



Computação Paralela: Introdução

Tipos de Paralelismo

Paralelismo por *Bit*

Paralelismo por Pipelining

Paralelismo por *Functional Units* (FU)

Paralelismo por Processos ou *Threads*

Processadores *Multicore*

Arquitetura

Conceitos Básicos

Organização de Memória

Execução

Arquiteturas

Single Instruction, Single Data

Single Instruction, Multiple Data

Multiple Instruction, Multiple Data

Implementação

Modelos

Modelo Máquina

Modelo Arquitetural

Modelo Computacional

Modelo de Programação

Paralelização de Programas

Padrões

Fork-Join

Parbegin-Parend

SPMD e SIMD

Master-Slave e Master-Worker

Client-Server

Pipelining

Producer-Consumer

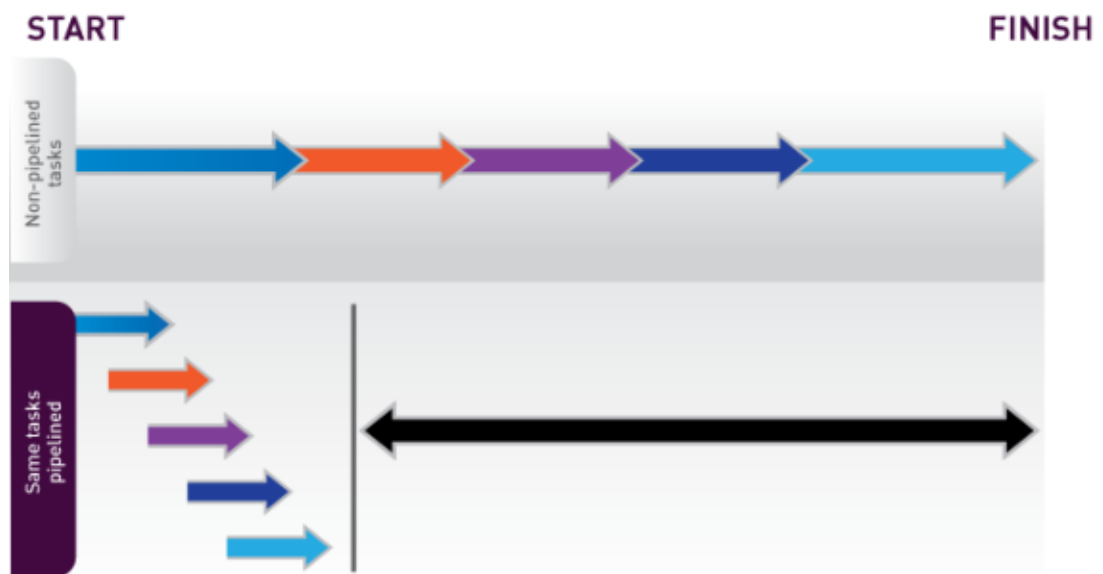
Tipos de Paralelismo

Paralelismo por *Bit*

- Relaciona-se com o aumento na capacidade de processamento de dados;
- **Mais bits permitem efetuar operações mais complexas de uma forma mais imediata.**

Paralelismo por Pipelining

- O pipeline pode ser comparado a uma linha de montagem:
 - Cada fase da linha poderá operar em paralelo caso não existam dependências entre as instruções;
 - Um *pipeline* eficiente possui instruções com durações semelhantes;
 - O *throughput* de um *pipeline* é o número de instruções finalizadas por unidade de tempo.



Paralelismo por *Functional Units* (FU)

- Existem várias unidades funcionais (ALU e FPU) independentes;
 - Estas unidades executam tarefas independente e paralelamente;
 - O número de FU a utilizar paralelamente poderá ser limitado consoante a dependência dos dados;
 - A sua utilização irá aumentar significativamente a complexidade do circuito.

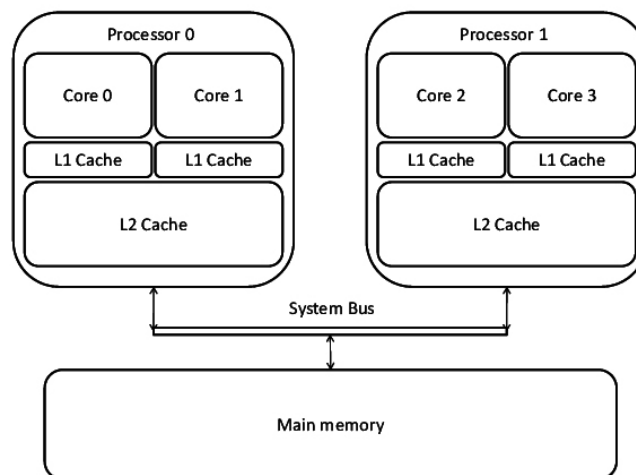
Paralelismo por Processos ou *Threads*

- Implica a utilização de sistemas *multicore/multiprocessor*;
 - Cada *core* acede à mesma memória partilhada;
 - Cada *core* executa um só fluxo;
 - Implica a utilização de técnicas de programação paralelas.

