Capítulo

2

Fundamentos das Arquiteturas para Processamento Paralelo e Distribuído

Philippe O. A. Navaux, César A. F. De Rose, Laércio L. Pilla

1. Introdução

Desde os primórdios da era da computação, já em trabalhos de Von Neumann (1945), as Arquiteturas Paralelas foram apresentadas como formas de obter uma maior capacidade de processamento para execução de tarefas com altas cargas de processamento.

Esse aumento na capacidade de processamento dos computadores só podia ser obtido através de processadores mais velozes ou através do aumento do número de processadores empregados em conjunto. Nos anos de 50 e 60, as tecnologias de fabricação de máquinas monoprocessadoras eram suficientes para atender à demanda de processamento daquela época. A partir dos anos 70, as necessidades crescentes de capacidade de processamento não eram mais atendidas pelas evoluções na tecnologia dos monoprocessadores, tornando-se necessária a utilização de técnicas de concorrência para alcançar o desempenho requerido.

Como o aumento da velocidade dos processadores esbarrava no custo e no limite da capacidade tecnológica para obtenção de circuitos rápidos, a solução tendeu para o emprego de vários processadores trabalhando em conjunto na obtenção de uma maior capacidade de processamento. É quando surge o termo Processamento Paralelo para designar técnicas diferentes de concorrência que atendem a essas necessidades.

O desenvolvimento da área de Processamento Paralelo levou ao aproveitamento destas técnicas no projeto de chips de processadores, microprocessadores, e também em máquinas especializadas em realizar grandes quantidades de operações por segundo, conhecidas como Supercomputadores. Estas últimas proporcionaram um maior desempenho na resolução de problemas, sendo, por isso, essa área conhecida também como Processamento de Alto Desempenho - PAD (em inglês, *High Performance Computing* - HPC).

1.1. Arquitetura de Processadores com Paralelismo – Chips

Uma das áreas de aplicação das técnicas de paralelismo é no projeto da arquitetura do(s) processador(es) dentro dos chips.

A evolução na integração de uma maior quantidade de transistores num chip, chegando a bilhões de transistores, permitiu que gradativamente fossem projetadas arquiteturas de processadores com mais unidades funcionais, mais memórias caches e maiores facilidades de barramentos e registradores. Aconteceu uma migração/ miniaturização das arquiteturas de máquinas para dentro dos chips.

Nas próximas seções será apresentada a evolução que aconteceu no projeto dos processadores nos chips, passando pelo emprego do pipeline, depois pelo superescalar, multithread, multi-core e por fim pelo atual many-core.

1.2. Máquinas para Processamento de Alto Desempenho

Apesar do crescente aumento de desempenho dos PC's encontrados no mercado (máquinas denominadas convencionais) existem usuários especiais que executam aplicações que precisam de ainda maior desempenho. Ex:

- Previsão do tempo
- Procura de petróleo
- Simulações físicas
- Matemática computacional

Se executadas em máquinas convencionais, estas aplicações precisariam de várias semanas ou até meses para executar. Em alguns casos extremos, nem executariam por causa de falta de memória.

Ex: Compro o melhor PC que encontro no mercado, utilizo um simulador para prever o tempo daqui a três (3) dias... A previsão pode só ficar pronta depois de 1 semana (7 dias) !!!

Processamento de Alto Desempenho (PAD) é uma área da computação que se preocupa com estes problemas complexos demais para máquinas convencionais. Estes usuários só têm duas alternativas:

- Simplificam o seu modelo e acabam tendo resultados menos precisos aumentando a margem de erro (Ex: a simulação de tempo indica uma alta probabilidade de chuva e não chove).
- Executam a sua aplicação em máquinas mais poderosas do que as convencionais, denominadas arquiteturas especiais ou arquiteturas paralelas.

As **arquiteturas especiais** obtêm um melhor desempenho replicando o número de unidades ativas (normalmente processadores).

Infelizmente um aumento do número de processadores na arquitetura acaba invariavelmente complicando ainda mais os problemas já encontrados em arquiteturas convencionais e criando novos complicadores.

- A dificuldade que se tinha para alimentar um processador com dados fica ainda maior para vários processadores.

- A programação de máquinas especiais é mais complicada, pois o problema tem que ser particionado (quebrado em partes e distribuído) entre as várias unidades ativas para que execute mais rápido.

O tipo de processamento que ocorre nestas máquinas é denominado **processamento paralelo** pelo fato de várias unidades ativas atuarem em paralelo no mesmo problema com o objetivo de reduzir o tempo total de execução

1.3. Tipos de Concorrência numa Arquitetura

As principais formas de concorrência encontradas numa arquitetura são a temporal e a de recursos. Numa concorrência temporal, existe uma sobreposição das execuções no tempo, criando, dessa forma, um ganho no desempenho final do processamento. Por outro lado, na concorrência de recursos, também chamada de espacial, a arquitetura possui vários processadores ou elementos de processamento que trabalham em paralelo na execução das tarefas que compõem o processamento. Nessa última concorrência, o ganho obtido no desempenho é fruto da quantidade de unidades de processamento que trabalham em conjunto. Esses dois tipos de concorrência resultam em três tipos de arquiteturas principais que hoje, juntas, representam a quase totalidade das máquinas existentes:

- **Concorrência temporal**, que resulta nas arquiteturas pipeline, onde existe uma sobreposição na execução temporal dos vários estágios que compõem uma instrução;
- **Concorrência de recursos síncrona**, que resulta nas arquiteturas SIMD, também conhecidas como arquiteturas matriciais (*array*), nas quais a concorrência existe entre elementos de processamento que executam em paralelo, de forma simultânea, a mesma operação;
- **Concorrência de recursos assíncrona**, que resulta nas arquiteturas MIMD, em que processadores atuam em paralelo para resolver uma tarefa, porém cada um executando dentro do seu ordenamento e tempo.

Num pipeline, uma tarefa é subdividida numa sequência de subtarefas, cada uma executada por um estágio de hardware específico, que trabalha concorrentemente com os outros estágios do pipeline, criando um paralelismo temporal na execução das subtarefas.

Os Processadores Matriciais (*Array Processors*), também conhecidos por SIMD, são representativos do Paralelismo Espacial Síncrono. Nessa arquitetura, uma unidade de controle distribui para vários elementos de processamento (EP) a mesma instrução que será executada em paralelo por esses EP's sobre seus próprios dados no mesmo instante de tempo.

O terceiro tipo de concorrência, o Paralelismo Espacial Assíncrono, é encontrado nos multiprocessadores, também conhecidos por arquiteturas MIMD. Nesse caso, diversos processadores trabalham em paralelo, processando suas tarefas concorrentemente de forma assíncrona para, num intervalo de tempo, concluírem a tarefa. Esses processadores geralmente são idênticos e possuem um sistema operacional

único que os gerencia. Portanto, um multiprocessador é um computador que possui vários processadores que se comunicam e cooperam para resolver uma dada tarefa.

1.4. Processamento Paralelo (PP)

Definição: várias unidades ativas colaborando na resolução de um mesmo problema.

As várias unidades ativas cooperam para resolver o mesmo problema, atacando cada uma delas uma parte do trabalho e se comunicando para a troca de resultados intermediários ou no mínimo para a divisão inicial do trabalho e para a junção final dos resultados. Exemplos de programas paralelos:

- Uma aplicação escrita em C ou Java com várias *threads.*
- Uma aplicação escrita em Java usando RMI (*Remote Method Invocation*).
- Uma aplicação escrita em C que foi quebrada em vários processos que se comunicam por *sockets.*

Um programa que não foi **preparado** para executar com várias unidades ativas (implementado com apenas um processo que não dispara múltiplas *threads*) **não** executa mais rápido em uma máquina dual!!!

O programa não é automaticamente "quebrado" pelo sistema operacional e só executa em um único processador, não se aproveitando de outros processadores que possam estar disponíveis na arquitetura (no caso da máquina dual).

Motivação para o uso de processamento paralelo

1. Desempenho: Espero reduzir o tempo de execução devido a utilização de diversas unidades ativas na resolução do problema.
2. Tolerância a falhas: Espero reduzir a probabilidade de falhas no cálculo pois cada unidade ativa calcula o mesmo problema e faço uma eleição no final.
3. Modelagem: Espero reduzir a complexidade da modelagem e consequentemente da implementação da aplicação utilizando uma linguagem que expresse paralelismo (em situações onde o problema é em sua essência paralelo).
4. Aproveitamento de recursos: Espero aproveitar melhor os recursos disponíveis na rede executando uma aplicação com múltiplos processos.

1.5. Níveis de Paralelismo

Exploração de paralelismo está presente nos diversos **níveis** de um sistema:

Grão de Paralelismo: é um conceito muito importante pois seu entendimento é fundamental para a modelagem de programas paralelos.

- Grão grosso: o trabalho a ser feito pode ser particionado em unidades de trabalho grandes. Mesmo pagando um alto custo de comunicação é grande a chance de se obter ganho de desempenho delegando estas unidades de trabalho para outras unidades ativas (o custo do envio é compensado pelo ganho de tempo em atacar o problema com mais unidades).

- Grão médio: o trabalho a ser feito só pode ser particionado em unidades de trabalho médio. Em caso de um alto custo de comunicação pode ser difícil se obter ganho de desempenho delegando estas unidades de trabalho para outras unidades ativas (o custo do envio não é necessariamente compensado pelo ganho de tempo em atacar o problema com mais unidades).
- Grão fino: o trabalho a ser feito só pode ser particionado em unidades de trabalho pequenas. Em caso de um alto custo de comunicação não vale a pena delegar estas unidades de trabalho para outras unidades ativas (o custo do envio não é compensado pelo ganho de tempo em atacar o problema com mais unidades).

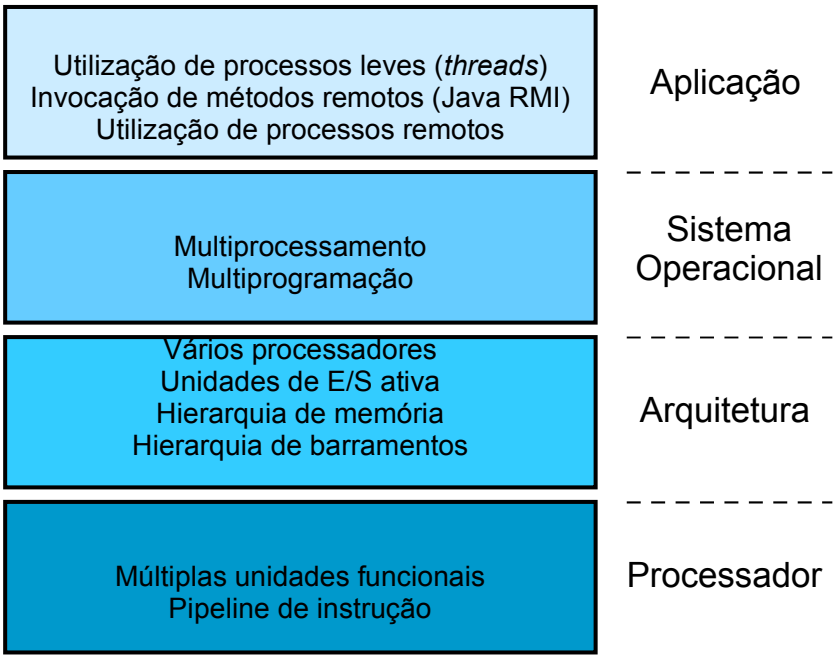


Figura 1. Diferentes níveis de paralelismo.

1.6. Processamento Paralelo x Distribuído

Processamento Paralelo x Distribuído: questões mais conceituais dependem muito de cada autor e de sua respectiva formação:

- Em ambas as áreas tenho os mesmos complicadores (custo de comunicação, distribuição dos dados, dependências) mas a motivação é diferente [De Rose 2003].
- No caso de **Processamento Paralelo** a motivação foi o ganho de desempenho e as unidades ativas estão normalmente na mesma máquina resultando em custos de comunicação menores (baixa latência).
- No caso de **Processamento Distribuído** a motivação foi a modelagem e o aproveitamento de recursos e as unidades ativas estão normalmente mais afastadas (em uma rede local ou até na Internet) resultando em custos de comunicação maiores (alta latência).

Poderia supor que se quero aumentar o desempenho de uma aplicação só precisaria executar o programa com mais unidades ativas... mas infelizmente tenho **complicadores**:

- Dependências de dados
- Distribuição dos dados
- Sincronização
- Áreas críticas

Exemplo: construção de um muro

- Um pedreiro faz o muro em 3 horas, dois pedreiros fazem em 2 horas, três pedreiros em 1 hora e meia, 4 pedreiros em duas horas (aumentou o tempo!!!)

A quantidade de trabalho a ser feita limita o número de unidades ativas que podem ser usadas de forma eficiente.

Um número muito grande de unidades ativas para uma quantidade limitada de trabalho faz com que os recursos mais se atrapalhem do que se ajudem (o que faz o tempo aumentar e não diminuir!!!).

A incidência de muitos complicadores faz com que o ganho de desempenho não seja proporcional ao acréscimo de unidades ativas utilizadas (a duplicação do número de pedreiros de 1 para dois não reduziu o tempo de execução pela metade no exemplo acima).

Os complicadores no caso da construção do muro são os seguintes:

- O muro só pode ser feito de baixo para cima (dependência de dados).
- Os tijolos têm que ser distribuídos entre os pedreiros (distribuição dos dados).
- Um pedreiro não pode levantar o muro do seu lado muito na frente dos outros pedreiros (sincronização – ritmo de subida do muro é dado pelo pedreiro mais lento).
- Se só existir um carrinho com cimento este será disputado por todos os pedreiros que farão uma fila para acessar o cimento. Enquanto um pedreiro acessa o cimento os outros perderão tempo esperando nessa fila (áreas críticas).

