Objetivos:

- Perceber a organização do sistema do computador (a sua arquitetura) para desenvolver algoritmos, programas e estruturas de dados eficientes.
- Profiling e medição da execução.

Onde:

- Na hierarquia de memória: em multi-level caches.
- Em código sequencial com ILP: pipelining, super-escalares sem ordem de execução, processamento vetorial.
- Em código com thread paralelismo: multithreading dentro do core, em vários cores e em múltiplos dispositivos.
- Em código com paralelismo entre processos: multiprocessing no cluster.

Níveis de paralelismo

Instrução (ILP):

- Execução de múltiplas instruções de um programa em paralelo.
- Processamento vetorial.
- Limitado pelas dependências de dados/controlo do programa.

Tarefas/fios de execução (threads):

- Múltiplos fluxos de instruções de um mesmo programa executam em paralelo.
- Limitado pelas dependências e características do programa.

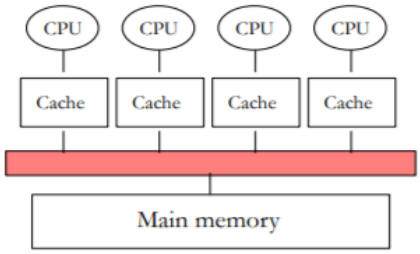
Processos:

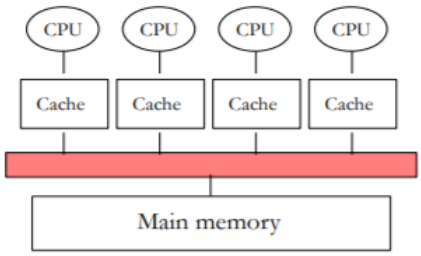
- Múltiplos processos de um mesmo programa ou de vários programas.

Memória partilhada VS Memória distribuída

Memória partilhada:

- Processadores partilham um canal de acesso à memória.
- Caches reduzem o tráfego e a latência nos acessos à memória.
- Largura de banda de acesso à memória partilhada por vários UPs: limitação à escalabilidade deste tipo de arquitetura.





Memória distribuída:

- Cada processador contém a sua memória, existindo uma rede de interligação entre os processadores.

Sistemas híbridos:

- Acesso não uniforme à memória (NUMA).

Programa

Desempenho “single core”:

- Medição de desempenho e métricas.
- Hierarquia de memória.
- Otimização para paralelismo ao nível da instrução.

Programação de sistemas de memória partilhada:

- Modelos baseados em fios de execução.
- Medição e otimização de desempenho.

Sistemas de memória distribuída:

- Modelo de processos comunicantes.
- Desenho de aplicações.

Algoritmos paralelos:

- Análise da complexidade e escalabilidade.

Desempenho do CPU

Requer um modelo que relacione o desempenho com as características do sistema de computação.

- Um programa numa máquina executa num determinado número médio de ciclos de relógio: #clock cycles.
- O período do relógio do CPU é constante: $T_{cc} = \frac{1}{f}$.

$$\mathrm{T_{exec}} = \#\text{clock cycles} * T_{cc}$$

De que depende o número médio de ciclos necessários para executar um programa?

- Número médio de ciclos necessários para executar uma instrução: CPI.
- Número de instruções executadas de um programa: #I.

$$\#\text{clock cycles} = \mathrm{CPI} * \#\mathrm{I}$$

$$\mathrm{T_{exec}} = \#\text{clock cycles} * T_{cc} = \mathrm{CPI} * \#\mathrm{I} * T_{cc} = \mathrm{CPI} * \frac{\#I}{f}$$

O paralelismo ajuda a reduzir significativamente o CPI.

- #I depende do algoritmo, do compilador e da arquitetura (ISA).
- CPI depende da arquitetura (ISA), da mistura de instruções efetivamente utilizadas, da organização do processador e da organização das restantes componentes do sistema (por exemplo, a memória).
- A frequência depende da organização do processador e da tecnologia utilizada.

Desempenho do CPU – MIPS

MIPS (milhões de instruções por segundo) – uma métrica enganadora.

$$MIPS = \frac{\#I}{T_{exec} * 10^6}$$

- MIPS especifica a taxa de execução das instruções, mas não considera o trabalho feito por cada instrução. CPUs com diferentes instruction sets não podem ser comparados.
- MIPS varia entre diferentes programas do mesmo CPU.

- MIPS pode variar inversamente com o desempenho.

Esta métrica pode ser usada para comparar o desempenho do mesmo programa em CPUs com o mesmo conjunto de instruções, mas micro-arquiteturas e/ou frequências do relógio diferentes.

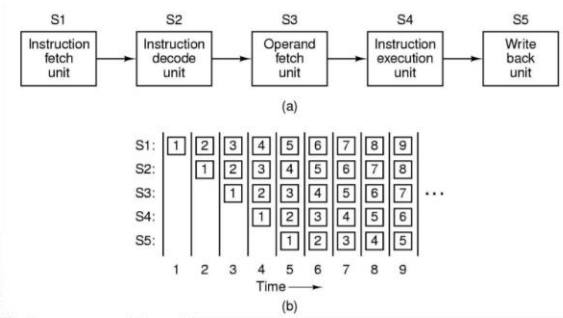
É a métrica mais enganadora, pois corresponde a sequências de código que apenas tenham instruções com o CPI mais baixo possível.

Este tipo de sequências de instruções não realizam, regra geral, trabalho útil. Consistem apenas em operações elementares com operandos em registos.

Pode ser visto como a velocidade da luz do CPU e, portanto, inatingível.

O principal problema é que é muitas vezes publicitada pelos fabricantes/vendedores como uma medida de desempenho das suas máquinas.

Paralelismo no processador – exemplo de pipeline



Objetivo: CPI=1.

Problemas:

- Dependência de dados.
- Latência nos acessos à memória.
- Saldos condicionais.

Propostas de solução para minimizar as perdas:

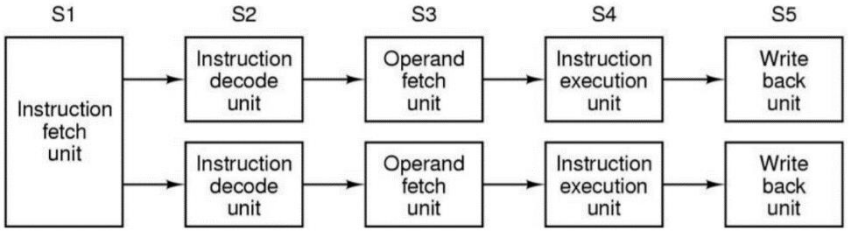
- Executar sempre a instrução que se segue.
- Usar o historial dos saltos anteriores (1 ou mais bits).
- Executar os dois percursos alternativos até à tomada de decisão.

O pipelining melhora a performance aumentado a taxa de transferência de instruções: executa múltiplas instruções em paralelo e cada instrução tem a mesma latência.

Sujeito a perigos: estrutura, dados e controlo.

O design do instruction set afeta a complexidade da implementação do pipeline.

Paralelismo no processador – exemplo de superescalaridade

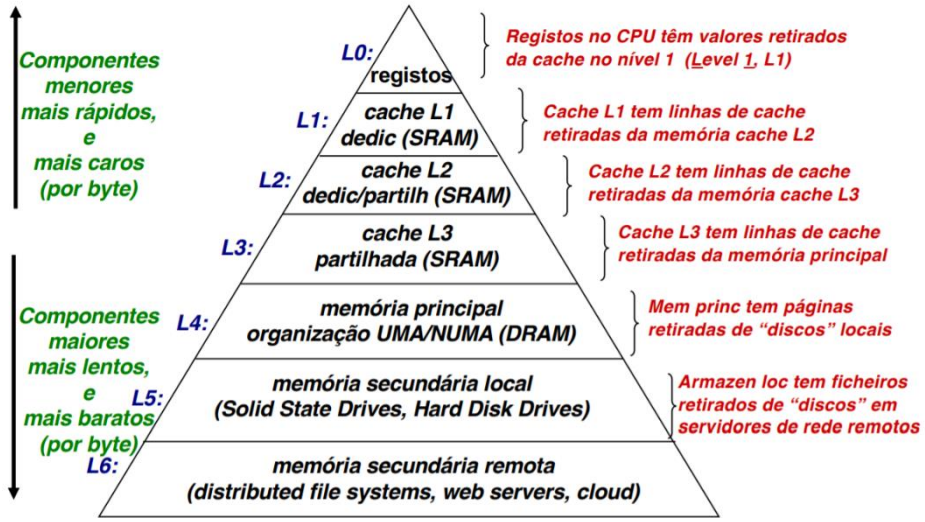


Executa duas instruções ao mesmo tempo (duas vias).

Funciona, mas não tão bem quanto esperávamos. Os programas têm dependências reais que limitam o ILP (Instruction Level Paralelism).

- Algumas dependências são difíceis de eliminar (por exemplo, pointer aliasing).
- Algum paralelismo é difícil de expor (tamanho da janela limitada durante um problema na instrução).
- Atrasos na memória e largura de banda limitada.

Organização hierárquica da memória



Notas:

- As caches L1 de dados e de instruções são normalmente distintas.
- As caches L2 em multicores podem ser partilhadas por outras cores.
- Muitos cores partilhando uma única memória traz complexidade: manutenção da coerência da informação nas caches; encaminhamento e partilha dos circuitos de acesso à memória.

Hiato Processador – Memória

Para cada instrução:

1. Ler instrução.
2. Ler operando.
3. Escrever resultado.

Suponhamos um processador a executar um programa que consiste numa longa sequência da instruções inteiras:

addl reg, [Mem]

- addl – 6 bytes
- reg – 4 bytes
- Mem – 4 bytes

Se cada instrução tiver 6 bytes de tamanho e cada inteiro 4 bytes, a execução de cada instrução implica 6+2*4=16 bytes.