Processadores *Multicore*

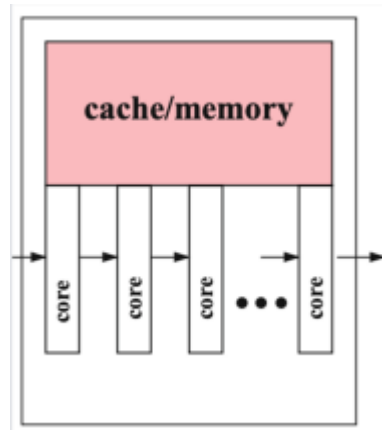
Arquitetura

- **Design hierárquico:** vários cores partilham várias *caches*, organizadas em forma de árvore, com o seu tamanho a aumentar desde as folhas até à raiz;



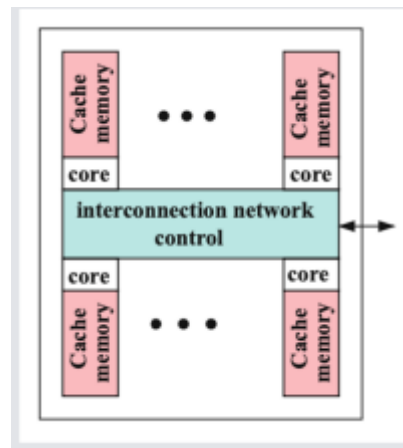
Arquitetura de sistemas *multicore*.

- **Design em *pipeline*:** os dados são processados por elemento sde processamento organizados em formato de *pipeline*; os dados são transferidos entre os *cores* de forma sucessiva até deixarem a unidade de processamento.



Arquitetura em *pipeline*.

- **Design em rede:** os *cores* de um processador e respectivas caches e memórias estão ligadas através de uma rede sob a qual ocorre a transferência de dados.



Conceitos Básicos

- A **decomposição da aplicação em *tasks*** é essencial para o design de um sistema paralelo;
 - O tamanho de cada *task* designa-se por **granularidade**;
 - As *tasks* são programadas através de processos ou *threads*, atribuídos posteriormente a diferentes *cores*;
 - **Scheduling.**

Organização de Memória

- As *tasks* partilham “conhecimento” de maneiras distintas:

- **Memória Partilhada:**
 - Pode ser acedida por todos os *cores*;
 - É feita através de R/W em variáveis partilhadas;
 - O acesso a esta memória tem de ser sincronizada.
- **Memória Distribuída:**
 - Cada processo ou *thread* tem a sua memória privada;
 - A partilha de informação ocorre com a troca de mensagens.

Execução

- O tempo de execução de um programa paralelo deverá ser menor que a execução das mesmas tarefas de forma sequencial;
- Existirão sempre *idle times* durante uma execução paralela:
 - O CPU está à espera do output de um evento;
 - Um **load balancing** equilibrado tenta mitigar os tempos de espera de um programa paralelo: distribuição de tarefas de maneira igual por todos os *cores*.

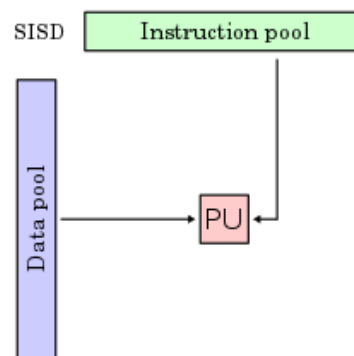
Arquiteturas

Single Instruction, Single Data

A sequential computer which exploits no parallelism in either the instruction or data streams. Single control unit (CU) fetches a single instruction stream (IS) from memory. The CU then generates appropriate control signals to direct a single processing element (PE) to operate on a single data stream (DS) i.e., one operation at a time

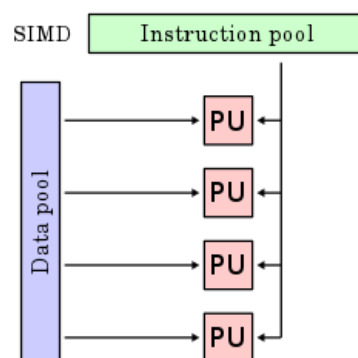
- Computador sequencial que não explora o paralelismo ao nível das instruções nem dos *stream* de dados;
- Uma única unidade de controlo (*control unit*, CU) carrega uma única instrução de memória;
- A CU gera os sinais de controlo apropriados para que uma única unidade de processamento (*processing element*, PE) opere sobre sobre uma único fluxo de

dados (*data stream*, DS), i.e uma operação de cada vez.

