Paralelismo de dados

(execução de simultaneidade)

Em métodos tradicionais de programação (processamento sequencial), **uma grande quantidade de dados é processada em um único núcleo de uma CPU**, **enquanto os outros núcleos permanecem livres.**

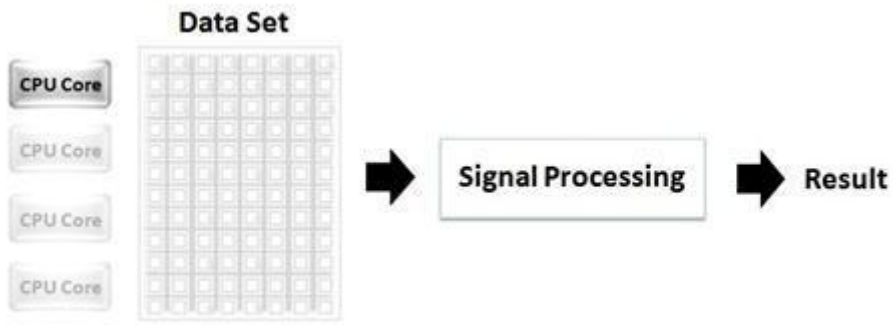


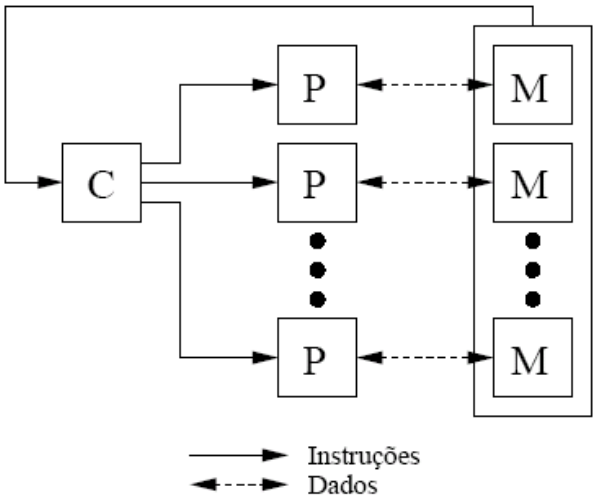
Figura 1 - Em métodos tradicionais de programação, o processamento sequencial de uma grande quantidade de dados é processada em um único núcleo da CPU, enquanto os outros núcleos permanecem livres.

Tipo de arquitetura paralela SIMD

- SIMD (**Single Instruction Multiple Data**)

- Significa que todas as unidades paralelas compartilham a mesma instrução, mas a realizam em diferentes elementos de dados.

Arquitetura SIMD



SIMD - Single Instruction Multiple Data

- A idéia é que podemos adicionar os arrays A=[1,2,3,4] e B=[5,6,7,8] para obter o array S=[6, 8, 10, 12] .
- Para isso, tem que haver quatro unidades aritméticas no trabalho, mas todos podem compartilhar a mesma instrução (aqui, "*add*").

Paralelismo de dados **refere-se a cenários de processamento em que a mesma operação (instrução) é executada simultaneamente (isto é, em paralelo) aos elementos em uma coleção de origem ou uma matriz.**

Em operações paralelas de dados, a coleção de origem é particionada para que vários threads podem funcionar simultaneamente em diferentes segmentos.

O **paralelismo de dados** é uma técnica de programação que divide uma grande quantidade de dados em partes menores que podem ser operadas em paralelo. Após os dados serem processados, eles são combinados novamente em um único conjunto.

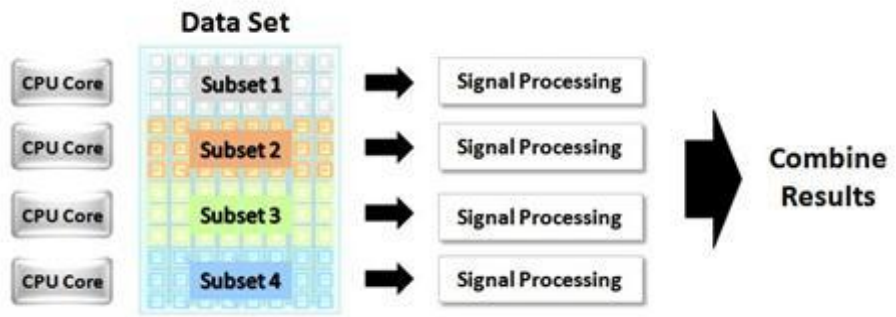


Figura 1 - Utilizando a técnica de programação com paralelismo de dados, uma grande quantidade de dados pode ser processada em paralelo em múltiplos núcleos da CPU/GPU.