1.7. Medidas Básicas de Desempenho

Índices que indicam o desempenho de diferentes aspectos de um programa paralelo.

- Desempenho da aplicação
- Desempenho da rede de interconexão

1.7.1. Desempenho da aplicação

Speed-Up (fator de aceleração): Indica quantas vezes o programa paralelo ficou mais rápido que a versão sequencial. É calculado pela razão entre o melhor tempo sequencial e o tempo da versão paralela.

$$SU_p(w) = \frac{T(w)}{T_p(w)}$$

Onde ***p*** é o número de unidades ativas utilizadas e ***w*** o trabalho que foi calculado.

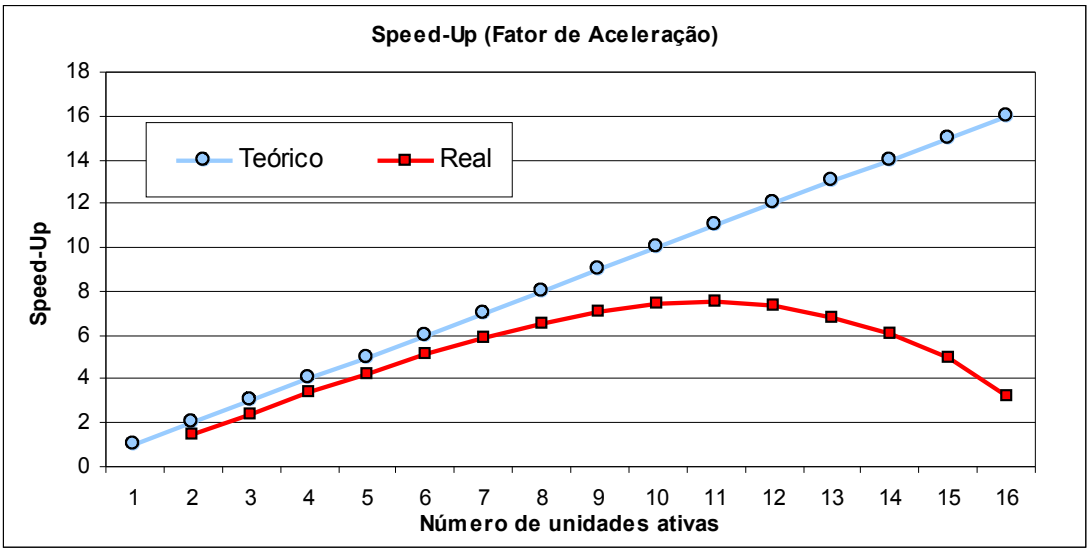


Figura 2. Comparação entre fator de aceleração teórico e obtido.

Se ***SU*** **> 1** a versão paralela reduziu o tempo de execução (ficou mais rápido que a sequencial) e se ***SU*** **< 1** a versão paralela aumentou o tempo de execução (ficou mais lenta que a sequencial).

Cada aplicação tem a sua curva (mais ou menos acentuada) dependendo do trabalho e da incidência de complicadores (aplicação mais ou menos amarrada).

Toda aplicação tem um número de unidades ativas ideal para a obtenção do melhor desempenho (*sweetspot*) não sendo verdade que quanto mais unidades ativas melhor.

Eficiência: Indica como foi a taxa de **utilização média** das unidades ativas utilizadas. Mostra se os recursos foram bem aproveitados. É calculado pela razão entre o *Speed-Up* e o número de *unidades ativas* utilizadas.

$$E_p(w) = \frac{SU_p(w)}{p}$$

Onde ***p*** é o número de unidades ativas utilizadas e ***w*** o trabalho que foi calculado.

O ideal seria que cada unidade ativa tivesse ficado 100% do tempo ativa (linha superior). Normalmente as unidades ativas ficam parte de seu tempo esperando por resultados de vizinhos o que reduz sua taxa de utilização.

A melhor taxa de utilização média não significa o menor tempo de execução (nas Figuras 2 e 3, o menor tempo de execução ocorreu com 11 unidades ativas e a melhor taxa de utilização média com 5).

1.7.2. Desempenho da rede de interconexão

Latência: É o tempo necessário para enviar uma mensagem através da rede de interconexão.

Unidade: medida de tempo, Ex: 4 microssegundos (4 µs).

Inclui o tempo de empacotamento e desempacotamento dos dados mais o tempo de envio propriamente dito.

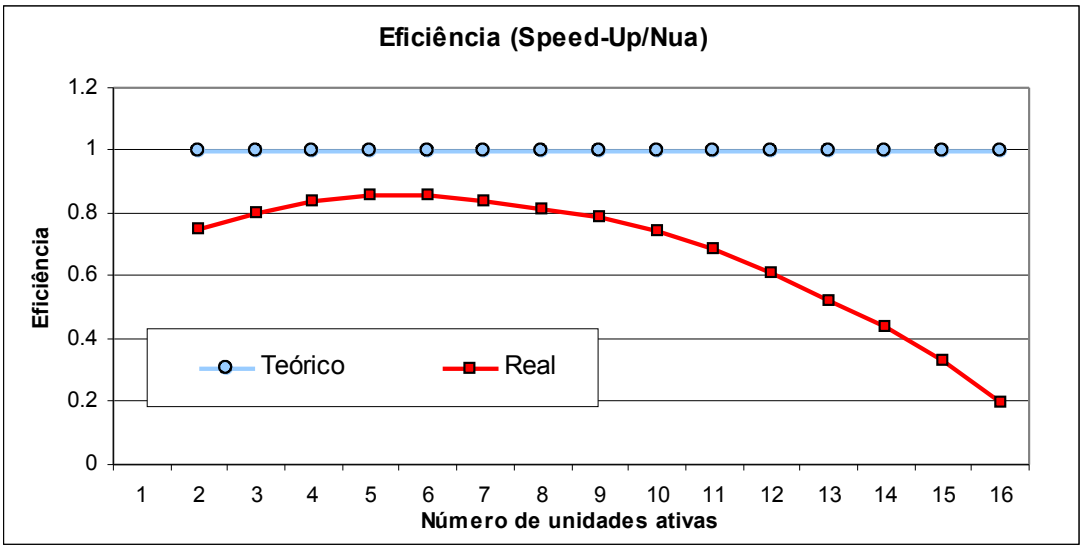


Figura 3. Comparação entre eficiência teórica e obtida.

A latência aumenta a medida que a quantidade de dados a serem enviados aumenta, mas não de forma linear, pois a componente do tempo referente ao custo de empacotamento e desempacotamento não varia tanto em relação ao tamanho da mensagem como a componente de custo de envio pela rede.

Vazão: Expressa a capacidade da rede de "bombear" dados entre dois pontos.

Unidade: quantidade de dados por unidade de tempo, Ex: 10 MBytes/segundo (10 MB/s).

A vazão é afetada pela "largura" do canal de comunicação (expressa normalmente em bits) e pela frequência da transmissão dos dados (expressa em MHz) segundo a seguinte fórmula:

$$V = L * F$$

2. Paralelismo na Arquitetura de Chips

2.1. Arquiteturas empregando a Técnica Pipeline

Conforme visto no capítulo de introdução, uma das primeiras formas de concorrência empregadas para aumento do desempenho de processadores foi a técnica pipeline. Neste capítulo, veremos o histórico da evolução do emprego da técnica pipeline para o projeto de arquiteturas de máquinas e microprocessadores, sua classificação, o pipeline de instrução, o pipeline aritmético, as principais técnicas de projeto para pipelines e, por fim, as dependências entre instruções que afetam o funcionamento de um pipeline. No

próximo capítulo, serão vistas as técnicas superescalares que representam a evolução das arquiteturas pipelines apresentadas aqui.

O reconhecimento do interesse no emprego do pipeline data do início dos projetos de computadores. O uso de técnicas de pipeline iniciou pela aceleração na busca de instruções na memória. O computador IBM 7094 empregava uma memória de 72 bits de largura e 36 bits de instruções. A cada acesso à memória eram trazidas duas instruções de 36 bits.

O primeiro emprego efetivo de técnicas pipeline foi no particionamento e na execução sobreposta de instruções nas máquinas STRETCH [Block 1959] e LARC [Eckert 1959]. O STRECH empregou o particionamento na execução das instruções em dois estágios: uma fase de busca e decodificação da instrução e uma fase de execução do operador. Já a máquina LARC melhorou a execução das instruções, quebrando o processo em quatro estágios: busca de instrução, operação de endereçamento e índice, busca do operando e execução.

2.1.1. Princípios de Funcionamento de um Pipeline

O princípio de funcionamento de um pipeline é a execução de uma tarefa através da sua divisão num conjunto de subtarefas, as quais, no seu conjunto, sejam equivalentes à execução da tarefa original, empregando concorrência temporal. Isso resulta em que a tarefa continua sendo executada de forma idêntica, só que dividida por etapas, chamadas de estágios. A cada vez que a tarefa passa para um novo estágio, é possível aproveitar o anterior para uma nova tarefa, portanto aproveitando melhor o hardware existente para a execução. Esse princípio é equivalente a uma linha de produção de automóveis, na qual cada estação é responsável pela montagem de uma parte do veículo.

Portanto, num pipeline, uma tarefa é dividida em subtarefas que são executadas cada uma por um dos estágios, de forma a ter na saída da cadeia do pipeline a tarefa completa executada. A Figura 4 mostra a estrutura básica de um pipeline linear; nela, os estágios são separados por registradores cuja função é armazenar o resultado do estágio anterior para execução no próximo estágio. Dessa forma, é criado um delimitador temporal que será empregado para sincronizar os estágios.

Observa-se que, para o bom funcionamento de um pipeline e para não haver perdas, é importante que as subtarefas dos diferentes estágios tenham um tempo de processamento o mais uniforme possível. O relógio, que cadenciará o pipeline através da carga dos registradores, será estabelecido pelo tempo da subtarefa mais lenta.

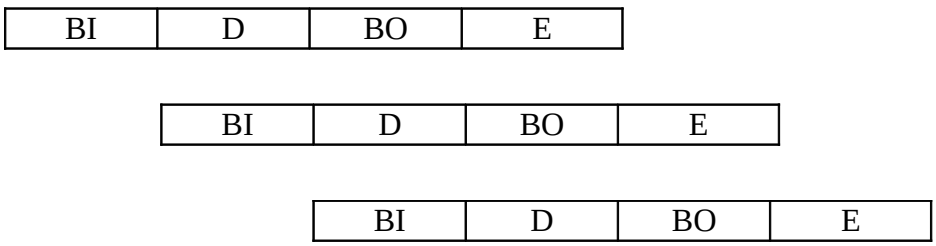


Figura 4. Estrutura de pipeline.

Conclui-se que um pipeline é uma cascata de estágios de processamento. Esses estágios são circuitos combinatórios que executarão operações lógicas e aritméticas sobre o fluxo de dados. Entre cada estágio, existem registradores que servem como barreiras lógicas e que sincronizam o pipeline. São estes que guardam os resultados intermediários de um estágio para outro. O relógio atuará simultaneamente sobre todos os registradores, cadenciando o fluxo das informações pelos estágios do pipeline.

Portanto, pipeline é uma técnica que pode ser aplicada em diversas circunstâncias, trazendo como resultado um aumento na vazão das tarefas. Foi o emprego dessa técnica nas arquiteturas de máquinas que propiciou o surgimento de computadores de alto desempenho. Por outro lado, hoje em dia, a maioria dos microprocessadores e computadores emprega alguma técnica pipeline, em maior ou menor grau, nas suas arquiteturas.

O pipeline de instruções é o mais simples e mais difundido dos dois tipos de pipeline. Consiste basicamente na execução concorrente das etapas de busca de instrução (BI), decodificação (D), busca de operando (BO), e execução (E).

Para um bom desempenho de um pipeline de instrução, é necessário que ele seja abastecido continuamente com instruções. Quando ocorre um salto no programa ou uma interrupção, diversos ciclos são perdidos para esvaziar e recarregar o pipeline.

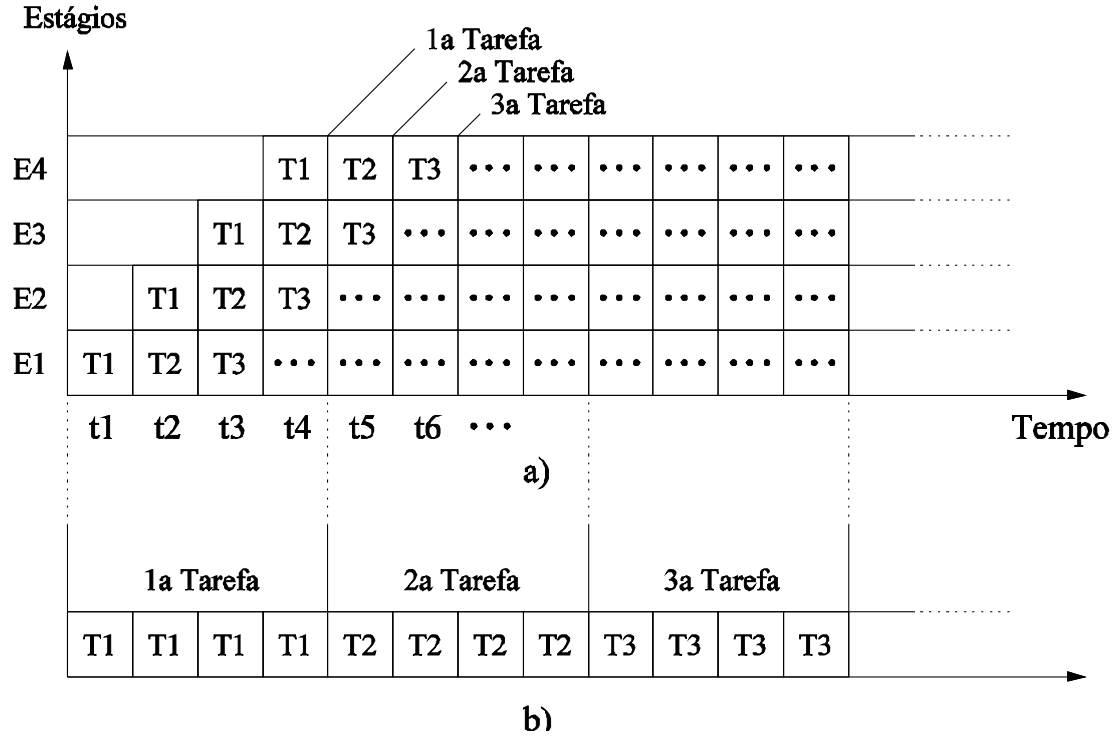


Figura 5. Exemplo de aceleração com pipeline.

