

Programação Paralela e Distribuída

Fundamentos

Porquê Programação Paralela?

“Se um único computador (processador) consegue resolver um problema em N segundos, podem N computadores (processadores) resolver o mesmo problema em 1 segundo?”

Porquê Programação Paralela?

- Dois dos principais motivos para utilizar programação paralela são:
 - Reduzir o tempo necessário para solucionar um problema.
 - Resolver problemas mais complexos e de maior dimensão.
- Outros motivos são:
 - Tirar partido de recursos computacionais não disponíveis localmente ou subaproveitados.
 - Ultrapassar limitações de memória quando a memória disponível num único computador é insuficiente para a resolução do problema.
 - Ultrapassar os limites físicos de velocidade e de miniaturização que actualmente começam a restringir a possibilidade de construção de computadores sequenciais cada vez mais rápidos.

Porquê Programação Paralela?

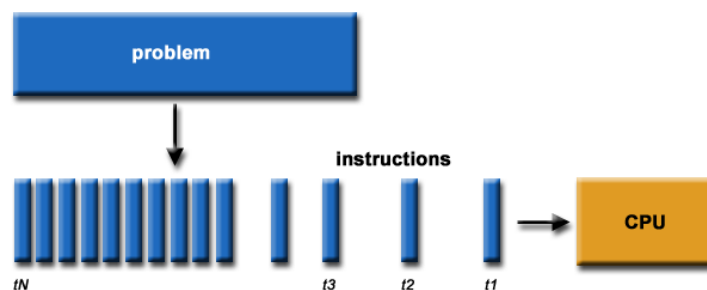
- Tradicionalmente, a programação paralela foi motivada pela resolução/simulação de problemas fundamentais da ciência/engenharia de grande relevância científica e económica, denominados como *Grand Challenge Problems (GCPs)*.
- Tipicamente, os GCPs simulam fenómenos que não podem ser medidos por experimentação:
 - Fenómenos climáticos (e.g. movimento das placas tectónicas)
 - Fenómenos físicos (e.g. órbita dos planetas)
 - Fenómenos químicos (e.g. reacções nucleares)
 - Fenómenos biológicos (e.g. genoma humano)
 - Fenómenos geológicos (e.g. actividade sísmica)
 - Componentes mecânicos (e.g. aerodinâmica/resistência de materiais em naves espaciais)
 - Circuitos electrónicos (e.g. verificação de placas de computador)
 - ...

Porquê Programação Paralela?

- Actualmente, as aplicações que exigem o desenvolvimento de computadores cada vez mais rápidos estão por todo o lado. Estas aplicações ou requerem um **grande poder de computação** ou requerem o **processamento de grandes quantidades de informação**. Alguns exemplos são:
 - Bases de dados paralelas
 - Mineração de dados (*data mining*)
 - Serviços de procura baseados na *web*
 - Serviços associados a tecnologias multimédia e telecomunicações
 - Computação gráfica e realidade virtual
 - Diagnóstico médico assistido por computador
 - Gestão de grandes indústrias/corporações
 - ...

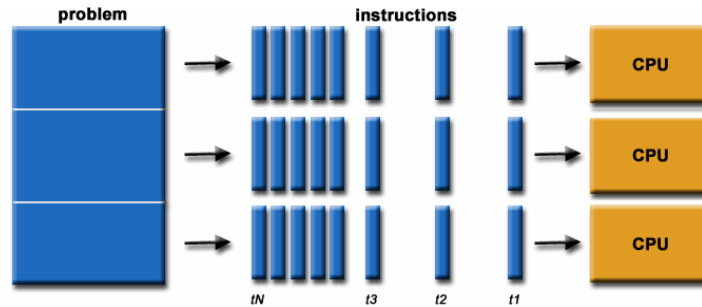
Programação Sequencial

- Um programa é considerado programação sequencial quando este é visto como uma série de instruções sequenciais que devem ser executadas num único processador.



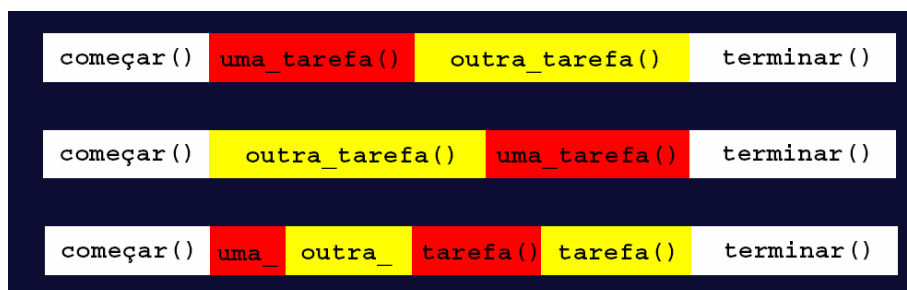
Programação Paralela

- Um programa é considerado programação paralela quando este é visto como um conjunto de partes que podem ser resolvidas **concorrentemente**. Cada parte é igualmente constituída por uma série de instruções sequenciais, mas que no seu conjunto podem ser executadas simultaneamente em vários processadores.



Concorrência ou Paralelismo Potencial

- Concorrência ou paralelismo potencial diz-se quando um programa possui **tarefas** (partes contíguas do programa) que podem ser executadas em qualquer ordem sem alterar o resultado final.



Paralelismo

- Paralelismo diz-se quando as tarefas de um programa são executadas em simultâneo em mais do que um processador.



Paralelismo Implícito

- O paralelismo diz-se implícito quando cabe ao **compilador** e ao **sistema de execução**:
 - Detectar o paralelismo potencial do programa.
 - Atribuir as tarefas para execução em paralelo.
 - Controlar e sincronizar toda a execução.
- Vantagens e inconvenientes:
 - (+) Liberta o programador dos detalhes da execução paralela.
 - (+) Solução mais geral e mais flexível.
 - (-) Difícil conseguir-se uma solução eficiente para todos os casos.

