídia

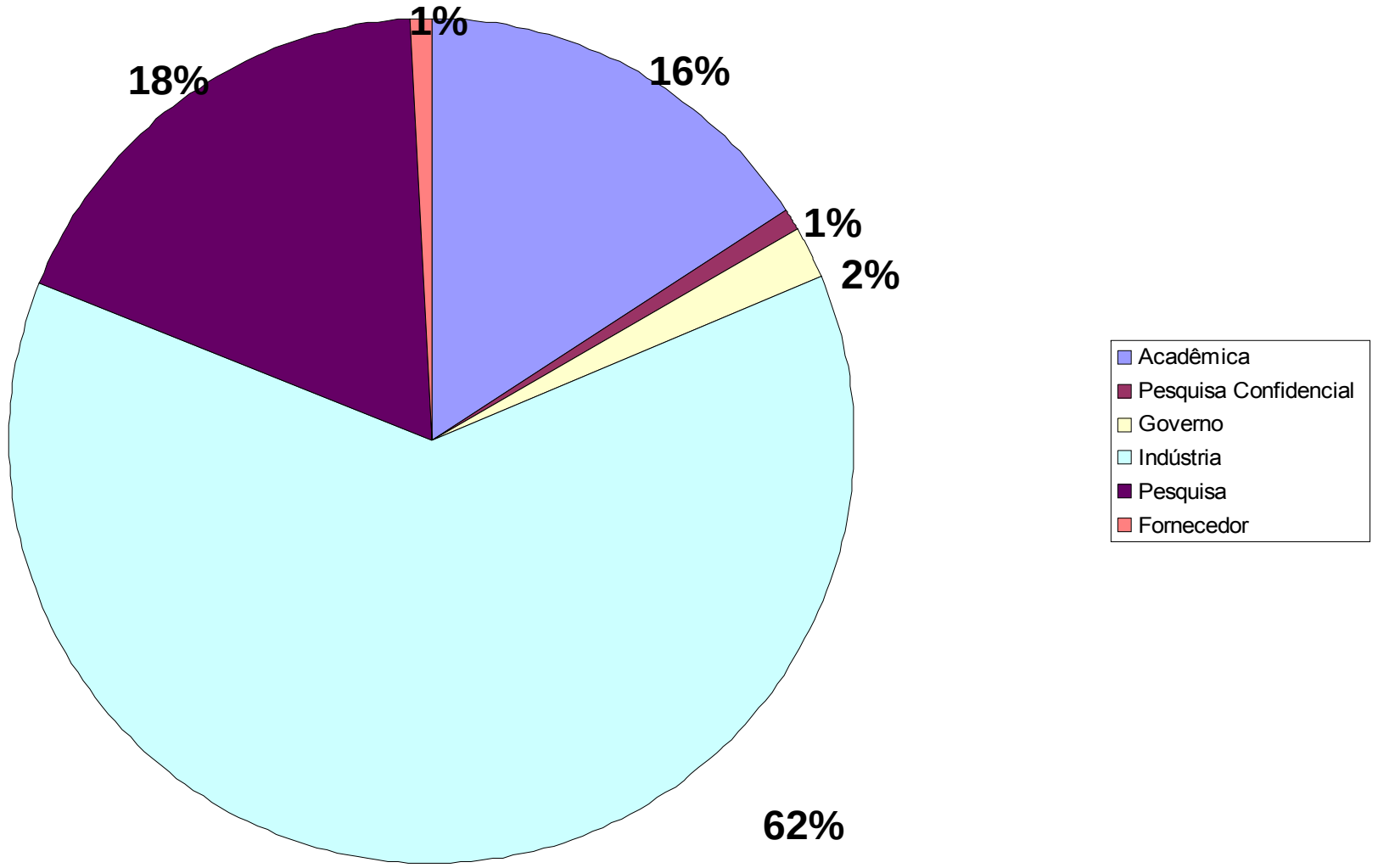
Porque utilizar programação paralela?

- Economizar tempo e/ou dinheiro
- Resolver problemas maiores
  - Grandes desafios da computação (DARPA 1980)
    - Dinâmica de fluidos
    - Projeto de novos materiais
    - Tecnologia para energia nuclear
    - Computação simbólica
  - Busca na Web e em bancos de dados com milhões de transações por segundo

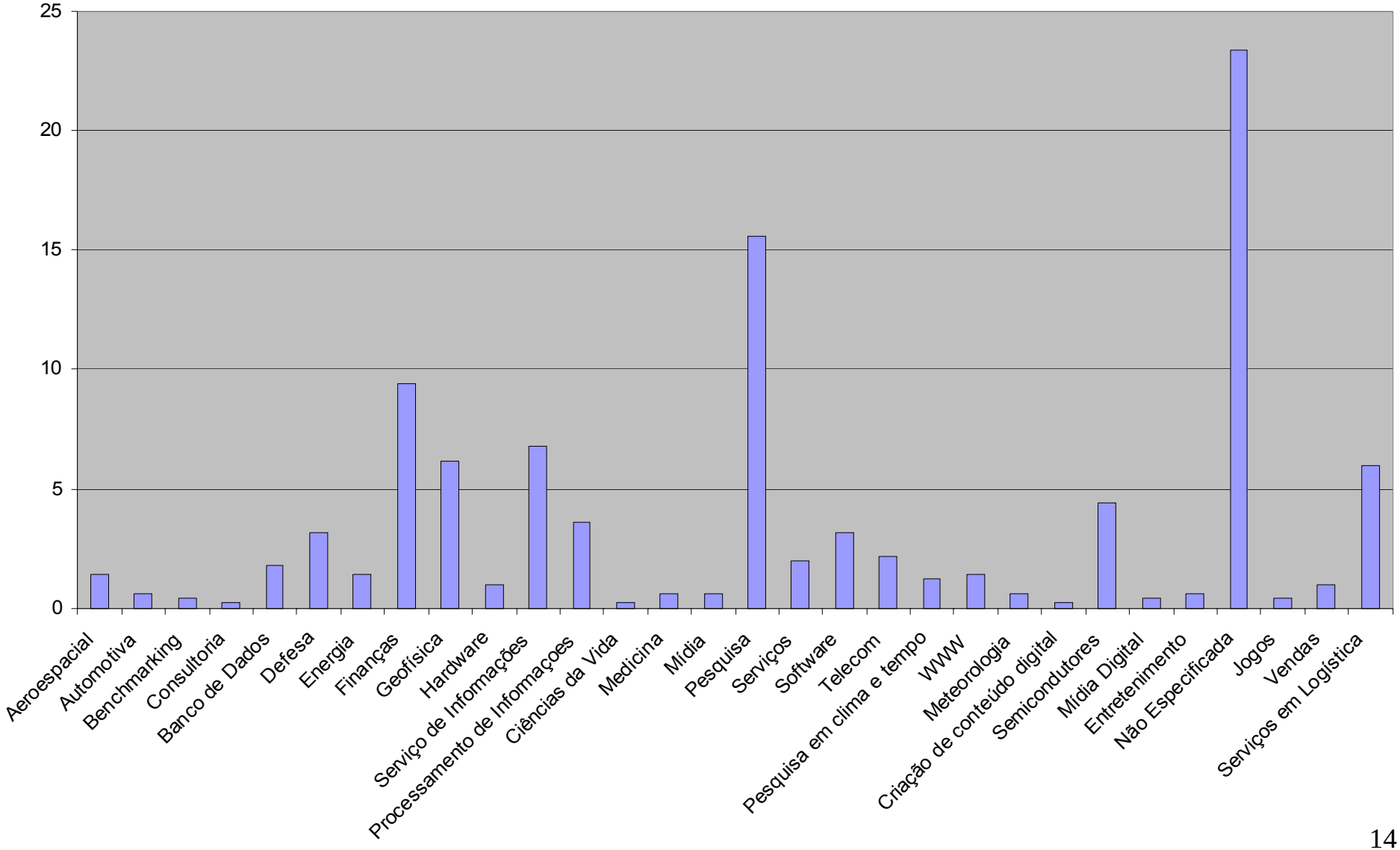
Porque utilizar programação paralela?

- Prover concorrência real
  - Uso de recursos globais (SETI@home)
- Limites da programação sequencial
  - Limite de transmissão dos sinais elétricos
    - Velocidade da luz (30 cm/ns). Necessita-se aproximar os componentes
    - Limites de miniaturização
    - Limitações de preço: mais econômico utilizar vários processadores mais baratos

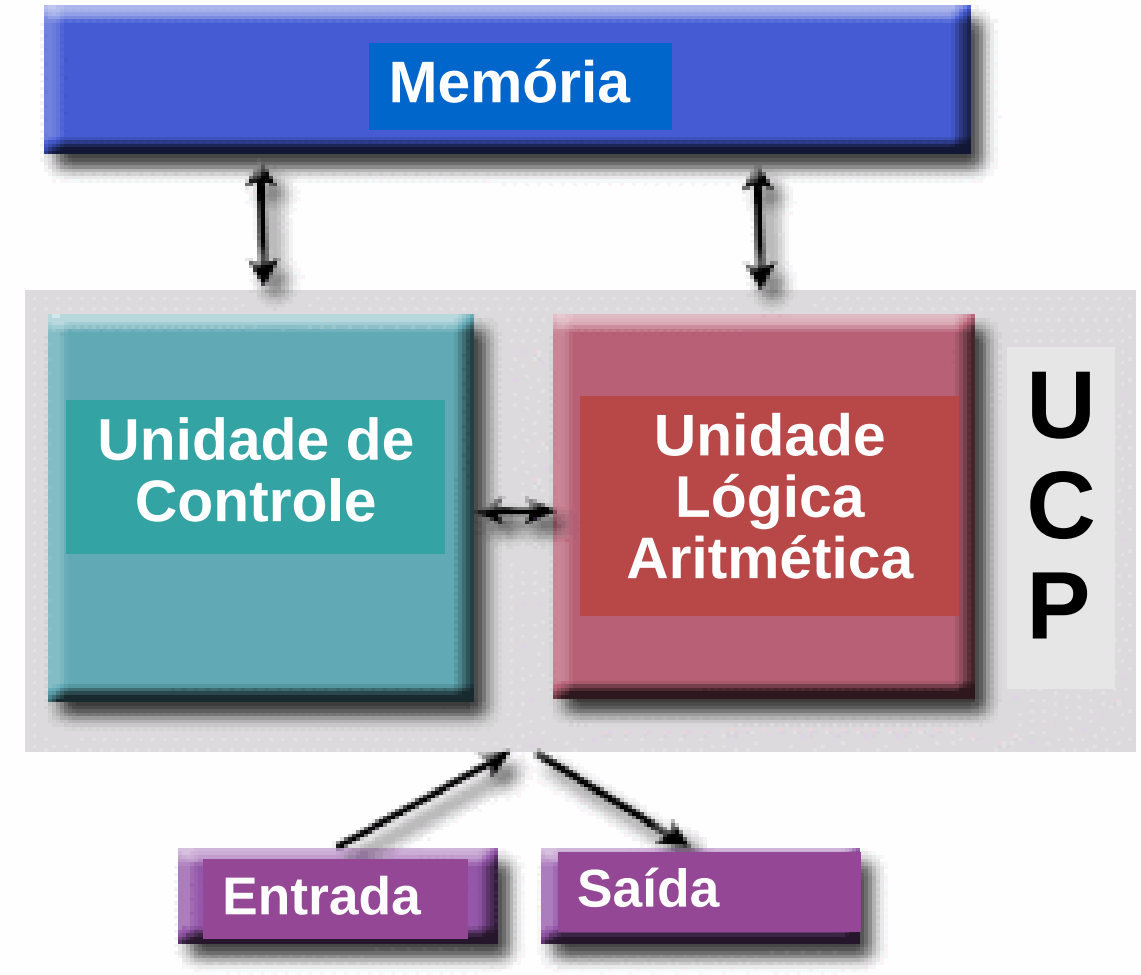
Setores que utilizam máquinas paralelas (Top500.org)



Atividades


25
20
15
10
5
0
Aeroespacial
Automotiva
Benchmarking
Consultoria
Banco de Dados
Defesa
Energia
Finanças
Geofísica
Hardware
Serviço de Informações
Processamento de Informaçoes
Ciências da Vida
Medicina
Mídia
Pesquisa
Serviços
Software
Telecom
Pesquisa em clima e tempo
WWW
Meteorologia
Criação de conteúdo digital
Semicondutores
Mídia Digital
Entretenimento
Não Especificada
Jogos
Vendas
Serviços em Logística

Arquitetura de von Neumann

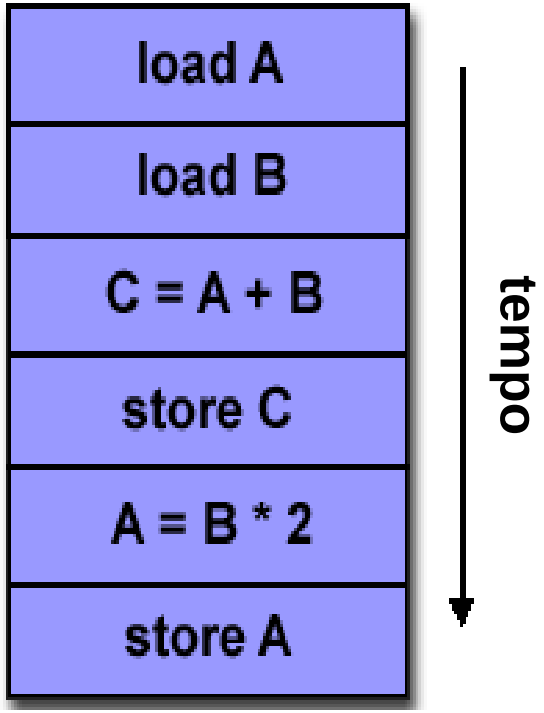


Classificação de Flynn para computadores paralelos

SISD **Single Instruction, Single Data**	**SIMD** **Single Instruction, Multiple Data**
MISD **Multiple Instruction, Single Data**	**MIMD** **Multiple Instruction, Multiple Data**

Single Instruction, Single Data (SISD)

- Computador serial
- Tipo mais comum



Exemplos de computadores SISD

Univac 1

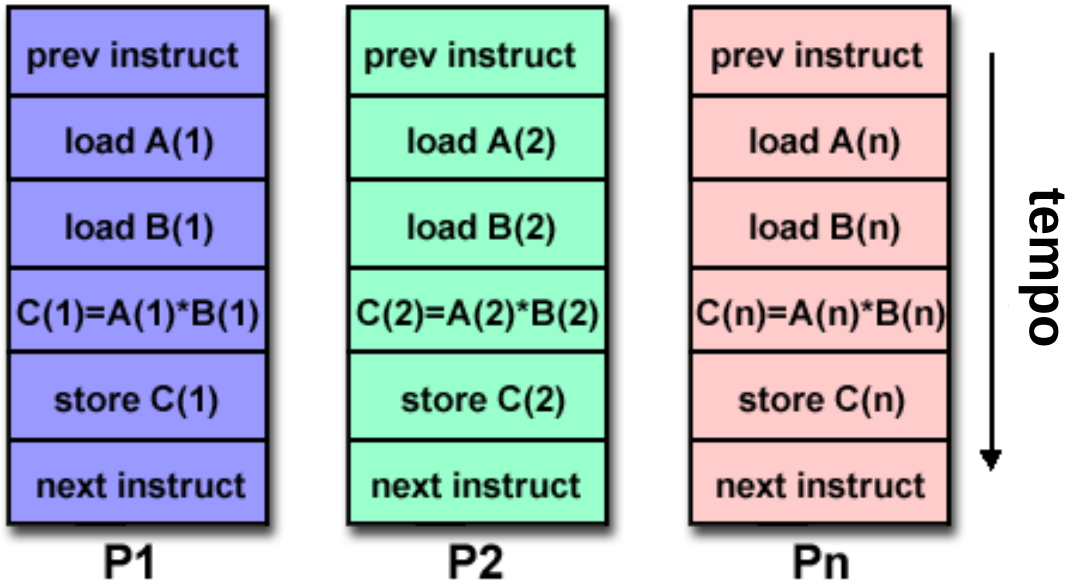
IBM 360

CDC 7600

Laptop Dell

Single Instruction, Multiple Data (SIMD)