Pelo exposto, verifica-se que um pipeline obtém seu máximo desempenho quando está cheio, completando uma tarefa a cada tempo T. Observando-se a Figura 5, conclui-se que, até a primeira tarefa ser concluída, ocorre um atraso de 4T, donde se obtém que um pipeline processa n tarefas em

$$(j+(n-1))T$$

Numa máquina comum, essas tarefas levariam n.j.T; portanto o ganho de desempenho (Speed-Up) G de um pipeline é dado por

$$G=\frac{(n.j.T)}{J(j-(n-1))}=\frac{(n.j)}{Tj+(n-1)}$$

2.1.2. Gerenciamento de Desvios

Um dos problemas na implementação dos pipelines é o fato de que quando existe uma instrução de desvio, ela depende de um resultado que na maioria das vezes ainda esta em execução no pipeline e portanto ainda não pronto, isto pode levar a parar a entrada de novas instruções enquanto o resultado não for calculado.

A distribuição típica da incidência das instruções pelas suas categorias fica com 60% para instruções aritméticas, 15% para instruções de armazenamento, 5% para desvios incondicionais e da ordem de 20% para os desvios condicionais [Hwang 1985].

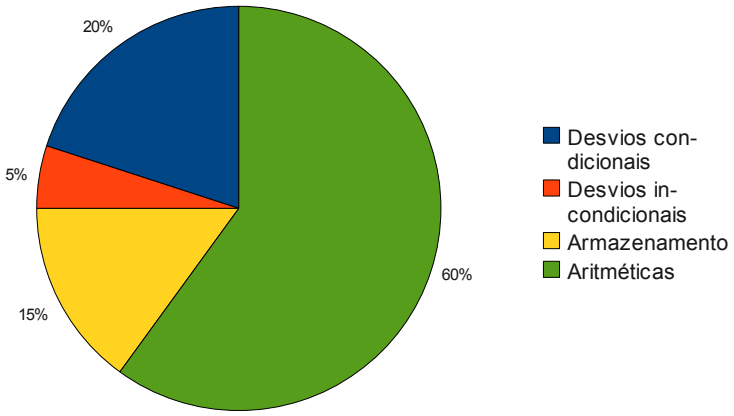


Figura 6. Distribuição típica de instruções.

Dos 20% de desvios condicionais, normalmente 8% não são tomados e, portanto, as instruções continuam sequencialmente sua execução, enquanto 12% são tomados, obrigando o cálculo do novo endereço para início da nova sequência de instruções. Essas duas situações, desvio incondicional e desvio condicional com sucesso, prejudicam grandemente o desempenho do pipeline, pois acarretam o esvaziamento deste para início da nova sequência de instruções, como ilustra a Figura 7. Além dos desvios, as interrupções também prejudicam o funcionamento de um pipeline de instruções.

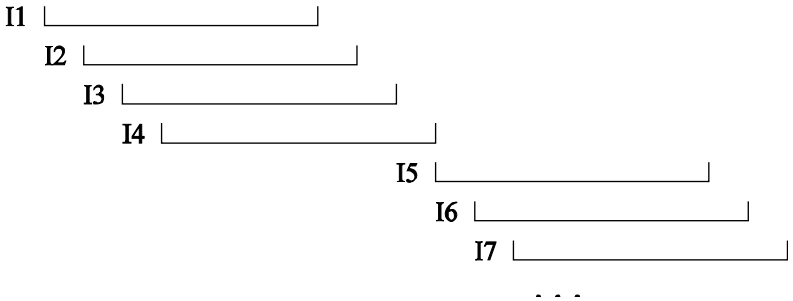


Figura 7. Efeito do esvaziamento de pipeline devido a desvios.

A impossibilidade de conhecer os resultados de um desvio condicional antes de executá-lo cria pontos de dependência conhecidos por "dependências de controle". O pipeline fica sem saber se continua executando as instruções pelo fluxo de instruções principal, ou se deve desviar e adotar o novo fluxo de instruções resultado do desvio. Esse tipo de situação gera uma dependência de controle, conhecida também por dependência procedural [Stallings 1996]. O processador só pode continuar a execução efetiva da instrução após conhecido o resultado da condição e, portanto, após ter o conhecimento de qual fluxo de instruções seguir.

Estabelecida a necessidade do desvio e, portanto, a adoção da nova sequência de instruções, é necessário descartar todas as instruções que já estão no pipeline. Para diminuir esse impacto no desempenho, foram criadas técnicas para **prever os desvios sob forma especulativa**, diminuindo a taxa de execução de sequências de instruções desnecessárias.

2.2. Arquiteturas Superescalares

Como visto anteriormente, o pipeline veio para acelerar a execução de instruções, empregando, para isso, a sobreposição na execução de etapas da instrução. Nos primeiros tempos do emprego da técnica pipeline, havia grandes diferenças de tempo entre os estágios de busca de instrução e de busca de operando na memória devido ao maior tempo necessário para acesso à memória. Esses problemas foram solucionados no final dos anos 70, na medida em que foram criadas memórias cache para as instruções e posteriormente para os dados.

Num segundo momento da evolução das técnicas de pipeline, em meados dos anos 80, surgiram as instruções RISC (*Reduced Instruction Set Computer*) em oposição ao emprego das instruções CISC (*Complex Instruction Set Computer*). Essa evolução visava a obter um conjunto de instruções com tempos de execução de um ciclo e, portanto, permitir que os pipelines conseguissem obter desempenhos melhores sem serem prejudicados por estágios de execução com tempos de vários ciclos, como eram os das instruções CISC.

O próximo objetivo a ser vencido foi permitir que uma arquitetura pipeline pudesse executar mais de uma instrução de cada vez. Para vencer essa barreira, era necessário que houvesse mais de uma instrução independente para ser executada e que houvesse hardware para tal. Dessa necessidade de aumento de desempenho dos processadores, surgiram os Processadores Superescalares, que possuem mais de um pipeline para permitir a execução de mais de uma instrução simultaneamente. A Figura 8 mostra essa nova estrutura de execução. Este mecanismo é conhecido por **ILP, *Instruction Level Parallelism*.**

Um processador superescalar típico busca e decodifica várias instruções por ciclo, enquanto as arquiteturas escalares buscam e executam apenas uma instrução por ciclo [Hwang 1993]. Dessa forma, pode-se esperar que múltiplas instruções sejam completadas a cada ciclo em um pipeline superescalar, o que não é possível em pipelines escalares [Pilla 2001]. Para permitir a execução de diversas instruções simultaneamente, é necessário verificar a não existência de dependências entre estas.

Portanto, é necessário que as instruções buscadas e decodificadas sejam analisadas na busca de dependências. Embora executando diversas instruções por ciclo, as arquiteturas superescalares possuem um compromisso com a semântica sequencial utilizada pelo programador e, portanto, devem manter o ordenamento total no final. Deve ser preservada a semântica sequencial ao mesmo tempo que se explora o paralelismo entre as instruções [Sohi 1995].

BI	D	BO	E
BI	D	BO	E

BI	D	BO	E
BI	D	BO	E

BI	D	BO	E
BI	D	BO	E

Figura 8. Estrutura de execução superescalar.

2.2.1. Unidades de uma Arquitetura Superescalar

Para a execução paralela de instruções, diversas premissas são necessárias para que se possa obter instruções disponíveis para sua execução. A arquitetura básica de um processador superescalar é apresentada na Figura 9, na qual se verifica a existência de diversas unidades.

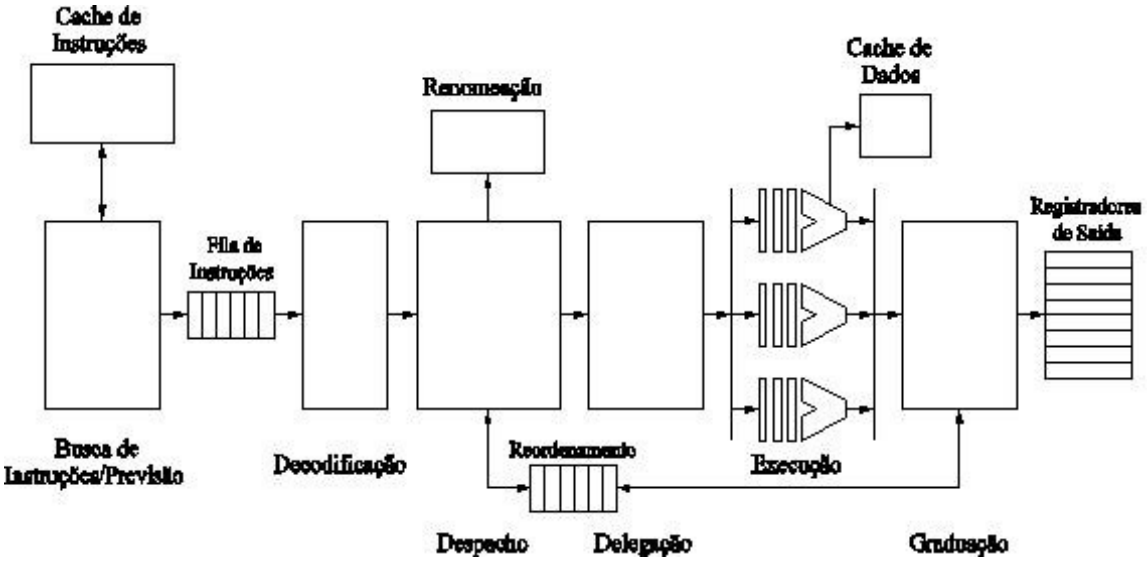


Figura 9. Arquitetura superescalar básica.

No modelo de processamento superescalar, diversas instruções são executadas concorrentemente e normalmente possuem códigos de operação diferentes. Essas arquiteturas começaram a surgir em 1980 e hoje integram a maioria dos

microprocessadores existentes. Diversas são as técnicas empregadas, em sua totalidade ou parcialmente que resultam no aumento do desempenho dessas arquiteturas. Abaixo, são apresentadas suas principais características [Smith 1995][Pizzol 2002]:

- técnicas para determinação e processamento de dependências entre os dados dos registradores;
- estratégias de busca de múltiplas instruções por ciclo;
- antever/prever os desvios condicionais;
- métodos para despacho de múltiplas instruções;
- métodos para recuperação do estado de ordenamento correto (semântica sequencial);
- estratégias para comunicação de dados, através da memória, empregando load/store;
- recursos para execução paralela de múltiplas instruções, incluindo múltiplas unidades funcionais e memória hierárquica para permitir múltiplos acessos.

2.3. Arquiteturas Multithread

Com o surgimento de processadores com arquiteturas superescalares, visto anteriormente, esperava-se que a paralelização das instruções conseguiria um melhor uso da várias unidades funcionais existentes, em geral de 6 a 8 unidades. No entanto em média as arquiteturas superescalares dobraram o número de instruções executadas, comparado com uma arquitetura pipeline escalar simples. A razão maior para este limite é a própria estrutura dos nossos programas que não permite extrair um maior número de instruções paralelas.

Portanto o desafio dos pesquisadores em aumentar o desempenho, levou a pensar que se de um fluxo de instruções eu só consigo obter um paralelismo da ordem de 2 instruções, porque não aumentar o número de fluxos de instruções a serem executadas no meu processador simultaneamente, permitindo desta forma a extração de mais instruções para execução pelas unidades funcionais do meu processador. Surge dai a arquitetura conhecida por Multi-thread. ou multithread, que permite o processamento de mais de um fluxo de instruções em paralelo pela arquitetura do processador.

Multithread demonstra ser uma boa alternativa para a execução de programas. Programas são representados pelo S.O. por processos e associado a cada processo existe um contexto que descreve o estado atual da execução do processo (registradores, PC, etc). Cada processo possui pelo menos uma thread, representada pelo fluxo de execução com o seu contexto local. A proposta da arquitetura multithread é executar mais do que um processo/thread simultaneamente, aproveitando melhor as unidades funcionais inativas. Este mecanismo é conhecido por **TLP - *Thread Level Parallelism***. A figura abaixo apresenta uma comparação entre o funcionamento de uma arquitetura Superescalar e uma arquitetura multithread.

Um processador multithread emprega uma técnica que permite que múltiplas threads despachem múltiplas instruções a cada ciclo para as unidades funcionais, UF,

de um processador. As arquiteturas MT combinam tanto o paralelismo no nível de instrução (ILP) como no paralelismo em nível de thread (TLP).