Paralelismo Explícito

- O paralelismo diz-se explícito quando cabe ao **programador**:
 - Anotar as tarefas para execução em paralelo.
 - Atribuir (possivelmente) as tarefas aos processadores.
 - Controlar a execução indicando os pontos de sincronização.
 - Conhecer a arquitectura dos computadores de forma a conseguir o máximo desempenho (aumentar localidade, diminuir comunicação, etc).
- Vantagens e inconvenientes:
 - (+) Programadores experientes produzem soluções muito eficientes para problemas específicos.
 - (–) O programador é o responsável por todos os detalhes da execução (*debugging* pode ser deveras penoso).
 - (–) Pouco portátil entre diferentes arquitecturas.

Computação Paralela

- De uma forma simples, a computação paralela pode ser definida como o uso simultâneo de vários recursos computacionais de forma a reduzir o tempo necessário para resolver um determinado problema. Esses recursos computacionais podem incluir:
 - Um único computador com múltiplos processadores.
 - Um número arbitrário de computadores ligados por rede.
 - A combinação de ambos.

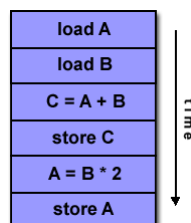
Taxonomia de Flynn

- Uma das metodologias mais conhecidas e utilizadas para classificar a arquitectura de um computador ou conjunto de computadores é a taxonomia de Flynn (1966).
 - Esta metodologia classifica a arquitectura dos computadores segundo duas dimensões independentes: **instruções** e **dados**, em que cada dimensão pode tomar apenas um de dois valores distintos: *single* ou *multiple*.

	Single Data	Multiple Data
Single Instruction	SISD <i>Single Instruction Single Data</i>	SIMD <i>Single Instruction Multiple Data</i>
Multiple Instruction	MISD <i>Multiple Instruction Single Data</i>	MIMD <i>Multiple Instruction Multiple Data</i>

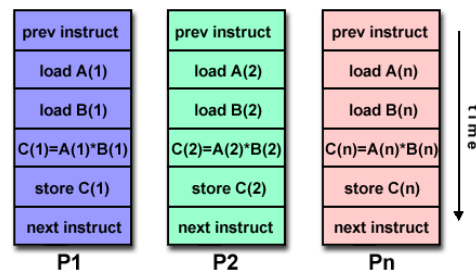
SISD – Single Instruction Single Data

- Corresponde à arquitectura dos computadores com um único processador.
 - Apenas uma instrução é processada a cada momento.
 - Apenas um fluxo de dados é processado a cada momento.
- Exemplos: PCs, *workstations* e servidores com um único processador.



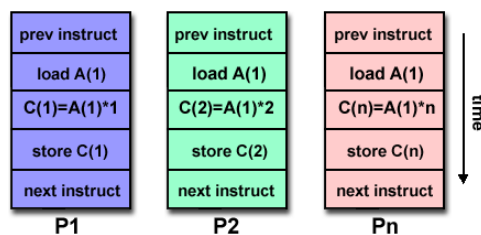
SIMD – Single Instruction Multiple Data

- Tipo de arquitetura paralela desenhada para problemas específicos caracterizados por um alto padrão de regularidade nos dados (e.g. processamento de imagem).
 - Todas as unidades de processamento executam a mesma instrução a cada momento.
 - Cada unidade de processamento pode operar sobre um fluxo de dados diferente.
- Exemplos: *processor arrays* e *pipelined vector processors*.



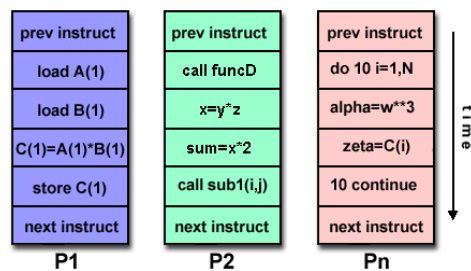
MISD – Multiple Instruction Single Data

- Tipo de arquitetura paralela desenhada para problemas específicos caracterizados por um alto padrão de regularidade funcional (e.g. processamento de sinal).
 - Constituída por uma *pipeline* de unidades de processamento independentes que operam sobre um mesmo fluxo de dados enviando os resultados duma unidade para a próxima.
 - Cada unidade de processamento executa instruções diferentes a cada momento.
- Exemplos: *systolic arrays*.



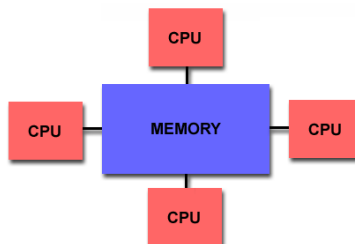
MIMD – Multiple Instruction Multiple Data

- Tipo de arquitectura paralela predominante actualmente.
 - Cada unidade de processamento executa instruções diferentes a cada momento.
 - Cada unidade de processamento pode operar sobre um fluxo de dados diferente.
- Exemplos: *multiprocessors* e *multicomputers*.



Multiprocessors

- Um *multiprocessor* é um computador em que todos os processadores partilham o acesso à memória física.
 - Os processadores executam de forma independente mas o espaço de endereçamento global é partilhado.
 - Qualquer alteração sobre uma posição de memória realizada por um determinado processador é igualmente visível por todos os restantes processadores.

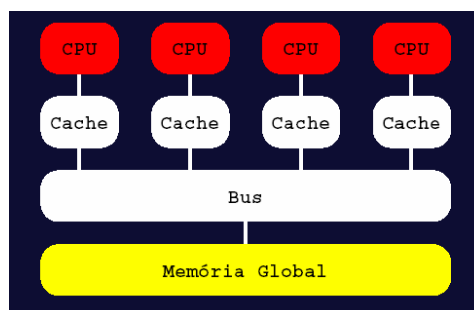


Multiprocessors

- Existem duas grandes classes de *multiprocessors*:
 - *Uniform Memory Access Multiprocessor (UMA)*
ou *Cache Coherent Uniform Memory Access Multiprocessor (CC-UMA)*
ou *Symmetrical Multiprocessor (SMP)*
ou *Centralized Multiprocessor*
 - *Non-Uniform Memory Access Multiprocessor (NUMA)*
ou *Distributed Multiprocessor*