Se a frequência for 2.5GHz e o CPI=1, então são executadas 2.5*10^9 instruções/segundo.

A largura de banda necessária para manter o processador alimentado é de:

$$2.5 * 10^9 * 16 = 40\text{GB/s}$$

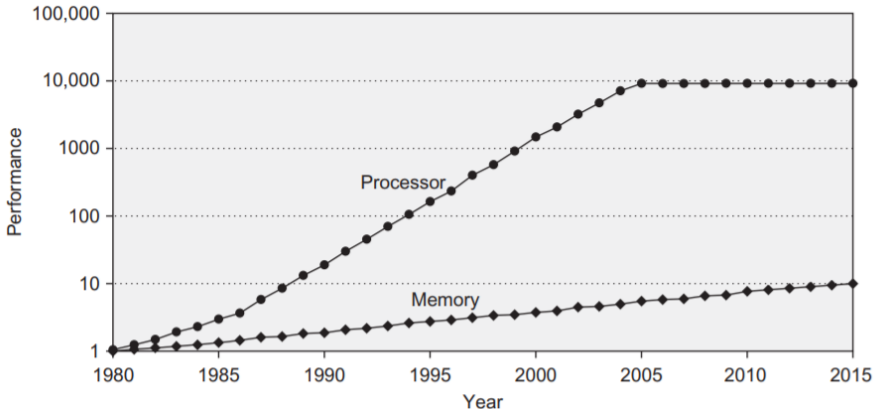
Quando temos multicores, o pico da largura de banda de todos os cores cresce com o número de cores.

$$f * \text{CPI} * \#I = \text{largura de banda ou débito}$$

Standard name (single channel)	Peak transfer rate
DDR2-800	6 400 MiB/s
DDR3-1066	8 533 MiB/s
DDR3-1600	12 800 MiB/s
DDR4-2666	21 800 MiB/s

Com esta tabela, vemos que a memória é incapaz de alimentar o processador com instruções e dados a uma taxa suficiente para o manter constantemente ocupado.

As diferentes taxas de desempenho destes dois componentes levam a um aumento do hiato (separação) Processador – Memória ("the memory gap") com o tempo.



O hiato é um dos principais obstáculos à melhoria do desempenho dos sistemas de computação.

Soluções: hierarquia de memória

- Latência VS largura de banda (débito): a evolução tem privilegiado a melhoria da largura de banda em detrimento da latência, isto é, aumentando a largura de banda, poderemos ter mais bytes a serem processados por segundo, o que vai diminuir o tempo para chegar ao local da memória a que queremos aceder (latência).

Latência: tempo que demora a aceder à informação requisitada.

Um registo é uma word na cache (load/store).

Uma cache é uma linha da memória primária (transferido automaticamente numa cache miss).

Uma linha da memória primária é uma página da memória secundária (transferido automaticamente quando surge um page fault).

Hierarquia da memória – Localidade

Os programas tendem a aceder a uma porção limitada de memória num dado período de tempo.

O princípio da localidade permite utilizar memória mais rápida para armazenar a informação usada mais frequentemente/recentemente. Também permite tirar partido da largura de banda, uma vez que a informação transferida entre diferentes níveis da hierarquia é efetuada por blocos.

Localidade temporal

Se um elemento de memória é acedido pelo processador, será, com grande probabilidade, acedido de novo num futuro próximo.

Exemplos: tanto as instruções dentro dos ciclos, como as variáveis usadas como contadores de ciclo, são acedidas repetidamente em curtos intervalos de tempo.

Consequência: a primeira vez que um elemento de memória é acedido, deve ser lido do nível mais baixo (por exemplo, a memória central). No entanto, da segunda vez que é acedido, existem grandes hipóteses que se encontre na cache, evitando-se o tempo de leitura da memória central.

Localidade espacial

Se um elemento de memória é acedido pelo CPU, então elementos com endereços na proximidade serão, com grande probabilidade, acedidos num futuro próximo.

Exemplos: as instruções do programa são normalmente acedidas em sequência, assim como, na maior parte dos programas, os elementos de vetores/matrizes.

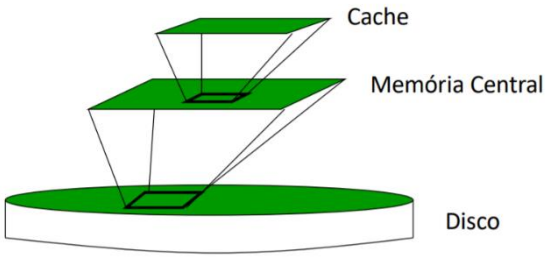
Consequência: a primeira vez que um elemento de memória é acedido, deve ser lido no nível mais baixo (por exemplo, memória central). Não apenas esse elemento, mas sim um bloco de elementos com endereços na sua vizinhança. Se o processador, nos próximos ciclos, acede a um endereço vizinho do anterior, aumenta a probabilidade de esta estar na cache.

Hierarquia da memória – Inclusão

Os dados contidos num nível mais próximo do processador são um sub-conjunto dos dados contidos no nível anterior.

O nível mais baixo contém a totalidade dos dados.

Os dados são copiados entre níveis em blocos.



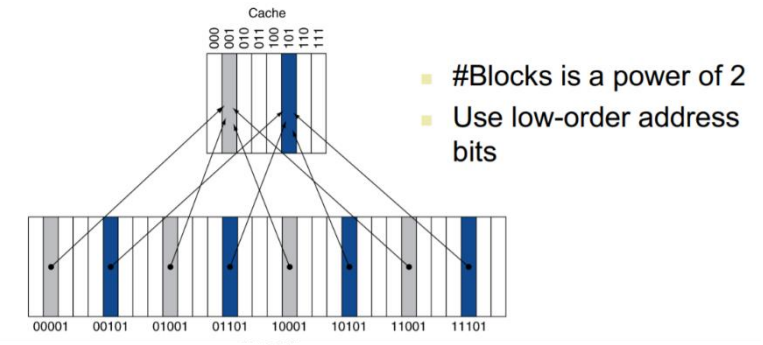
Hierarquia da memória – Terminologia

- Linha: a cache está dividida em linhas. Cada linha teu o seu endereço (índice) e tem a capacidade de um bloco.
- Bloco: quantidade de informação que é transferida de cada vez da memória central para a cache (ou entre níveis na cache). É igual à capacidade da linha.

- Hit: diz-se que ocorreu um hit quando um elemento é encontrado na cache.
- Miss: diz-se que ocorreu um miss quando o elemento de memória acedido pelo CPU não se encontra na cache, sendo necessário lê-lo do nível inferior da hierarquia.
- Hit rate: percentagem de hits ocorridos relativamente ao total de acessos à memória.
- Miss rate: percentagem de misses ocorridos relativamente ao total de acessos à memória (Miss rate=1-hit rate).
- Hit time: tempo necessário para aceder à cache, incluindo o tempo necessário para determinar se elemento a que o CPU está a aceder se encontra ou não na cache.
- Miss penalty: tempo necessário para carregar um bloco da memória central (ou de um nível inferior) para a cache quando ocorre um miss.

Mapeamento direto

A localização onde a informação que vem da memória central, ou nível inferior, tem de ir na cache é determinada por um endereço, que é o índice da linha do nosso bloco. Assim, só há uma única escolha. Tem o menor tempo de acesso.



Caches associativas

São caches que permitem que o bloco proveniente de um nível inferior se aloque em qualquer sítio e entre por qualquer entrada. Daí temos de ter o número de vias de entrada (n-way). Se aumentarmos o número de n-way, diminuímos a nossa miss rate, isto é, porque quantas mais n-ways, maior a nossa cache. Contudo, tem maior custo, complexidade e tempo de acesso.

Política de escrita

Write-through:

- Ambos os níveis são atualizados (superior e inferior).
- Simplifica a substituição, mas pode precisar de um buffer.

Write-back:

- Atualiza apenas o nível superior.
- Atualiza o nível inferior quando o bloco for substituído.
- Precisa de manter a condição.

Memória virtual:

- Apenas write-back é praticável, dada a latência na escrita do disco.

Hierarquia da memória – básicos

$$\text{CPU}_{\text{exec-time}} = \left(\text{CPU}_{\text{clock-cycles}} + \text{Mem}_{\text{stall-cycles}}\right) * \text{Clock cycle time}$$

$$\text{CPU}_{\text{exec-time}} = \left(\text{IC} * \text{CPI}_{\text{CPU}} + \text{Mem}_{\text{stall-cycles}}\right) * \text{Clock cycle time}$$

Single-level cache:

$$\text{Mem}_{\text{stall-cycles}} = \text{IC} * \ldots \text{Miss rate} \ldots \text{Mem acess} \ldots \text{Miss penalty} \ldots$$

$$\text{Mem}_{\text{stall-cycles}} = \text{IC} * \text{Misses/Intruction} * \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate} * \text{Memory accesses}}{\text{Instruction count}} = \text{Miss rate} * \frac{\text{Memory acesses}}{\text{Instruction}}$$

Por cada nível de cache adicional:

$$\text{Mem\_acesses}_{\text{level\_i}} = \frac{\text{Misses}}{\text{Instruction}_{\text{leveli}-1}}$$

$$\text{Miss\_penalty}_{\text{level\_i}} = \left(\text{Hit}_{\text{rate}} * \text{Hit}_{\text{time}} + \text{Miss}_{\text{rate}} * \text{Miss}_{\text{penalty}}\right)_{\text{level}_i+1}$$

Hierarquia da memória – Desempenho

Como é que a hierarquia de memória influencia o tempo de execução?