Você pode aplicar paralelismo de dados para grandes conjuntos de dados como vetores e matrizes (arrays), dividindo esse conjunto em subgrupos, realizando operações e combinando seus resultados.

Com esta técnica, programadores podem modificar um processo que tipicamente não seria capaz de utilizar as potencialidades dos processadores **multicore** (CPU) ou **manycore** (GPU), e assim, particionando os dados (paralelizando operaçoes sobre os mesmos), pode-se utilizar toda a força de processamento disponível.

Visto de outra forma, em primeiro lugar, considere a aplicação seqüencial, na qual uma única CPU (um único núcleo) processa todo o conjunto de dados. Neste caso, tem-se um **único núcleo de processamento**.

Em vez disso, considere o exemplo de um mesmo conjunto de dados divididos em quatro partes. Você pode distribuir este conjunto de dados em todos os núcleos disponíveis para alcançar um aumento significativo de velocidade.

Ou múltiplos núcleos de processamento -

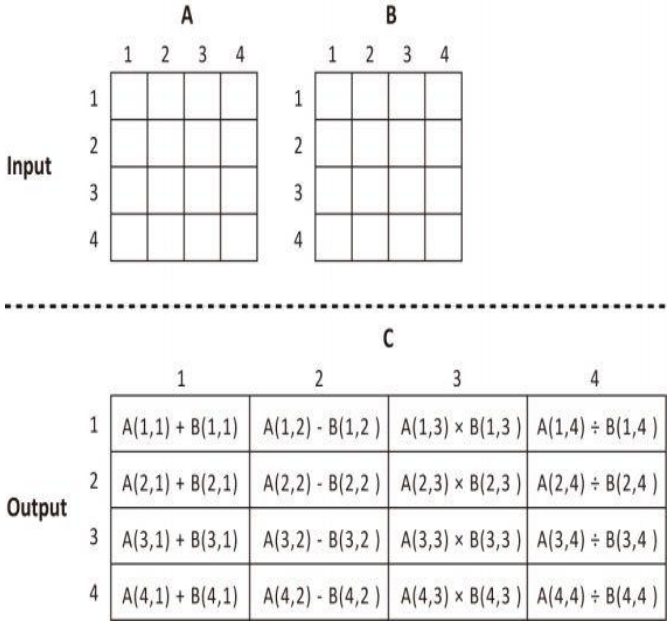
Em aplicações computacionais de alto desempenho (High-Performance Computing - HPC) em tempo real, como sistemas de controles, uma estratégia comum e eficiente é a realização paralela de multiplicações de matrizes de dimensões consideráveis. Normalmente a matriz é fixa, e é possível realizar sua decomposição. A medição colhida dos sensores fornece o vetor (ou a matriz) a cada iteração do loop. Você pode, por exemplo, usar os resultados da matriz para controlar atuadores.

Exemplo

- O código de exemplo executa as operações aritméticas básicas, que são adição, subtração, multiplicação e divisão, entre valores de ponto flutuante (reais, nas matrizes dadas).

- A visão geral é mostrada na Figura 4.4.

Figure 4.4:
Basic arithmetic operations between floats



Na Figura 4.4

- Como mostra a Figura 4.4, os dados de entrada consistem em 2 conjuntos de matrizes 4x4, A e B. Os dados de saída são uma matriz 4x4, C.

- Veja primeiro a implementação paralela de dados (Lista 4.8, Lista 4.9).

- Este programa trata cada linha de dados como um *work-group* para executar a computação.