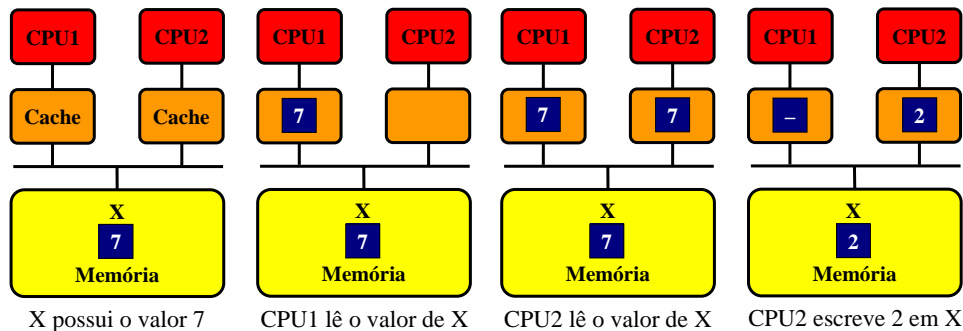
Multiprocessors

- ***Uniform Memory Access Multiprocessor (UMA)***
 - Todos os processadores têm tempos de acesso idênticos a toda a memória.
 - A coerência das *caches* é implementada pelo *hardware* (*write invalidate protocol*).



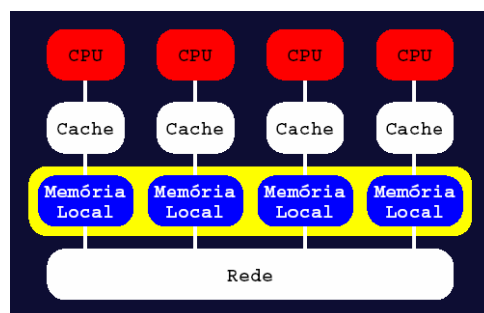
Write Invalidate Protocol

- Antes de se escrever um valor em memória, todas as cópias existentes nas *caches* dos outros processadores são invalidadas. Quando mais tarde, esses outros processadores tentam ler o valor invalidado, acontece um *cache miss* o que os obriga a actualizar o valor a partir da memória.



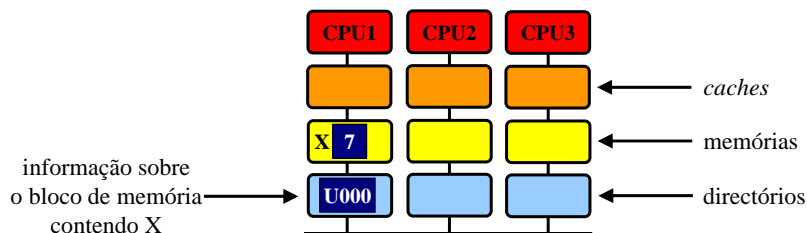
Multiprocessors

- **Non-Uniform Memory Access Multiprocessor (NUMA)**
 - Os processadores têm tempos de acesso diferentes a diferentes áreas da memória.
 - Se a coerência das *caches* for implementada pelo *hardware* (*directory-based protocol*) são também designados por *Cache Coherent NUMA (CC-NUMA)*.

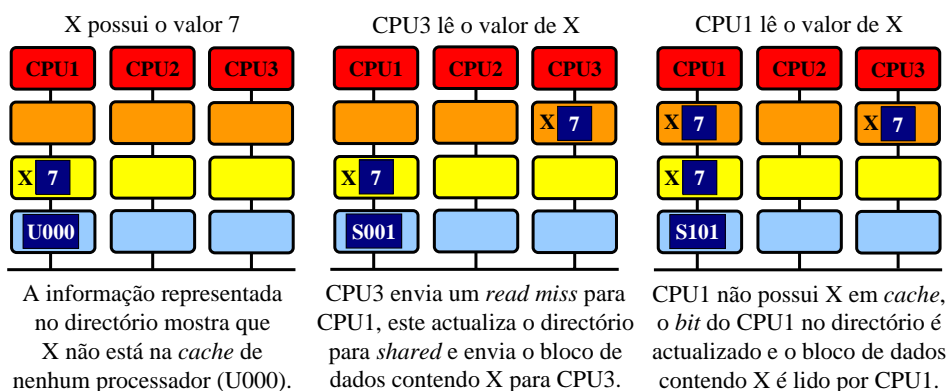


Directory-Based Protocol

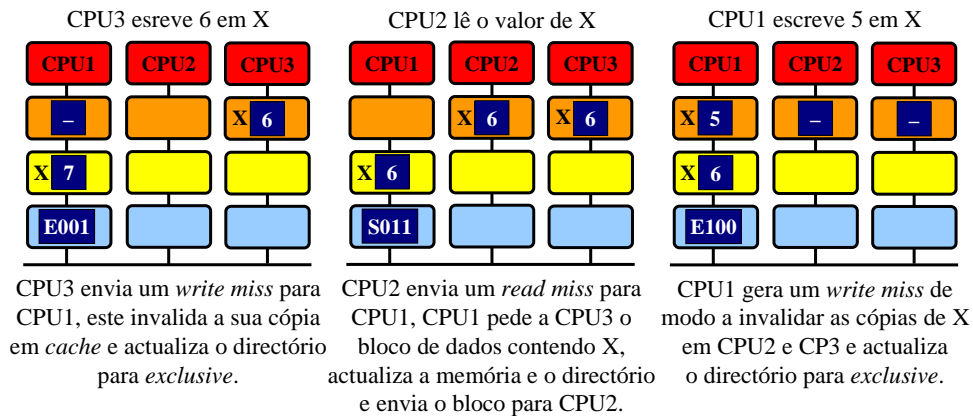
- Associado a cada processador existe um directório com informação sobre o estado dos seus blocos de memória. Cada bloco pode estar num dos seguintes estados:
 - **Uncached**: não está na *cache* de nenhum processador.
 - **Shared**: encontra-se na *cache* de um ou mais processadores e a cópia em memória está correcta.
 - **Exclusive**: encontra-se apenas na *cache* de um processador e a cópia em memória está obsoleta.