Cada acesso à memória vai originar ciclos adicionais na execução do programa (#CC_Mem) devido aos misses:

$$\text{T}_{\text{exec}} = \left(\#\text{CC}_{\text{CPU}} + \#\text{CC}_{\text{Mem}}\right) * \text{T}_{\text{CC}}$$

Cada miss implica um aumento do #CC em misspenalty ciclos, logo:

$$\#\text{CC}_{\text{Mem}} = \text{nºmiss} * \text{miss penalty}$$

Onde, nº miss=miss rate* nº acessos à memória.

$$\text{T}_{\text{exec}} = \left(\#\text{CC}_{\text{CPU}} + \#\text{CC}_{\text{Mem}}\right) * \text{T}_{\text{CC}}$$

Como #CC=#I*CPI, temos:

$$\text{T}_{\text{exec}} = \#\text{I} * \left(\text{CPI}_{\text{CPU}} + \text{CPI}_{\text{Mem}}\right) * \text{T}_{\text{CC}}$$

CPI_{CPU} : número de ciclos que o processador necessita, em média, para executar cada instrução. O hit time considera-se incluído no CPI_{CPU}.

CPI_{Mem} : número de ciclos que o processador para, em média, à espera de dados da memória, porque não encontrou estes dados na cache. Estes são vulgarmente designados por memory stall cycles ou wait states.

$$\text{CPI}_{\text{Mem}} = \%\text{acessosMem} * \text{missrate} * \text{misspenalty}$$

Os acessos à memória devem-se a:

- Acessos a dados (instruções de Load e Store do programa).
- Busca de instruções.

Como estes têm comportamentos diferentes, usam-se diferentes percentagens:

- Dados – apenas uma determinada percentagem de instruções acede à memória (%Mem). Missrate_D refere-se ao acesso a dados.
- Instruções – todas as instruções são lidas da memória, logo a % de acesso à memória é de 100%. Missrate_I refere-se ao acesso às instruções.

Missrate_I é, geralmente, menor do que missrate_D devido à localidade espacial.

$$\text{CPI}_{\text{Mem}} = (\text{missrate}_{\text{I}} + \%\text{Mem} * \text{missrate}_{\text{D}}) * \text{misspenalty}$$

Otimização de código sequencial (ILP)

Consideremos, por exemplo, as capacidades do Intel P6 para a execução de várias instruções em paralelo:

- 2 integer (1 pode ser branch).
- 1 FP Add.
- 1 FP Multiply or divide.
- 1 load.
- 1 store.

Algumas instruções requerem mais do que um ciclo de clock, mas podem ser pipelined:

Instruction	Latency	Cycles/Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

Nota: é útil fazer o inverso da multiplicação para obtermos a divisão.

Exemplo de 3 iterações do mesmo ciclo combine:

- Com recursos ilimitados, conseguimos executar a função com paralelismo e pipeline, executando as instruções fora de ordem e de forma especulativa.
- Em termos de performance, vemos que o fator limitador é a latência da multiplicação de integer. Logo, CPE=4 (clock cycles por elemento).

Exemplo de 4 iterações do add cycle em combine:

- Com recursos ilimitados, podemos iniciar uma nova iteração a cada ciclo de clock, obtendo um CPE=1. Isto é conseguido porque se faz 4 operações em paralelo sobre integer.
- Na realidade, conseguimos apenas executar 2 operações sobre integer ao mesmo tempo, o que implica que algumas operações têm de esperar até os operandos estarem disponíveis. A prioridade é, contudo, executar o código por ordem. CPE=2.

Técnicas de otimização dependentes da máquina

Loop unroll: dentro da nossa função vamos fundir n interações dentro de um único loop cycle, O que isto vai fazer é reduzir para 1/n o número de iterações que são precisas fazer.

- Performance esperada: 1 iteração fosse completa em 3 ciclos. CPE=1.
- Performance real: 1 iteração a cada 4 ciclos. CPE=1.33.

Na realidade, os loop unrolls não melhoram linearmente os resultados. Contudo, nas operações de adição de integer os resultados melhoram. Em todas as outras operações, a execução está limitada à latência.

Loop unroll com paralelismo: considerando o exemplo, cada produto dentro do ciclo não depende um do outro, logo podem ser pipelined – iteration splitting. Há uma acumulação em 2 variáveis e ,no fim, elas são unidas. CPE esperado =2.0 (a unidade de multiplicação é ocupada com 2 operações em simultâneo.

```
void combine6(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  int *data = get_vec_start(v);
  int x0 = 1;
  int x1 = 1;
  int i;
  /* junta 2 elem's de cada vez */
  for (i = 0; i < limit; i+=2) {
    x0 *= data[i];
    x1 *= data[i+1];
  }
  /* completa os restantes elem's */
  for (; i < length; i++) {
    x0 *= data[i];
  }
  *dest = x0 * x1;
}
```

No geral, o uso destas técnicas varia em função do tipo do programa em questão; das suas operações.

Multithreading

Execução de vários fios/threads em paralelo.

1. Grão fino/time-multiplexed:
   - Troca de fio após cada ciclo.
   - Execução de instruções intercaladas.
   - Se um fio empata, os outros são executados.
2. Grão grosso multithreading:
   - Só troca de thread quando uma thread tem um long stall (por exemplo, L2 cache miss).
   - Simplifica o hardware, mas não esconde short stalls (por exemplo, perigo dos dados).
3. Multithreading simultâneo: em múltiplos problemas, agenda dinamicamente as threads no processador.
   - Agenda instruções de múltiplas threads.
   - As instruções de fios independentes são executadas quando as unidades de função estão disponíveis.
   - Dentro de threads, as dependências são lidadas através do agendamento e renomear os registos.
   - Por exemplo, Intel from Pentium-4 HT. 2 fios: registos duplicados, partilham unidades de funcionamento e caches.

Data paralelismo

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

SPMD: Single Program Multiple Data

- Um programa em paralelo num computador MIMD.
- O código é sequencial para diferentes processadores.

A arquitetura SIMD pode explorar um grande número de data a vários níveis de paralelismo para:

- Computação científica matrix orientada.
- Processamento de imagem e som media orientada.

SIMD é mais energeticamente eficiente do que MIMD:

- Apenas vai buscar uma instrução por cada operação de data.
- Faz SIMD atrativa para uso pessoal de tecnologia móvel.

SIMD permite que os programadores continuem a pensar sequencialmente.

SIMD: Single Instruction Stream with Multiple Data.

MIMD: Multiple Instruction Stream with Multiple Data.

Paralelismo SIMD

- Arquitetura de vetores.
- SIMD e extensões.
- Graphics Processor Units (GPU).

Arquitetura de vetores

Ideia básica:

- Lê set de data elements (retirado da memória) em registos vetoriais.
- Opera sobre esses mesmos registos.
- Armazena ou distribui os resultados de volta à memória.

Os registos são controlados por um compilador:

- São usados para esconder latência de memória.
- Aproveita a largura de banda da largura de memória.

Programação de vetores:

- Compiladores são o elemento chave para dar dicas se a secção de código será ou não vetorizada.
- Vê se as iterações num loop têm dependência de data. Em caso contrário, a vetorização fica comprometida.

Arquiteturas de vetores têm um grande custo, mas variáveis mais simples estão agora disponíveis para dispositivos fáceis de encontrar, como extensões do processador escalar.

Extensões SIMD

As aplicações dos meios de comunicação operam em tipos de data mais estreitos do que o tamanho normal utilizado pelo mundo.

Limitações comparativamente com arquitetura de vetores:

- Um número de data operands está codificada em op code.
- Não há modos de endereçamento sofisticados.

- Não há registos de máscaras.

Nota: os operandos têm/devem estar em memória consecutiva e alinhado.

Programar em memória partilhada

Processos:

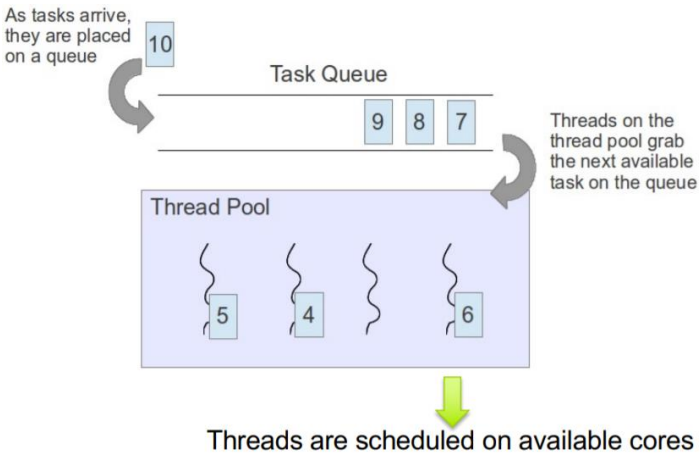
- Usados para tarefas que não estão relacionadas.
- Tem um espaço próprio de endereçamento (este espaço é protegido de outros processos).
- Swithching at the kernel level (cada processo tem data, files, registos, código e stack).

Threads:

- São parte do mesmo trabalho.
- Partilham um espaço de endereçamento, código, data e ficheiros.
- Swithching at the use ror kernel level (partilham data, files, registos, código e stack).

Cada processo tem pelo menos uma thread.

- Tarefa: sequência de instruções.
- Thread/process: contexto de execução para uma tarefa.
- Processador/core: hardware que corre threads/processos.



Nota:

- OMP: memória partilhada, dentro do processador.
- MPI: memória distribuída, fora do processador.
- GPU: comunicação entre processadores (interface).