- Todas as UCPs executam as mesmas instruções sincronamente
- Cada UCP pode operar com diferentes dados
- Arrays de processadores e Vetores em pipeline
- GPUs (Graphical Processing Unit) utilizam processadores com esta arquitetura



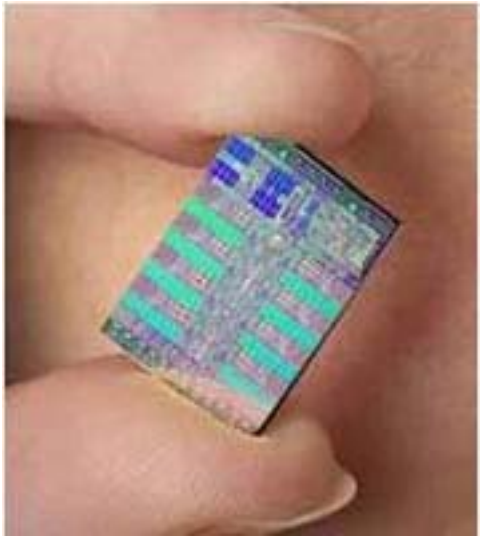
Exemplos de computadores SIMD

ILLIAC IV

MASPAR

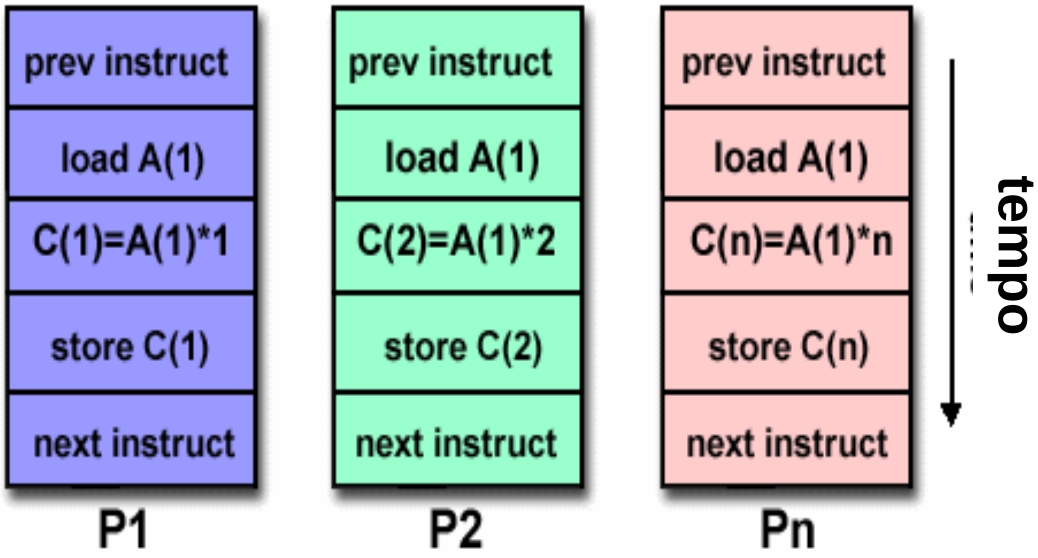
CRAY X-MP

Processador da GPU



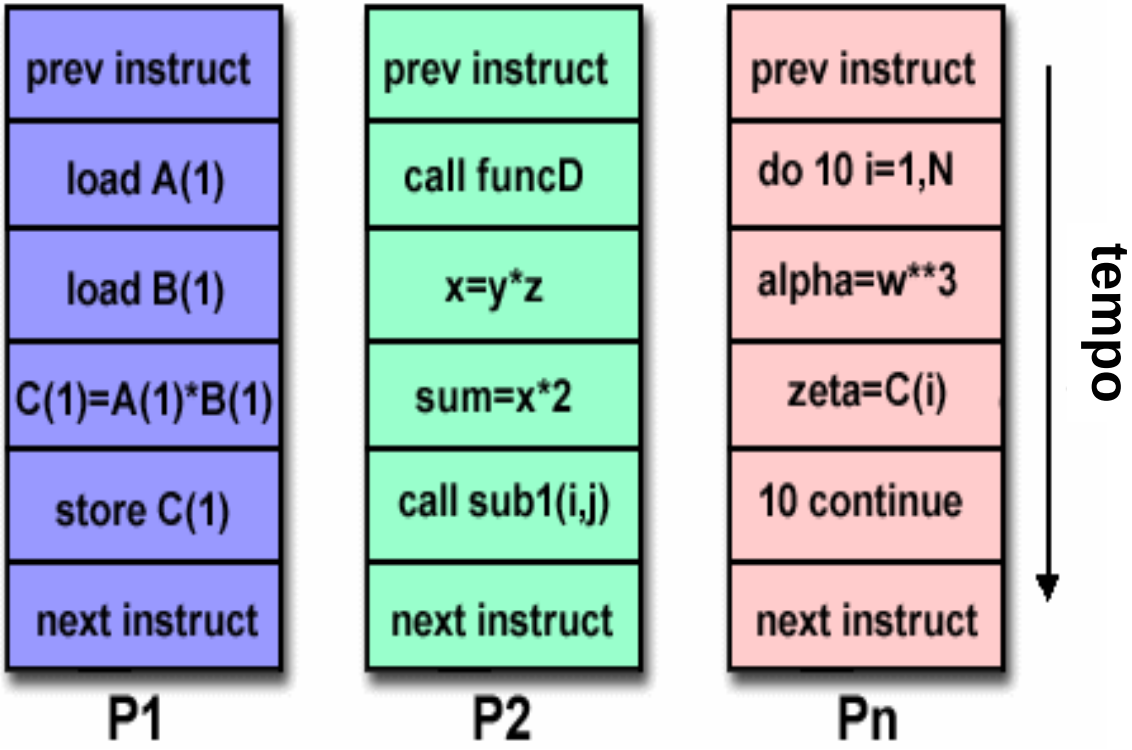
Multiple Instruction, Single Data (MISD)

- Um único fluxo de dados é executado por várias UCPs com instruções diferentes
- Poucos exemplos reais de arquitetura deste tipo
- Completar a classificação
- Exemplos:
  - Tentar quebrar criptografia
  - Aplicação de filtros em um sinal



Multiple Instruction, Multiple Data (MIMD)

- Tipo mais comum de computador paralelo
- Cada UCP pode executar instruções diferentes
- Cada UCP pode utilizar dados diferentes
- Super-computadores, clusters, grids, multi-core



Exemplos de computadores MIMD

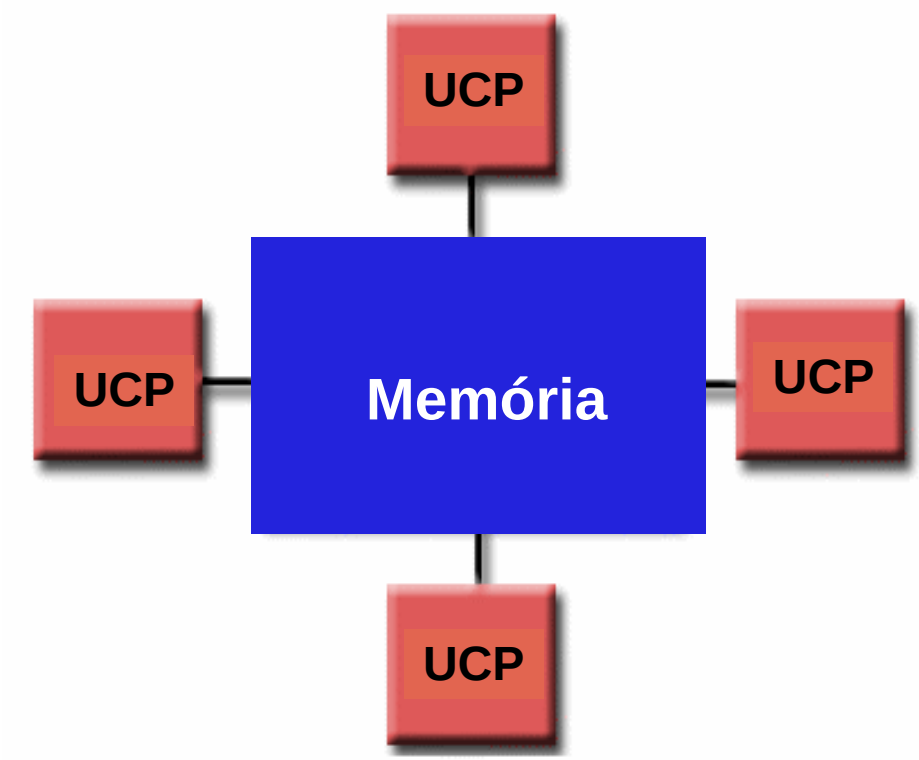
IBM POWER 5

IBM BG/L

AMD Opteron

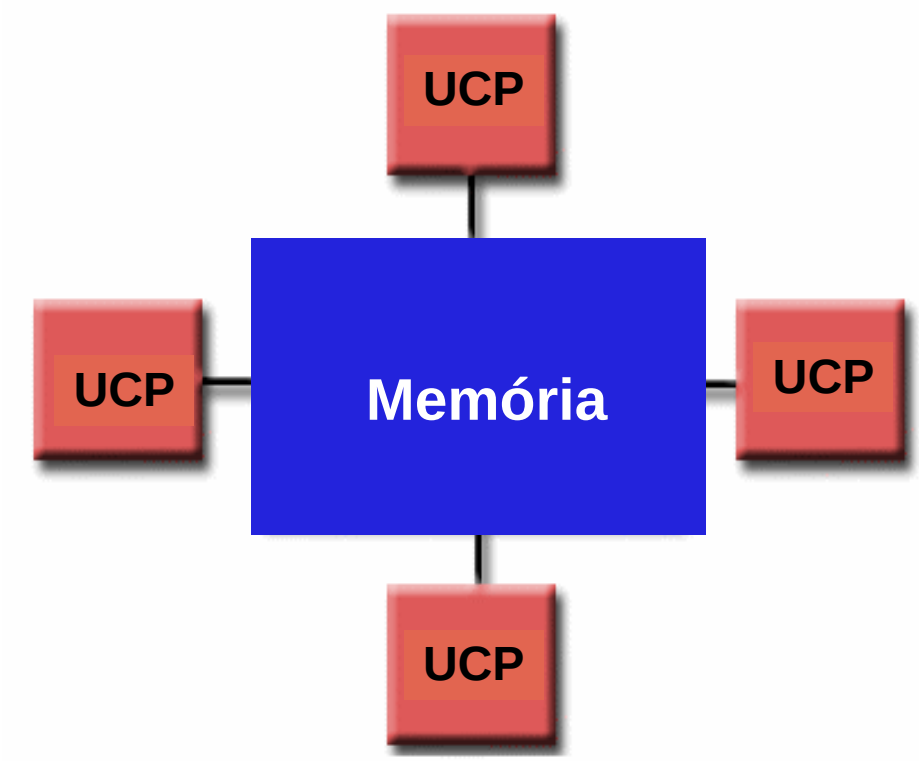
Arquitetura de memória: memória compartihada

- Todos os processadores acessam um espaço de endereçamento global
- Processadores podem executar de forma independente mas compartilham a memória
- Mudanças realizadas na memória por um processador são visíveis a todos processadores
- 2 classes: UMA e NUMA



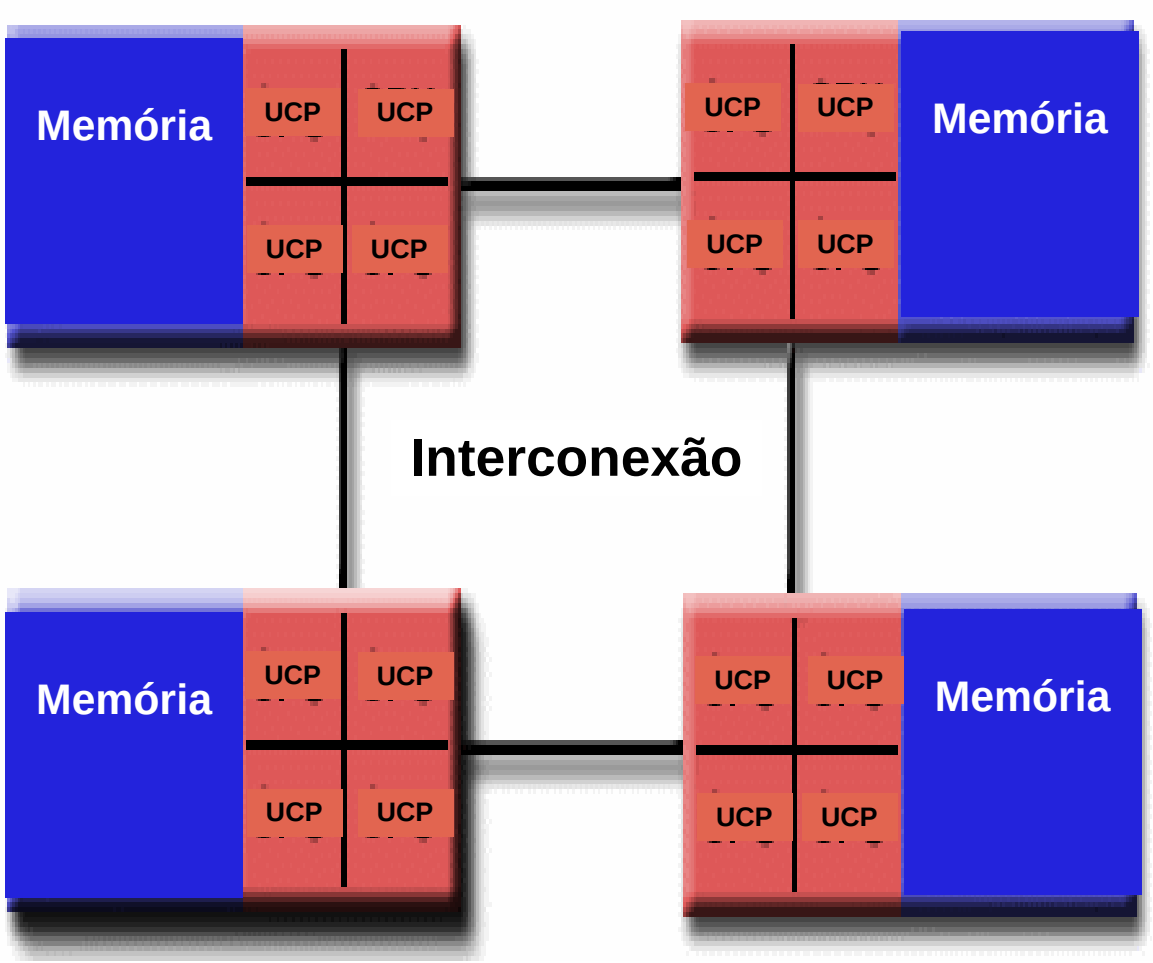
Arquitetura UMA (Uniform Memory Architecture)