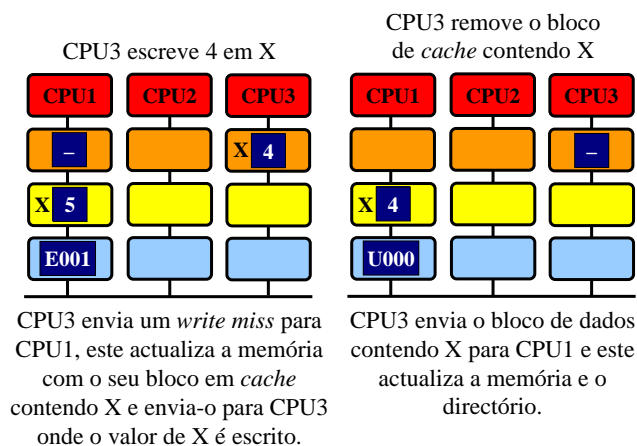
Directory-Based Protocol



Directory-Based Protocol



Directory-Based Protocol



Multiprocessors

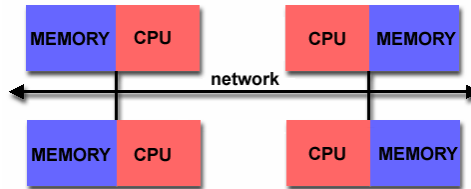
- Vantagens e inconvenientes:
 - (+) Partilha de dados entre tarefas é conseguida de forma simples, uniforme e rápida.
 - (-) Necessita de mecanismos de sincronização para obter um correcto manuseamento dos dados.
 - (-) Pouco escalável. O aumento do número de processadores aumenta a contenção no acesso à memória e torna inviável qualquer mecanismo de coerência das *caches*.
 - (-) Custo elevado. É difícil e bastante caro desenhar e produzir computadores cada vez com um maior número de processadores.

Multicomputers

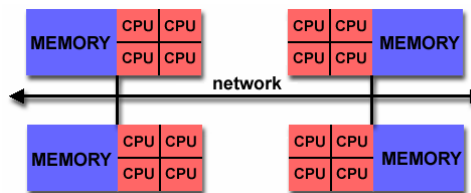
- Um *multicomputer* é um conjunto de computadores ligados por rede em que cada computador tem acesso exclusivo à sua memória física.
 - O espaço de endereçamento de cada computador não é partilhado pelos restantes computadores, ou seja, não existe o conceito de espaço de endereçamento global.
 - As alterações sobre uma posição de memória realizada por um determinado processador não são visíveis pelos processadores dos restantes computadores, ou seja, não existe o conceito de coerência das *caches*.
- Existem duas grandes classes de *multicomputers*:
 - *Distributed Multicomputer*
 - *Distributed-Shared Multicomputer*

Multicomputers

■ *Distributed Multicomputer*



■ *Distributed-Shared Multicomputer*



Multicomputers

■ Vantagens e inconvenientes:

- (+) O aumento do número de computadores aumenta proporcionalmente a memória disponível sem necessitar de mecanismos de coerência das *caches*.
- (+) Fácil escalabilidade a baixo custo. O aumento do poder de computação pode ser conseguido à custa de computadores de uso doméstico.
- (-) Necessita de mecanismos de comunicação para partilha de dados entre tarefas de diferentes computadores.
- (-) O tempo de acesso aos dados entre diferentes computadores não é uniforme e é por natureza mais lento.
- (-) Pode ser difícil converter estruturas de dados previamente existentes para memória partilhada em estruturas de dados para memória distribuída.

Arquitecturas MIMD

	<i>Multiprocessor</i>		<i>Multicomputer</i>	
	<i>CC-UMA</i>	<i>CC-NUMA</i>	<i>Distributed</i>	<i>Distributed-Shared</i>
Escalabilidade	10s de CPUs	100s de CPUs	1000s de CPUs	
Comunicação	Segmentos memória partilhada <i>Threads</i> <i>OpenMP</i> (<i>MPI</i>)		<i>MPI</i>	<i>MPI/Threads</i> <i>MPI/OpenMP</i>

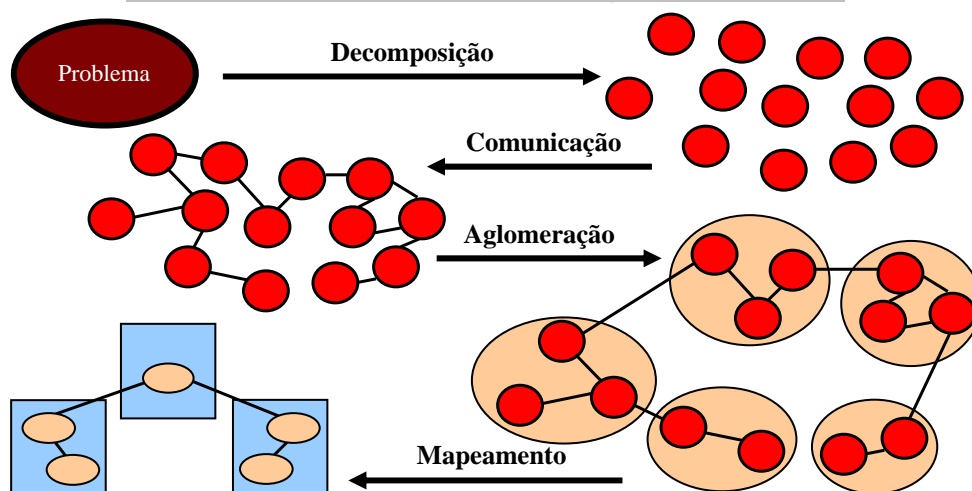
Programação Paralela

- Apesar das arquitecturas paralelas serem actualmente uma realidade, a programação paralela continua a ser uma tarefa complexa. Para além de depender da disponibilidade de ferramentas/ambientes de programação adequados para memória partilhada/distribuída, debate-se com uma série de problemas não existentes em programação sequencial:
 - **Concorrência:** identificar as partes da computação que podem ser executadas em simultâneo.
 - **Comunicação e Sincronização:** desenhar o fluxo de informação de modo a que a computação possa ser executada em simultâneo pelos diversos processadores evitando situações de *deadlock* e *race conditions*.
 - **Balanceamento de Carga e Escalonamento:** distribuir de forma equilibrada e eficiente as diferentes partes da computação pelos diversos processadores de modo a ter os processadores maioritariamente ocupados durante toda a execução.