O paralelismo em nível de thread é obtido através de programas paralelos (multithreaded) ou de programas individuais (multiprogramação).

Por explorar os dois tipos de paralelismo, ILP e TLP, multithread usa de forma mais eficiente os recursos disponíveis e os speedups são maiores. O emprego de mutithread cria, em última análise, **processadores virtuais** para o usuário destes processadores.

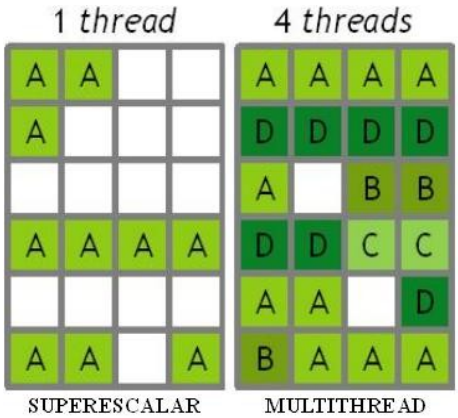


Figura 10. Comparação de eficiência entre o escalonamento de tarefas em ambientes superescalar e multithread.

As arquiteturas multithread possuem um hardware para armazenar o contexto de vários programas (threads) em execução no processador. Este hardware a mais incluído numa arquitetura superescalar para atender o multithread não passa de 5% de acréscimo o que torna bastante vantajoso para os fabricantes. A Figura 11 apresenta a estrutura geral de uma arquitetura multithread.

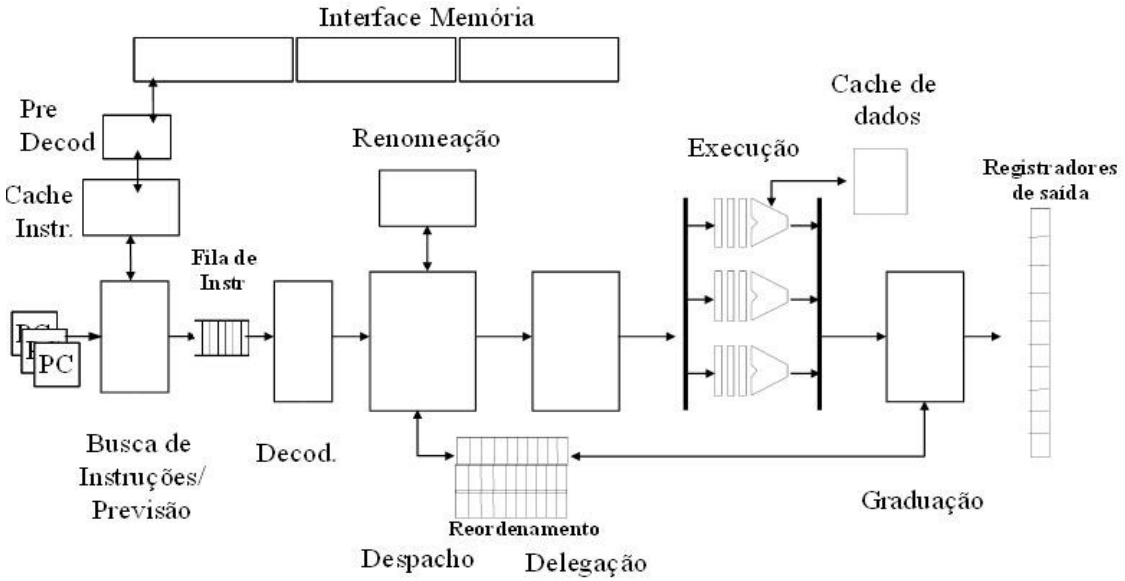


Figura 11. Arquitetura multithread básica.

Esta arquitetura busca reduzir o número de slots de despacho não ocupados a cada ciclo (comum em arquiteturas multithread) e o número de ciclos sem que instruções tenham sido despachadas (comum em arquiteturas superescalares). Associação das unidades funcionais com as threads é feita de forma completamente dinâmica.

2.3.1. Tipos de Multithread

As arquiteturas multithread dividem-se em tipos diferentes dependendo da forma que serão executados as threads no processador.

Explicit multithreading é uma abordagem de execução (simultânea ou não) de vários fluxos de instruções (threads) de diferentes origens. Esta forma de execução divide-se em três tipos diferentes que serão apresentados abaixo.

- Multithreading de grão fino, **IMT** - *Interleaved Multi Thread*, é o método em que a cada novo ciclo uma nova instrução de uma thread diferente é executada. Portanto existe uma alternância na execução das instruções das diferentes threads a cada ciclo. O processador muda de contexto a cada novo ciclo. Neste caso é possível eliminar conflitos de dependências de dados e aumentar o ganho na execução do pipeline. Exemplo do emprego de IMT é na arquitetura do processador Ultra Sparc T1.
- Multithreading de grão grosso, **BMT** - *Blocked Multi Thread*, é o método em que a execução de uma thread só é interrompida quando um evento de alta latência ocorre, tal como um acesso à memória. Neste caso, a cada novo ciclo a execução permanece com a mesma thread até acontecer uma mudança de contexto, por exemplo quando é realizado um acesso à memória. Sendo assim, é possível esperar o resultado do acesso á memória executando uma nova thread. Exemplo do emprego da arquitetura BMT é na arquitetura do processador IBM PowerPC RS64-III.
- **SMT**- *Simultaneous Multi Thread*, é uma técnica que permite múltiplas threads despacharem múltiplas instruções a cada ciclo para as UF, unidades funcionais, de um processador superescalar adaptado para execução multithread. As arquiteturas SMT combinam tanto o paralelismo em nível de instrução, **ILP**, como paralelismo em nível de thread, **TLP**. Por explorar os dois tipos de paralelismo, SMT usa de forma mais eficiente os recursos disponíveis e os speedups são maiores A figura abaixo apresenta comparativamente o funcionamento dos tipos diferentes de multithread.

Para o bom funcionamento dos processadores multithread é necessário caches que tragam suficientes instruções e dados para atender o volume de processamento. Observou-se que as Cache L2 tem um grande impacto sobre o desempenho dos processadores multithread, em especial os SMT, que requerem caches maiores e com maior associatividade.

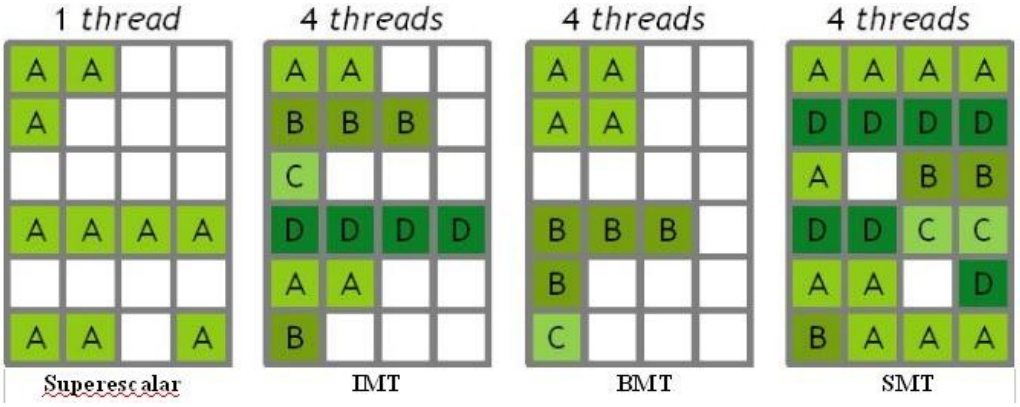


Figura 12. Comparação de eficiência entre o escalonamento de tarefas em ambientes superescalar e diferentes tipos de multithread.

2.4. Arquiteturas Multi-core

O surgimento dos multi-core foi motivada pela demanda contínua por desempenho e os limites físicos perto de serem alcançados nas frequências de relógio dos processadores, aliados ao alto consumo de potência e consequente geração de calor.

Chegou-se a conclusão que o aumento da frequência de relógio não garantiria ganhos reais, pois entre outras razões a latência de memória não acompanha a velocidade dos processadores. A Arquitetura Von Neumann tradicional é um gargalho em várias aplicações.

A alternativa ao aumento da frequência de relógio é a divisão das tarefas em operações concorrentes e distribuídas entre várias unidades de processamento, conhecidas como cores ou núcleos. Esta abordagem ficou conhecida por **CMP (*Chip Multi-processing*)**.

O uso de múltiplos cores em um único chip apresenta um melhor layout do circuito, com mais unidades funcionais (UFs) na mesma área, do que um processador com apenas um core, e portanto permite trabalhar com uma frequência de relógio menos elevada.

Contribuíram também para o surgimento dos multi-cores o fato que os processadores multithread demandarem caches grandes para atender o processamento das várias threads no mesmo processador físico, demandas dos processadores virtuais.

Além disto a arquitetura multi-core permite que o usuário continue trabalhando enquanto outras tarefas de processamento intensivo são executadas em paralelo.

Durante o processamento, alguns cores inativos podem ser desligados. Isso proporciona uma economia no consumo de potência e uma menor dissipação de calor. Uma das últimas gerações de chips sem multi-core era a do Itanium que possuía um consumo de potência na ordem dos 110W. Em contrapartida, primeiro Dual-core da Intel, o Montecito, já diminuía este consumo para 90W.

Além de possuir cores de propósito geral, os CMP's podem ter cores específicos, como para processamento de imagens, para algoritmos de reconhecimento de voz, para protocolos de comunicação, entre outros.

Além disso, pode-se reconfigurar dinamicamente os cores, a rede de interconexão e as caches para uma melhor adaptação a determinados cenários.

De forma simplificada um processador multi-core pode ser considerado como colocar dois ou mais processadores num mesmo processador/chip, como na Figura 13.

As vantagens desta arquitetura com vários cores é melhorar o paralelismo no nível de threads, como ilustrado na Figura 14. Além disto, permite ajudar aplicações que não conseguem se beneficiar dos processadores superescalares atuais por não possuírem um bom paralelismo no nível de instruções

Outras vantagens que podem ser enumeradas são a melhoria na localidade de dados, a melhoria na comunicação entre as unidades, a economia de espaço e de energia do chip. A vazão em um processador multi-core não aumentará para aplicações únicas não paralelizáveis, mas sim nos outros casos e no sistema como um todo.

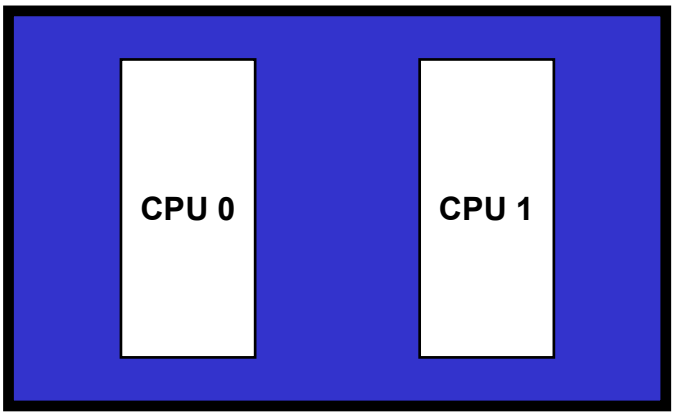


Figura 13. Visão simplificada de uma arquitetura multi-core.

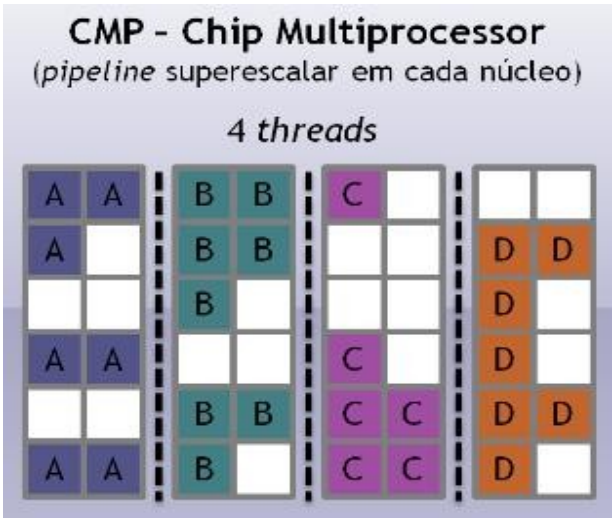


Figura 14. Paralelismo de tarefas em arquitetura multi-core.

2.4.1. Virtualização

A Virtualização surgiu nas arquiteturas multi-core como uma forma para mascarar a complexidade do gerenciamento do hardware com vários cores para a camada de

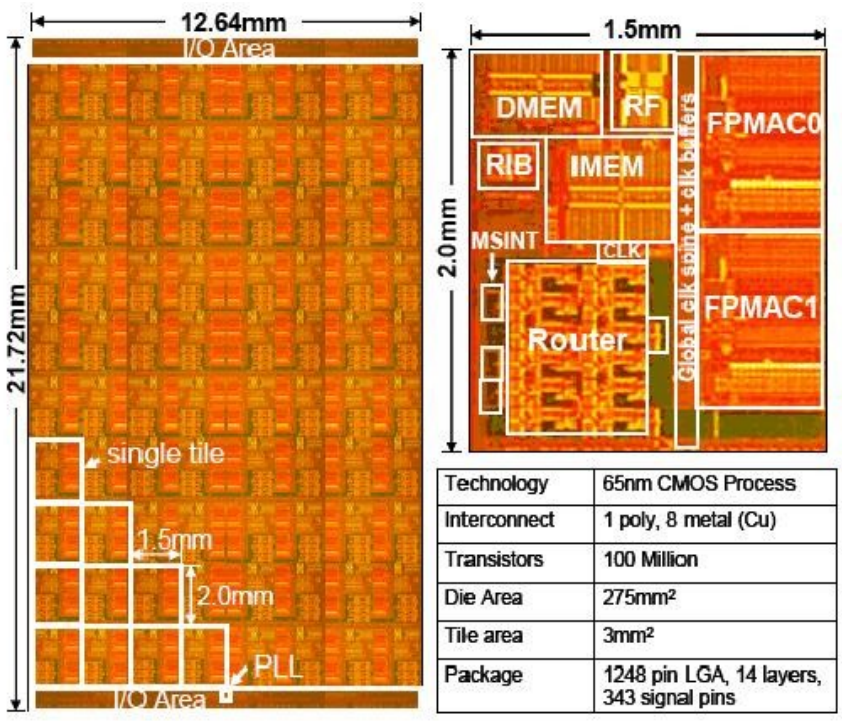
software. Ao invés do usuário se preocupar com a execução nos cores o sistema, a virtualização, se encarrega deste gerenciamento fornecendo ao usuário. Dessa forma, software vê o processador como uma série de máquinas virtuais (abstração).