- Processadores idênticos
- Todos os processadores acessam endereços de memória em tempos iguais
- Exemplo: SMP (Symmetric Multiprocessor)



Arquitetura NUMA (Non-Uniform Memory Architecture)

- Os processadores acessam endereços de memória em tempos diferentes
- Geralmente implementada interconcetando-se SMPs
- Um SMP pode acessar diretamente a memória de outro SMP
- Acesso à memória através da interconexão é mais lenta

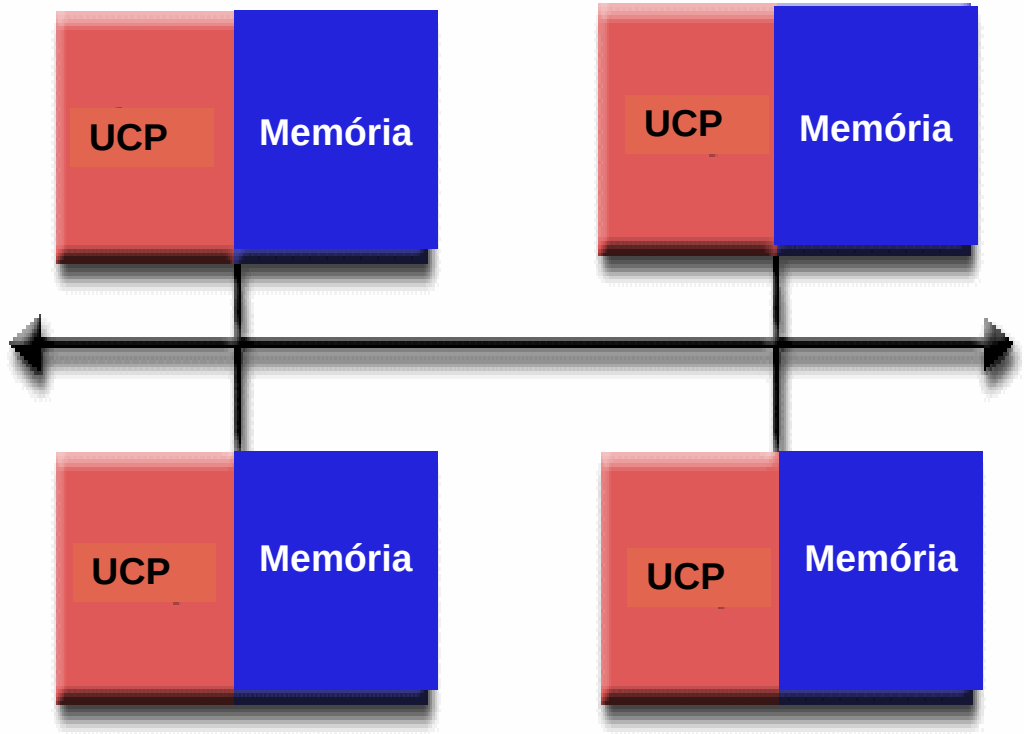


Arquitetura com memória compartihada

- Vantagens:
  - Facilidade de utilização de variáveis compartilhadas
  - Tempo de acesso às variáveis compartilhadas é rápido e uniforme
- Desvantagens:
  - Aumento no número dos processadores leva a um aumento do tráfego na interconexão entre processadores e memória causando instabilidade
  - O programador é responsável pela sincronização do acesso à memória compartilhada para evitar erros no seu acesso

Arquitetura de memória: memória distribuída

- Cada processador acessa somente sua memória local
- Não existe o conceito de endereçamento global
- Mudanças realizadas na memória local não afetam a memória dos outros processadores
- Processadores se comunicam através de uma rede de comunicação
- Acesso à memória não local realizado através de programação explícita

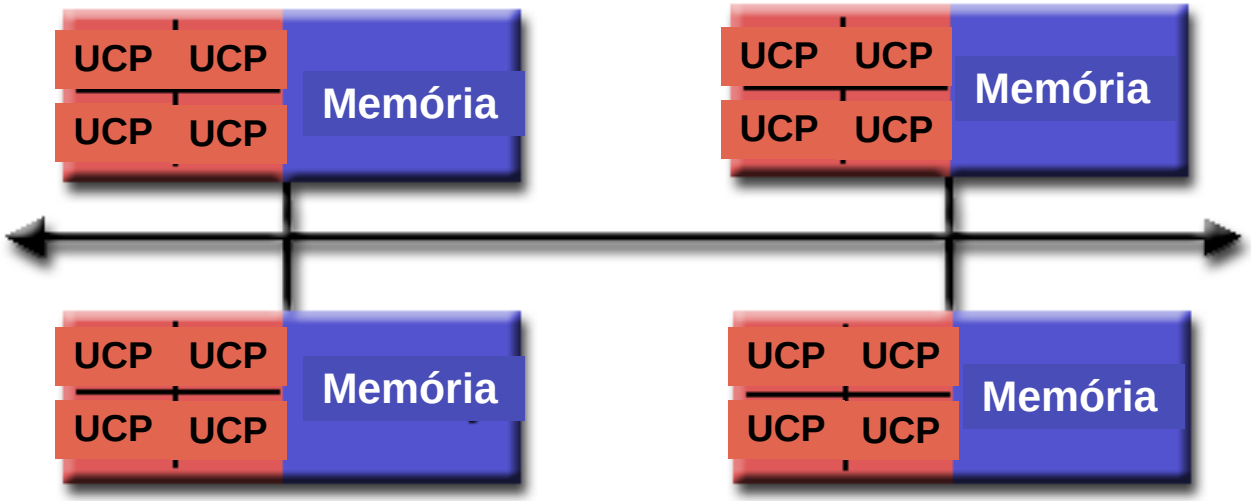


Arquitetura com memória distribuída

- Vantagens:
  - A quantidade de memória aumenta proporcionalmente à quantidade de processadores
  - Cada processador acessa a sua memória de forma independente dos outros processadores, não necessitando de manter a coerência entre processadores
  - Pode utilizar processadores e redes já existentes
- Desvantagens:
  - Programador responsável pelo compartilhamento de dados entre processadores
  - Difícil mapeamento de estruturas de dados baseadas em endereçamento global
  - Tempos não uniformes de acesso à memória

Arquitetura de memória: memória híbrida distribuída-compartilhada

- Componentes SMPs que compartilham endereço global
- Componentes SMPs são conectados através de um canal de comunicação para possibilitar o acesso à memória de outros SMPs
- Vantagens e desvantagens de cada arquitetura



Modelos de programação paralela

- Memória compartilhada
- Threads
- Passagem de mensagem
- Paralelismo de dados
- Modelos híbridos

Modelos de programação paralela

- Modelos de programação paralela fornecem abstração da arquitetura do hardware e de memória
- Os modelos de programação não são específicos para uma determinada arquitetura de computador
  - Modelo de programação com memória compartilhada em um computador com memória distribuída
    - LINDA em um cluster
  - Modelo de programação com passagem de mensagens em um computador com memória compartilhada
    - MPI em uma ORIGIN SGI

Qual modelo de programação paralela utilizar?

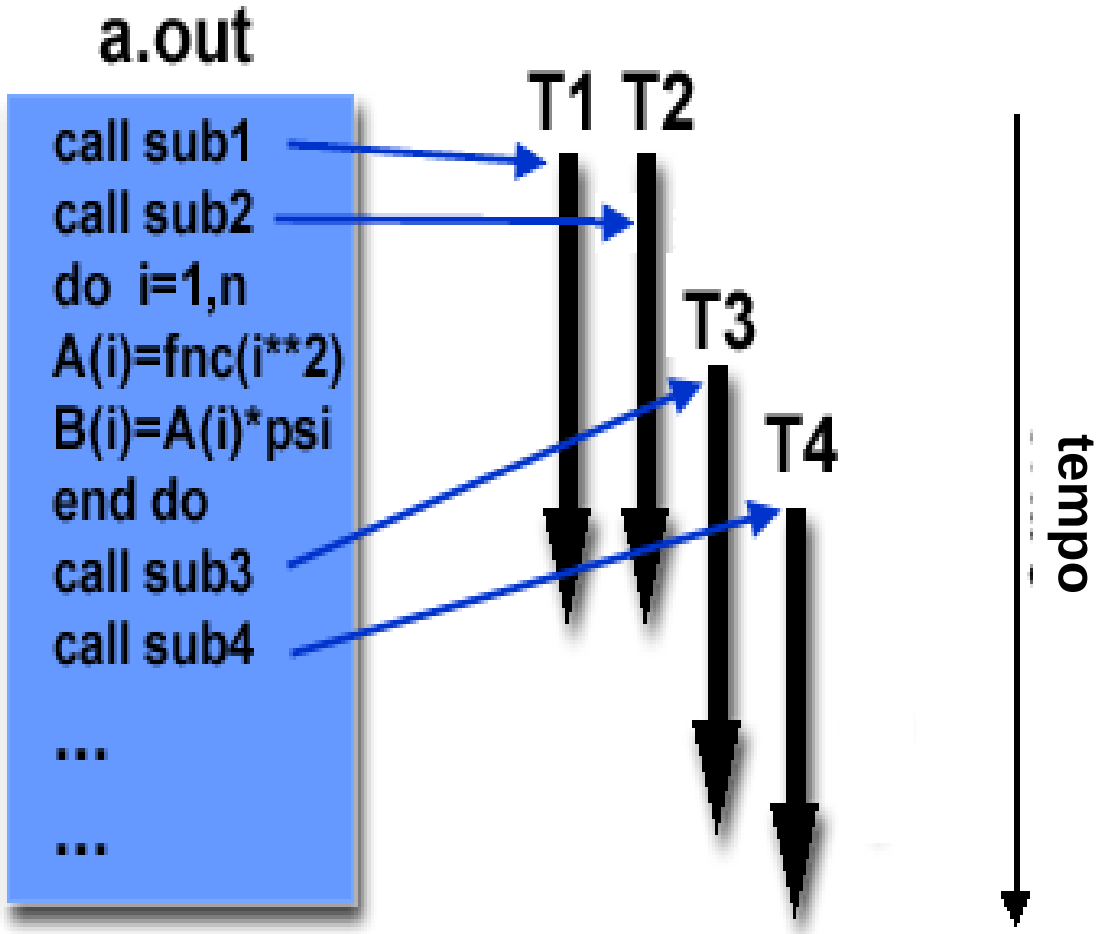
- Combinação de disponibilidade e escolha pessoal
- Qualidade da implementação do modelo é muito importante

Modelo de memória compartilhada

- Tarefas compartilham um espaço de endereçamento o qual pode ser lido e escrito assíncronamente
- Controle de acesso à memória realizado pelo programador
- Não necessita de comunicação explícita entre as tarefas para compartilhar dados
- Controle da localidade dos dados é difícil de ser realizado pelo programador

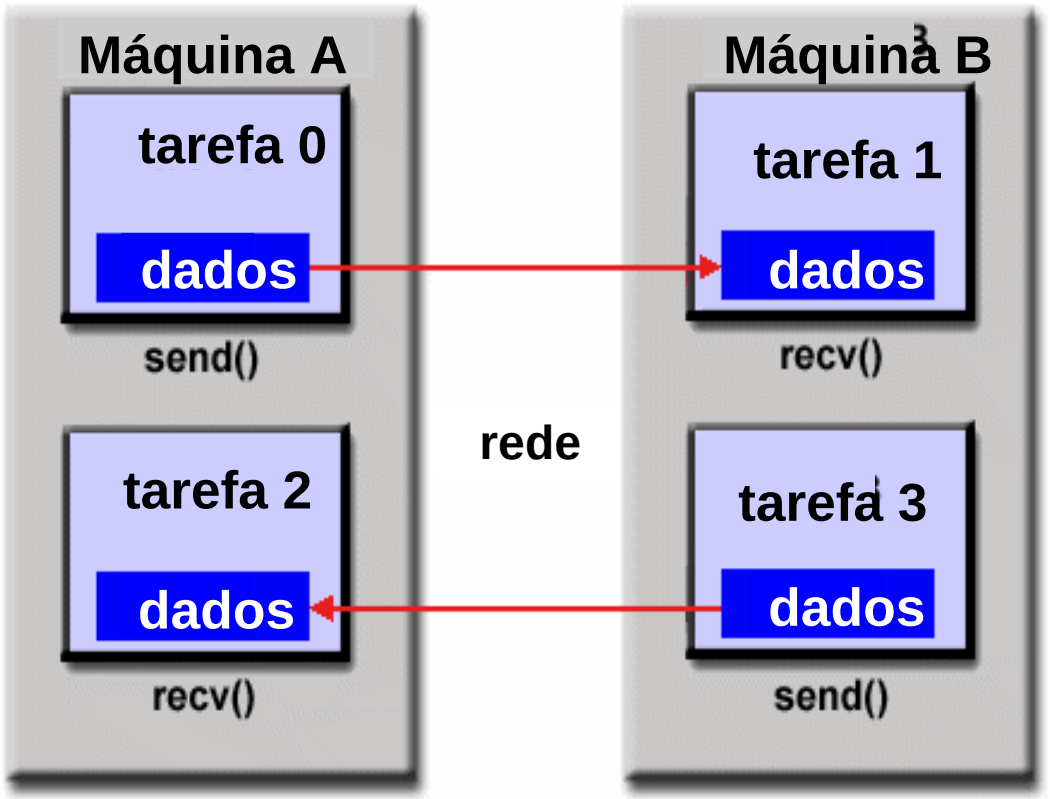
Modelo de threads

- Um único processo pode ter múltiplos e concorrentes caminhos
- Programa principal *a.out* é escalonado para ser executado no computador
- *a.out* executa serialmente e cria várias threads que podem ser executadas de forma concorrente
- Threads possuem memória local e compartilham todos os recursos de *a.out* (memória global)