Metodologia de Programação de Foster

- Um dos métodos mais conhecidos para desenhar algoritmos paralelos é a metodologia de Ian Foster (1996). Esta metodologia permite que o programador se concentre inicialmente nos aspectos não-dependentes da arquitectura, como sejam a concorrência e a escalabilidade, e só depois considere os aspectos dependentes da arquitectura, como sejam aumentar a localidade e diminuir a comunicação da computação.
- A metodologia de programação de Foster divide-se em 4 etapas:
 - Decomposição
 - Comunicação
 - Aglomeração
 - Mapeamento

Metodologia de Programação de Foster

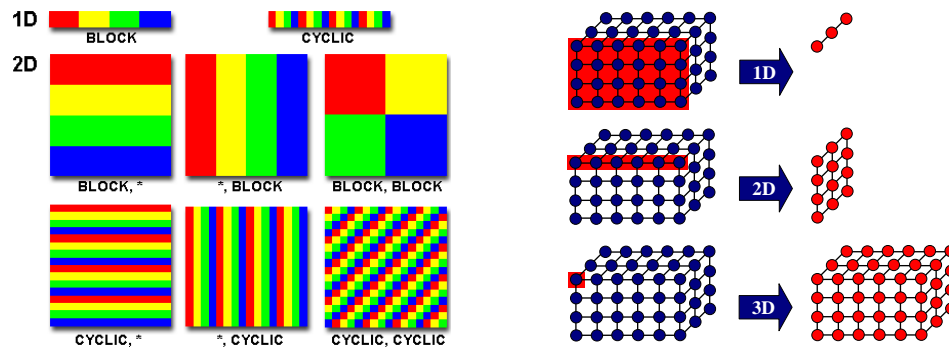


Decomposição

- Uma forma de diminuir a complexidade de um problema é conseguir dividi-lo em tarefas mais pequenas de modo a aumentar a concorrência e a localidade de referência de cada tarefa.
- Existem duas estratégias principais de decompor um problema:
 - **Decomposição do Domínio:** decompor o problema em função dos dados.
 - **Decomposição Funcional:** decompor o problema em função da computação.
- Um boa decomposição tanto divide os dados como a computação em múltiplas tarefas mais pequenas.

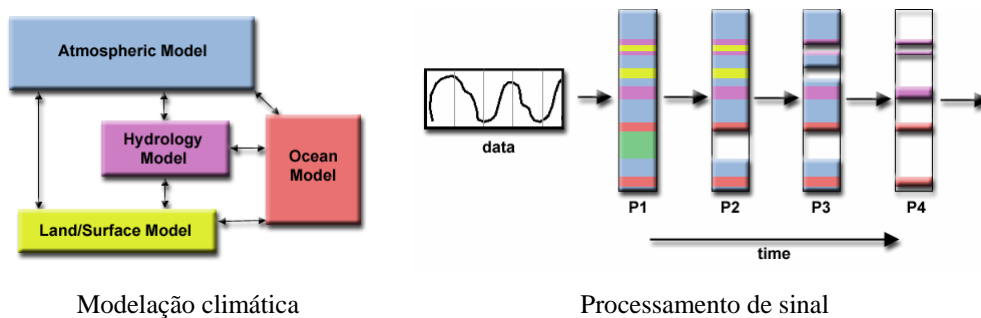
Decomposição do Domínio

- Tipo de decomposição em que primeiro se dividem os dados em partições e só depois se determina o processo de associar a computação com as partições.
- Todas as tarefas executam as mesmas operações.



Decomposição Funcional

- Tipo de decomposição em que primeiro se divide a computação em partições e só depois se determina o processo de associar os dados com cada partição.
- Diferentes tarefas executam diferentes operações.



Comunicação

- A natureza do problema e o tipo de decomposição determinam o padrão de comunicação entre as diferentes tarefas. A execução de uma tarefa pode envolver a sincronização/acesso a dados pertencentes/calculados por outras tarefas.
- Para haver cooperação entre as tarefas é necessário definir algoritmos e estruturas de dados que permitam uma eficiente troca de informação. Alguns dos principais factores que limitam essa eficiência são:
 - **Custo da Comunicação:** existe sempre um custo associado à troca de informação e enquanto as tarefas processam essa informação não contribuem para a computação.
 - **Necessidade de Sincronização:** enquanto as tarefas ficam à espera de sincronizar não contribuem para a computação.
 - **Latência** (tempo mínimo de comunicação entre dois pontos) e **Largura de Banda** (quantidade de informação comunicada por unidade de tempo): é boa prática enviar poucas mensagens grandes do que muitas mensagens pequenas.

Padrões de Comunicação

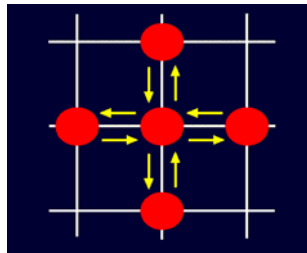
■ Comunicação Global

- Todas as tarefas podem comunicar entre si.

■ Comunicação Local

- A comunicação é restrita a tarefas vizinhas (e.g. método de Jacobi de diferenças finitas).

$$X_{i,j}^{t+1} = \frac{4X_{i,j}^t + X_{i-1,j}^t + X_{i+1,j}^t + X_{i,j-1}^t + X_{i,j+1}^t}{8}$$



Padrões de Comunicação

■ Comunicação Estruturada

- Tarefas vizinhas constituem uma estrutura regular (e.g. árvore ou rede).

■ Comunicação Não-Estruturada

- Comunicação entre tarefas constitui um grafo arbitrário.

■ Comunicação Estática

- Os parceiros de comunicação não variam durante toda a execução.