Modelos de programação em paralelo

Tarefas em paralelo: o problema é partido em tarefas para ser executado; tarefas individuais são criadas e comunicam entre si para coordenar operações.

Data em paralelo: o problema é visto como operações de dados em paralelo; a data é distribuída por processos e computados localmente.

Característica de “escalar” programas em paralelo:

- Decomposição do domínio de dados para melhorar a localidade da data.
- Comunicação e latência não aumentam significativamente.

Introdução ao OpenMP

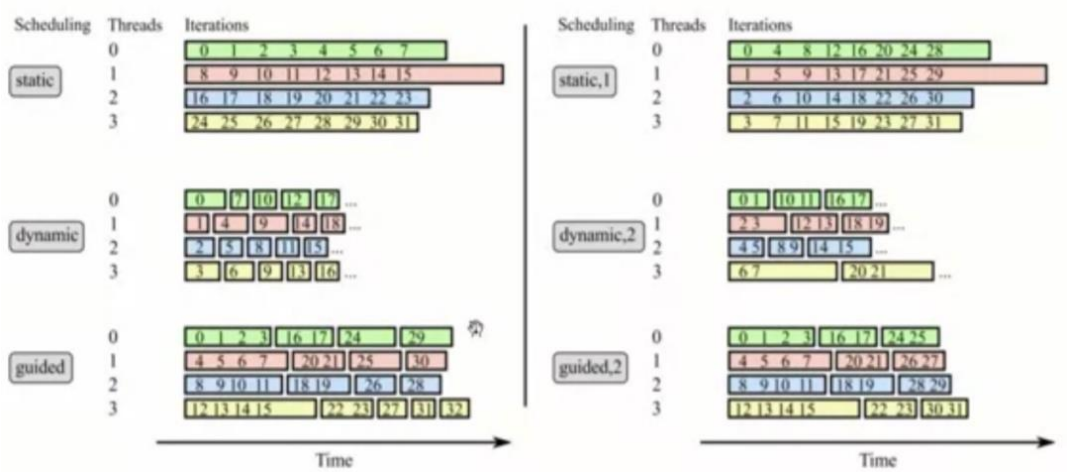
Se não tivermos o número de threads este é definido pelo número de cores disponíveis.

Para evitar race conditions usa-se:

- #pragma omp critical.
- #pragma omp atomic (não pode ser aplicado num conjunto de instruções; melhor do que critical).
- #pragma omp reduction (melhor do que atomic).

Loop Schedule:

- Static: as tarefas são divididas por blocos de tamanho n e executadas por essa exata ordem. Cada fio executa igual número de tarefas.
- Dynamic: as tarefas são distribuídas conforme a sua execução, isto é, mal uma thread acabe de executar uma tarefa, outra ser-lhe-á atribuída efetivamente (avisa o distribuidor). O fio mais rápido executa mais tarefas.
- Guided: igual à dinâmica, contudo, são distribuídos pedaços de tarefas que vai gradualmente diminuir.



Parallel Construct

- #pragma omp parallel: cria uma equipa de threads.

Work-sharing Constructs

- #pragma omp for: atribuição de iterações em loop a cada thread, sem distribuição da carga (fazem todas a mesma coisa).
- #pragma omp sections: atribuição de blocos de código de cada thread (secções).
- #pragma omp single: restringe um bloco de código para ser executado por uma única thread (executa a thread que chegar primeiro).

Tasking Constructs

- #pragma omp task: cria uma piscina de tarefas para serem executadas pelas threads.

Master and Synchronization Constructs

- #pragma omp master: restringe um bloco de código para ser executado pela master thread.

- #pragma omp critical: restringe a execução de um bloco de código a uma única thread de cada vez. Não há divisão de carga, há sincronização, o output é sempre o mesmo entre #pragma omp parallel e for.
- #pragma omp barrier: faz com que todas as threads sejam uma equipa para esperar pela restante. Dentro do loop depois do print. Executa uma thread de cada vez sem distribuição.
- #pragma omp taskwait: espera pela conclusão da tarefa atual.
- #pragma omp atomic: garante que um específico local de armazenamento seja gerenciado atomicamente.
- #pragma omp flush: identifica um ponto em que o compilador garante que toda as threads numa região paralela tenham a mesma visão de objetos especificados na memória.
- #pragma omp ordered: especifica um bloco de código dentro do loop que as iterações serão executadas por ordem. Dentro do loop antes do print. Há distribuição da carga.

Nested Parallel Region

Quando uma thread encontra uma região em paralelo, ela vai criar uma nova equipa de threads, sendo ela a master da nova equipa.

Para ativar/desativar: omp_set_nested(x).

Loop Construct

As iterações num for loop são distribuídas ao longo de threads na equipa. A distribuição é baseada em:

- Chunk_size, por default é 1.
- Schedule, por default é static.

Loop Schedule

- Static: as iterações são divididas em chunk de tamanho n pelas threads da equipa numa round-robin fashion.

Se colocarmos, por exemplo, #pragma parallel for Schedule (static, 10): cada fio executa 10 iterações de cada vez, sendo geradas aleatoriamente (Round-robin).

- Dynamic: os pedaços são distribuídos pelas threads na equipa à medida que todas vão pedindo. Diminui o tempo de execução em relação a static.
- Guided: semelhante a dynamic, mas tamanho do pedaço diminui durante a execução. Pode ser melhor que a dynamic.
- Auto: OpenMP decide a implementação.

Data sharing

Variáveis declaradas dentro das regiões paralelas são locais a cada thread.

Variáveis declaradas fora dessa região são variáveis partilhadas.

Data sharing clauses

- Private (varlist): cada variável em varlist torna-se privada para cada thread. O valor inicial não é especificado; inicialização própria.
- Firstprivate (varlist): o mesmo que private, mas as variáveis são inicializadas com um valor fora do loop.

- Lastprivate (varlist): o mesmo que private, mas o valor final é o último valor da iteração no último loop. Admite inicialização inicial.
- Reduction (op:var): o mesmo que lastprivate, mas o valor final é o resultado da redução dos valores do private usando o operador "op". Por exemplo, reduction(+:w).

Diretivas para data sharing:

- #pragma omp threadlocal: cada thread tem uma cópia local do valor.
- Copyni clause copia o valor que está na master thread para todas as outras threads.

Otimização da Performance com OpenMP

As medidas mais comuns nas aplicações paralelas são: tempo de execução e speedup.

Strong scability analysis:

- O speedup aumenta com o PU para um problema com o dada size fixo.
- Speedup ideal seria proporcional ao PU.

Weak scability analysis:

- Aumentando o data size do problema também se aumento o número de PU.
- Idealmente, o tempo de execução deveria manter-se constante.

Amdahl's law (strong scability analysis)

Mede o tempo de execução da versão paralela à medida que o número PU aumenta e mede também o tempo de execução da versão sequencial. Ao comparar os dois, vemos que a fração do trabalho não paralelizável (trabalho sequencial) limita o speedup.

$$S_P = \frac{1}{f + (1-f)/P}$$

Onde, P é o número de PU e f a fração que limita (série).

Nota: em superlinearidade, o speedup é superior ao número de cores (PU) devido aos esforços da cache.

Gustafson's law (weak scability analysis)

Aumentando o tamanho do problema, aumenta-se o número do PU.

A fração do trabalho sequencial geralmente diminui com o tamanho do problema.

Hot-spots: funções que demoram grande parte do tempo de execução a serem executadas.

Scability problems in shared memory

Por que é que as aplicações de paralelismo não têm um speedup ideal?

- A percentagem de trabalho sequencial.
- A parede de memória: o acesso à memória é sequencial.
- Paralelismo/granularidade da tarefa: há trabalho adicional na paralelização (supervisionar tarefas, computações redundantes).
- Sobrecarga do sincronismo: pode também adicionar execuções sequenciais (critical); inclui chamadas externas de rotina (por exemplo, malloc).
- Desequilíbrio de carga: a decomposição em demasia pode melhorar este desequilíbrio.

Motivos para a falta de escalabilidade

Limitação de largura de banda na cache/memória.

- Resolução: otimizações locais (tirar proveito das localidades).

Paralelismo de grão fino/sobrecarga de paralelismo.

- Resolução: aumentar o grão (melhorar o multithreading).

Tarefas com sincronismo excessivo.

- Resolução: medir programas sem sincronismo ou diminuir dependências.

Má distribuição da carga.

- Resolução: computar tempo por cada tarefa paralela.

Medir a performance

Quanto tempo é necessário para medir a execução de uma aplicação?

- CPU-time: tempo dedicado exclusivamente à execução do programa; não depende de outras atividades.
- Wall time: tempo medido desde o início da execução ate ao final desta; depende do load system, I/O…
- Complexidades: processo de agendamento (10ms?); load introduzido por outros processos.

Passagem de mensagem – conceitos básicos

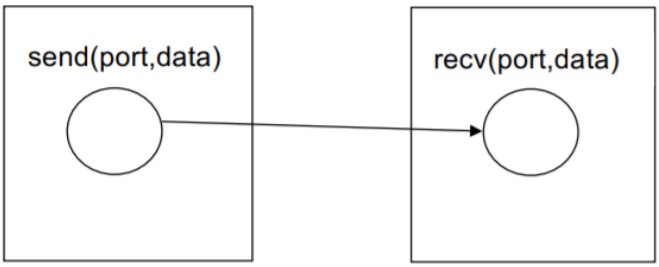
Especificações de atividades paralelas através de processos com espaço de endereçamento separados.

- Não há partilha de memória entre processos: passagem de mensagem paralelismo.
- Os processos podem ser idênticos (Single Program Multiple Data, SPMD) ou não (Multiple Instructions Multiple Data, MIMD).

As atividades paralelas comunicam através de portos/canais.