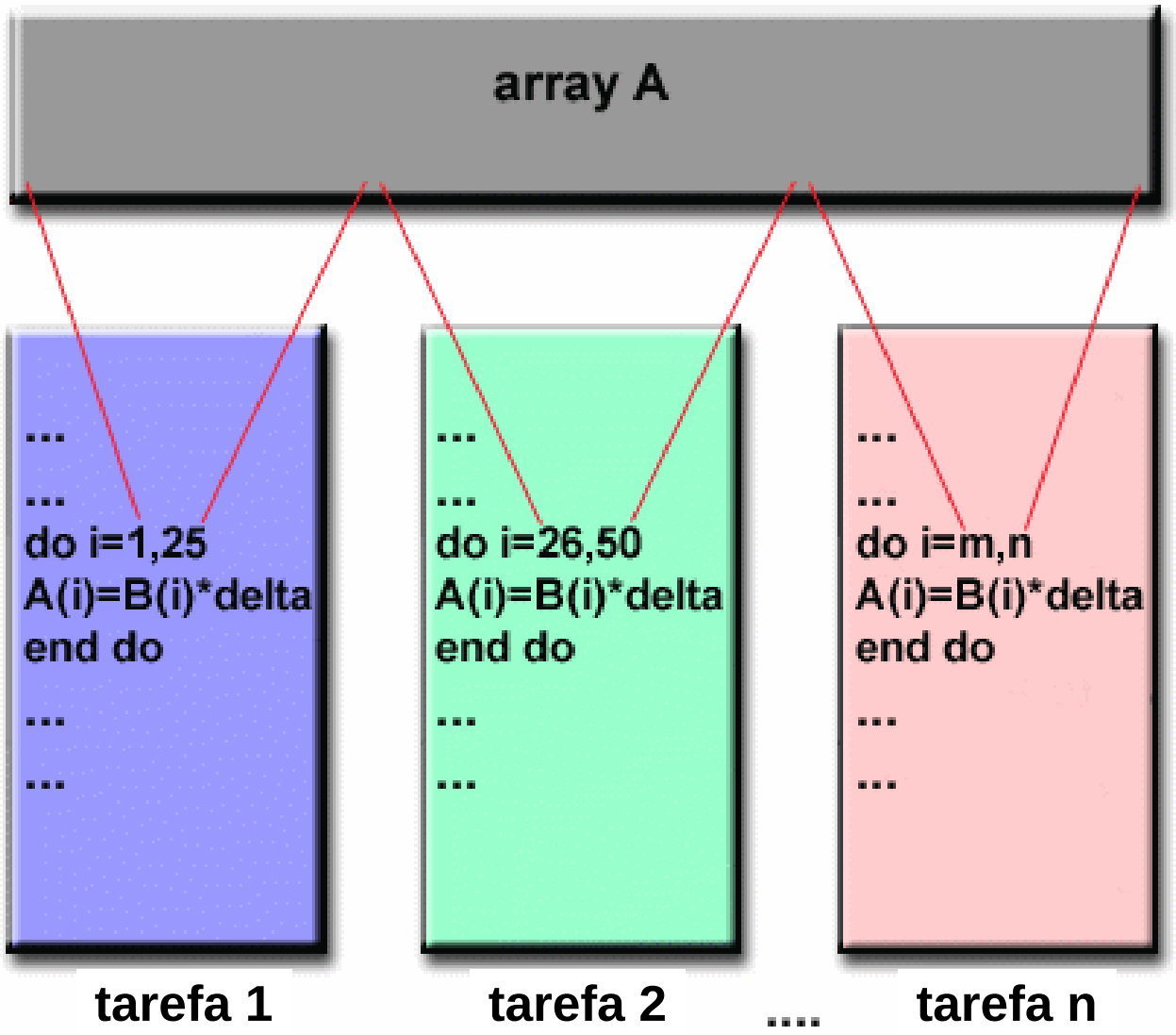
Passagem de mensagens

- Cada tarefa utiliza sua própria memória local
- Tarefas compartilham dados através da troca de mensagens
- Biblioteca de subrotinas que são inseridas no código
- Programador determina envio e recebimento de mensagens
- Padrão MPI



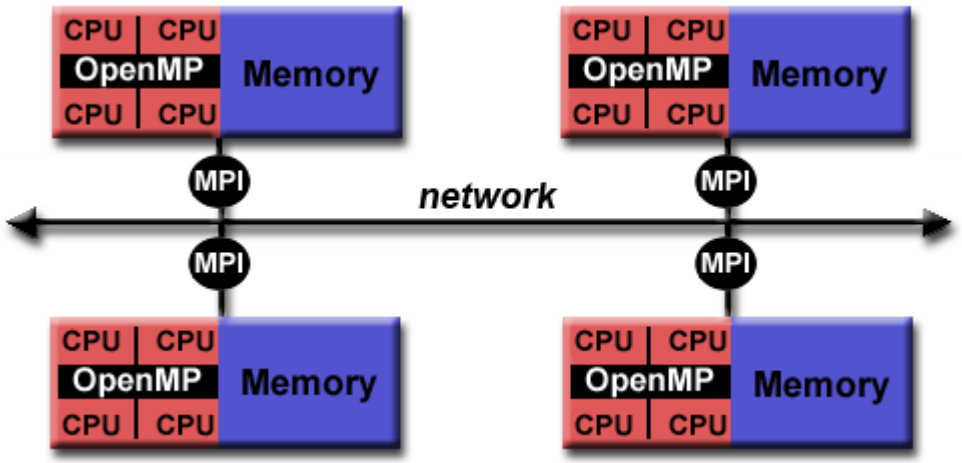
Paralelismo de dados

- Tarefas são executadas sobre o mesmo conjunto de dados organizados como um vetor ou cubo
- Tarefas trabalham sobre diferentes partes da mesma estrutura de dados
- Tarefas executam a mesma operação sobre dados diferentes



Modelo híbrido

- Dois ou mais modelos de programação paralela são combinados
- Combinação de passagem de mensagens (MPI) com threads (OPENMP)

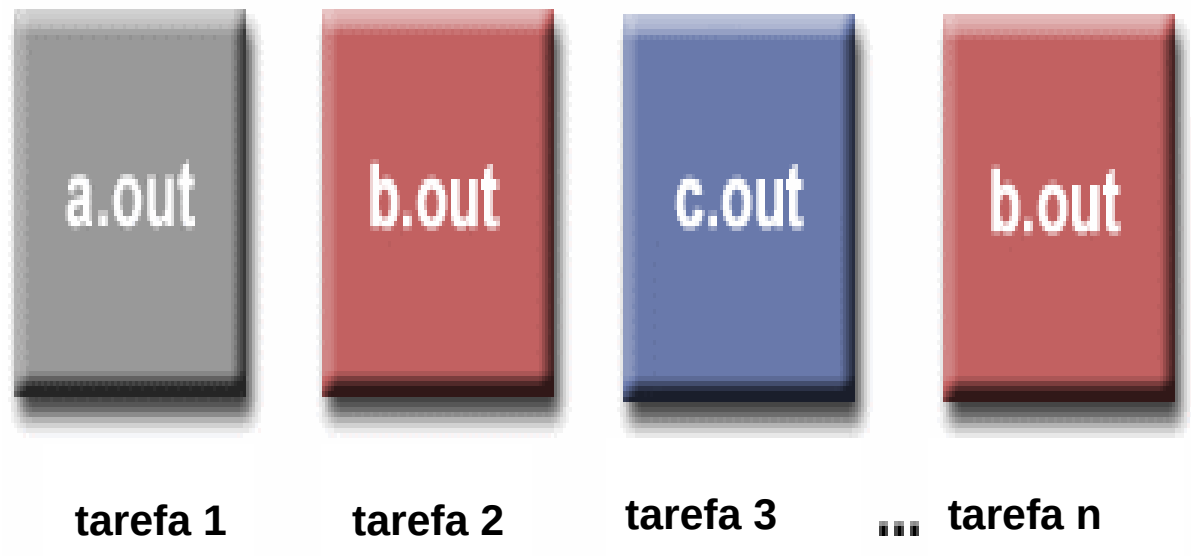


Single Program Multiple Data(SPMD)

- Um único programa executado por todas tarefas simultaneamente
- Tarefas podem estar executando instruções iguais ou diferentes de um mesmo programa
- Tarefas podem utilizar diferentes dados

Mulitple Program Multiple Data(MPMD)

- As tarefas podem estar executando programas iguais ou diferentes
- Todas as tarefas podem estar utilizando dados diferentes



Projeto de programas paralelos

- Paralelização automática ou manual
- Geralmente processo de paralelização realizado de form manual pelo programador
- Existem algumas ferramentas para auxiliar o programador
  - Pré-processadores ou compiladores "paralelizadores"

Projeto de programas paralelos

- Compiladores
  - Totalmente automáticos
    - Compilador analisa o código fonte de um programa fonte sequencial e identifica possibilidades de paralelização
    - Em geral "loops" são o alvo mais frequente de paralelização
  - Diretivas de programação
    - O programador explicitamente indica ao compilador o que paralelizar

Problemas de paralelização automática

- Podem ser produzidos resultados errados
- Queda de desempenho
- Menor flexibilidade no desenvolvimento
- Limitado a uma parte do código
- Difícil de identificar paralelismo no código sequencial

Entedimento do problema e do código sequencial

- Exemplo de um programa trivialmente paralelizável:
  - Encontrar a energia potencial de várias conformações de uma molécula e identificar a conformação com valor mínimo de energia
- Exemplo de um programa não paralelizável:
  - Calcular a séria de Fibonacci
    - F(k+2) = F(k+1) + F(k)

Entedimento do problema e do código sequencial

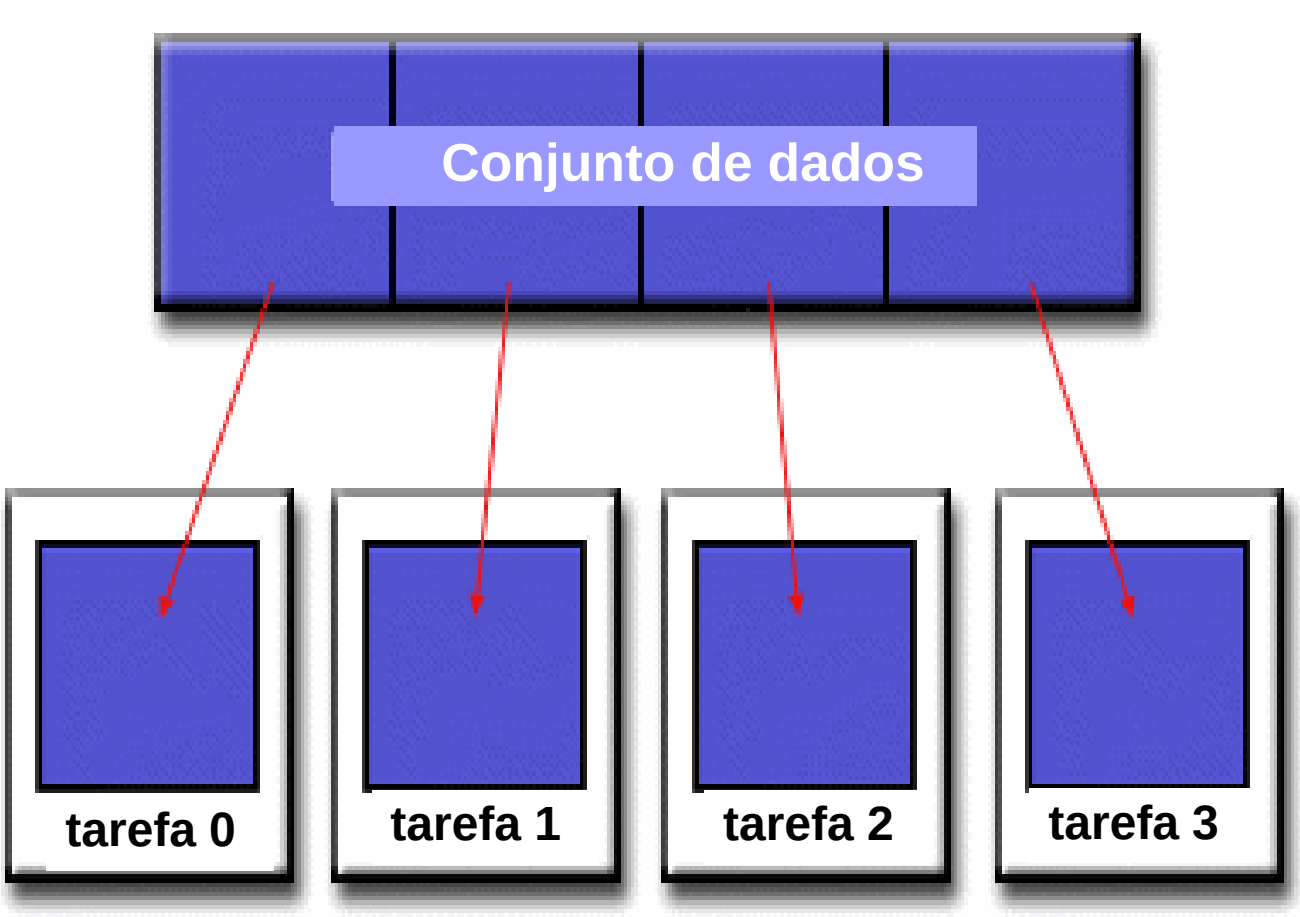
- Identificação das partes do programa que requerem muito processamento da UCP
- Identificação de gargalos do programa (Entrada/Saída)
- Identificação de inibidores de paralelismo (dependência de dados)
- Investigar diferentes algoritmos
  - Algoritmo utilizado para desenvolvimento sequencial pode não ser o melhor para desenvolvimento paralelo

Particionamento

- Dividir o problema em partes discretas de trabalho que podem ser distribuídas a múltiplas tarefas
- Existem dois tipos básicos de particionamento:
  - Particionamento de dados
  - Particionamento funcional