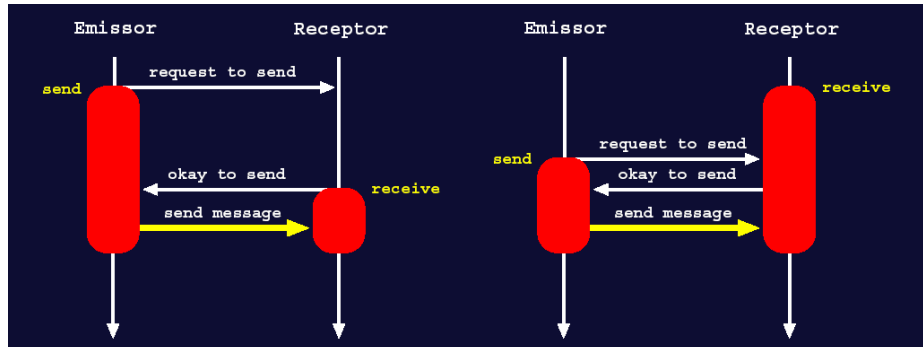
■ Comunicação Dinâmica

- A comunicação é determinada pela execução e pode ser muito variável.

Padrões de Comunicação

■ Comunicação Síncrona

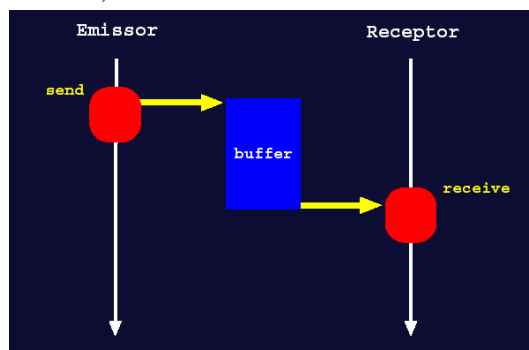
- As tarefas executam de forma coordenada e sincronizam na transferência de dados (e.g. protocolo das 3-fases ou *rendez-vous*: a comunicação apenas se concretiza quando as duas tarefas estão sincronizadas).



Padrões de Comunicação

■ Comunicação Assíncrona

- As tarefas executam de forma independente não necessitando de sincronizar para transferir dados (e.g. *buffering* de mensagens: o envio de mensagens não interfere com a execução do emissor).

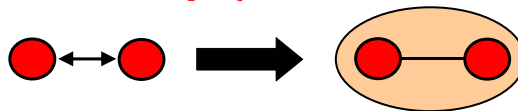


Aglomeración

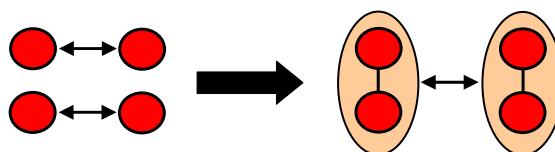
- Aglomeração é o processo de agrupar tarefas em tarefas maiores de modo a diminuir os custos de implementação do algoritmo paralelo e os custos de comunicação entre as tarefas.
- Custos de implementação do algoritmo paralelo:
 - O agrupamento em tarefas maiores permite uma maior reutilização do código do algoritmo sequencial na implementação do algoritmo paralelo.
 - No entanto, o agrupamento em tarefas maiores deve garantir a escalabilidade do algoritmo paralelo de modo a evitar posteriores alterações (e.g. optar por aglomerar as duas últimas dimensões duma matriz de dimensão $8 \times 128 \times 256$ restringe a escalabilidade a um máximo de 8 processadores).

Aglomeración

- Custos de comunicação entre as tarefas:
 - O agrupamento de tarefas elimina os custos de comunicação entre essas tarefas e aumenta a **granularidade da computação**.



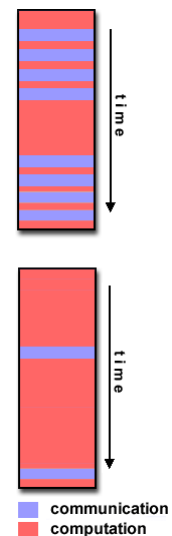
- O agrupamento de tarefas com pequenas comunicações individuais em tarefas com comunicações maiores permite aumentar a **granularidade das comunicações** e reduzir o número total de comunicações.



Granularidade

- Períodos de computação são tipicamente separados por períodos de comunicação entre as tarefas. A granularidade é a medida qualitativa do rácio entre computação e comunicação.
- O número e o tamanho das tarefas em que a computação é agrupada determina a sua granularidade. A granularidade pode ser fina, média ou grossa.

“Como agrupar a computação de modo a obter o máximo desempenho?”



Granularidade

- **Granularidade Fina**
 - A computação é agrupada num grande número de pequenas tarefas.
 - O rácio entre computação e comunicação é baixo.
 - (+) Fácil de conseguir um balanceamento de carga eficiente.
 - (-) O tempo de computação de uma tarefa nem sempre compensa os custos de criação, comunicação e sincronização.
 - (-) Difícil de se conseguir melhorar o desempenho.
- **Granularidade Grossa**
 - A computação é agrupada num pequeno número de grandes tarefas.
 - O rácio entre computação e comunicação é grande.
 - (-) Difícil de conseguir um balanceamento de carga eficiente.
 - (+) O tempo de computação compensa os custos de criação, comunicação e sincronização.
 - (+) Oportunidades para se conseguir melhorar o desempenho.

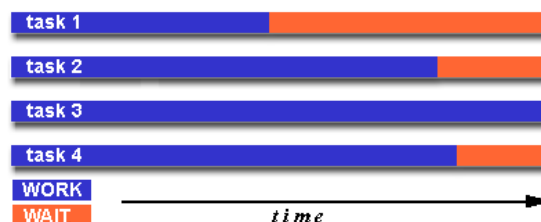
Mapeamento

- Mapeamento é o processo de atribuir tarefas a processadores de modo a maximizar a percentagem de ocupação e minimizar a comunicação entre processadores.
 - A percentagem de ocupação é óptima quando a computação é balanceada de forma igual pelos processadores, permitindo que todos comecem e terminem as suas tarefas em simultâneo. A percentagem de ocupação decresce quando um ou mais processadores ficam suspensos enquanto os restantes continuam ocupados.
 - A comunicação entre processadores é menor quando tarefas que comunicam entre si são atribuídas ao mesmo processador. No entanto, este mapeamento nem sempre é compatível com o objectivo de maximizar a percentagem de ocupação.