Você pode ver, no link seguinte, um exemplo do modelo de paralelismo de dados, *List-4-8-List-4.9* *onde a mesma instrução atua sobre diferentes elementos de dados, tratando cada linha de dados como um work-group em OpenCL, ou o que é o mesmo como um bloco de threads em CUDA.*

Na Lista 4.8, a linha de código abaixo define um kernel para o modelo de paralelismo de dados, como segue:

__kernel void **dataParallel**(__global float * A, __global float * B, __global float * C)

Quando o kernel para paralelismo de dados é enfileirado, work-items (threads) são criados. Cada destes work-items (threads) executa o mesmo kernel em paralelo.

```
int base = 4*get_global_id(0);
```

Na Lista 4.8, esta linha de código significa que o get_global_id(0) obtém o ID global do work-item (thread), o qual é usado para decidir qual dado é para processar, de modo que cada work-item (thread) pode processar diferentes conjuntos de dados em paralelo.

Paralelismo de Tarefa

(execução de simultaneidade)

Publicado: julho de 2016 - https://msdn.microsoft.com/pt-br/library/dd492427.aspx

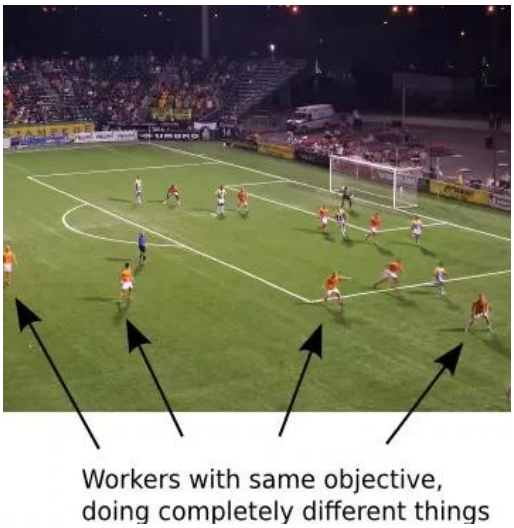
.

No tempo de execução de simultaneidade, um *tarefa* é uma unidade de trabalho que executa um trabalho específico e normalmente é executado em paralelo com outras tarefas.

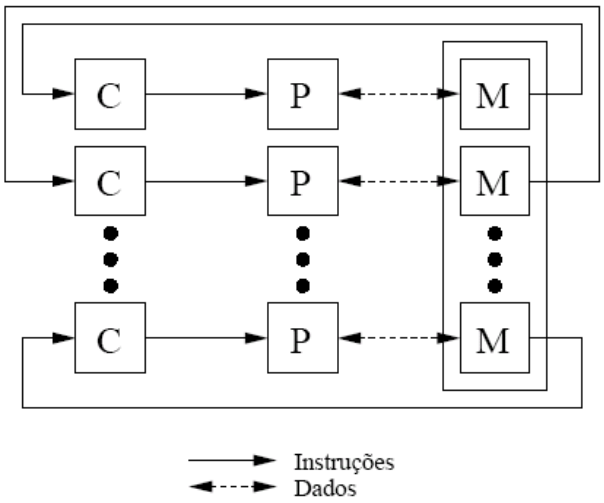
Tipo de arquitetura paralela MIMD

- MIMD (**Multiple Instruction Multiple Data**)

- Significa que as unidades paralelas têm instruções distintas, então cada uma delas pode fazer algo diferente em um dado momento.



Do ponto de vista da arquitetura visualizada em termos de hardware, você pode ver a figura que segue:



MIMD

Uma tarefa pode ser decomposta em tarefas mais refinadas adicionais que são organizadas em um ***grupo tarefas*** (podendo-se definir um **bloco** de tarefas).

Paralelismo de tarefa é a forma mais simples de programação paralela, onde as aplicações estão divididas em tarefas exclusivas que são independentes umas das outras e podem ser executadas em processadores diferentes.

Como exemplo, considere um programa com dois loops (loop A e loop b); o loop A executa uma rotina de processamento de sinais e o loop b realiza atualizações na interface de usuário. Isto representa paralelismo de tarefa, em que uma aplicação multithread pode executar dois loops em threads separadas para utilizar dois núcleos de uma CPU.

Você usa tarefas quando você quer escrever código assíncrono e deseja alguma operação após a operação assíncrona ser concluída.

Num código assíncrono, tarefas do seu programa poderão ser executadas sem interferir em nada do fluxo de execução dos códigos de outras tarefas. Essas tarefas são conhecidas como fluxos assíncronos.

Você pode disparar uma série dessas tarefas sem precisar esperar que cada uma se complete para prosseguir.