O emprego da virtualização fornece segurança para as aplicações, separando aquelas que são confiáveis das demais. Além disso, é capaz de realizar o balanceamento de carga entre os processadores.

2.5. Arquiteturas Many-core

Com o surgimento de vários cores num chip a forma de conexão destes precisa ser mais complexa que um simples compartilhamento de cache, para tal é necessário empregar redes de interconexão denominadas de NoC, *Network on Chip.*

A Figura 15 apresenta a arquitetura do chip experimental Tera-Scale da Intel, onde pode ser observado que além dos núcleos existe uma estrutura para o roteador responsável pela comunicação entre os núcleos (cores). Verifica-se que quando o volume de cores fica grande não é mais possível fazer a comunicação destes através de memórias caches) compartilhadas e que a solução passa pelo emprego de redes de comunicação.



Technology	65nm CMOS Process
Interconnect	1 poly, 8 metal (Cu)
Transistors	100 Million
Die Area	275mm²
Tile area	3mm²
Package	1248 pin LGA, 14 layers, 343 signal pins

Figura 15. Arquitetura do processador experimental Tera-Scale.

Características principais do Intel Tera-Scale:

- Capacidade de processamento superior a 1 Tera-flops.
- 80 Núcleos VLIW.
- Cada núcleo com um roteador integrado.

- Emprego de Caches L1 integradas.
- Interconexão usando NoC.

3. Paralelismo em Arquiteturas com Múltiplos Processadores

3.1. Classificação de Flynn (fluxo de instruções / fluxo de dados)

Para uma classificação inicial de arquiteturas paralelas, pode ser usada a classificação genérica de Flynn. Apesar de ter sua origem em meados dos anos 70, é ainda válida e muito difundida.

Baseando-se no fato de um computador executar uma sequência de instruções sobre uma sequência de dados, diferenciam-se o fluxo de instruções (*instruction stream*) e o fluxo de dados (*data stream*). Dependendo de esses fluxos serem múltiplos ou não, e através da combinação das possibilidades, Flynn propôs quatro classes:

	SD (*Single Data*)	**MD** (*Multiple Data*)
SI (*Single Instruction*)	SISD Máquinas von Neumann convencionais	SIMD Máquinas *Array* (CM-2, MasPar)
MI (*Multiple Instruction*)	MISD Sem representante (até agora)	MIMD Multiprocessadores e multicomputadores (nCUBE, Intel Paragon, Cray T3D)

Figura 16. Classificação de Flynn.

Na classe **SISD** (*Single Instruction Single Data*), um único fluxo de instruções atua sobre um único fluxo de dados.

- O fluxo de instruções (linha contínua) alimenta uma unidade de controle (C) que ativa a unidade central de processamento (P).
- A unidade P, por sua vez, atua sobre um único fluxo de dados (linha tracejada), que é lido, processado e reescrito na memória (M).
- Nessa classe, são enquadradas as máquinas von Neumann tradicionais com apenas um processador, como microcomputadores pessoais e estações de trabalho.

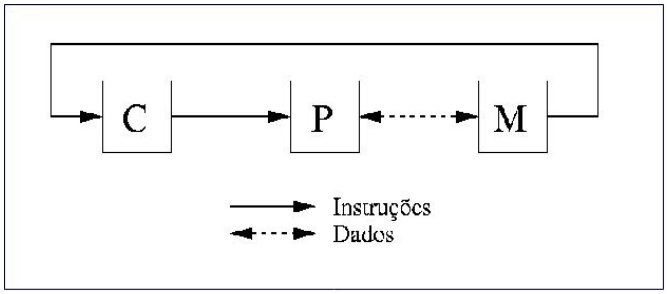


Figura 17. Classe SISD.

Na classe **MISD** (*Multiple Instruction Single Data*), múltiplos fluxos de instruções atuariam sobre um único fluxo de dados.

- Múltiplas unidades de processamento P, cada uma com sua unidade de controle própria C, recebendo um fluxo diferente de instruções.
- Essas unidades de processamento executam suas diferentes instruções sobre o mesmo fluxo de dados.
- Na prática, diferentes instruções operam a mesma posição de memória ao mesmo tempo, executando instruções diferentes.
- Como isso, até os dias de hoje, não faz qualquer sentido, além de ser tecnicamente impraticável, essa classe é considerada vazia.

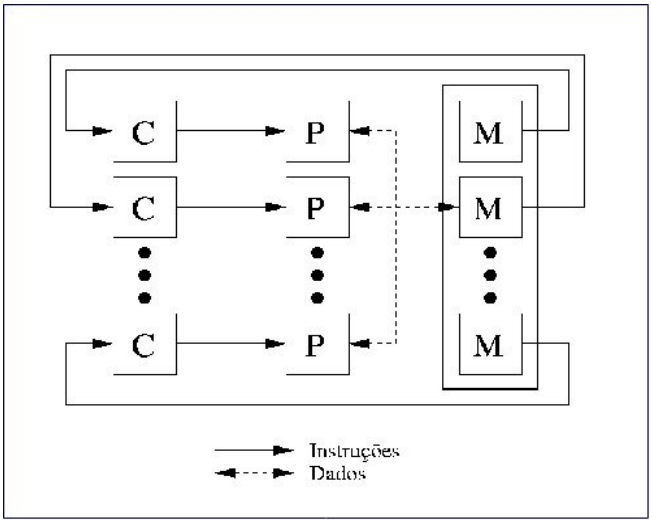


Figura 18. Classe MISD.

As **máquinas paralelas** concentram-se nas duas classes restantes, **SIMD** e **MIMD**. No caso **SIMD** (*Single Instruction Multiple Data),* uma única instrução é executada ao mesmo tempo sobre múltiplos dados.

- O processamento é controlado por uma única unidade de controle C, alimentada por um único fluxo de instruções.
- A mesma instrução é enviada para os diversos processadores P envolvidos na execução.

- Todos os processadores executam suas instruções em paralelo de forma **síncrona** sobre diferentes fluxos de dados.
- Na prática, pode-se dizer que o mesmo programa está sendo executado sobre diferentes dados, o que faz com que o princípio de execução SIMD assemelhe-se bastante ao paradigma de execução sequencial.
- É importante ressaltar que, para que o processamento das diferentes posições de memória possa ocorrer em paralelo, a unidade de memória M não pode ser implementada como um único módulo de memória, o que permitiria só uma operação por vez.

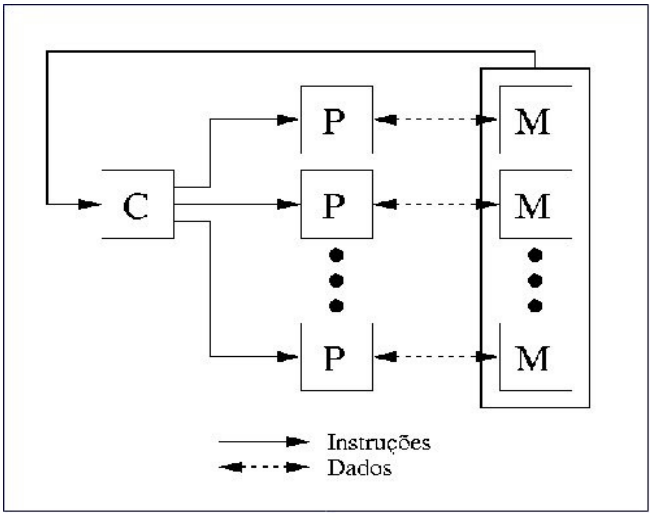


Figura 19. Classe SIMD.

Enquanto, em uma máquina SIMD, só um fluxo de instruções, ou seja, só um programa, está sendo executado, em uma máquina **MIMD** (*Multiple Instruction Multiple Data*), cada unidade de controle C recebe um fluxo de instruções próprio e repassa-o para sua unidade processadora P para que seja executado sobre um fluxo de instruções próprio.

- Dessa forma, cada processador executa o seu próprio programa sobre seus próprios dados de forma **assíncrona.**
- Sendo assim, o princípio MIMD é bastante genérico, pois qualquer grupo de máquinas, se analisado como uma unidade (executando, por exemplo, um sistema distribuído), pode ser considerado uma máquina MIMD.
- Nesse caso, como na classe SIMD, para que o processamento das diferentes posições de memória possa ocorrer em paralelo, a unidade de memória M não pode ser implementada como um único módulo de memória, o que permitiria só uma operação por vez.
- Nessa classe, enquadram-se máquinas com multi-core ou com múltiplos processadores.

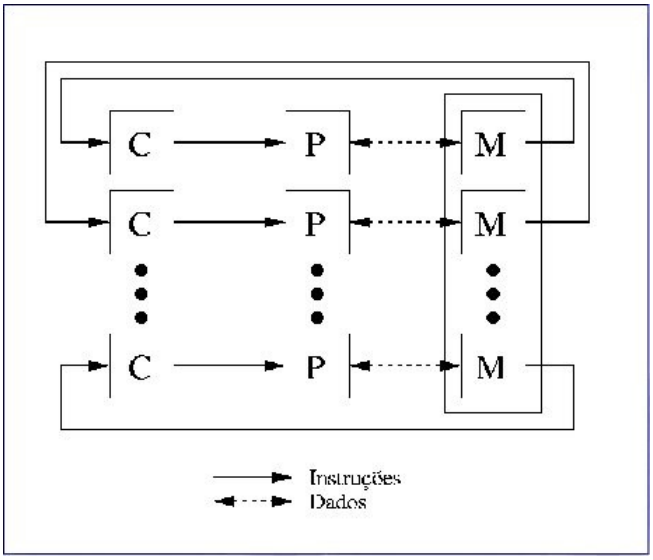


Figura 20. Classe MIMD.

3.2. Classificação segundo o compartilhamento de memória

Um outro critério para a classificação de máquinas paralelas é o compartilhamento da memória.

Quando se fala em **memória compartilhada** (*shared memory*), existe um único espaço de endereçamento que será usado de forma implícita para comunicação entre processadores, com operações de load e store.

Quando a memória não é compartilhada, existem **múltiplos espaços de endereçamento privados** (*multiple private address spaces*), um para cada processador. Isso implica comunicação explícita através de troca de mensagens com operações send e receive.

Memória distribuída (*distributed memory*), por sua vez, refere-se à localização física da memória. Se a memória é implementada com vários módulos, e cada módulo foi colocado próximo de um processador, então a memória é considerada distribuída.

Outra alternativa é o uso de uma **memória centralizada** (*centralized memory*), ou seja, a memória encontra-se à mesma distância de todos os processadores, independentemente de ter sido implementada com um ou vários módulos .

Dependendo de uma máquina paralela utilizar-se ou não de uma memória compartilhada por todos os processadores, pode-se diferenciar:

- Multiprocessadores
- Multicomputadores

3.2.1. Multiprocessadores

Todos os processadores *P* acessam, através de uma rede de interconexão, uma memória compartilhada *M.*

Esse tipo de máquina possui apenas um espaço de endereçamento, de forma que todos os processadores *P* são capazes de endereçar todas as memórias *M*.

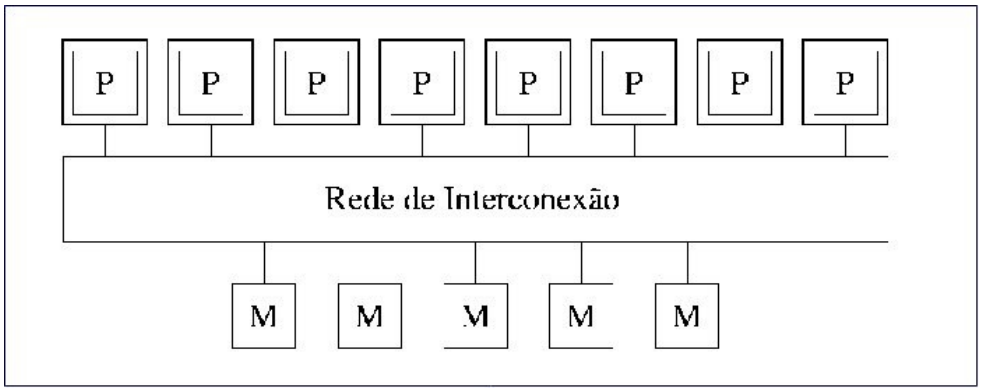


Figura 21. Organização de multiprocessadores.

A comunicação entre processos é feita através da memória compartilhada de forma bastante eficiente com operações do tipo load e store.