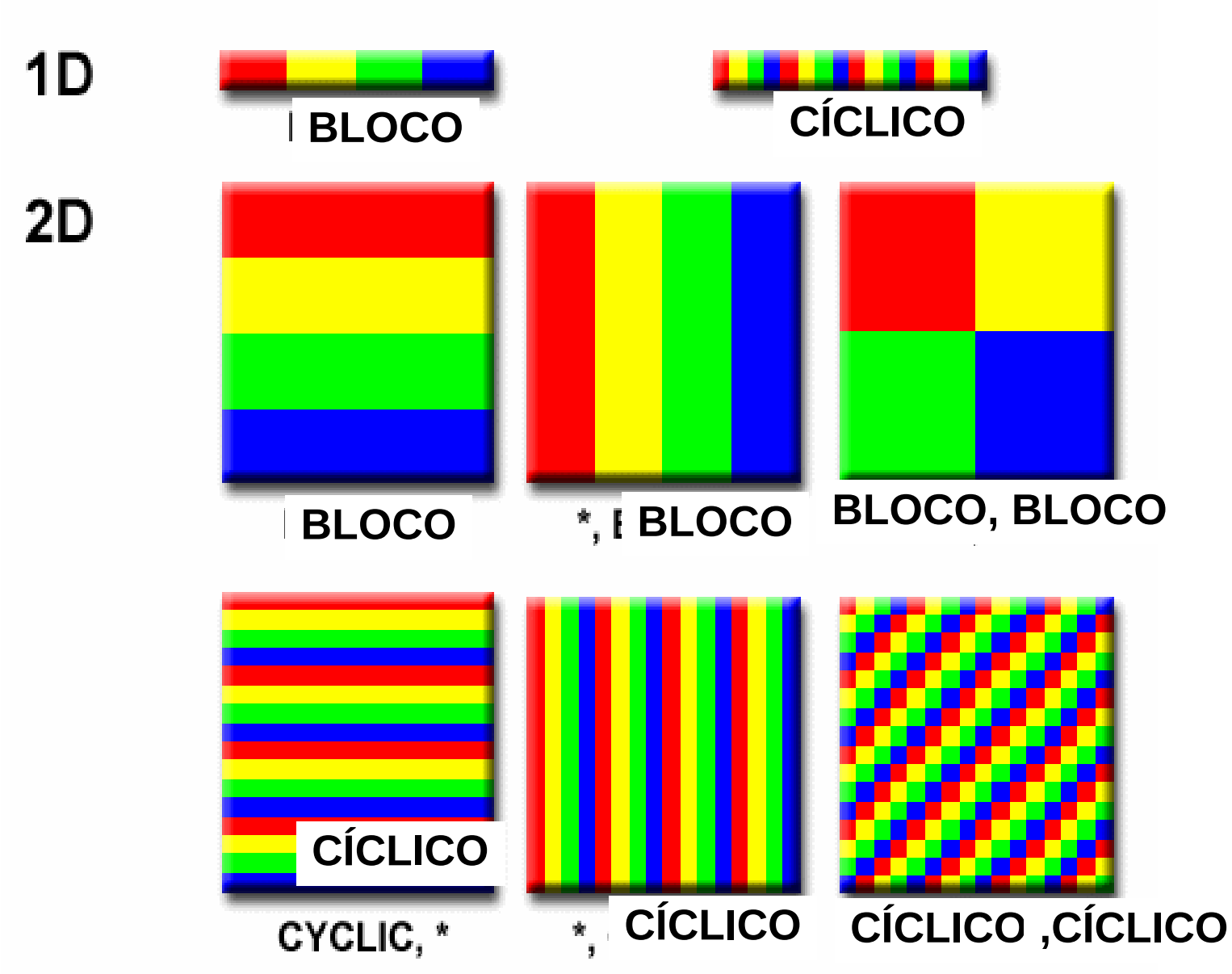
Particionamento de dados

- Os dados são particionados entre as tarefas



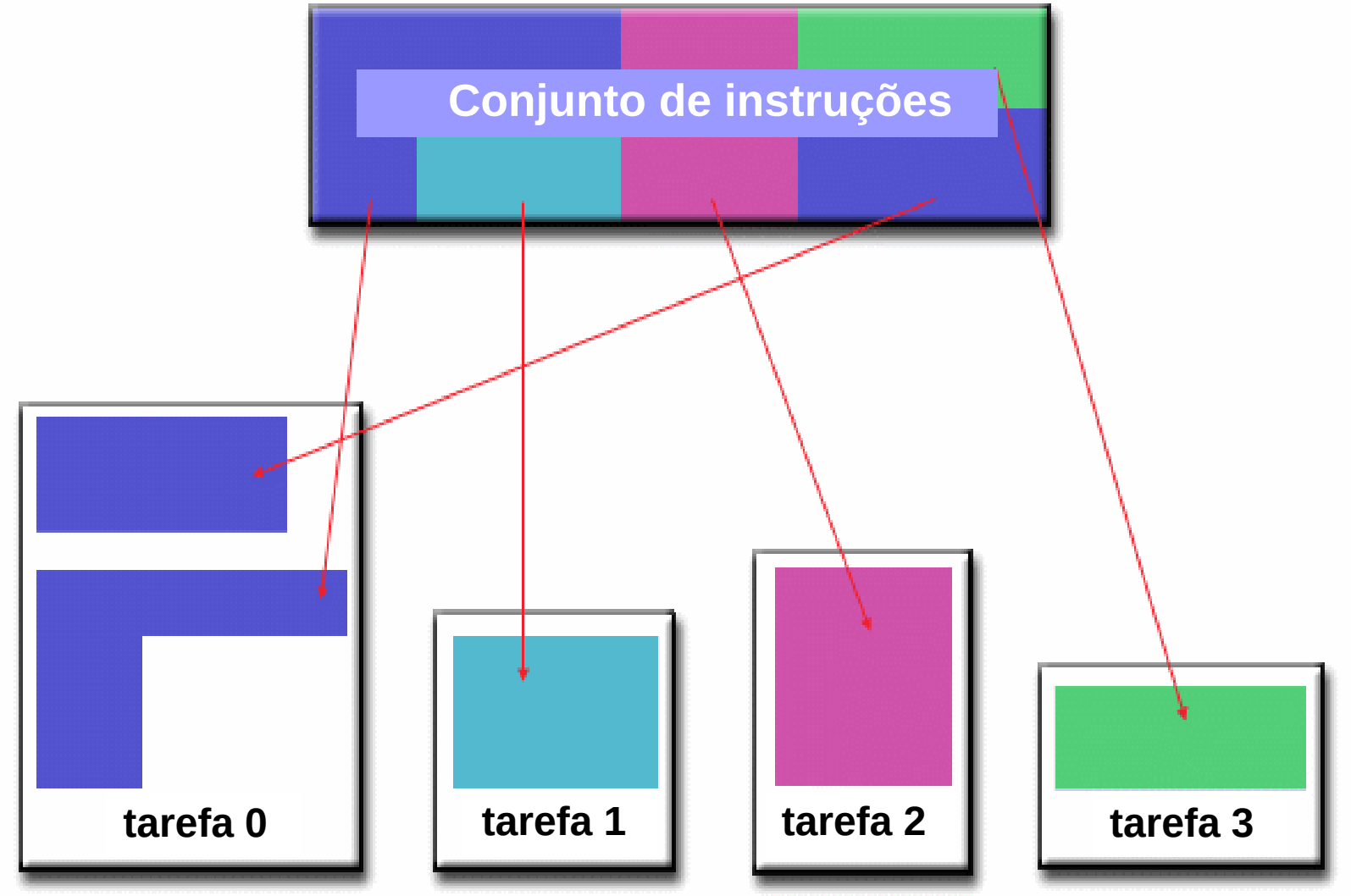
Particionamento de dados

- Diferentes formas de particionamento



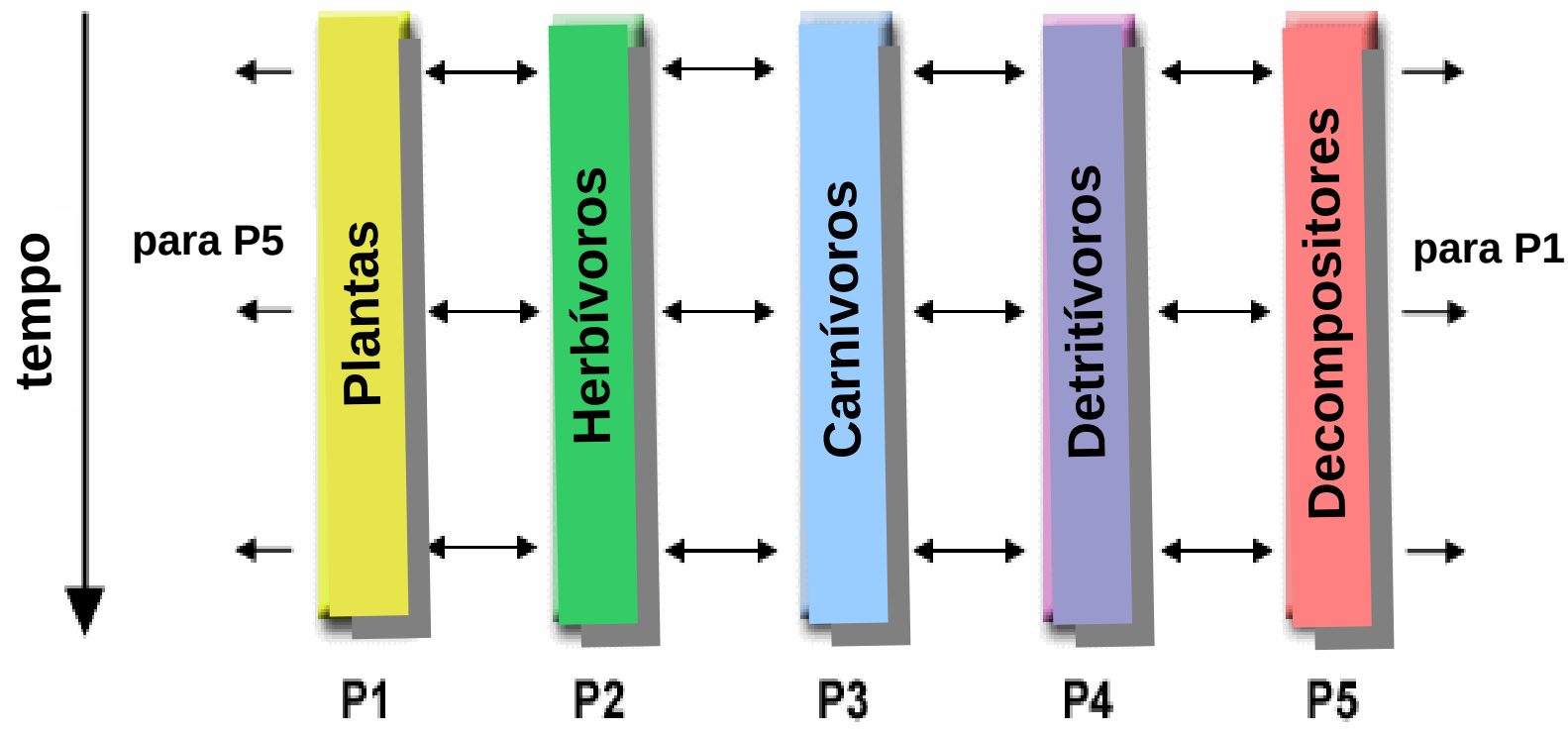
Particionamento funcional

- O problema é decomposto de acordo com o trabalho a ser executado e cada tarefa executa uma porção do trabalho total



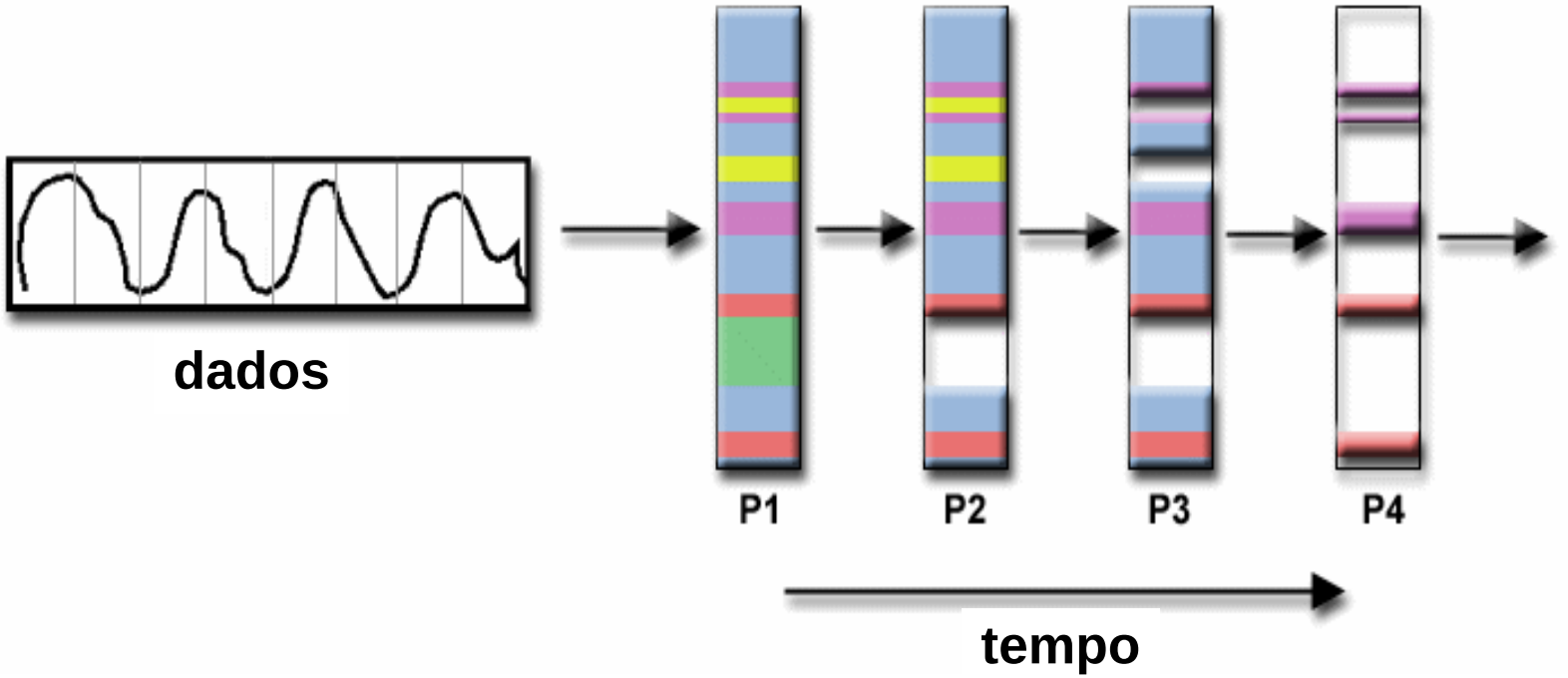
Exemplo de particionamento funcional

- Modelagem de um ecossistema



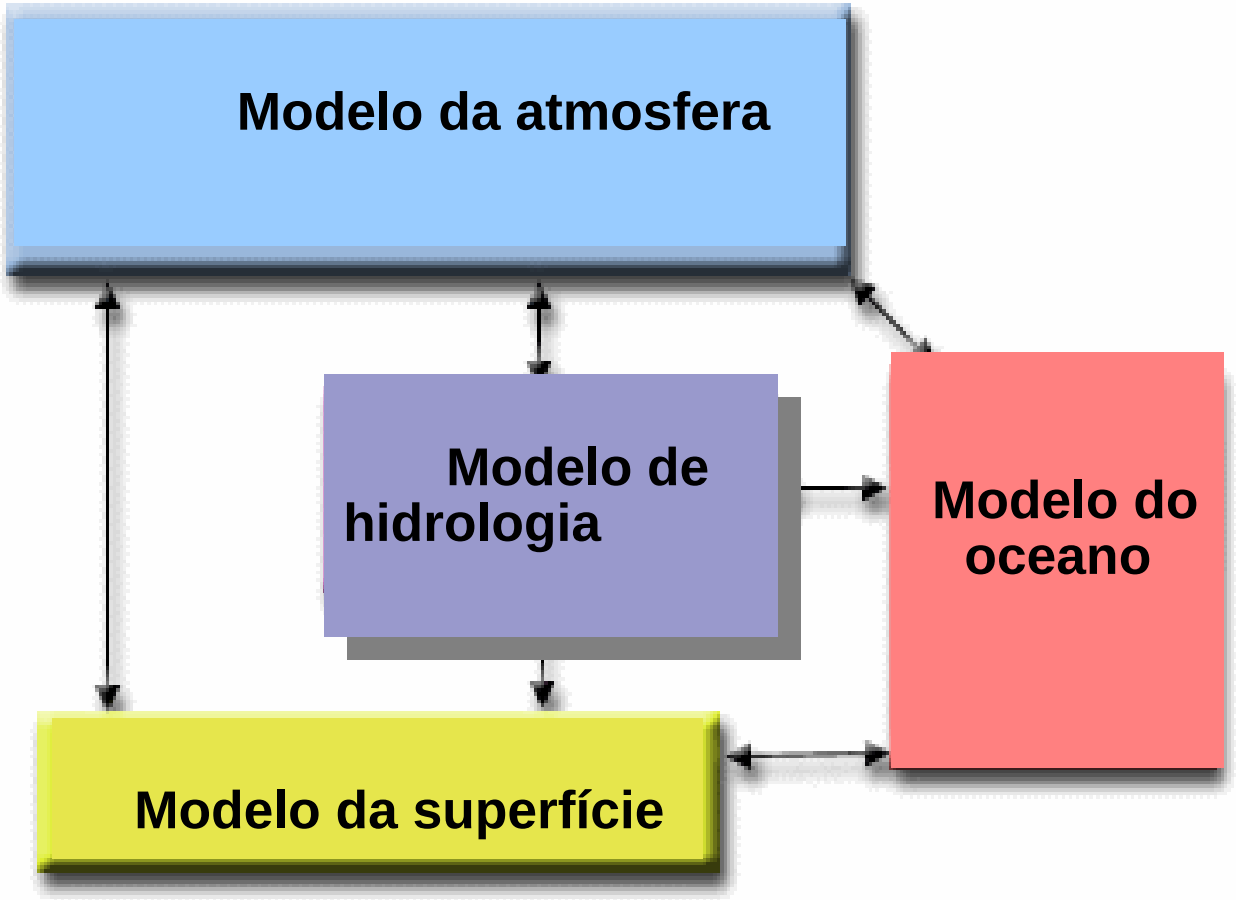
Exemplo de particionamento funcional

- Processamento de sinal
  - Sinal deve passar por 4 filtros diferentes



Exemplo de particionamento funcional

- Modelo do clima
  - Modelo da atmosfera gera dados sobre velocidade dos ventos utilizados pelo modelo do oceano que gera temperaturas dos oceanos utilizadas pelo modelo da atmosfera