Por exemplo, você poderia usar uma tarefa para ler de um arquivo de forma assíncrona e, em seguida, usar outra tarefa — um ***tarefa de continuação***, que é para processar os dados depois que ele fica disponível.

Por outro lado, você pode usar grupos de tarefas para decompor o trabalho paralelo em partes menores.

Por exemplo, suponha que você tenha um algoritmo recursivo que divide o trabalho restante em duas partições. Você pode usar grupos de tarefas para executar essas partições simultaneamente e aguarde até que o trabalho dividido para ser concluído.

Você pode ver, no link seguinte, um exemplo de código usando o modelo de paralelismo de tarefas: *List-4-10-List-4-11* onde as tarefas tem instruções (operações) diferentes, e cada uma delas faz algo diferente em um dado momento, de forma paralelizada.

- Conclusão -

  Em OpenCL, O código paralelizável é implementado como:

  "**Paralelismo de dados**" ou "**Paralelismo de Tarefa**"

- No OpenCL, a diferença entre os dois modelos de paralelismo é, se o **mesmo kernel** (dados) ou **kernels diferentes** (tarefas) são executados em paralelo.

Estratégias de Programação para Processamento Multicore: Pipelining

Pipelining é similar a uma linha de montagem. Considere esta abordagem em aplicações de streaming ou qualquer outra aplicação na qual você terá que modificar um algoritmo seqüencial de processamento intenso na CPU, onde cada etapa leva um tempo considerável.

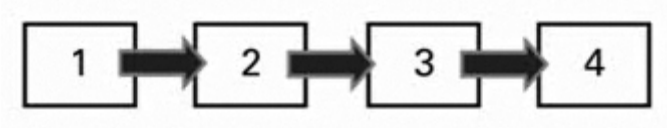


Figura 1 – Etapas sequenciais de um algoritmo

Como uma linha de montagem, cada estágio concentra-se em uma unidade de trabalho. Cada resultado passa para o próximo estágio até chegar ao último.

Para aplicar uma estratégia pipelining em uma CPU multicore, o algoritmo é separado em etapas que têm aproximadamente a mesma unidade de trabalho e executa cada passo com um núcleo distinto. O algoritmo pode repetir a execução em vários conjuntos de dados ou em fluxo de dados contínuo.

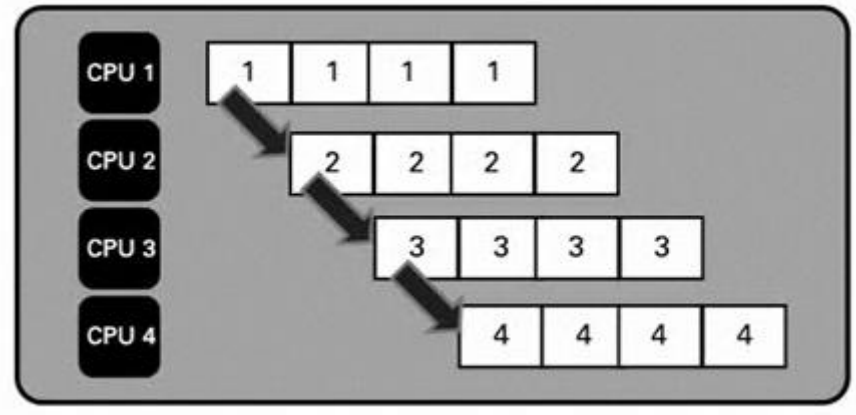


Figura 2 - Abordagem Pipeline

A chave do desmembramento do algoritmo em etapas que levem a mesma quantidade de tempo para ser executadas, uma vez que a etapa que leva mais tempo determinará o tempo global de cada iteração. Por exemplo, se o passo 2 leva um minuto para ser executado, mas os passos 1,3 e 4 levam 10 segundos cada, cada iteração terá uma duração de um minuto.

Visão geral

Quando desenvolvemos aplicações multicore, considerações especiais devem ser feitas para aproveitar o poder dos processadores de hoje. Este documento discute o pipelining, uma técnica que pode ser usada para aumentar o ganho de desempenho (em processadores multicore) quando executam tarefas sequenciais inerentes.