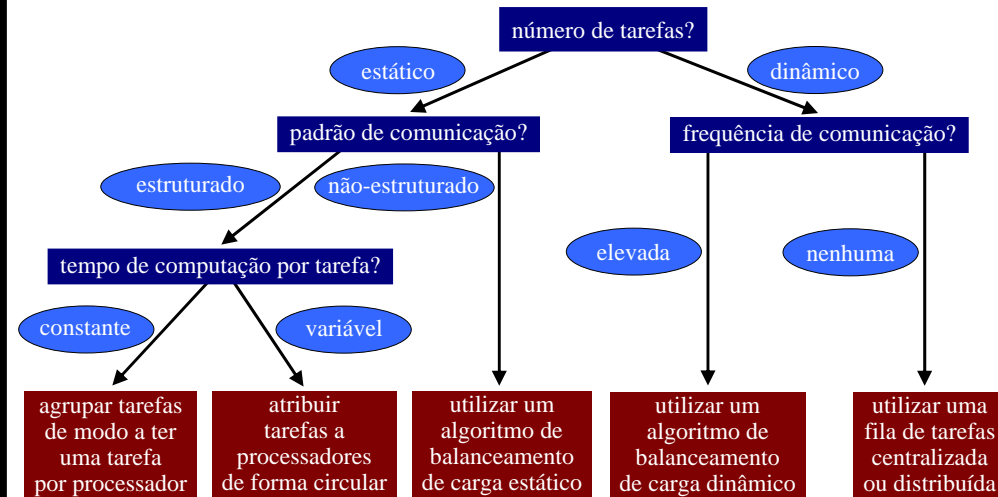
“Como conseguir o melhor compromisso entre maximizar ocupação e minimizar comunicação?”

Balanceamento de Carga

- O balanceamento de carga refere-se à capacidade de distribuir tarefas pelos processadores de modo a que todos os processadores estejam ocupados todo o tempo. O balanceamento de carga pode ser visto como uma função de minimização do tempo em que os processadores não estão ocupados.
- O balanceamento de carga pode ser estático (em tempo de compilação) ou dinâmico (em tempo de execução).



Balanceamento de Carga



Factores Limitativos do Desempenho

- **Código Sequencial:** existem partes do código que são inerentemente sequenciais (e.g. iniciar/terminar a computação).
- **Concorrência:** o número de tarefas pode ser escasso e/ou de difícil definição.
- **Comunicação:** existe sempre um custo associado à troca de informação e enquanto as tarefas processam essa informação não contribuem para a computação.
- **Sincronização:** a partilha de dados entre as várias tarefas pode levar a problemas de contenção no acesso à memória e enquanto as tarefas ficam à espera de sincronizar não contribuem para a computação.
- **Granularidade:** o número e o tamanho das tarefas é importante porque o tempo que demoram a ser executadas tem de compensar os custos da execução em paralelo (e.g. custos de criação, comunicação e sincronização).
- **Balanceamento de Carga:** ter os processadores maioritariamente ocupados durante toda a execução é decisivo para o desempenho global do sistema.

Principais Modelos de Programação Paralela

■ Programação em Memória Partilhada

- Programação usando processos ou *threads*.
- Decomposição do domínio ou funcional com granularidade fina, média ou grossa.
- Comunicação através de memória partilhada.
- Sincronização através de mecanismos de exclusão mútua.

■ Programação em Memória Distribuída

- Programação usando troca de mensagens.
- Decomposição do domínio com granularidade grossa.
- Comunicação e sincronização por troca de mensagens.

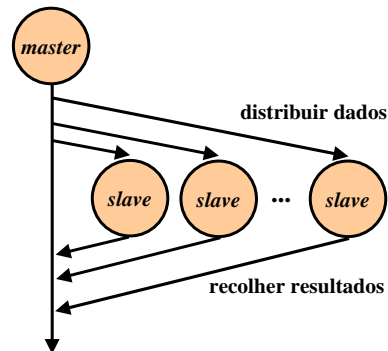
Principais Paradigmas de Programação Paralela

- Apesar da diversidade de problemas aos quais podemos aplicar a programação paralela, o desenvolvimento de algoritmos paralelos pode ser classificado num conjunto relativamente pequeno de diferentes paradigmas, em que cada paradigma representa uma classe de algoritmos que possuem o mesmo tipo de controle:
 - *Master/Slave*
 - *Single Program Multiple Data (SPMD)*
 - *Data Pipelining*
 - *Divide and Conquer*
 - *Speculative Parallelism*
- A escolha do paradigma a aplicar a um dado problema é determinado pelo:
 - Tipo de paralelismo inerente ao problema: decomposição do domínio ou funcional.
 - Tipo de recursos computacionais disponíveis: nível de granularidade que pode ser eficientemente suportada pelo sistema.

Master/Slave

- Este paradigma divide a computação em duas entidades distintas: o processo *master* e o conjunto dos processos *slaves*:

- O *master* é o responsável por decompor o problema em tarefas, distribuir as tarefas pelos *slaves* e recolher os resultados parciais dos *slaves* de modo a calcular o resultado final.
- O ciclo de execução dos *slaves* é muito simples: obter uma tarefa do *master*, processar a tarefa e enviar o resultado de volta para o *master*.