- Message send e receive é explicita (de/para o porto ou canal)

.



A data deve ir explicitamente montada em mensagens.

Há formas mais sofisticadas de comunicar entre PU, tal como broadcast, reduction e barrier.

Forma standart de aplicar MPI

Cada processo é identificado com uma determinada rank, única dentro de um grupo de processos.

Processos que fazem computações e interagem uns com os outros por explicitamente enviarem e receberem mensagens.

O MPI suporta multithreading dentro dos processos do MPI.

- MPI-1: não há conceito de memória partilhada.
- MPI-2: memória partilhada limitada.
- MPI-3: programação em memória partilhada explícita.
- MPI-4: versão atual.

Modos de comunicação:

- De ponto em ponto.
- Coletiva/em grupo.

Sincronização de comunicação.

- Síncrona (blocking).
- Assíncrona (non-blocking).

MPI: Message Passing Interface.

MPI-Send: quando esta função retorna, a informação já foi entregue ao sistema e o buffer está pronto a ser usado novamente.

MPI-Recv: espera até que a mensagem correspondente seja recebida do sistema e depois o buffer pode ser utilizado novamente.

Modos de comunicação

De ponto em ponto.

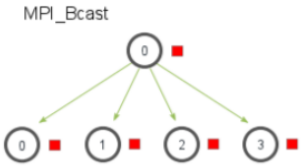
- A mensagem é transferida de rank em rank (thread em thread); tempo de transferência de mensagem equivale ao tempo de copiar para o network, do network enviar e, finalmente, de ser entregue ao recetor do buffer.

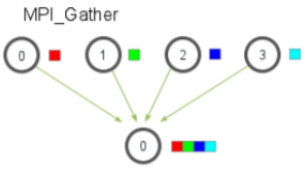
Modos de implementação do SEND (diferentes opções para sincronização e buffers).

- MPI_Send (síncrono): o que envia espera até que a mensagem seja recebida.
- MPI_Rsend (ready send): volta assim que a mensagem seja colocada na network/interface. O lado do recetor devia já ter postado MPI-Recv para evitar impasses.
- MPI_Bsend (buffer send): volta assim que a mensagem é colocada no buffer do lado do recetor. Não sofre de sobrecarga de sincronização, mas pode copiar para um buffer local.
- MPI_Ixxx (non-blocking send) com MPI_wait/MPI_Test/MPI_Probe: volta imediatamente; o programador deve verificar se a operação foi completada, usando wait.

MPI – comunicações coletivas

- MPI_Barrier: espera até o processo chegar à barreira.
- MPI_Bcast: envia a informação da raiz até aos outros processos.
- MPI_Gather: junta a informação de todos os processos para a raiz.
- MPI_Scather: espalha da raiz até todos os processos.
- MPI_Reduce: combina/junta os resultados de todos os processos para a raiz, usando o operador MPI_Op.





Tempo de execução

A medição do tempo inicia no primeiro processo (ou thread) e acaba na execução do último processo/thread (wall time).

$$T_{exec} = T_{comp} + T_{comm} + T_{free}$$

- T_comp: tempo de computação. Excluí comunicações/sincronização e tempo livre. Por exemplo, versão sequencial (execução).
- T_comm: tempo de comunicação, tempo gasto a receber/mandar mensagens. Para cada mensagem, $T_{msg} = t_s + t_b * L$, onde ts é o tempo de setup da comunicação, tb é a largura de banda da comunicação e L é a largura da mensagem (bytes).
- T_free: quando o PU fica sem trabalho. Pode ser difícil de medir uma vez que depende da ordem das tarefas.

Otimização: memória distribuída VS memória partilhada

	Memória distribuída (MPI)	Memória partilhada (OpenMP)
Distribuição de informação	Explícita	Implícita
Schedule	Static	Dynamic
Sincronização	Dispendiosa	Apenas é feita por barreiras globais e envio de mensagem

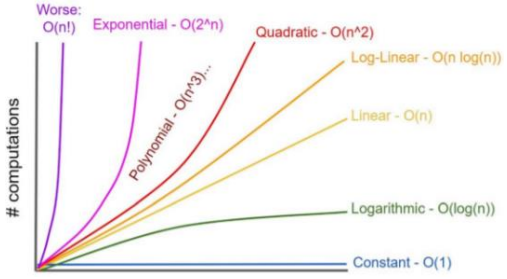
Como melhorar a escalabilidade em memória partilhada?

- Minimizar comunicações entre processos (o que duplica a computação).
- Minimizar tempo parado com uma boa distribuição.

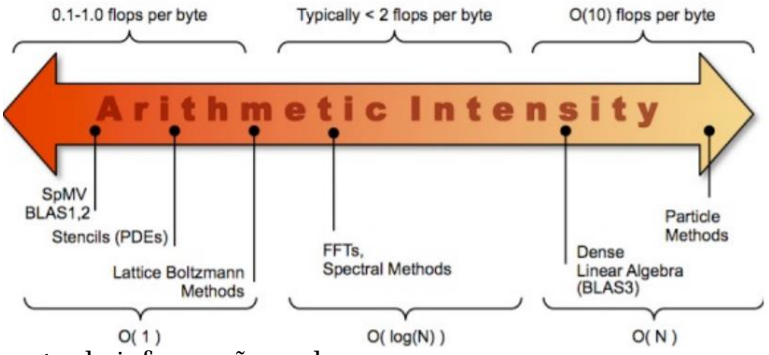
Ordenação em Paralelo

Normalmente, analisa-se os algoritmos através da notação: O.

- Multiplicação de matrizes: $O(n^{\frac{3}{2}})$, onde n é o número de elementos de C.
- Ordenação (n keys):
  - Má ordenação (força bruta): $O(n^2)$.
  - Melhor algoritmo: $n\log_2(n)$.
  - O melhor: ([k]n).
- Inserção em estrutura de dados (n inserções).
  - Lista ordenada: $O(n^2)$.
  - Árvore binária: $O(n\log_2(n))$.
  - Mesa de confusão: O(n).



Intensidade Aritmética: operações por data moves.



O armazenamento de informação pode ser:

- Regular: vetores, matrizes.
- Irregular (baseado no apontador): árvores e grafos.

Assim, o acesso à informação pode ser feito de forma regular ou irregular.

Algoritmos de ordenação sequencial

- Insertion sort: insere elementos numa lista organizada;
  Complexidade n^2.
- Bubble sort: compara e troca elementos sucessivos.
  Complexidade n^2.
- Quicksort: ordena os elementos maiores/menores que um determinado pivô, recursivamente.
  Complexidade nlog(n).
- Merge sort: une sucessivamente prquenas listas ordenadas, começando por listas de um único elemento.
  Complexidade nlog(n).
- Heap sort: insere elementos numa árvore binária.
  Complexidade nlog(n).
- Radiz sort: ordena elementos dígito por dígito (d) (variáveis: MSD e LSD, onde MSD é most significant digit e LSD é least significant digit).
  Complexidade nd (melhor).

Localidade

Radix MSG VS Radix LSD: ambos são baseados numa ordenação chave indexada.

- LSD: como começa no dígito menos significativo, ele terá mais movimentos na informação: D x N data moves (onde D é número de dígitos e N número de data). Ordena o menos significativo e vai avançando para os mais significativos. Tem de comparar sempre todos os N.
- MSD: como começa no dígito mais significativo, posteriormente, ele vai ter de comparar apenas pequenos grupos daquele dígito específico, logo terá menos movimentos: I x N(global) + local data moves (I é passos dados por todos os data). Posteriormente a ordenar o bit mais significativo, ele ordena pequenos grupo. Por exemplo, tendo 5 ele irá ordenar recursivamente apenas estes grupos não tendo de procurar em toda a data. Isto são os local data moves.

Quick-sort VS Heap-sort: ambos fazem nlog(n) de passos pela informação, mas o heap-sort mostra mais acessos irregulares aos data indexes. Quicl-sort tem um armazenamento de data regular, enquanto que o heap-sort tem um irregular.

Resumo – referência de localidade em algoritmos de ordenação

- Quicksort: tem uma boa localidade espacial, mas má localidade temporal na fase inicial.
  Melhorias: partição do set inicial usando k keys.
- Mergesort: boa localidade espacial, mas má localidade temporal na fase final da união.
  Melhorias: realizar um único merge quando a data excede o tamanho da cache.
- Heap sort: má localidade.
  Melhorias: tem uma cache "consciente" de árvores e espalhar d.
- Insertion sort: má localidade.
- Radix sort: boa localidade quando MSD é primeiro (só quando se processar LSDs).
  Melhorias: reduzir o número de passos pela data.

Paralelismo em algoritmos de ordenação

Métodos:

- Quicksort: ordena sublistas em paralelo/começa com p listas.
- Merge-sort: une listas em paralelo.
- Heap sort.
- Insertion sort.
- Radiz sort: ordena sets de dígitos em paralelo.

Em memória distribuída

As chaves são inicialmente distribuídas pelos processos.

- Fase intermédia para outros algoritmos paralelos.

Paralelismo explorável: baseados em merger e splitter.

Considerações eficientes para paralelismo:

- Data movements (através de processadores).
- Load balacing.
- Evitar tempo parado (fases sequenciais).

Parallel Merge-sort:

- Localmente ordenada cada set.
- Troca os sets entre processadores.
- Só é eficaz quando n/p ~ 1.
- Há uma extensão de data movements quando n/p>>1.

Parallel Quick-sort:

- O master escolhe e transmite a chave pivot.
- Cada processador, localmente, divide-se usando o pivot, tendo assim cada processo partições maiores e menores.
- Divide-se o processador em sets de maiores e menores. Envia data para um processador de outro set.
- Repete-se este processo até que #sets=#p.
- Ordenação local em cada processador p.
- Complexidade: requere log(p) passos de comunicação.