1. Sumário

Atualmente, com processadores multicore e aplicações multithread, os programadores necessitam pensar constantemente sobre qual melhor maneira de aproveitar ao máximo as CPUs quando estão desenvolvendo suas aplicações.

Embora estruturar um código em paralelo em uma linguagem tradicional baseada em texto pode ser difícil de se programar e visualizar, ambientes de desenvolvimento gráfico, que permitem cada vez mais aos engenheiros e cientistas diminuir o tempo de desenvolvimento e executar rapidamente suas idéias.

O fato de se ter paralelismo inerente (baseada em fluxo de dados), programar aplicações multithread é tipicamente uma tarefa muito simples. Tarefas independentes do diagrama de blocos executam paralelamente sem nenhum trabalho extra necessário por parte do programador.

Mas, e as partes do código que não são independentes? Quando implementamos uma programação sequencial inerente, o que fazer para aproveitar o poder dos processadores multicore?

2. Introdução ao Pipelining

Uma técnica muito utilizada para aumentar o desempenho de tarefas sequenciais de software é o Pipelining. Pode se dizer que Pipelining é o processo que divide tarefas sequenciais em estágios distintos que podem ser executados no modelo de linha estruturada.

Considere o seguinte exemplo: suponha que você está produzindo carros em uma linha de produção automatizada.

Sua tarefa final é construir um carro completo, mas você pode separar isso em três estágios distintos: construção da armação, instalação de peças internas (como o motor), e pintura do carro quando terminado.

Levando em conta que a contrução da armação, instalação das peças, e pintura levem uma hora cada parte serem concluídas, então, se você construir apenas um carro de cada vez, cada carro levará três horas para ficar pronto (veja a Figura 1 abaixo).

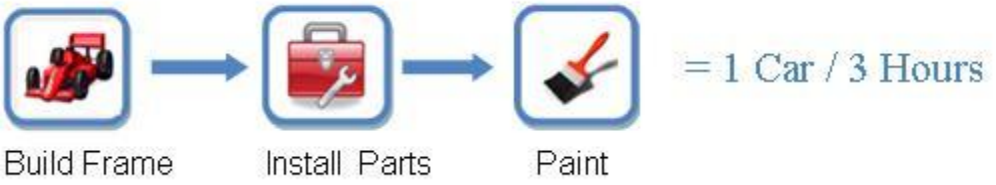


Figura 3. Nesse exemplo (sem Pipeline), a contrução do carro leva 3 horas para ser concluída.

Como esse processo pode ser melhorado?

Se fosse criado uma estação para a construção da armação, outra para a instalação das peças, e uma terceira pra pintura, agora quando um carro estiver

sendo pintado, o segundo carro pode estar instalando as peças, e o terceiro carro pode estar na construção da armação.

3. Como o Pipelining Melhora o Desempenho

Embora cada carro ainda leve três horas para ser finalizado usando o novo processo, nós podemos agora produzir um carro a cada hora, melhor que um a cada três horas - uma melhoria de 3x no processo de produção do carro. Note que este exemplo foi bem simplificado para propósitos de demonstração; veja a seção de considerações Importantes abaixo para mais detalhes sobre Pipelining.