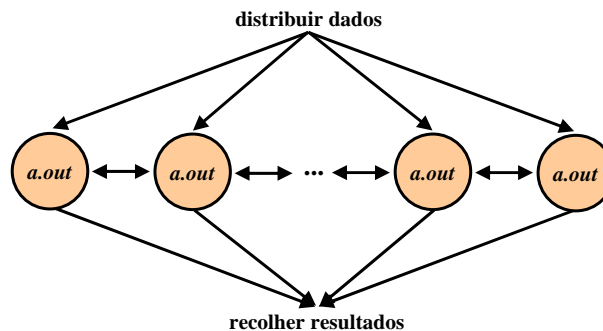


Master/Slave

- O balanceamento de carga pode ser estático ou dinâmico:
 - É estático quando a divisão de tarefas é feita no início da computação. O balanceamento estático permite que o *master* também participe na computação.
 - É dinâmico quando o número de tarefas excede o número de processadores ou quando o número de tarefas ou o tempo de execução das tarefas é desconhecido no início da computação.
- Como só existe comunicação entre o *master* e os *slaves*, este paradigma consegue bons desempenhos e um elevado grau de escalabilidade.
 - No entanto, o controle centralizado no *master* pode ser um problema quando o número de *slaves* é elevado. Nesses casos é possível aumentar a escalabilidade do paradigma considerando vários *masters* em que cada um controla um grupo diferente de *slaves*.

Single Program Multiple Data (SPMD)

- Neste paradigma todos os processos executam o mesmo programa (executável) mas sobre diferentes partes dos dados. Este paradigma é também conhecido como:
 - *Geometric Parallelism*
 - *Data Parallelism*



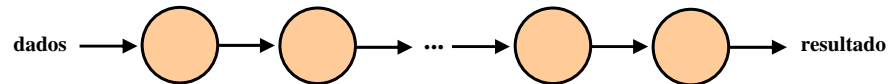
Single Program Multiple Data (SPMD)

- Tipicamente, os dados são bem distribuídos (mesma quantidade e regularidade) e o padrão de comunicação é bem definido (estruturado, estático e local):
 - Os dados ou são lidos individualmente por cada processo ou um dos processos é o responsável por ler todos os dados e depois distribui-los pelos restantes processos.
 - Os processos comunicam quase sempre com processos vizinhos e apenas esporadicamente existem pontos de sincronização global.
- Como os dados são bem distribuídos e o padrão de comunicação é bem definido, este paradigma consegue bons desempenhos e um elevado grau de escalabilidade.
 - No entanto, este paradigma é muito sensível a falhas. A perda de um processador causa necessariamente o encravamento da computação no próximo ponto de sincronização global.

Data Pipelining

- Este paradigma utiliza uma decomposição funcional do problema em que cada processo executa apenas uma parte do algoritmo total. Este paradigma é também conhecido como:

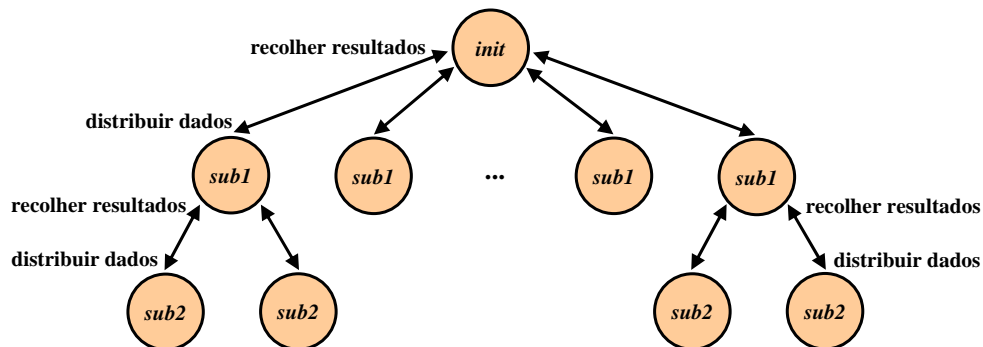
- *Data Flow Parallelism*



- O padrão de comunicação é bem definido e bastante simples:
 - Os processos são organizados em sequência (*pipeline*) e cada processo só troca informação com o processo seguinte.
 - Toda a comunicação pode ser completamente assíncrona.
- O bom desempenho deste paradigma é directamente dependente da capacidade de balancear a carga entre as diferentes etapas da *pipeline*.

Divide and Conquer

- Este paradigma utiliza uma divisão recursiva do problema inicial em sub-problemas independentes (instâncias mais pequenas do problema inicial) cujos resultados são depois combinados para obter o resultado final.



Divide and Conquer

- A computação fica organizada numa espécie de árvore virtual:
 - Os processos nos nós folha processam as subtarefas.
 - Os restantes processos são responsáveis por criar as subtarefas e por agregar os seus resultados parciais.
- O padrão de comunicação é bem definido e bastante simples:
 - Como as subtarefas são totalmente independentes não é necessário qualquer tipo de comunicação durante o processamento das mesmas.
 - Apenas existe comunicação entre o processo que cria as subtarefas e os processos que as processam.
- No entanto, o processo de divisão em tarefas e de agregação de resultados também pode ser realizado em paralelo, o que requer comunicação entre os processos:
 - As tarefas podem ser colocadas numa fila de tarefas única e centralizada ou podem ser distribuídas por diferentes filas de tarefas associadas à resolução de cada sub-problema.

Speculative Parallelism

- É utilizado quando as dependências entre os dados são tão complexas que tornam difícil explorar paralelismo usando os paradigmas anteriores.
- Este paradigma introduz paralelismo nos problemas através da execução de computações especulativas:
 - A ideia é antecipar a execução de computações relacionadas com a computação corrente na assumpção otimista de que essas computações serão necessariamente realizadas posteriormente.
 - Quando isso não acontece, pode ser necessário repor partes do estado da computação de modo a não violar a consistência do problema em resolução.
- Uma outra aplicação deste paradigma é quando se utiliza simultaneamente diversos algoritmos para resolver um determinado problema e se escolhe aquele que primeiro obtiver uma solução.

Principais Paradigmas de Programação Paralela

	Decomposição	Distribuição
<i>Master/Slave</i>	estática	dinâmica
<i>Single Program Multiple Data (SPMD)</i>	estática	estática/dinâmica
<i>Data Pipelining</i>	estática	estática
<i>Divide and Conquer</i>	dinâmica	dinâmica
<i>Speculative Parallelism</i>	dinâmica	dinâmica