Parallel Radix sort:

- Cada processador é responsável por um subset de valores de dígitos.

- Ordena e conta o número de valores de dígitos.
- Todos os processadores reduzem o número total de dígitos.
- Envia chave para o processador responsável por aquele alcance de dígitos.
- Repete-se para o próximo dígito.
- Complexidade: LSG D communication steps; MSD I communication steps.

Paralelismo em algoritmos de ordenação

Sampling based:

- Divide data em p steps usando p-1 splitters.
- Cada processador age sobre um local set.
- Minimiza-se os data movements.

Sampling alternatives:

- Sampling regular (p*(p-1)keys) não é eficaz para grandes p.
- Sampling aleatório.
- Histogram sampling.

Regular sampling

1. Divide o conjunto em p sets não unidos e localmente ordena-se esses sets.
    - Aplica-se localmente um quicksort.
    - Escolhe-se p-1 local samples, que divide as samples uniformemente cada set em p subsets.
2. Ordena-se p*(p-1) samples e escolhe-se os melhores p-1 pivot.
3. Partição de cada set usando a chave pivot p-1.
4. Une-se p*p sets: o processador i merges as i partições.

GPU e CUDA

Melhor acelerador para processamento de números, ou seja, computação vetorial/de matrizes intensiva.

CPU – SIMD: Single Instruction Multiple Data.

GPU – SIMT: Single Instruction Multiple Threads.

Dado o hardware investido para haver bons gráficos, como é que o podemos complementar de modo a melhorar a performance de um maior alcance de aplicações?

Key ideas:

- Modelo de execução heterogéneo: CPU é o host e o GPU o dispositivo.
- Desenvolver um programa em C para o GPU.
- Unificar todas as formas de paralelismo no GPU como Cuda – threads.
- O modelo de programação segue SIMT.

Semelhanças do GPU com “máquinas de vetores”:

- Trabalha bem com problemas de paralelismo em data-level.
- Transferências de dispersão (espalha) – união (coleta).
- Mask register.
- Maiores ficheiros de registos.

Diferenças:

- Não tem processador escalar.
- Usa multithreading para esconder a latência de memória.

- Tem muitas unidades de funcionamento, em contraste com algumas profundas pipelined unidades como processamento de vetores.

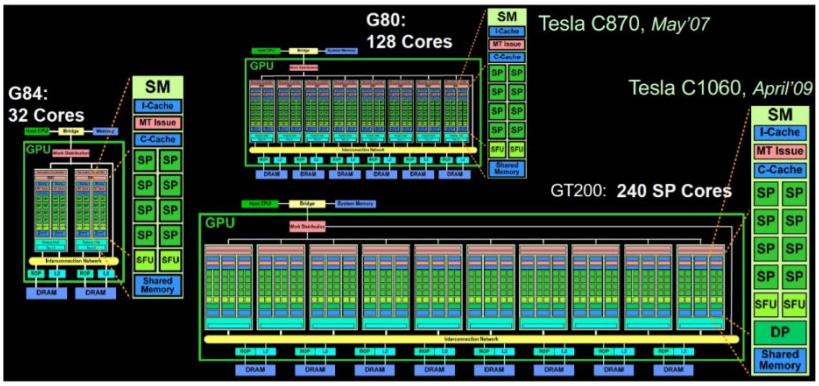
Estruturas de memória

Cada faixa de SIMD tem a sua parte privada de DRAM fora da DRAM.

- Private memory (local memory).
- Contém uma stack frame, spilling registers e variáveis privadas.

Cada multireadeed SIMD processador (SM) também tem uma memória local (shared memory) partilhada com as faixas/threads dentro de um bloco.

A memória partilhada por cada SIMD processador (SM) é um GPU memory, off-chip DRAM (global memory): o host pode ler e escrever na memória do GPU.



The CUDA programming model

Compute Unified Device Architecture.

Cuda é um programa recente, desenhado para:

- Um multicore CPU host acoplado com um dispositivo de many-core.
- Onde os dispositivos têm um grande SIMT/SIMD paralelismo.
- O host e o dispositivo não partilham memória.

O Cuda fornece abstração nas threads para lidar com SIMD, sincronismo e partilha de data dentro de pequenos grupos de threads.

O dispositivo de computação:

- É um co-processador ao CPU ou um host.
- Tem a sua própria DRAM (shared memory).
- Consegue executar muitas threads em paralelo.
- É tipicamente um GPU, mas também pode ser outro tipo de dispositivo de executar em paralelo.

As porções de data-parallel de uma aplicação são expressas como kernel devices que executam muitas threads – SIMT.

Diferenças entre GPU e CPU threads:

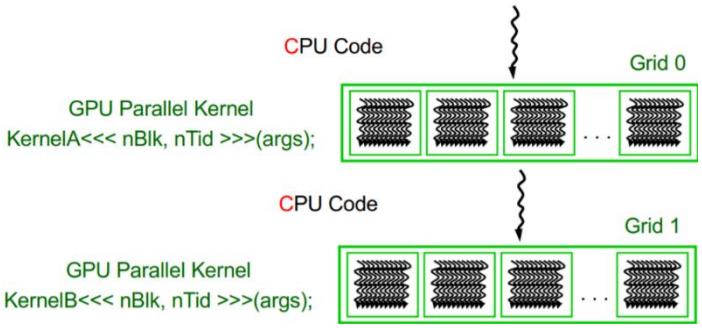
- GPU threads são muito leves, enquanto que, nas CPU threads, cada pequena sobrecarga de criação leva à requisição de grandes bancos de regisots.
- GPU precisa de 1000s de threads para total eficiência, enquanto que multi-core PU precisa apenas de alguns.

SPMD – Cuda Basic Model

Single Program Multiple-Data.

O Cuda está integrado na aplicação em C de CPU+GPU.

- O código sequencial em C é executado no CPU.
- O código C em paralelo, kernel, é executado no GPU em thread blocks.



Nota: Grid – todas as Cuda threads geradas.

Programming Model: SPMD + SIMT/SIMD

Hierarquia:

- Dispositivo > Grid > Blocks > Warps > Threads.

Single Kernel executa em múltiplas threads (SPMD).

Threads dentro de uma warp são executadas numa forma de etapa de bloqueio chamada Single Instruction Multiple-Thread (SIMT).

Single instructions são executadas em multiple-threads (SIMD): o warp size define o tamanho/granularidade do SIMD (32 threads).

A sincronização dentro de um bloco usa memória partilhada.

A Kernel executa numa grid computacional thread blocks: as threads partilham global memory.

Cada thread usa ID's para decidir que data irão trabalhar:

- Block ID: 1D ou 2D.
- Thread ID: 1D, 2D ou 3D.

Uma thread block é um conjunto de threads que podem trabalhar juntas por:

- Sincronizar a sua execução com uma barreira.
- Partilhar eficientemente data através de low latência na shared memory.
- Duas threads de dois blocos diferentes não podem trabalhar juntas.

As threads das instruções do SIMD são warps.

- Cada uma tem o seu IP.
- O "agendador" das threads usa scoreboard para despachá-las.
- Não há dependência de data entre threads.
- Elas são organizadas em blocos e executadas em grupos de 32 threads (thread block): os blocos são organizados numa grid.

O threadblock scheduler agenda blocos para o processador SIMD (streaming multiprocessors).

Dentro de cada SIMD processador há 32 SIMD faixas (thread processors).

Cuda Thread Block

O programador declara (thread) block:

- Todas as threads no mesmo bloco executam a mesma thread do programa.
- As threads partilham data e sincronizam-se enquanto fazem parte delas (seu trabalho).
- As threads têm uma thread ID dentro do bloco.
- A thread do programa usa essa thread ID para selecionar trabalho e endereçar memória partilhada.

Parallel Memory Sharing

- Local memory: privada por thread.
- Shared memory: partilhada por threads do mesmo bloco.
- Global memory: partilhada por todas as threads.

From multicore to manycore

Dicas para ter vários cores num único chip:

- Diminuir a capacidade de computação de cada core, mas não demasiado.
- Usar uma rede de interconexão escalável no chip (NoC) para minimizar as latências de acesso a cache/memória partilhada; providenciar uma largura de banda de comunicação de data suficiente e minimizar traffic bottlenecks.
- Agrupar cores nos clusters para aumentar a qualidade do NoC.
- Reduzir o nível/tamanho da cache.
- Processos de fabricação menores.
- Mudar para MCM/chiplets.
- Misturar PUs de uso geral com módulos orientados a aplicações: GPUs para computação de vetores.

Interconnect fundamentals

Networks on-chip: sistema de interconexão para ligar servidores em super computadores.

Parâmetros chave que definem um NoC:

- Topologia: define como os nodos e links estão conectados, nomeadamente, todos os caminhos possíveis que uma mensagem pode percorrer na network.
- Algoritmo de roteamento: seleciona um caminho específico que uma mensagem vai percorrer da fonte ao destino.
- Protocolo de controlo de fluxo: determina como uma mensagem realmente percorre a rota atribuída.
- Micro arquitetura routeamento: implementa os protocolos de roteamento e controle de fluxo e molda criticamente os seus circuitos.

Manycore chips/packages: na overview

Key server chips/packages that addresses those issues:

- AMD: the Epyc Zen family.
- Sunway: the SX260x0 family.
- ARM: the ARMv8 server-level competiors.
- Cerebras: a Wafer Scale Engine.
- Apple: the SoC approach.

MACC (Minho Advanced Computing Center)

MACC é um supercomputador sustentável e bigdata infraestrutura atendendo a comunidades científicas e industriais nacionais e complementar a parceiros internacionais.