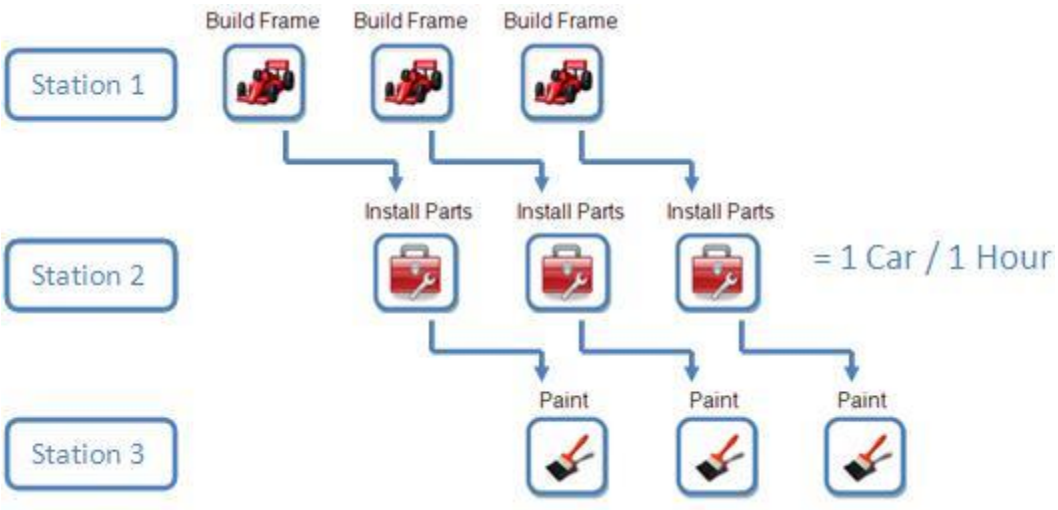


Figura 4 . Pipelining pode aumentar muito o rendimento da sua aplicação

4. Pipelining Básico

O mesmo conceito de pipelining como visto no exemplo do carro, pode ser usado em várias aplicações, onde você estará executando uma tarefa sequencial.

A seguinte ilustração mostra como uma aplicação pipelining pode funcionar em diferentes núcleos (cores):

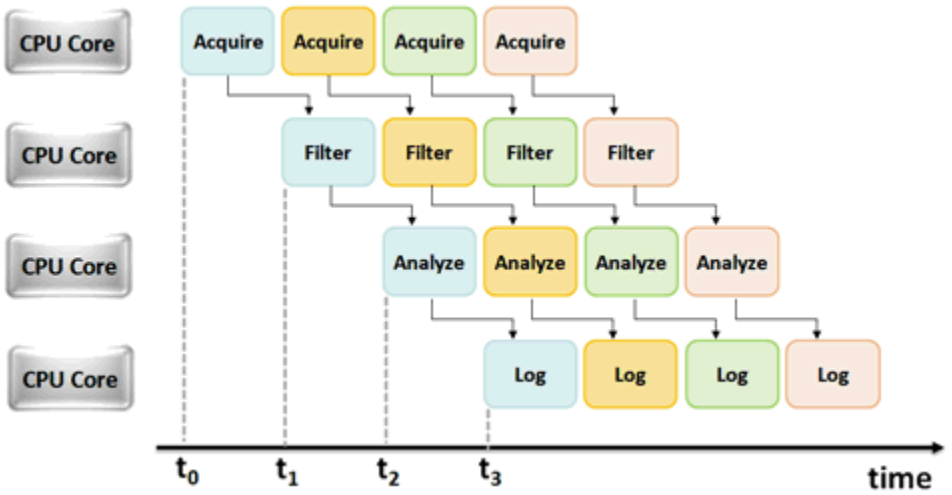


Figura 3. Carta dos tempos para aplicação pipelining funcionando em cores diferentes.

Considerações Importantes

Quando são criadas aplicações multicore usando pipelining, o programador deve levar em conta diversos parâmetros importantes. Em especial, equillibrar os estágios em pipelining e minimizar a transferência de memória entre os cores são fatores importantes para aumentar o desempenho com o pipelining.

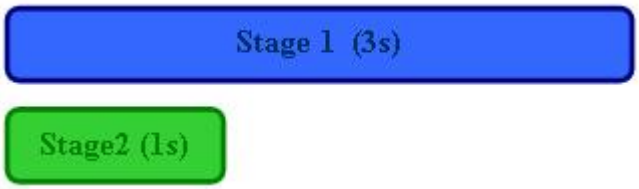
Balanceando Estágios

Em ambos os processos citados acima, na produção do carro e em outros exemplos do LabVIEW, cada estágio pipelining assumiu o mesmo montante de tempo de execução; e podemos dizer que este exemplo de estágios de pipeline estavam equilibrados.

Entretanto, em aplicações reais isso raramente acontece. Considere o diagrama a seguir: se o Estágio 1 toma três vezes o tempo do Estágio 2, então os dois estágios produzem um aumento pequeno de desempenho.

Sem Pipeline (tempo total = 4s):

Com Pipeline (tempo total = 3s):



Obs: Aumento de desempenho = 1.33X (caso não ideal para pipelining)

Para melhorar essa situação, o programador deve mover as tarefas para o Estágio 1 e o Estágio 2 até que ambos os estágios demorem aproximadamente o mesmo tempo de execução. Com um grande números de estágios no pipeline, essa pode ser uma tarefa difícil.