Comunicação entre tarefas

- Aplicações que não necessitam de comunicação
  - Inverter cor de uma imagem preto e branco
    - Basta inverter cada pixel independentemente
  - São denominados embaraçosamente paralelos porque paralelização é trivial
- Aplicações que necessitam de comunicação
  - Difusão de calor em uma superfície
    - Uma tarefa necessita saber a temperatura dos seus vizinhos para calcular a temperatura da parte da superfície que ela está processando

Fatores a considerar sobre a comunicação

- Custo de comunicação
  - Comunicação entre tarefas implica em custo extra de processamento
  - Ciclos de máquina e recursos que poderiam ser utilizados no processamento do problema são utilizados para empacotar e desempacotar dados
  - Requer sincronização entre tarefas que pode levar as tarefas a ficarem esperando ao inves de estarem processando
  - Saturação da banda passante do meio de comunicação

Fatores a considerar sobre a comunicação

- Latência vs Banda Passante
  - Latência é o tempo gasto para enviar uma mensagem de tamanho mínimo (0 bytes) de um ponto A a B
  - Banda passante é a quantidade de dados que pode ser transmitida por unidade de tempo
  - O envio de muitas mensagens pequenas pode implicar em muito tempo gasto em latência
  - Geralmente é mais eficiente empacotar várias mensagens pequenas em uma grande

Fatores a considerar sobre a comunicação

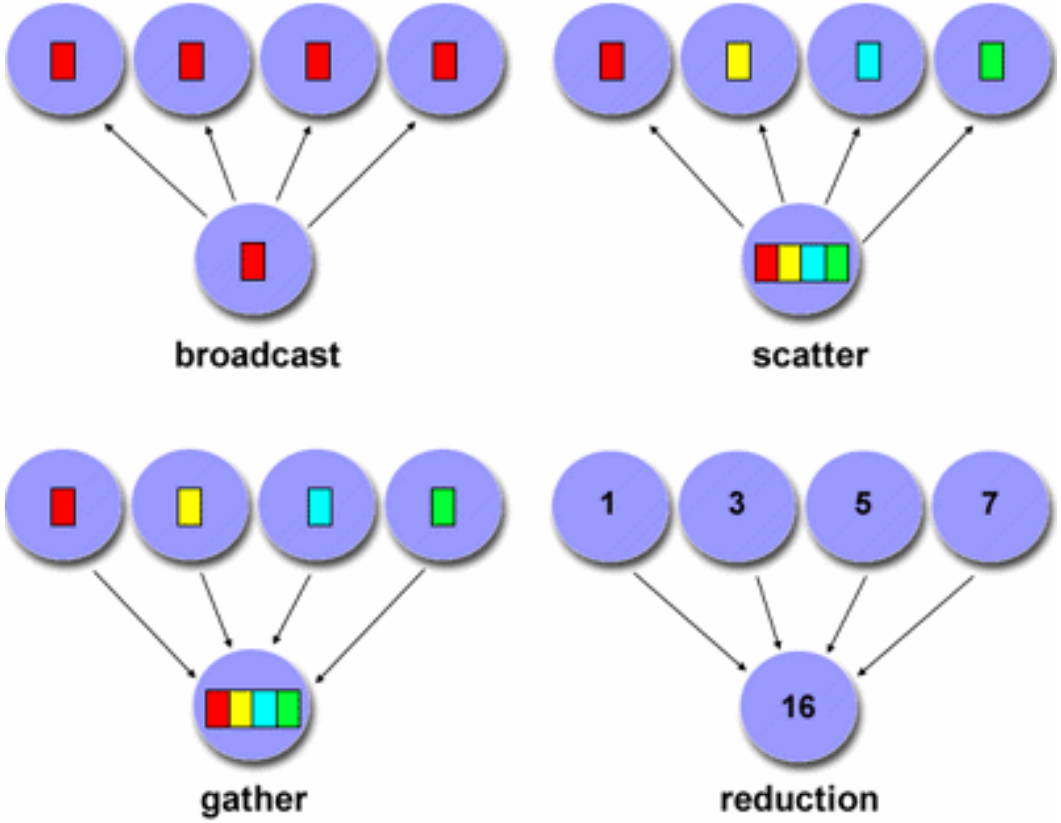
- Visibilidade da comunicação
  - No modelo de passagem de mensagens a comunicação é explícita e visível e controlável pelo programador
  - No modelo de paralelismo de dados a comunicação é transparente para o programador

Fatores a considerar sobre a comunicação

- Síncrona vs Assíncrona
  - Comunicação síncrona requer algum tipo de protocolo de comunicação entre as tarefas que pode ser executado explicitamente pelo programador ou não
  - Comunicações síncronas em geral são bloqueantes porque as tarefas necessitam que a comunicação termine para continuar o processamento
  - Comunicação assíncrona permite que as tarefas transfiram dados independentemente umas das outras
  - Comunicações assíncronas em geral são não bloqueantes porque as tarefas podem executar processamento enquanto a comunicação ocorre

Fatores a considerar sobre a comunicação

- Ponto-a-ponto: envolve a comunicação entre duas tarefas
- Coletiva: envolve comunicação entre tarefas de um grupo



Sincronização

- Barreira
  - Geralmente implica no envolvimento de todas as tarefas
  - Quando uma tarefa chega em uma barreira, ela para
  - Quando a última tarefa atinge a barreira, todas as tarefas estão sincronizadas
- Lock/semáforo
  - Pode envolver um número qualquer de tarefas
  - Utilizados para proteger dados ou código compartilhados
  - A primeira tarefa a obter o lock/semáforo pode acessar a área compartilhada e bloqueia o acesso das outras tarefas
  - As outras tarefas só podem acessar a área compartilhada quando a tarefa que obteve o lock/semáforo liberar

Sincronização

- Operações de comunicação síncronas
  - Envolve somente tarefas que estão executando comunicação
  - Utilizada para coordenar as operações de comunicação entre tarefas
    - Por exemplo, uma tarefa só pode enviar um determinado dado depois de receber uma confirmação de recebimento de um outro dado previamente enviado

Dependência de dados

- Existe dependência entre as instruções de um programa, quando sua ordem de execução afeta o resultado do programa
- Dependência de dados ocorre entre tarefas quando elas utilizam os mesmos dados
- É um dos maiores inibidores para a paralelização

Dependência de dados

- Exemplo de laço com dependência de dados:

```
FOR(J=1; J<FIM;J++)
  A(J)=A(J-1)*2.0
```

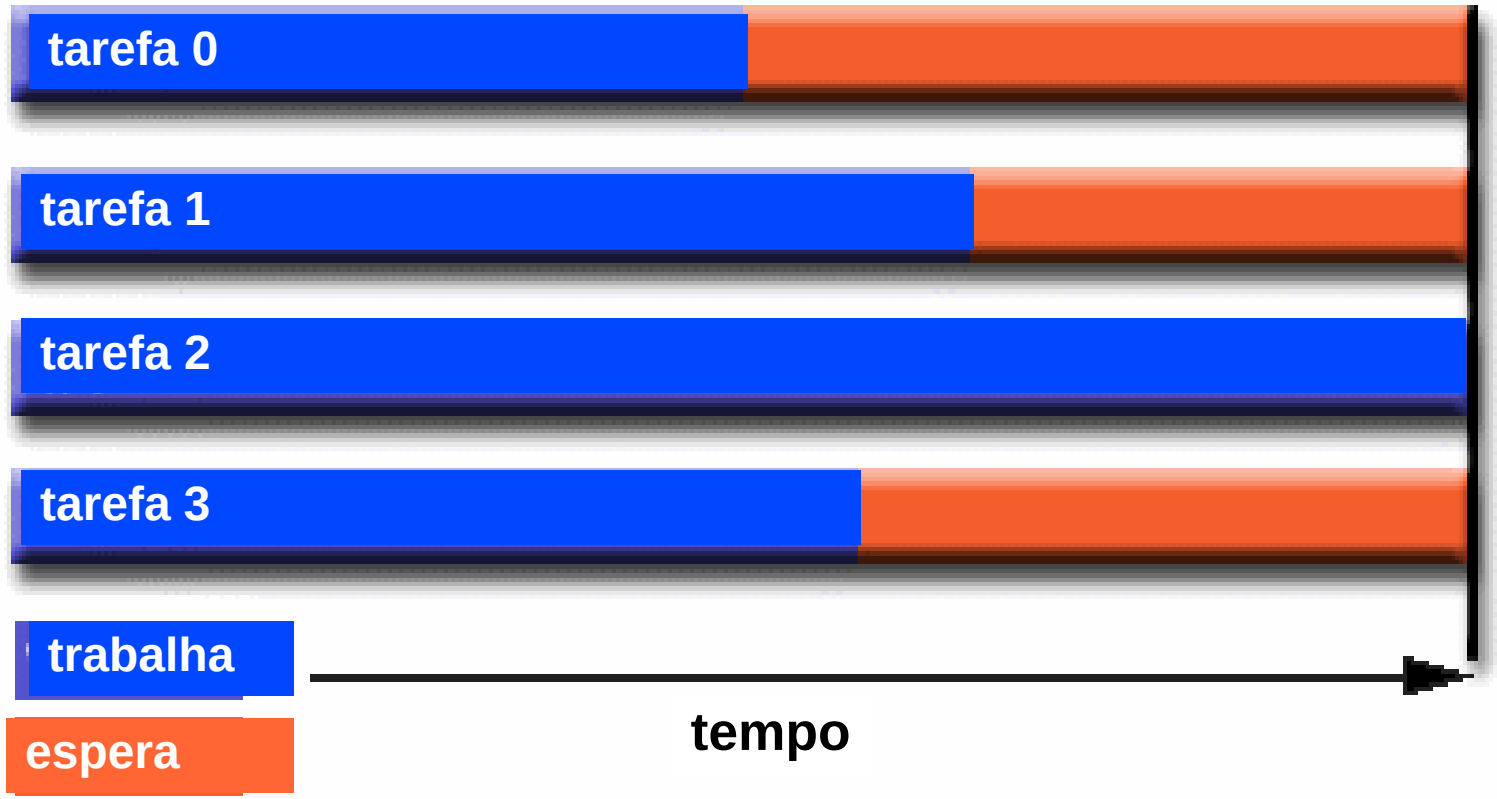
- Outro exemplo:

```
Tarefa  1                    Tarefa 2

-----------                 ------------

 X = 2                       X = 4

    .                           .                    .

 Y = X**2                    Y = X**3
```

  - Memória distribuída: Valor de Y depende se valor de X é transmitido entre tarefas
  - Memória compartilhada: Valor de Y depende de qual tarefa atualiza X por último

Balanceamento de carga

- Distribuição de trabalho entre tarefas de modo a mantê-las ocupadas todo o tempo



Balanceamento de carga

- Dividir o trabalho igualmente entre as tarefas
  - Distribuir elementos de um vetor ou matriz
  - Distribuir iterações de um laço
  - Analisar desempenho das máquinas caso sejam diferentes
- Alocação dinâmica
  - Pode ocorrer desbalanceamento mesmo que os dados sejam distribuídos igualmente:
    - Matrizes esparsas
    - Métodos de malha
  - Quantidade de trabalho não é previsível

Granularidade

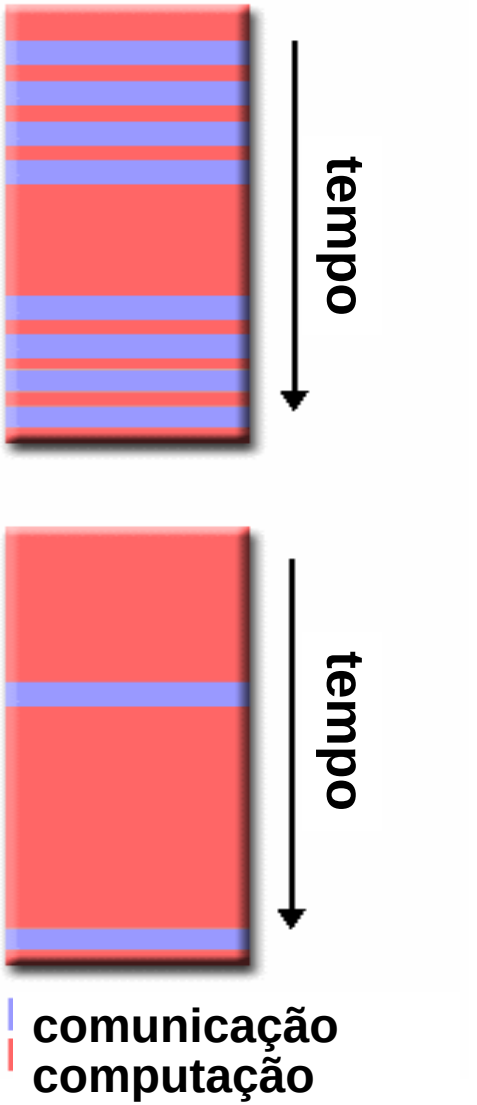
- Medida qualitativa da razão entre processamento de instruções e comunicação
- Paralelismo de granularidade fina
  - Pouco processamento é executado entre eventos de comunicação
  - Baixa razão entre processamento e comunicação
  - Facilita balanceamento de carga
  - Pode acontecer que o processamento exigido para a comunicação entre tarefas seja maior que o processamento para executá-las

Granularidade

- Paralelismo de granularidade grossa
  - Muito processamento é executado entre eventos de comunicação
  - Alta razão entre processamento e comunicação
  - Dificulta balanceamento de carga