Foi criado em 2017 pelo Governo português através de FCT, INCodeDE.2030 e a Universidade do Minho como uma infraestrutura colaboradora para promover e apoiar iniciativas científicas em computação avançada.

O MACC centra-se em promover descobertas científicas e industriais inovadores abrindo o High-Performance Computing para pesquisar comunidades ao longo de um vasto espectro de disciplinas. Para completar a sua missão, o MACC conduz cinco atividades instrumentais:

- Pesquisa e inovação em supercomputação: para apoiar e encorajar a pesquisa em tópicos entrais para implementar, otimizar e explorar performance computacional.
- Treino e desenvolvimento de habilidades: para habilitar e encorajar treino de recursos humanos no design, desenvolvimento e operação de centros supercomputacionais
- R&D em engenharia computacional/informática, ciências da computação e inteligência computacional: para promover a criação, expansão e consolidação da pesquisa e inovação de centros em diferentes domínios da ciência onde se usa uma ampla capacidade de processamento de data e poder computacional é crucial, nomeadamente em áreas de simulação digital, máquinas de aprendizagem, IA.
- Serviço público e negócio: para oferecer recursos avançados HPC.
- Internacionalização: participar em redes internacionais de ciência avançada na área da computação.

Bob (MACC)

O Bob é composto por 600 nodos computacionais e 9600 cores instalados no centro de data, localizado em Riba de Ave, altamente alimentado por recursos de energia sustentável.

Cada nodo do Bob tem 2 Intel 8-core “Sandy Bridge” generation Xeon processadores a 2.7GHz com 32GB de RAM e memória partilhada com 1.5PB. Os nodos são integrados com a Mellanox FDR 56 Gb/s InfiniBand network.

Bob é parte do antigo supercomputador to Texas Advanced Computing Center (TACC).

OpenMP

OpenMP é uma implementação de multithreading, um método de paralelização no qual o "master thread" (uma série de instruções executadas consecutivamente) forks ("bifurca") um específico número de threads escravos e uma tarefa é dividida entre eles. Os thread são então executados simultaneamente, com ambiente de execução distribuindo as threads para diferentes processadores.

A parte de código que é criada para funcionar em paralelo é marcada de acordo com uma diretriz pré-processador que criará os thread para formar a secção antes de ser executada. Cada thread tem um "id" (endereço) anexado a ele, que pode ser obtido através de uma função (chamada omp_get_thread_num() em C. O thread "id" é um número inteiro e o master thread possui o id "0". Após a execução do código em paralelo, os "threads" retornam ao master thread, o qual continua progressivo até o fim do programa.

Construção do compartilhamento

Usado para especificar como atribuir um trabalho independente para um ou todos as threads.

- omp for ou omp do: usado para fracionar os laços entre os caminhos de execução, também chamados "construtores de laço".
- sections: atribuindo consecutivos, porém independentes, blocos de código para os diferentes threads.
- single: especificando um bloco de código, que é então executado por apenas uma thread, com uma barreira implícita no fim
- master: semelhante ao single, mas o bloco de código será executado apenas pela thread mestre e nenhuma barreira estará implícita no final.

Exemplo: inicializar o valor de uma grande variedade em paralelo, utilizando cada fio de execução para fazer parte do trabalho.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```

Cláusulas do OpenMP

Desde que foi compartilhada a programação modelo do OpenMP, a maior parte das variáveis no código do OpenMP são visíveis a todos threads por padrão. Porém algumas variáveis individuais são necessárias para evitar race conditions e existe a necessidade de passar valores entre a parte sequencial e a região paralela (o código executado em paralelo), para que os dados de gestão ambiental sejam introduzidos como as cláusulas atribuídas ao compartilhamento de dados colocando-lhes as diretrizes do OpenMP. Os diferentes tipos de cláusulas são:

Características da Partilha de Dados

- shared: os dados dentro de uma região paralela são compartilhados, o que significa visível e acessível por todas as threads em simultâneo. Por padrão, todas as variáveis da região de compartilhamento são partilhados, exceto o laço (loop).
- private: os dados em uma região paralela são individuais para cada thread, com este recurso cada thread terá uma cópia local e será utilizada como uma variável temporária. Uma variável não é inicializada e o valor não é mantido para o uso fora da região paralela. Por padrão, os contadores no "construção do laço OpenMP" (OpenMP loop constructs) são individuais.
- default: permite ao programador declarar a extensão do default data scoping (padrão de dados) que será compartilhado (ou nenhuma região) na região paralela, em C/C++, ou shared, firstprivate ou none para Fortran. A opção none força o programador a declarar cada variável na região paralela utilizando as cláusulas atribuídas ao compartilhamento de dados (data sharing attribute clauses).

- firstprivate: como private exceto ao inicializar pelo valor original.
- lasprivate: como private exceto que o valor original é atualizado depois da construção.
- reduction: um caminho seguro de juntar todas as threads após a construção.

Sincronização

- critical section: o código incluso será executado por somente um thread por vez e não simultaneamente executado por múltiplos threads. É frequentemente usado para proteger os dados compartilhados das race conditions.
- atomic: semelhante à critical section, mas informamos ao compilador para usar instruções especias de hardware para um melhor desempenho. Os compiladores podem optar por ignorar essa sugestão dos usuários e usar a critical section em vez da atomic.
- ordered: o bloco estruturado é executado na ordem em que as iterações seriam executadas em um loop sequencial.
- barrier: cada thread espera até que todos os outros threads de um grupo tenham alcançado este ponto. Uma construção de partilha tem uma barreira de sincronização implícita no final.
- nowait: especifica quais threads podem completar sua instrução sem esperar todos os outros threads do grupo para concluir. Na ausência de tal cláusula acontece o mesmo da barrier.

Programação

- schedule (type chunk): É útil se a construção de partilha for um do-loop ou for-loop. A iteração, ou as iterações, na construção de partilha são atribuídas para as threads de acordo com o método programado definido nesta cláusula. Os três tipos de programações são:

1. static: Aqui, a todos os threads são atribuídas as iteração antes de executar o laço de repetição. As interações são divididas igualmente entre os threads por padrão. No entanto, especificando um inteiro para um parâmetro "chunk" fixará um número "chunk" de iterações sequenciadas a uma determinada lista de threads.

2. dynamic: Aqui, algumas das iterações são atribuídas a um número menor de threads. Após o término da iteração atribuída a um thread em particular, ele retorna para buscar uma das iterações restantes. O parâmetro "chunk" define o número de iterações sequenciais que são atribuídas a um thread por vez.

3. guided: Um grande "chunk" de iterações sequenciadas são atribuídos a cada thread dinamicamente (como acima). O tamanho do "chunk" diminui exponencialmente com cada atribuição sucessiva até um tamanho mínimo especificado no parâmetro chunk.

Comando if

- if: Isto fará com que as threads paralelizem a tarefa se a condição for satisfeita. Caso contrário, o bloco de código é executado em série.

Inicialização

- firstprivate: os dados são individuais para cada thread, mas inicializando o valor da variável usando o mesmo nome a partir da thread master.
- lastprivate: os dados são individuais para cada thread. Se a iteração atual for a última iteração no loop paralelizado, o valor desses dados individuais serão copiados para uma variável global, usando o mesmo nome fora da região paralela. Uma variável pode ser ambas, firstprivate e lastprivate.
- threadprivate: O dado é um dado global, mas é individual para cada região paralela durante a execução. A diferença entre threadprivate e private é que

o escopo global é associado com o threadprivate e o valor é preservado além das regiões paralelas.

Cópia de dados

- copyin: semelhante ao firstprivate para variáveis privadas, variáveis threadprivate não são inicializadas, a não ser usando copyin para passar o valor das variáveis globais correspondentes. O copyout é necessária porque o valor de uma variável threadprivate é mantida durante toda a execução do programa inteiro.
- copyprivate: usado como único apoio para a cópia dos valores dos dados a partir de objetos particulares em uma thread (a thread única) para os objetos correspondentes em outras threads do grupo.

Redução

- reduction(operador | intrinsic: list) a variável tem uma copia local em cada thread, mas os valores das copias locais são resumidos (reduzidos) em uma variavel global compartilhada. Isso é muito útil se uma operação particular (especificada em "operador" para essa clausula particular) em um datatype que roda interativelmente então esse valor em uma interação particular depende do valor da interação anterior. Basicamente, os passos que mais dão importância ao incremento operacional são paralelizados, mas os threads reúnem-se e esperam antes de fazer o update dos datatype, depois incrementa o datatype em ordem para evitar perda de dados. Isso seria requerido paralelizando funções de integração numérica e equações diferenciadas, como um exemplo comum.

Message Passing Interface (MPI)

Message Passing Interface (MPI) é um padrão para comunicação de dados em computação paralela. Existem várias modalidades de computação paralela, e dependendo do problema que se está tentando resolver, pode ser necessário passar informações entre os vários processadores ou nodos de um cluster, e o MPI oferece uma infraestrutura para essa tarefa.

O objetivo de MPI é prover um amplo padrão para escrever programas com passagem de mensagens de forma prática, portátil, eficiente e flexível. MPI não é um IEEE ou um padrão ISO, mas chega a ser um padrão industrial para o desenvolvimento de programas com troca de mensagens.