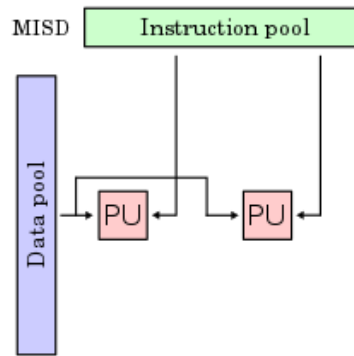


Single Instruction, Multiple Data

- Uma única instrução é simultaneamente aplicada a vários DS;
- As instruções podem ser executadas sequencialmente (e.g. via *pipelining* ou em paralelo por múltiplas FU).
- Multiple Instruction, Single Data

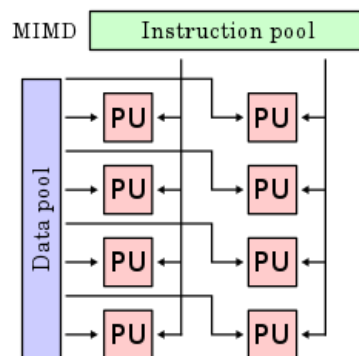


- Várias instruções operam sobre um único fluxo de dados;
- Sistemas deste tipo devem concordar no *output*/resultado;
- Arquitetura pouco comum, utilizada para tolerância de falhas.



Multiple Instruction, Multiple Data

- Vários CPU autónomos executam, simultaneamente, diferentes instruções em DS diferentes;
- Estas arquiteturas tipicamente incluem **sistemas distribuídos**.



Implementação

Modelos

- Podemos diferenciar os modelos para computação paralela de acordo com o seu nível de abstração.

Modelo Máquina

- Descreve o *hardware* e o SO (e.g. registos e *buffers*);
- Linguagens *assembly* usam estes tipos de modelos.

Modelo Arquitetural

- Descreve ligações entre processos, organização e sincronização de memória, bem como se as instruções são executadas por SIMD ou MIMD.

Modelo Computacional

- Fornece funções de custo para o tempo necessário para a execução do algoritmo;
- Modelo analítico para o desenho e avaliação de um algoritmo paralelo.

Modelo de Programação

- Representa a visão de um programador de um determinado sistema paralelo;
- **Deverá fornecer um mecanismo para o programador especificar um sistema paralelo:** especificar que operações são executadas em paralelo.
 - Como uma sequência de instruções aritméticas ou lógicas;
 - Conjuntos de *statements*;
 - Uma função ou método que contem diversos *statements*.

Paralelização de Programas

- Necessário considerar as estruturas de controlo e dependências de dados;
- **Garantir que a versão paralela do programa retorna o mesmo resultado para todos os inputs possíveis;**

Padrões

- Criação de processos e *threads*:
 - **Estaticamente:** número fixo de processos ou *threads* é criado quando o programa é iniciado, e finalizados quando o programa finaliza;
 - **Dinamicamente:** estes elementos poderão ser criados de forma arbitrária durante a execução do programa.

Fork-Join

- Uma thread T cria um conjunto de *threads* filhas T_1, \dots, T_n :
 - Trabalham em paralelo e executam uma ou mais instruções/funções.

- T pode executar a mesma parte ou outra qualquer do programa;
- O comando `join` é utilizado para T esperar pelo resultado de T_1, \dots, T_n .

Parbegin-Parend

- Na construção, são definidas uma série de instruções que deverão ser executadas em paralelo;
- Quando um programa chega a um comando `parbegin-parend`, um conjunto de *threads* é criado e as *statements* são atribuídas às mesmas;
- O restante prorama só é executado depois de todas as *threads* finalizarem as operações.

SPMD e SIMD

- Utilizam um número fixo de *threads*;
- SIMD: a mesma ionstrução é executada de forma síncrona por diferentes *threads* em diferentes conjuntos de dados;
- SPMD: diversas threads trabalham de forma assíncrona entre sí, podendo executar diferentes partes do programa.

Master-Slave e Master-Worker

- A *thread master* é responsável pela execução do programa (cria *workers* em diferentes partes do programa para realizar operações paralelas).

Client-Server

- Utilizado em sistemas distribuídos;
- Vários sistemas clientes comunicam com um main-frame que fornece acesso distribuído a uma base de dados;
- Devem ser criadas diversas *threads client*, que geram algum tipo de pedido ao *server*.

Pipelining

- Feature de *hardware* para computação paralela;

- As *threads* T_1, \dots, T_n são organizadas de forma a que uma *thread* T_i receba como *input* o *output* da *thread* T_{i-1} ;
- As *threads* utilizadas no *pipelining* poderão ser utilizadas de forma paralela.

Producer-Consumer

- Os *Producers* geram dados que serão consumidos e processados pelos *Consumers*;
- É utilizado um *buffer* partilhado;
- O *producer* só conseguem adicionar dados ao *buffer* se este não estiver cheio;
- O *consumer* só consegue processar dados de um *buffer* não vazio.