Granularidade

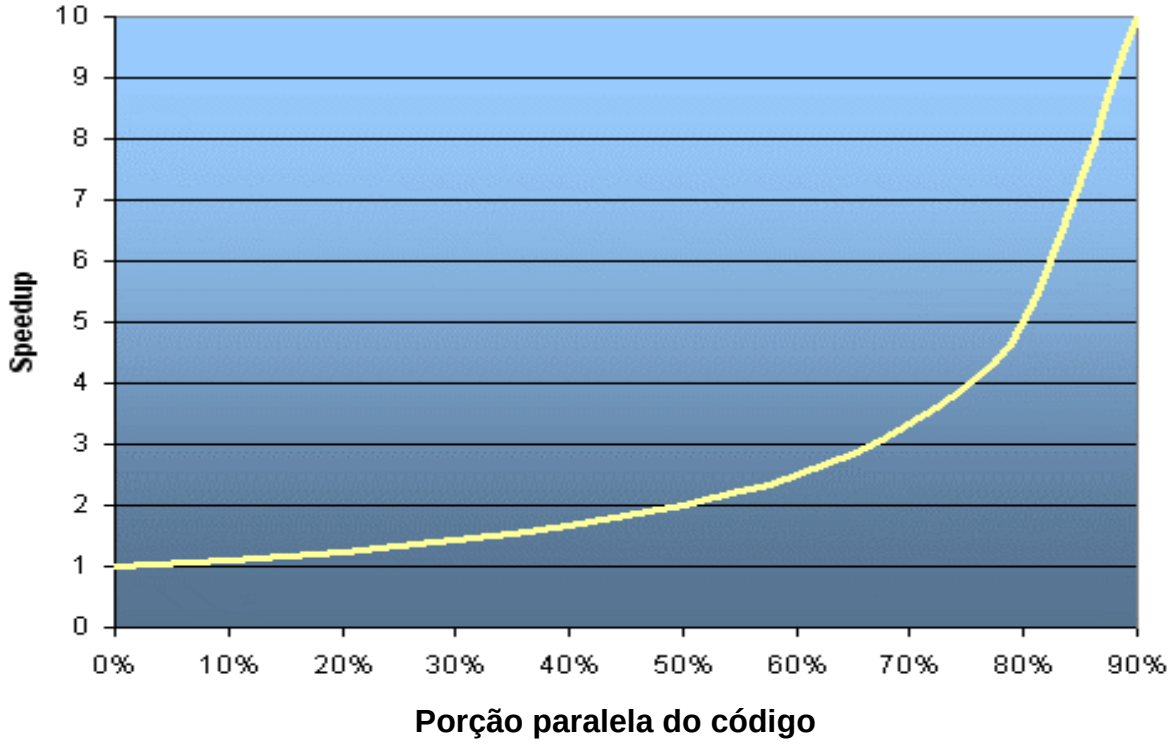
- Qual utilizar?
  - Depende do algoritmo e do ambiente em que o programa será executado
  - Em geral, os tempos de processamento para comunicação são relativamente altos em relação ao processamento de instruções, então prefere-se granularidade grossa
  - Granularidade fina permite um melhor balanceamento de carga



Limites e custos da programação paralela

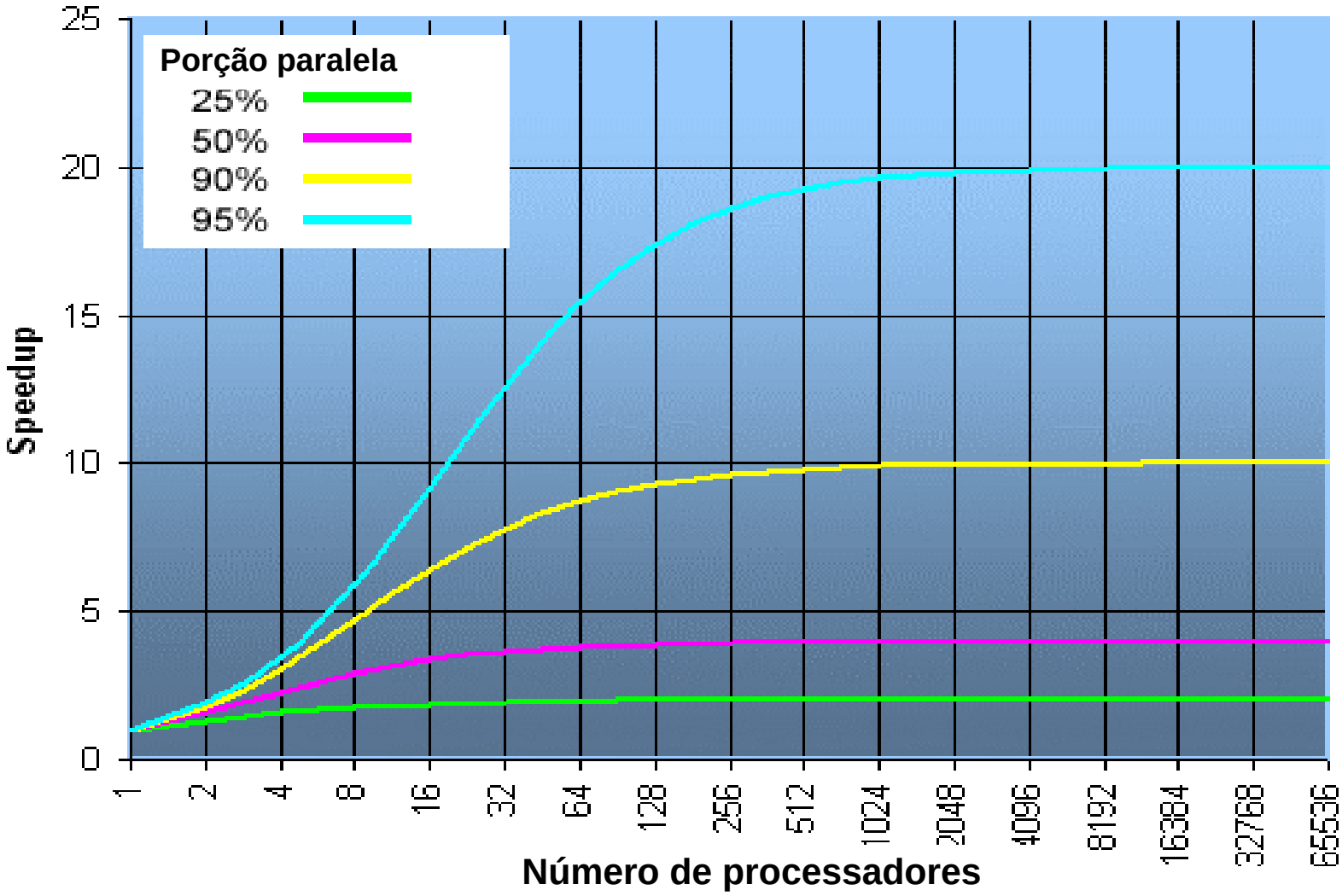
- Lei de Amdhal
  - Aceleração (speed-up) potencial de um programa é definida pela sua fração de código paralelizável (P)

Speed-up= 1/1-P



Speed-up em relação ao número de processadores

- **Speed-up= 1/(P/N+S)**



Utilização de recursos

- Objetivo de paralelizar programas é obter resultados em tempo mais rápido. Isto pode implicar em maior utilização de UCP. Por exemplo, um programa paralelo que executa em 1 hora e utiliza 10 UCPs, utiliza 10 horas de UCP
- Pode-se utilizar mais memória em programas paralelos devido a replicação de dados e a utilização de bibliotecas e sistemas de suporte à programação paralela
- Programas paralelos que demandam pouco processamento podem ser mais ineficientes que a versão sequencial devido a overhead de criação de tarefas e comunicação

Escalabilidade

- Tempo de execução deveria ser proporcional ao número de processadores
- Espera-se que quanto maior o número de processadores, menor o tempo de execução
- Algoritmo utilizado pode inibir a escalabilidade
- Hardware:
  - Banda passante entre UCP e memória em computadores SMP
  - Banda passante da rede de comunicação em clusters
  - Memória disponível
- Implementação das rotinas de biblioteca e subistemas de suporte à programaçao paralela

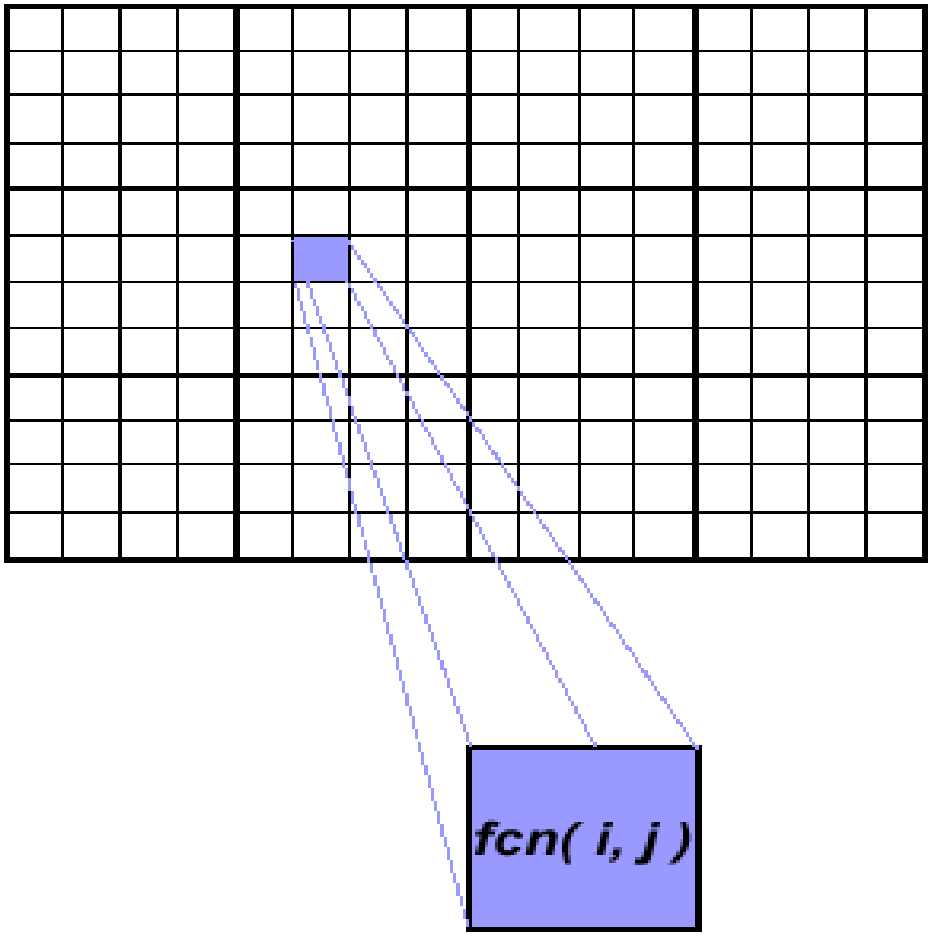
Análise de desempenho e depuração

- Mais difícil depurar e analisar desempenho de programas paralelos que programas sequenciais
- Existem algumas ferramentas disponíveis
- Bastante trabalho a ser realizado em relação a este assunto

Exemplo: Processamento de matriz

- Código sequencial:

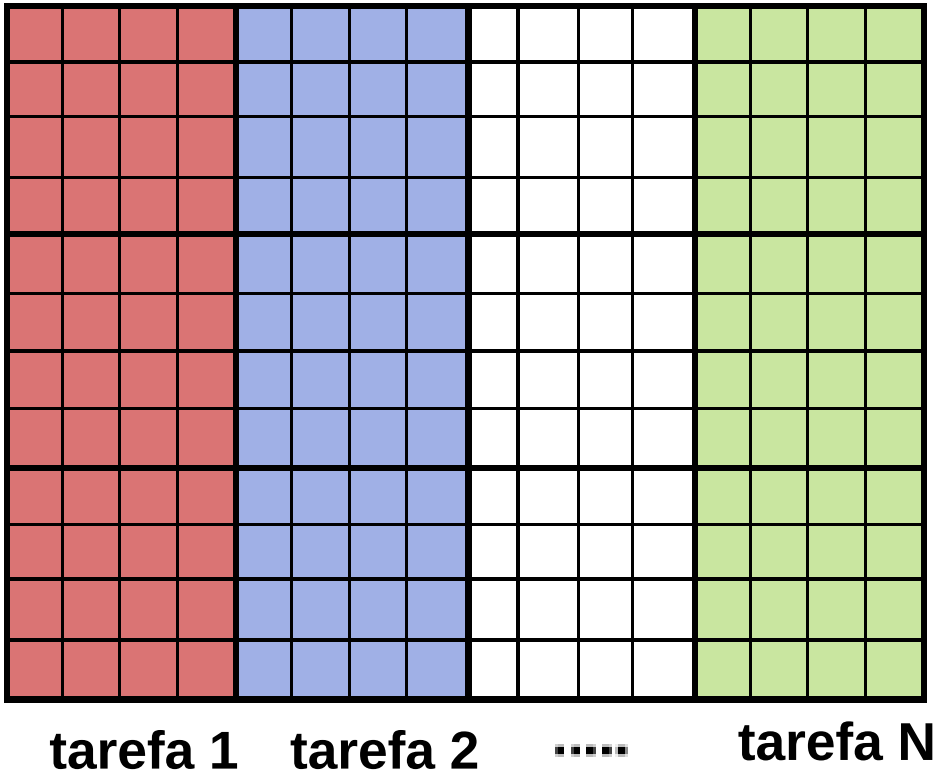
```
for (j=0;j<n;j++)
  for (i=0;i<n;i++)
     A[i,j]=fcn(i,j)
```



Primeira solução paralela

- Cada processador executa o loop sobre uma parte da matriz

```
for (j=meucomeco;j<meufim;j++)
  for (i=0;i<n;i++)
     A[i,j]=fcn(i,j)
```



Solução SPMD

- Processador mestre inicializa matriz, envia dados para os trabalhadores e recebe resultados
- Cada trabalhador recebe dados do mestre, executa o laço e envia resultado

```
Verifica se é MESTRE ou TRABALHADOR
Se é MESTRE
  inicializa matriz
  envia informação da parte da matriz para cada
  trabalhador
  espera resultados de cada TRABALHADOR
Senão
  recebe informação da matriz do MESTRE
 executa loop
 j=meucomeco;j<meufim;j++)
  for (i=0;i<n;i++)
     A[i,j]=fcn(i,j)
 envia resultados para o MESTRE
```

Segunda solução: repositório de tarefas

- Processador mestre inicializa matriz e gerencia repositório de tarefas
- Cada trabalhador solicita uma tarefa de cada vez ao mestre.
- Cada tarefa consiste no cálculo da função para um elemento da matriz

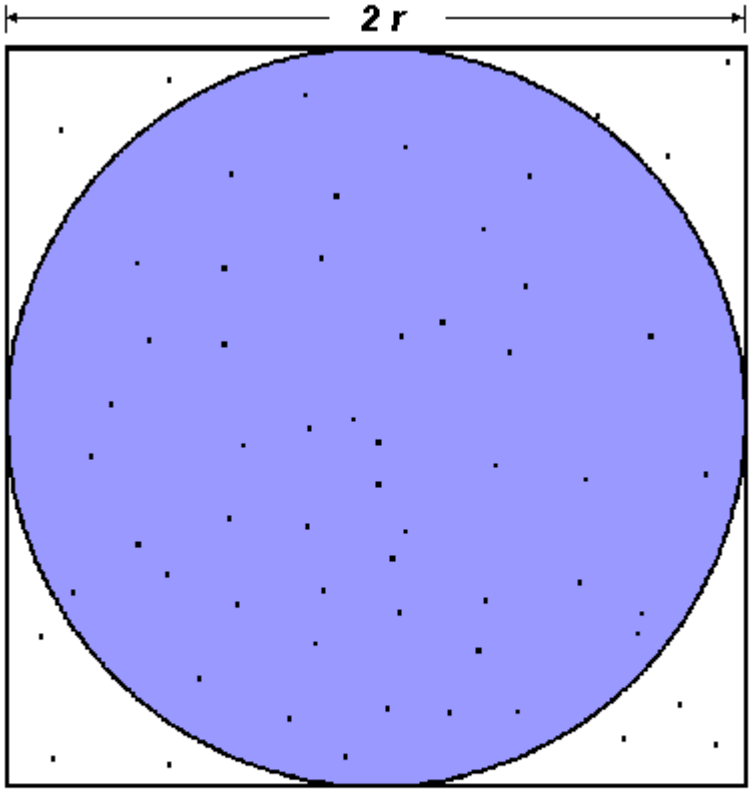
```
Verifica se é MESTRE ou TRABALHADOR
Se é MESTRE
  enquanto existirem tarefas
        envia próxima tarefa para trabalhador
     recebe resultados de trabalhador
  avisa trabalhadores que não existem mais tarefas
Senão
  enquanto existirem tarefas
        recebe tarefa do mestre
        A[i,j]=fcn(i,j)
        envia resultados para o MESTRE
```

- Qual deve ser o tamanho da tarefa?