Essas características resultam do fato de esse tipo de máquina paralela ser construída a partir da replicação apenas do componente processador de uma arquitetura convencional (destacados com uma moldura mais escura na figura). Daí o nome **múltiplos processadores.**

Em relação ao tipo de acesso às memórias do sistema, multiprocessadores podem ser classificados como:

- UMA
- NUMA
- COMA

Acesso uniforme à memória (*uniform memory access*, UMA): A memória usada nessas máquinas é centralizada e encontra-se à mesma distância de todos os processadores, fazendo com que a latência de acesso à memória seja igual para todos os processadores do sistema (uniforme).

Como o barramento é a rede de interconexão mais usada nessas máquinas e suporta apenas uma transação por vez, a memória principal é normalmente implementada com um único bloco.

É importante ressaltar que máquinas com outras redes de interconexão e com memórias entrelaçadas (implementadas com múltiplos módulos e, dessa forma, permitindo acesso paralelo a diferentes) também se enquadram nessa categoria se mantiverem o tempo de acesso à memória uniforme para todos os processadores do sistema.

Acesso não uniforme à memória (*non-uniform memory access*, NUMA): A memória usada nessas máquinas é distribuída, implementada com múltiplos módulos que são associados um a cada processador.

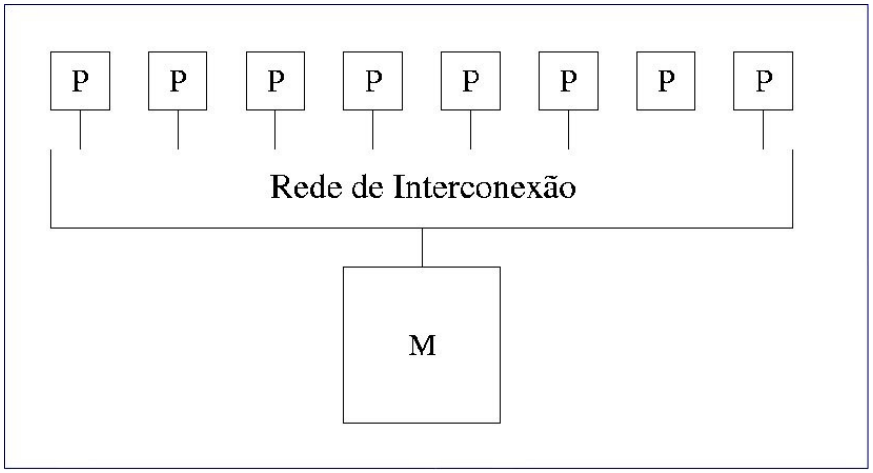


Figura 22. Arquitetura UMA.

O espaço de endereçamento é único, e cada processador pode endereçar toda a memória do sistema.

Se o endereço gerado pelo processador encontrar-se no módulo de memória diretamente ligado a ele, dito local, o tempo de acesso a ele será menor que o tempo de acesso a um módulo que está diretamente ligado a outro processador, dito remoto, que só pode ser acessado através da rede de interconexão.

Por esse motivo, essas máquinas possuem um acesso não uniforme à memória (a distância à memória não é sempre a mesma e depende do endereço desejado).

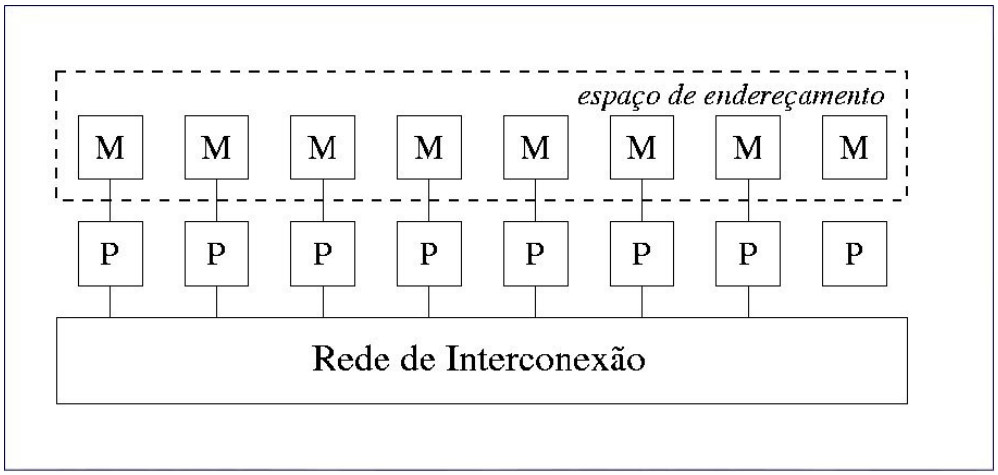


Figura 23. Arquitetura NUMA.

Arquiteturas de memória somente com *cache* (*cache-only memory architecture,* COMA): Em uma máquina COMA, todas as memórias locais estão estruturadas como memórias *cache* e são chamadas de COMA *caches*. Essas *caches* têm muito mais capacidade que uma *cache* tradicional.

Arquiteturas COMA têm suporte de hardware para a replicação efetiva do mesmo bloco de *cache* em múltiplos nós fazendo com que essas arquiteturas sejam mais caras de implementar que as máquinas NUMA.

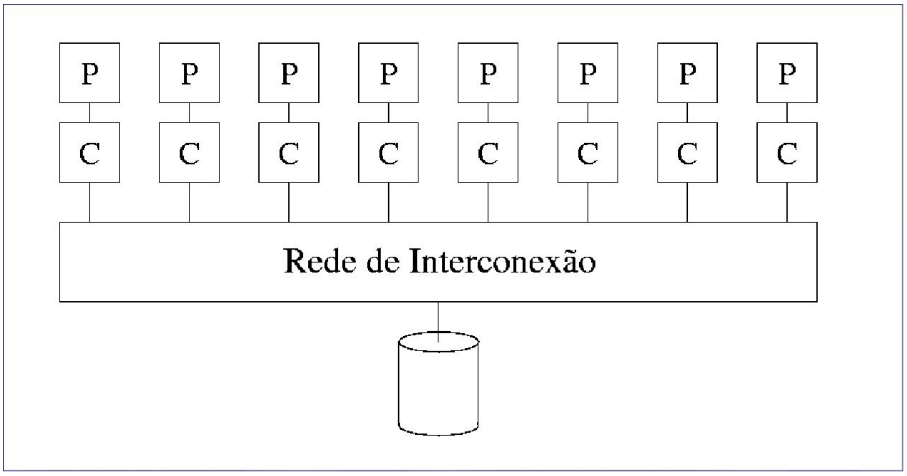


Figura 24. Arquitetura COMA.

3.2.2. Multicomputadores

Cada processador *P* possui uma memória local *M*, à qual só ele tem acesso. As memórias dos outros processadores são consideradas memórias remotas e possuem espaços de endereçamento distintos (um endereço gerado por P_1 só é capaz de endereçar M_1).

Como não é possível o uso de variáveis compartilhadas nesse ambiente, a troca de informações com outros processos é feita por envio de mensagens pela rede de interconexão.

Por essa razão, essas máquinas também são chamadas de sistemas de **troca de mensagens** (*message passing systems*).

Essas características resultam do fato de esse tipo de máquina paralela ser construído a partir da replicação de toda a arquitetura convencional (destacadas com uma moldura mais escura na figura abaixo), e não apenas do componente processador como nos multiprocessadores (daí o nome **múltiplos computadores).**

Em relação ao tipo de acesso às memórias do sistema, multicomputadores podem ser classificados como:

- NORMA

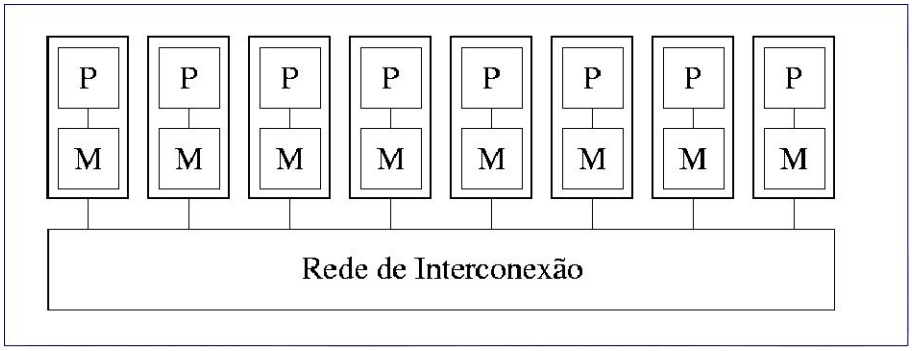


Figura 25. Arquitetura NORMA.

Sem acesso a variáveis remotas (*non-remote memory access*, NORMA): Como uma arquitetura tradicional inteira foi replicada na construção dessas máquinas, os registradores de endereçamento de cada nó só conseguem endereçar a sua memória local.

3.2.3.Visão geral da classificação segundo o compartilhamento de memória

A Figura 26 apresenta uma visão geral da classificação segundo o compartilhamento de memória:

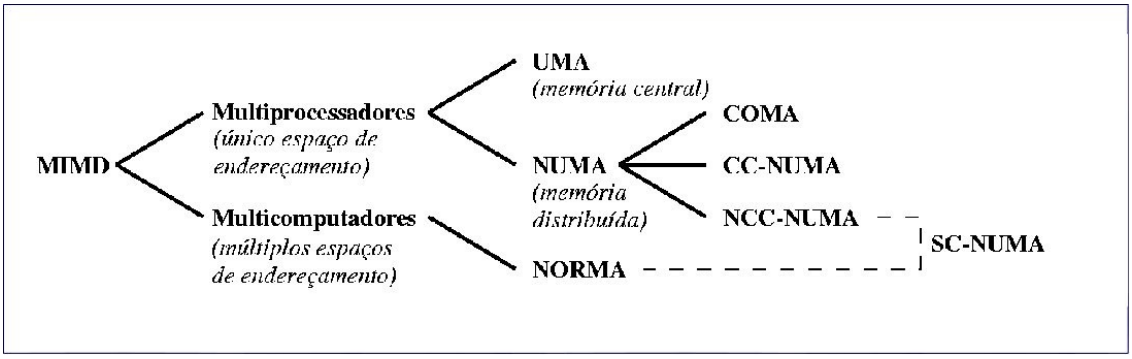


Figura 26. Classificação segundo o compartilhamento de memória.

A linha tracejada indica que as máquinas das classes NCC-NUMA e NORMA podem ser transformadas em máquinas SC-NUMA através da inclusão de uma camada de software que implemente coerência de *cache.*

3.3. Organização da Memória Principal

Em uma máquina paralela, a memória principal tem um papel fundamental, como vimos na seção de classificações, especialmente em multiprocessadores onde é compartilhada por todos os processadores.

É necessário que se tenha cuidado na escolha do tipo de organização de memória para que se evite uma drástica degradação de desempenho causada por dois ou mais processadores tentando acessar os mesmos módulos do sistema de memória.

3.3.1. Memórias entrelaçadas

Não é desejável que a memória principal seja implementada por um único módulo de forma monolítica mas sim particionada em vários módulos independentes com um espaço de endereçamento único distribuído entre eles.

Essa forma de implementação é chamada de **entrelaçamento** (*interleaving*) e atenua a interferência entre processadores no acesso à memória, permitindo acessos concorrentes a diferentes módulos.

O entrelaçamento de endereços entre *M* módulos de memória é chamado de entrelaçamento de *M*-vias.

É importante destacar que de nada adianta a quebra da memória principal em vários módulos se a rede de interconexão utilizada na máquina não suporta múltiplas transações.

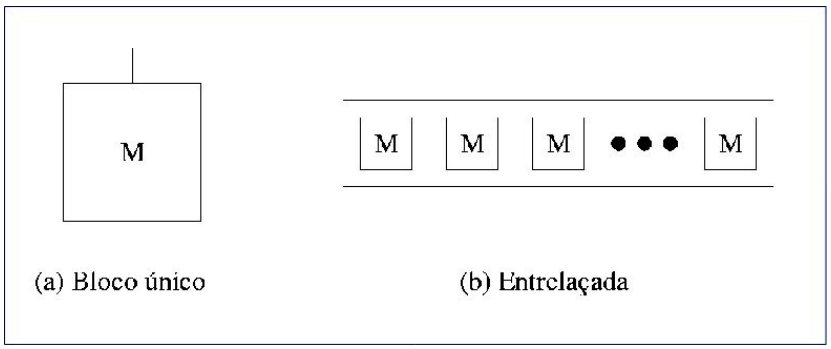


Figura 27. Diferentes organizações de memória.

Uma máquina UMA, por exemplo, que se utilize de um barramento para ligar os processadores à memória principal, não se beneficia dos múltiplos canais de uma memória entrelaçada, já que o próprio barramento é o gargalo.

3.4. Redes de Interconexão

A forma como os processadores de uma arquitetura são ligados entre si e com outros componentes do sistema (normalmente com a(s) memória(s)) é dada pela rede de interconexão.

Nas duas principais classes de arquiteturas vistas acima, a rede de interconexão desempenha um papel muito importante: nos multiprocessadores, ela pode ajudar a amenizar o problema dos conflitos de acesso, e em multicomputadores, ela influencia diretamente a eficiência da troca de informações.

As redes de interconexão se dividem em dois grandes grupos:

- Estáticas
- Dinâmicas

3.4.1. Redes estáticas

Se os componentes da máquina (processadores, memórias) estão interligados através de ligações fixas, de forma que, entre dois componentes, exista uma ligação direta dedicada, a rede de interconexão é denominada **estática** (ponto-a-ponto).

Redes estáticas são utilizadas, na maioria dos casos, em multicomputadores. Nesse caso, a **topologia** (estrutura da interligação) determina as características da rede.