Cálculo de ¶

- Gera pontos aleatórios e contabiliza pontos que estão dentro círculo

```
npontos=10000
contcirc=0
Para j=1 até npontos
  gera 2 números aleatórios entre 0
  e 1
  coordx=num1
  coordy=num2
  se (coordx,coordy) dentro do
  círculo
        contcirc=contcirc+1
PI=4.0*contcirc/npontos
```



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

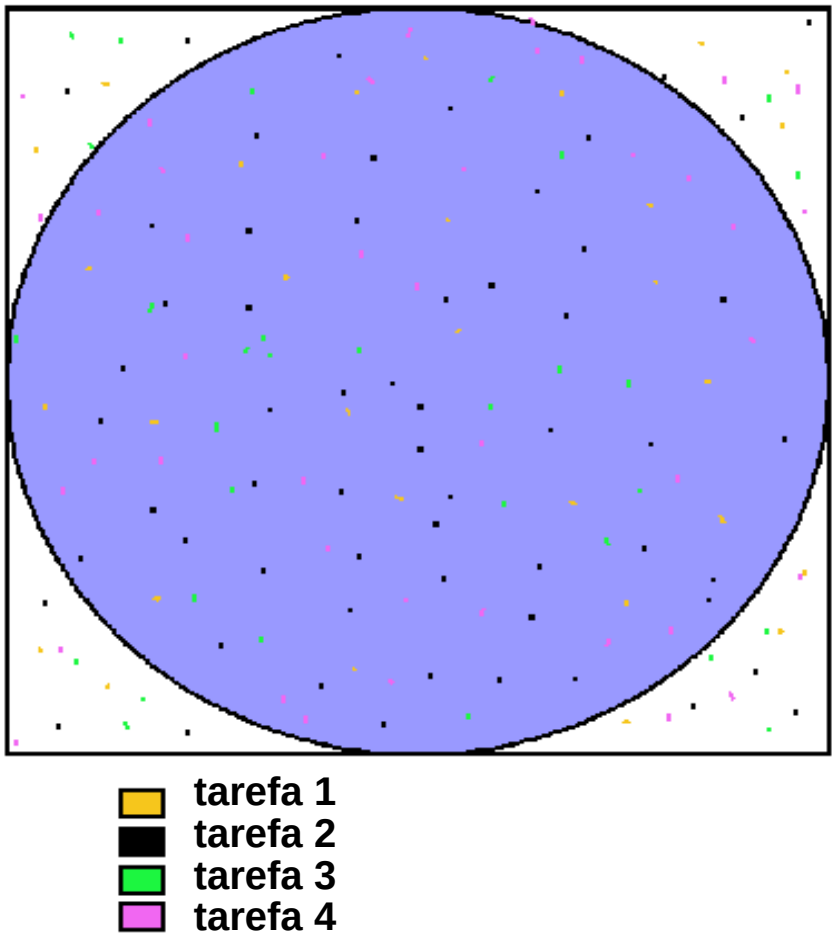
$$A_C = \pi r^2$$

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

Solução paralela: Cálculo de ¶

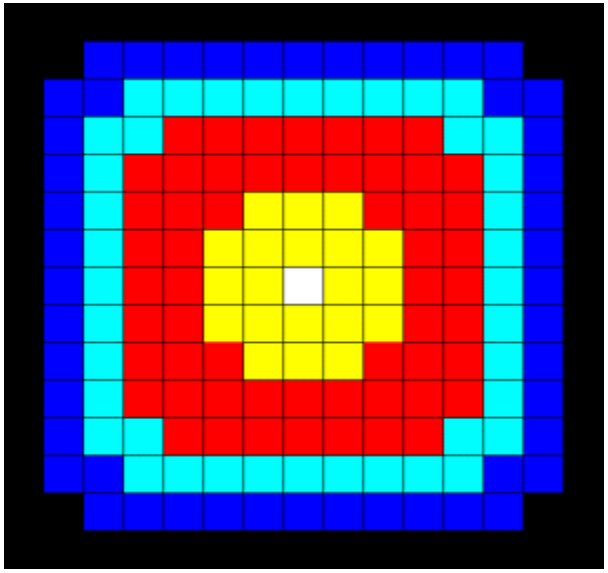
- Cada tarefa executa uma parte do loop

```
npontos=10000
contcirc=0
p=número de tarefas
num=npontos/p
Para j=1 até num
  gera 2 números aleatórios entre 0 e
  1
  coordx=num1
  coordy=num2
  se (coordx,coordy) dentro do
  círculo
        contcirc=contcirc+1
Se sou MESTRE
  recebe contcirc de cada trabalhador
  calcula PI
Senão
  envia contcirc para MESTRE
```



Cálculo de temperatura

- Variação de temperatura em uma superfície dadas as temperaturas iniciais e condições de fronteira
- Inicialmente temperatura mais alta no meio e zero nas bordas
- Temperatura das bordas mantida em zero durante todo o tempo

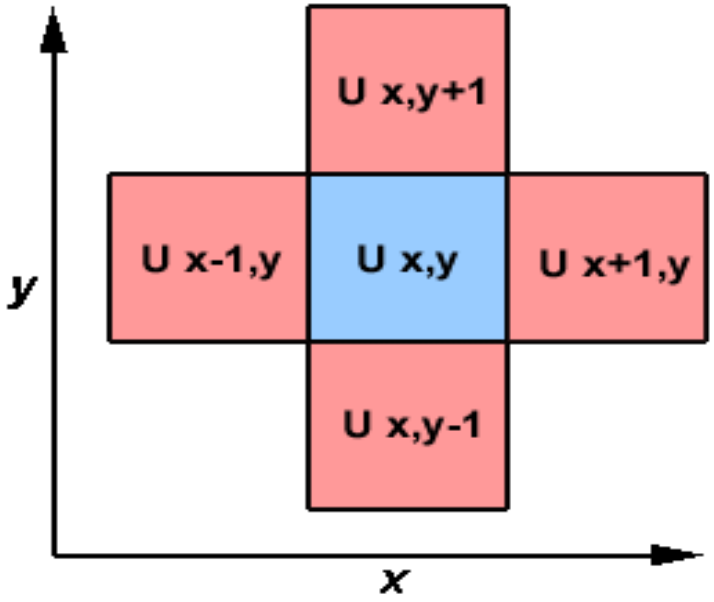


Cálculo de temperatura

$$U_{x,y} = U_{x,y}$$

$$+ C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy})$$

$$+ C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})$$



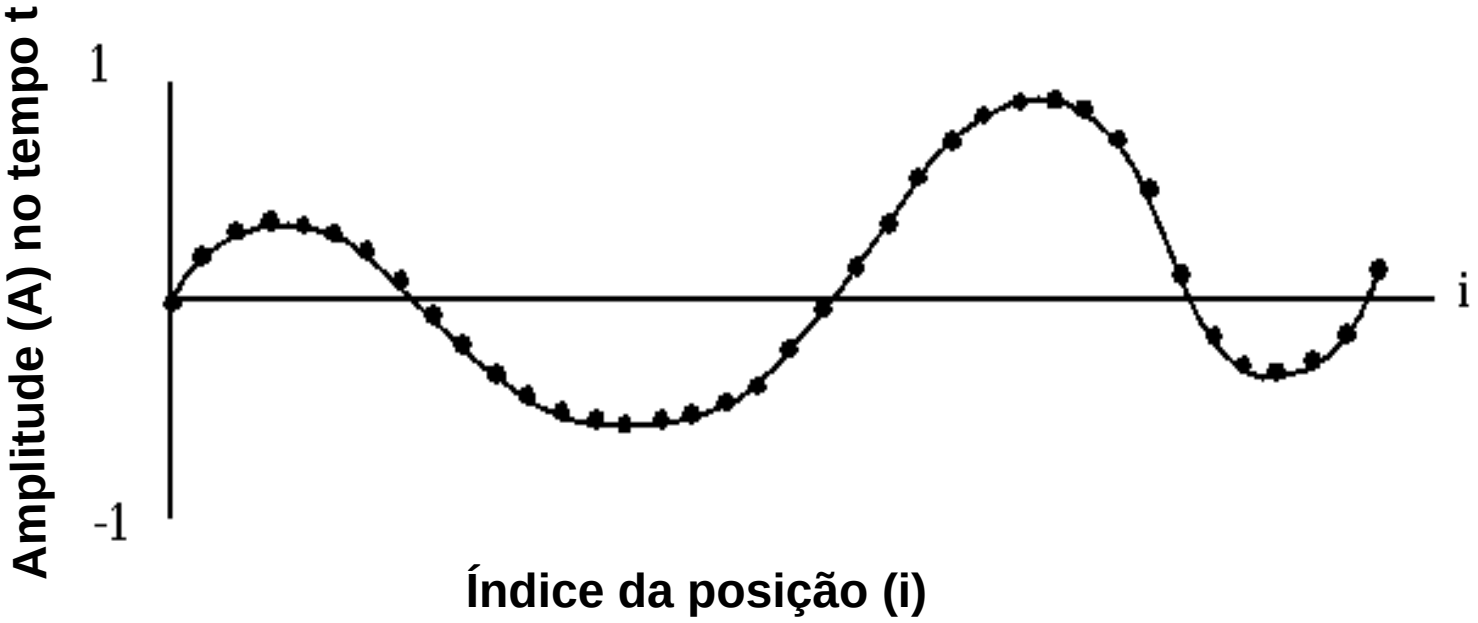

```
Para iy=2, ny-1
  Para ix=2,ix-1
          u2(ix, iy) = u1(ix, iy) + cx*(u1(ix+1,iy) + u1(ix-1,iy)-2*u1(ix,iy))
  +
                       cy*(u1(ix,iy+1) + u1(ix,iy-1) - 2*u1(ix,iy))
```

Solução1: Cálculo de temperatura

```
Verifica se é MESTRE ou TRABALHADOR
Se é MESTRE
  inicializa matriz
  envia informação inicial da matriz para cada trabalhador
  espera resultados de cada TRABALHADOR
Senão
  recebe informação da matriz do MESTRE
  faça de t=1 até npassos
         atualiza tempo
         envia informação de borda aos vizinhos
         recebe informação de borda dos vizinhos
         atualiza valores da submatriz
  envia resultados para o MESTRE
```

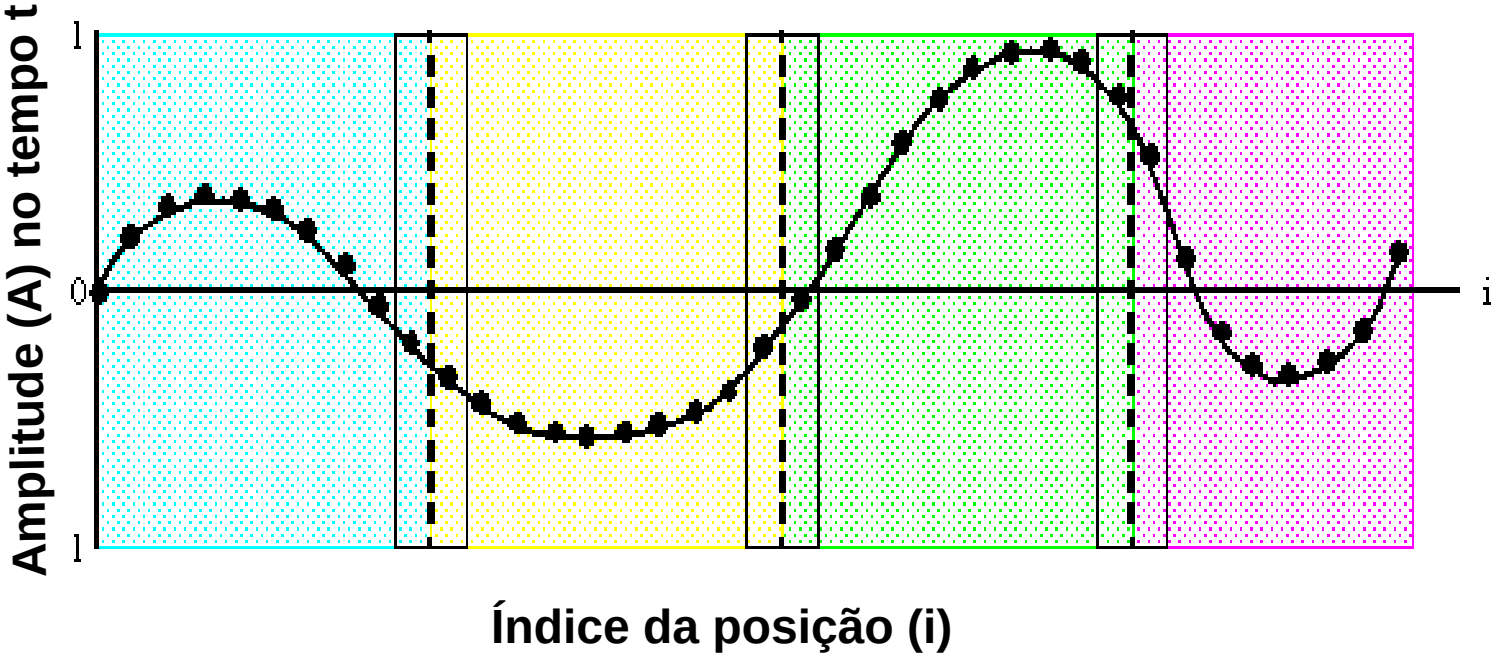
Equação de onda uni-dimensional

```
A(i,t+1) = (2.0*A(i,t))- A(i,t-1)
+ (c*(A(i-1,t)-(2.0*A(i,t))+A(i+1,t)))
```



Equação de onda uni-dimensional

- Vetor dividido entre tarefas
- Tarefas requerem mesma quantidade de processamento
- Só ocorrre comunicação entre as bordas



Equação de onda uni-dimensional

```
Identifica quantidade e identificadores de tarefas
vizesq=meuid-1
vizdir=meuid+1
Se meuid=primeiro então esqviz=último
Se meuid=último então esqdir=primeiro
Se é MESTRE
  inicializa matriz
  envia informação inicial da matriz para cada trabalhador
Senão
  recebe informação da matriz do MESTRE
Para t=1,npassos
  envia ponto da esquerda para vizinho da esquerda
  recebe ponto da esquerda do vizinho da direita
  envia ponto da direita para vizinho da direita
  recebe ponto da direita do vizinho da esquerda
  para i=1, npontos
      novval[i] = (2.0*val[i])- velhoval[i]+(c*(val[i-1]-
  (2.0*val[i])+val[i+1]))
Se é MESTRE
  recebe resultados dos TRABALHADORES
  escreve resultados em arquivo
Senão
  envia resultados para MESTRE
```

Referência

- **Material baseado no tutorial Introduction to Parallel Computing, Blaise Barney (computing.llnl.gov/tutorials/parallel_comp)**