Máquinas paralelas possuem quase sempre estruturas regulares com ligações homogêneas, enquanto sistemas distribuídos, por causa da posição geográfica e de sua integração gradual, possuem estruturas irregulares com ligações heterogêneas (ligações heterogêneas entre redes locais).

Os principais critérios para a avaliação de uma topologia são o número total de ligações entre componentes, quantas ligações diretas cada componente possui (**grau do nó**) e a maior distância entre dois componentes quaisquer da rede (**diâmetro**).

O grau do nó pode ser constante ou variar de acordo com o número total de nós da rede (por exemplo, em uma rede em forma de estrela).

Para cada ligação que um componente possui, é necessária uma interface física correspondente (porta), o que aumenta consideravelmente o custo do componente para muitas ligações.

Em relação ao custo, o melhor caso seria uma rede com um grau de nó baixo e constante.

A estrutura com a menor conectividade (relação entre o número ligações e o número de nós) é o **anel**:

- Com um grau de nó constante igual a 2, o seu custo é baixo, mas seu diâmetro cresce de forma linear em relação ao número total de nós.
- Se todos os processadores necessitarem trocar dados entre si, pode ocorrer uma sobrecarga do anel, o que acarretaria atraso na transferência dos dados.
- Outra desvantagem do anel é a falta de caminhos alternativos entre os nós, o que resulta em uma baixa confiabilidade.

O outro extremo, em relação ao anel, é a topologia **totalmente conectada**, com um grau de nó igual ao número de nós e um crescimento quadrático do número de conexões em relação ao número de nós, mas com o diâmetro ideal de 1.

Outro fator que pode ser decisivo na escolha de uma topologia, além da relação custo/desempenho, é sua adequação a uma classe específica de algoritmos.

No caso ideal, o padrão de interconexão da topologia corresponde exatamente ao padrão de comunicação da aplicação paralela que executa na máquina.

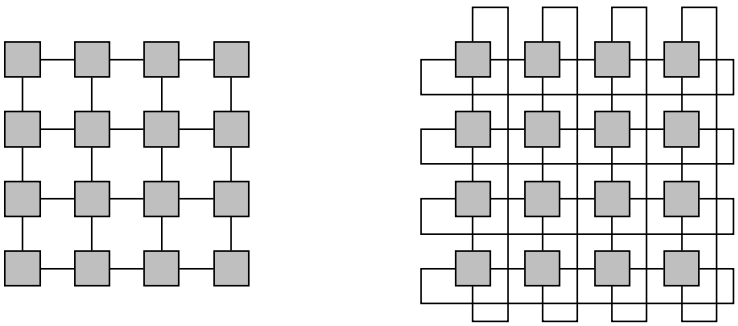
A **árvore binária**, por exemplo, favorece a execução de algoritmos de divisão e conquista (*divide and conquer*).

- O seu diâmetro cresce de forma linear em relação à altura *h* da árvore *A(h)* e de forma logarítmica em relação ao número de nós.
- Outras características das árvores binárias são o seu grau de nó máximo de 3 (a raiz tem grau igual a 2) e sua baixa confiabilidade, já que a falha de um nó resulta na perda da ligação com toda a subárvore abaixo dele (particionamento da estrutura).
- Mesmo que não ocorram falhas, o uso de árvores pode-se tornar problemático, pois todo o fluxo de dados entre a subárvore esquerda e a direita tem que passar pela raiz, a qual se torna rapidamente o gargalo da rede.

Uma topologia muito utilizada em máquinas paralelas é a **malha bidimensional** *M(x,y)*, podendo ter as bordas não conectadas (a) ou ter bordas conectadas de forma cíclica, formando um *torus* bidimensional (b):

- O grau do nó é constante e igual a 4 (se não forem consideradas as bordas da malha), possibilitando facilmente um aumento do número de processadores em qualquer uma das dimensões (inclusão de linhas ou de colunas) e resultando em uma boa escalabilidade.
- Malhas não precisam necessariamente ser quadradas como na figura, podendo uma das dimensões ser menor do que a outra, resultando em diferentes

retângulos (o que pode ser vantajoso no caso da escalabilidade, pois permite o aumento do número de processadores em múltiplos da menor dimensão).



(a) Malha (b) *Torus*

Figura 28. Redes estáticas em malha e em Torus.

- O diâmetro da malha cresce proporcionalmente à raiz quadrada do número de nós, e a existência de caminhos alternativos entre nós aumenta a confiabilidade da rede e diminui o risco de gargalos.
- Malhas são adequadas aos problemas nos quais uma estrutura de dados bidimensional tem que ser processada de forma particionada.
- Muitos problemas atuais, que necessitam de grande poder de processamento, possuem essas características, como, por exemplo, operações com matrizes, processamento de imagens e equações diferenciais .
- Malhas também podem ter mais de duas dimensões ($d > 2$). O grau do nó é, nesses casos, $2d$.
- Uma atenção especial vem sendo dada às estruturas tridimensionais, pois cada vez mais aplicações que necessitam de alto desempenho modelam aspectos físicos do nosso mundo tridimensional. Alguns exemplos são: previsão do tempo, simulação de partículas e aerodinâmica.

3.4.2. Redes dinâmicas

Na interconexão de componentes com redes **dinâmicas,** não existe uma topologia fixa que defina o padrão de comunicação da rede.

Quando uma conexão entre dois pontos faz-se necessária, a rede de interconexão adapta-se dinamicamente para permitir a transferência dos dados.

Uma rede dinâmica é dita **bloqueante** quando uma conexão estabelecida entre dois pontos P_1 e P_2 impede o estabelecimento de outra conexão entre componentes quaisquer que não P_1 e P_2.

Em uma rede dinâmica **unilateral,** cada componente possui uma ligação bidirecional com a rede. No caso de uma rede dinâmica **bilateral,** cada componente possui uma ligação de envio e outra de recebimento .

Enquanto as redes estáticas vistas até agora são utilizadas na maioria dos casos na interconexão de nós de multicomputadores, redes dinâmicas, por sua vez, são tipicamente responsáveis pela interligação de processadores com memórias em multiprocessadores. Algumas redes dinâmicas são também empregadas em multicomputadores.

Para facilitar sua análise, as redes dinâmicas foram divididas em três grupos de acordo com suas características:

- Barramento
- Matriz de chaveamento
- Redes multinível

Barramento: Dentre as redes dinâmicas, o barramento é a alternativa de menor custo. Porém, por tratar-se de um canal compartilhado por todas as conexões possíveis, tem baixa tolerância a falhas (baixa confiabilidade) e é altamente bloqueante.

Sendo assim, acaba tendo sua escalabilidade comprometida, sendo utilizado em multiprocessadores com um número moderado de processadores (em torno de 100).

As duas deficiências podem ser amenizadas com o uso de vários barramentos em paralelo.

Matriz de Chaveamento: A rede dinâmica de maior custo é a matriz de chaveamento (*crossbar switch*), que permite o chaveamento entre dois componentes quaisquer, desde que estes não se encontrem já ocupados.

Uma matriz de chaveamento pode ser usada como rede unilateral para ligar processadores a memórias em um multiprocessador (a) ou como rede bilateral para interligar processadores de um multicomputador (b):

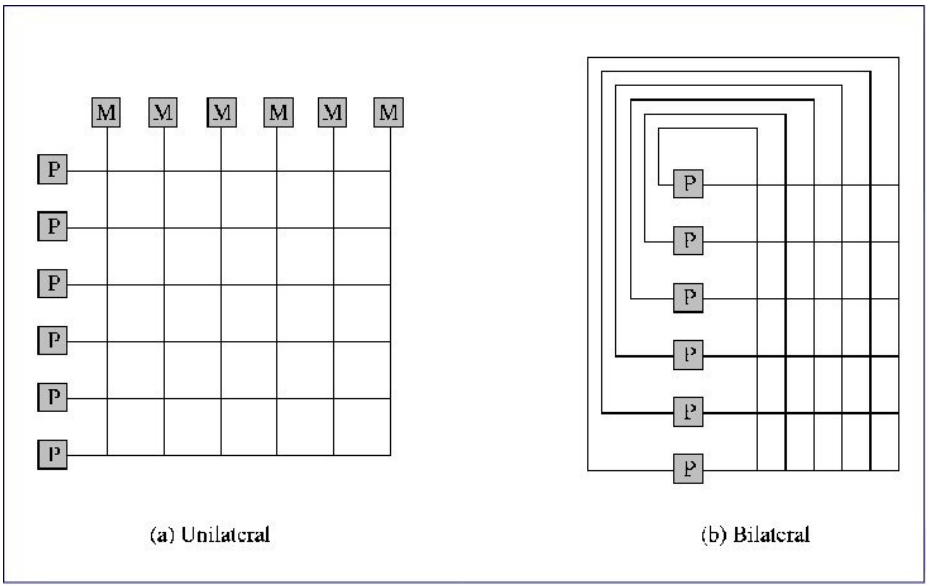


Figura 29. Matrizes de chaveamento.

A matriz de chaveamento não é bloqueante e tem uma escalabilidade boa, permitindo o acréscimo de componentes aos pares.

O alto custo, que cresce de forma quadrática em relação ao número de componentes interligados, inviabiliza, por razões econômicas, a sua utilização para a interconexão de muitos processadores.

Redes Multinível: Uma outra aplicação para redes hierárquicas de matrizes de chaveamento é a construção de redes de permutação multinível. A ideia básica é a ligação de pequenas matrizes de chaveamento (normalmente de tamanho 2 ▪ 2) em vários níveis consecutivos e conectá-las de forma a reduzir a probabilidade de conflitos entre conexões de diferentes pares.

Diferentemente das redes estáticas, a latência, nesse tipo de rede, é igual para qualquer par comunicante, crescendo, porém, de forma logarítmica de acordo com o número de possíveis conexões.

As matrizes chaveadoras presentes na maioria das redes multinível têm tamanho 2 x 2 e permitem no mínimo 2 e, na maioria das vezes, 4 posições de chaveamento:

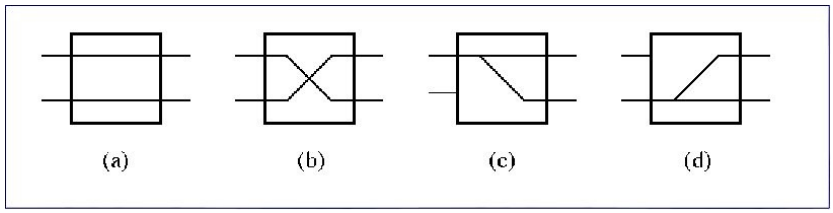


Figura 30. Diferentes formas de conexão.

A Figura 31 mostra um exemplo de rede multinível denominada Banyan:

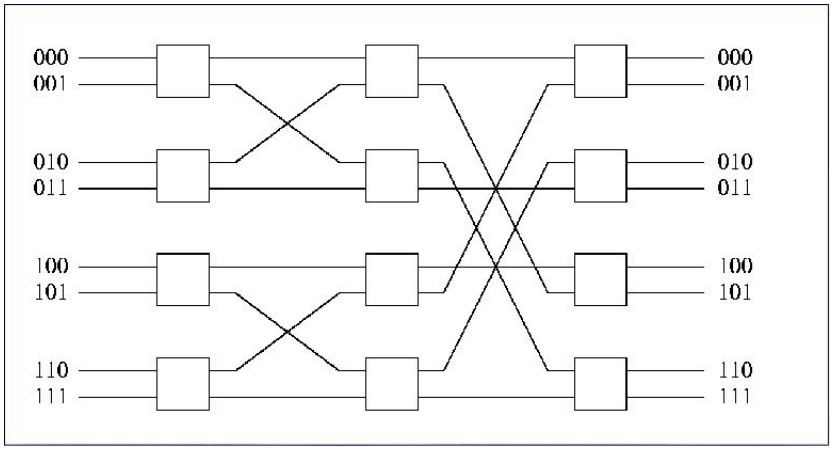


Figura 31. Rede de Banyan.

- O número de linhas é dado pela metade do número de possíveis componentes *n*, o número de níveis de $\log_2 n$, e, no total, (n/2).rede
- $\log_2 n$ matrizes de chaveamento são utilizadas.
- Nessa rede, existe apenas um caminho possível entre uma entrada e uma saída.
- Sendo assim, a escolha do caminho é muito eficiente e pode ser feita de forma descentralizada.

- Porém, por causa dessa falta de redundância, a rede é bloqueante.

3.4.3. Roteamento de Mensagens

É muito comum que na construção de máquinas paralelas, por motivos de custo, sejam utilizadas redes de interconexão que não possuem ligações diretas entre todos os componentes de um sistema. Sendo assim, uma mensagem, para chegar ao seu destino, pode precisar trafegar por nós intermediários.

É dado o nome de **roteamento** ao procedimento de condução de uma mensagem, através de nós intermediários, até seu destino.

Todos os nós envolvidos nessa condução participam do roteamento, identificando se a mensagem já chegou ao seu destino e, se não for o caso, reenviando-a para um próximo nó.

Fora o nó destino, todos os outros nós envolvidos nesse procedimento decidem o caminho seguido pela mensagem através da rede de interconexão e são chamados de nós roteadores.

Existem duas formas básicas de conduzir uma mensagem ao seu nó destino:

- Chaveamento de circuito
- Chaveamento de pacotes

Nas redes de telecomunicações, é tradicionalmente usado o **chaveamento de circuito** (*circuit switching*), pelo qual, inicialmente, é estabelecido um caminho fixo da origem ao destino, e só depois são enviadas todas as mensagens.

- Esse estabelecimento de conexão tem naturalmente um custo associado, que, no caso das telecomunicações, é pequeno em relação à duração da chamada.
- Essa forma de roteamento é usada por poucas máquinas paralelas, pois a comunicação entre dois nós, nesses casos, tem pouca duração (mensagens pequenas).
- Sendo assim, o estabelecimento da conexão teria uma representatividade significativa no tempo total de comunicação, e a reserva de circuitos na rede para poucas mensagens subutilizaria os canais reservados e poderia ainda atrasar o estabelecimento de outras conexões.

Mais comum em máquinas paralelas é o **chaveamento de pacotes** (*packet switching*) onde não existe caminho pré-definido, mas cada mensagem decide, a cada nó, qual a direção que irá seguir na rede.

- Isso elimina o custo inicial de estabelecimento de circuito, mas embute um custo adicional para o roteamento de cada mensagem em cada um dos nós visitados.
- Duas grandes vantagens do chaveamento de pacotes que tornam esse tipo de roteamento atrativo para máquinas paralelas são a **inexistência de uma reserva de canais** da rede para uma única operação de comunicação e o **estabelecimento dinâmico do caminho** a ser seguido por uma mensagem.

- O estabelecimento dinâmico do caminho, por sua vez, pode permitir que os algoritmos de roteamento reajam mais rapidamente a congestionamentos e falhas na rede de interconexão, optando por caminhos alternativos.

3.5. Coerência de Cache (*cache coherence*)

É muito comum que as máquinas paralelas atuais sejam construídas com processadores produzidos em larga escala com o objetivo de reduzir os custos de projeto.

Dessa fora as máquinas paralelas com múltiplos processadores acabam por incorporar *caches* em suas arquiteturas.

Porém a presença de *caches* privadas em multiprocessadores necessariamente introduz problemas de **coerência de *cache*** (*cache coherence* [Hwanh 1993][Hwang 1998]).

Como múltiplas cópias de uma mesma posição de memória podem vir a existir em *caches* diferentes ao mesmo tempo, cada processador pode atualizar sua cópia local, não se preocupando com a existência de outras cópias da mesma posição em outros processadores.

Se essas cópias existirem, cópias de um mesmo endereço de memória poderão possuir valores diferentes, o que caracteriza uma situação de inconsistência de dados.

Outra alternativa para eliminar completamente o problema seria não permitir que dados compartilhados para operações de escrita sejam colocados nas *caches* do sistema.

- Nesse caso, somente instruções e dados privados são *cacheable* (passíveis de serem colocados na *cache*), e dados compartilhados são considerados *noncacheable.*
- O compilador fica responsável pela colocação da respectiva etiqueta (*tag*) nos dados para que os mecanismos de cópia de dados entre os níveis da hierarquia de memória estejam cientes de quais os dados que devem ser repassados diretamente ao processador sem passar pelas *caches* privadas.

Em multicomputadores, esse problema não ocorre, já que cada nó possui uma hierarquia de memória inteira associada à sua memória local, e não existe um espaço de endereçamento global compartilhado por todos os nós.

DEFINIÇÃO:

- Uma arquitetura multiprocessada com *caches* privadas em cada processador **é coerente** se e somente se uma leitura de uma posição *x* de memória efetuada por qualquer processador *i* retorne o valor mais recente desse endereço.
- Ou seja, toda vez que uma escrita for efetuada por um processador *i* em um endereço de memória *x*, tem que ser garantido que todas as leituras subsequentes de *x*, independentemente do processador, forneçam o novo conteúdo de *x.*

3.5.1. O problema da inconsistência de dados

É imprescindível que o resultado de um programa composto por múltiplos processos não seja diferente quando o programa roda em vários processadores, do que quando roda em apenas um, utilizando-se de multiprogramação.

Porém, quando dois processos compartilham a mesma memória através de *caches* diferentes, existe o risco de que uma posição de memória, em um determinado momento, não possua mais o valor mais recente desse dado.

As causas mais comuns desse tipo de inconsistência de dados serão detalhadas abaixo e são:

- Inconsistência no compartilhamento de dados
- Inconsistência na migração de processos
- Inconsistência de E/S

Inconsistência no compartilhamento de dados:

- Os processadores P_1, P_2 e P_3 possuem *caches* privadas e estão interconectados a uma memória principal compartilhada através de um barramento. Eles efetuam uma sequência de acessos à posição de memória *u.*
- Inicialmente, P_1 tenta ler *u* de sua *cache*. Como *u* não está presente, o dado é lido da memória principal e copiado para sua *cache*. Na sequência, P_3 faz o mesmo, gerando também uma cópia de *u* em sua *cache*. Então P_3 efetua uma escrita em *u* e altera o conteúdo da posição de 5 para 7.
- Quando P_1 efetuar uma leitura de *u* novamente, ele recebe o conteúdo da posição *u* que se encontra em sua *cache*, ou seja, 5, e não o valor mais recente de *u*, que é 7.

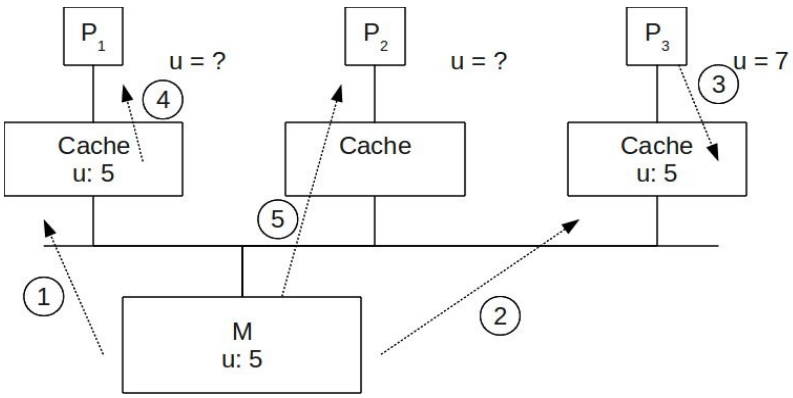


Figura 32. Exemplo de inconsistência de cache.

Vale destacar que esse problema ocorre independentemente da política de atualização da memória principal utilizada pela *cache.*

No caso de uma política de ***write-through*** (escrever através), em que, a cada escrita na *cache*, a posição da memória principal é atualizada também, a alteração de P_3

teria sido repassada também à memória principal, o que não impediria P_1 de ler o valor menos recente de *u* de sua *cache.*

Em se tratando da política ***write-back*** (escrever de volta), a situação seria ainda pior, pois a alteração de P_3 teria apenas marcado o bloco de *u* na *cache* como sujo (através de seu *dirty-bit*), e a memória principal não seria atualizada imediatamente. Somente em uma eventual substituição desse bloco na *cache* é que a memória principal seria atualizada. Se P_2 efetuasse uma leitura em *u* nesse meio-tempo, copiaria para sua *cache* o valor menos recente de *u* (valor 5).

Inconsistência na migração de processos: Quando um processo perde o processador por causa de uma operação de E/S, não existe qualquer garantia de que ele vá retornar sua execução no mesmo processador de uma máquina multiprocessada.

O escalonador do sistema associará esse processo a um dos processadores livres segundo sua política de escalonamento quando a operação de E/S estiver concluída.

Isso faz com que, muitas vezes, o processo volte a executar em outro processador perdendo as informações de sua antiga *cache.*

Inconsistência de E/S: Problemas de inconsistência de dados também podem ocorrer durante operações de E/S que façam acesso direto à memória principal (*DMA - Direct Memory Access*).

Operações de E/S desse tipo têm como origem ou destino à memória principal, e não se preocupam se os dados em questão estão sendo compartilhados por vários processadores, com possíveis cópias em diferentes *caches* privadas.

Sendo assim, quando uma controladora de E/S carrega um dado *x'* (atualizando o valor *x* que tinha sido escrito na memória principal por *write-through*) em uma posição da memória principal, ocorre inconsistência de dados entre *x'* e as cópias dessa posição de memória nas *caches* privadas dos processadores P_1 e P_2 que possuem o antigo valor *x.*

Quando um valor é lido da memória principal em uma operação de E/S, e as *caches* atualizam essa memória com a política de *write-back*, também pode ocorrer inconsistência de dados. Basta, para isso, que o valor atual das cópias dessa posição de memória nas *caches* privadas dos processadores P_1 e P_2 ainda não tenha sido substituído e, portanto, ainda não tenha sido copiado para a memória principal.

3.5.2. Estratégias de coerência de cache

Como vimos até agora, o problema da coerência de *cache* resume-se ao fato de que, em um determinado momento, possam existir simultaneamente múltiplas cópias de um mesmo dado da memória principal **o qual pode ser alterado localmente sem que se faça algo em relação às outras cópias** (o que geraria inconsistência de dados)

Ou seja, o problema de coerência de *cache* seria resolvido se essa inconsistência de dados fosse eliminada. Isso pode ser obtido através de duas estratégias básicas:

- Uma operação de escrita em uma posição da *cache* resulta na **atualização** de outras cópias desse mesmo dado em outras *caches* do sistema (*write-update*).

- Uma operação de escrita em uma posição da *cache* resulta na **invalidação** de outras cópias desse mesmo dado em outras *caches* do sistema (*write-invalidate*).

As duas estratégias resolvem o problema, impedindo que sejam geradas múltiplas cópias diferentes da mesma posição de memória.

A estratégia de **invalidação** tem um custo menor, mas resulta em uma maior latência de acesso caso as cópias invalidadas (eliminadas da *cache*) sejam novamente acessadas, o que resultaria em uma busca na memória principal.

A **atualização** das cópias tem naturalmente um custo mais alto, especialmente em máquinas com muitos processadores (e muitas cópias potenciais de um mesmo endereço), mas faz com que um novo acesso a essas cópias seja resolvido em nível de *cache* (menor latência).

A questão de qual das estratégias acima resulta em melhor desempenho para o sistema como um todo está diretamente ligada ao padrão de compartilhamento da carga de trabalho, ou seja, dos programas que executam na máquina:

- Se os processadores que estavam usando as cópias antes de serem atualizadas fizerem novos acessos a esses dados, o custo das **atualizações** vai ter valido a pena.
- Se os processadores não utilizarem esses dados novamente, o tráfego gerado pelas atualizações só onerou a rede de interconexão e não teve utilidade. Nesse segundo caso, a **invalidação** eliminaria as cópias antigas e acabaria com uma situação de compartilhamento aparente.

Um fenômeno que onera bastante a estratégia de atualizações é denominado *"pack rat"*. Ele é uma consequência da migração de processos executada pelo sistema operacional durante a multiprogramação. Quando um processo perde o processador por causa de uma operação de E/S, ele pode voltar a executar em outro processador, dependendo da disponibilidade dos recursos do sistema naquele momento e da política empregada pelo escalonador do sistema operacional. Nesse caso, além de não poder acessar mais os dados de sua antiga *cache* e ter que buscá-los novamente da memória principal, os dados antigos continuam sendo atualizados em vão até que sejam eliminados da *cache* antiga por alguma política de substituição de blocos (como, por exemplo, *LRU - Least Recent Used* [Stallings 1996], que elimina da *cache* os blocos menos recentemente usados).

Como é muito fácil construir casos nos quais uma determinada estratégia vai desempenhar melhor que a outra e vice-versa, alguns autores propõem como alternativa que as duas estratégias sejam implementadas em hardware e que o sistema permita a troca entre atualização e invalidação dinamicamente em tempo de execução.

O momento da troca pode ser:

- Determinado pelo programador através de uma chamada de sistema.
- Seguir uma certa probabilidade alterável na configuração do sistema.
- Depender do padrão de acesso observado em tempo de execução.

Referências

Block, E. (1959) "The Engineering design of the STRETCH computer" Em: Proceedings of Eastern Joint Computer Conference. [S.l. : s.n].

Eckert, J. P. et al. (1959) "Design of UNIVAC-LARC system 1", Em: Proceedings of Eastern Joint Computer Conference. [S.l. : s.n]. p. 60-65.

Hwang, K. e Briggs, F. A. (1985) "Computer Architecture and Parallel Processing". [S.l.] : McGraw-Hill.

Hwang, K. (1993) "Advanced Computer Architecture: Parallelism, Scalability, Programmability". [S.l.] : McGraw-Hill.

Hwang, K. e Xu, Z. (1998) "Scalable Parallel Computing: technology, architecture, programming". Boston: WCB.

Pilla, M. (2001) "Arquiteturas Superescalares: Exploração Dinâmica da Previsibilidade e Redundância de Valores". Exame de Qualificação (Doutorado Ciência da Computação) - Instituto Informática, UFRGS, Porto Alegre. 92p.

Pizzol, G. D. (2002) "SimMan: Simulation Manager. Definição e Implementação de um Ambiente de Simulação de Arquiteturas Superescalares para a Ferramenta SimpleScalar." Projeto de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre. 64p.

De Rose, C. A. F. e Navaux, P. O. A. (2003) "Arquiteturas Paralelas", Série de Livros Didáticos, vol. 15, Editora Artmed.

Sohi, G. S. e Breach, S. e Vijaykumar, T. N. (1995) "Multiscalar Processor". Em: SIGARCH Computer Architecture News, New York, v. 23, n. 2 May 1995. Annual International Symposium on Computer Architecture, ISCA, 22.

Smith, J. E. e Sohi, G. S. (1995) "The Microarchitecture of Superscalar Processors" Em: Proceedings of the IEEE, New York, v. 83, n.12, p. 1609-1624.

Stallings, W. (1996) "Computer Organization and Architecture: designing for performance". Upper Saddle River, Prentice Hall.

Von Neumann, J. (1945) "First Draft of a Report on the EDVAC". [S.l. : s.n.].d