Diferenciando Tarefas através de Rank

Cada tarefa MPI deve ser identificada por um id, essa identificação deve ser feita no momento que um processo envolvendo aplicação MPI está sendo iniciada. O ambiente MPI atribui a cada processo um rank (guardado como um int) e um grupo de comunicação. O rank é um tipo de identificador de processo para cada tarefa MPI. É contíguo e começa por zero. Usado pelo programador para especificar a origem e o destino de mensagens:mensagem e frequentemente usado em condições para controle de execução (if rank == 0 faça isso / if rank == 1 faça aquilo). O grupo de comunicação define quais processos podem chamar comunicações ponto a ponto. Inicialmente, todos os processos MPI são associados a um grupo de comunicação default. Os membros de um grupo de comunicação podem mudar depois da aplicação ter iniciado. A atribuição do rank deve ser feita pela rotina MPI_Comm_rank() assim que a aplicação MPI está sendo iniciada. O primeiro argumento dessa rotina especifica qual comunicador será associado e o rank é retornado pelo segundo argumento. O exemplo abaixo mostra como a rotina MPI_Comm_rank() é usada.

```
//…
int Tag = 33;
int WorldSize;
int TaskRank;
MPI_Status Status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&TaskRank);
MPI_Comm_size(MPI_COMM_WORLD,&WorldSize);
//…
```

O comunicador MPI_COMM_WORLD é o comunicador default que todas as tarefas MPI são associadas quando iniciadas. Tarefas MPI são agrupadas por comunicadores. Um comunicador é o que identifica um grupo de comunicação.

Segue abaixo algumas rotinas usadas em MPI:

- MPI_INIT: Inicializa o ambiente de execução de MPI. Deve ser chamada apenas uma vez em todos os programas de MPI e antes de qualquer outra função. Um exemplo em C seria:

```
MPI_Init (&argc,&argv)
```

- MPI_Comm_size: Determina o número de processos em um grupo associado a um comunicador. Geralmente usado com o comunicador MPI_COMM_WORLD para determinar o número de processes que estão sendo usados por uma aplicação.

```
MPI_Comm_size (comm,&size)
```

- MPI_Comm_rank: Utilizado para associar um *rank* do processo requisitante ao comunicador. Inicialmente, cada processo será associado a um único *rank* inteiro entre 0 e o número de processos -1 com o comunicador MPI_COMM_WORLD. Se um processo se tornar associado a outros comunicadores, ele terá um único *rank* com cada um desses comunicadores.

```
MPI_Comm_rank (comm,&rank)
```

- MPI_Abort: Termina todos os processos MPI associados com o comunicador. Em várias implementações de MPI, essa rotina termina todos os processos independentemente do comunicador especificar.

```
MPI_Abort (comm,errorcode)
```

- MPI_Get_processor_name: Retorna o nome do processador. Também retorna o tamanho do nome. O buffer do “nome” deve ser pelo menos MPI_MAX_PROCESSOR_NAME caracteres em tamanho.

```
MPI_Get_processor_name (&name,&resultlength)
```

- MPI_Initialized: Usado para saber se MPI_Init já foi chamado, retornando true (1) se já foi ou false (0) caso contrário.

```
MPI_Initialized (&flag)
```

- MPI_Wtime: Retorna o tempo decorrido em segundos (precisão dupla) no processador requisitante.

```
MPI_Wtime ()
```

- MPI_Finalize: Termina a execução do ambiente MPI. Esta função deve ser a última rotina MPI chamada em todo programa MPI, nenhuma outra rotina MPI deve ser chamada após ela.

```
MPI_Finalize ()
```

Os principais propósitos de um grupo e objetos comunicadores

1. Permitir organizar tarefas em grupos de tarefas;

2. Permitir operações de comunicação coletiva através de subconjuntos de tarefas relacionadas;

3. Prover comunicação segura.

Importante:

- Grupos/comunicadores são dinâmicos, eles podem ser criados e destruídos durante a execução do programa;

- Processos podem estar em mais de um grupo ou comunicador, no entanto eles terão um único rank para cada grupo/comunicador;

- MPI provê mais de 40 rotinas relacionadas a grupos, comunicadores e topologias virtuais.

É comum que

1. Um processo se agrupe ao grupo MPI_COMM_WORLD usando a rotina MPI_Comm_group;

2. Forme novos grupos como um subconjunto do grupo global utilizando MPI_Comm_incl;

3. Crie novo comunicador para o novo grupo utilizando MPI_Comm_create;

4. Determine um novo rank no novo comunicador utilizando MPI_Comm_rank;

5. Conduza comunicações usando qualquer rotina de troca de mensagem MPI;

6. Quando acabado, libere o novo comunicador e grupo (opcional) usando MPI_Comm_free e MPI_Group_free.

O exemplo abaixo mostra um exemplo de um programa em C que faz a criação de dois grupos de processos diferentes que requer a criação de novos comunicadores também.

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8

int main(argc, argv)
        int argc;char *argv[]; {
        int rank,new_rank,sendbuf,recvbuf,numtasks,ranks1[4] =
{0,1,2,3},
                        ranks2[4] = {4,5,6,7};
        MPI_Group orig_group, new_group;
        MPI_Comm new_comm;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

        if (numtasks != NPROCS) {
                printf("Deve especificar MP_PROCS= %d.
Terminando.\n", NPROCS);
                MPI_Finalize();
                exit(0);
        }

        sendbuf = rank;

        /* Extrai o grupo original */
        MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

        /* Divide tarefas em dois grupos distintos baseados no
''rank'' */
        if (rank < NPROCS / 2) {
                MPI_Group_incl(orig_group, NPROCS/2, ranks1,
&new_group);
        } else {
                MPI_Group_incl(orig_group, NPROCS/2, ranks2,
&new_group);
        }

/* Cria novos comunicadores e então realiza comunicação coletiva.
*/
        MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
        MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM,
new_comm);

        MPI_Group_rank(new_group, &new_rank);
        printf("rank= %d newrank= %d recvbuf= %d\n", rank, new_rank,
recvbuf);

        MPI_Finalize();
}
```

Rotinas de comunicação ponto a ponto

Operações MPI ponto a ponto envolvem troca de mensagem entre APENAS duas tarefas distintas. Uma tarefa executa a operação de envio enquanto outra executa a operação de recebimento que casa com a de envio.

Há diferentes tipos de rotinas de envio e recebimento, tais como:

- Envio síncrono;
- Envio bloquente/recebimento bloqueante;
- Envio não bloqueante /recebimento não bloqueante;
- Envio buferizado;
- Envio e recebimento combinado;

Todos esses tipos de envio e recebimento de mensagens podem ser usados para diferentes propósitos e qualquer tipo de rotina de envio pode ser usado com qualquer tipo de rotina de recebimento.

Envio bloqueante de mensagens só irá retornar da rotina quando a mensagem estiver no buffer da fonte ou no buffer do destino. Vai depender da rotina que foi invocada. Analogamente podemos dizer o mesmo das rotinas de recebimento bloqueante. São rotinas que só retornam quando o seu buffer tiver recebido a mensagem ou quando tiver lido a mensagem. Abaixo segue as rotinas de envio bloqueantes mais usadas:

- MPI_Send: Operação básica de bloqueio de envio. A rotina retorna apenas depois da aplicação buferizar na tarefa de envio.

```
MPI_Send (&buf,count,datatype,dest,tag,comm)
```

- MPI_Recv: Recebe uma mensagem e bloqueia até que o dado que está sendo recebido esteja disponível no buffer da aplicação.

```
MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
```

- MPI_Ssend: Bloqueia até que o destino tenha recebido a mensagem.

```
MPI_Ssend (&buf,count,datatype,dest,tag,comm)
```

- MPI_Bsend: Permite ao programador escolher um certo valor de buffer que torna a chamada da rotina bloqueada até que esse valor de buffer tenha sido atingido.

```
MPI_Bsend (&buf,count,datatype,dest,tag,comm)
```

- MPI_Buffer_attach
- MPI_Buffer_detach: Usado para alocar e desalocar o espaço de buffer de mensagem que será usado pela rotina MPI_Bsend.

```
MPI_Buffer_attach (&buffer,size)
     MPI_Buffer_detach (&buffer,size)
```

- MPI_Rsend: Dispensa o handshake, é mais eficiente, porém só pode ser usado se o recetor estiver em receive().

```
MPI_Rsend (&buf,count,datatype,dest,tag,comm)
```

- MPI_Sendrecv: Envia uma mensagem e coloca um receive antes de bloquear. Essa rotina bloqueia até que o buffer da aplicação fonte esteja livre para reuso e até o buffer da aplicação destino contenha a mensagem recebida.

```
MPI_Sendrecv (&sendbuf,sendcount,sendtype,dest,sendtag,
                  &recvbuf,recvcount,recvtype,source,recvtag,
                  comm,&status)
```

- MPI_Wait
- MPI_Waitany
- MPI_Waitall
- MPI_Waitsome: MPI_Wait boqueia até que um específico envio não bloquente ou operação de recebimento tenha completado. Para múltiplas operações não bloqueantes, o programador pode especificar qualquer, todas ou algumas conclusões.

```
MPI_Wait (&request,&status)
     MPI_Waitany (count,&array_of_requests,&index,&status)
     MPI_Waitall (count,&array_of_requests,&array_of_statuses)
     MPI_Waitsome (incount,&array_of_requests,&outcount,
                  &array_of_offsets, &array_of_statuses)
```

- MPI_Probe: Executa um teste de bloqueio para uma mensagem.

```
MPI_Probe (source,tag,comm,&status)
```

O código em C abaixo mostra um exemplo da utilização dessas rotinas.

```
#include "mpi.h"
#include <stdio.h>

int main(argc, argv)
        int argc;char *argv[]; {
        int numtasks, rank, dest, source, rc, count, tag = 1;
        char inmsg, outmsg = 'x';
        MPI_Status Stat;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        if (rank == 0) {
                dest = 1;
                source = 1;
               rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
MPI_COMM_WORLD);
               rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
MPI_COMM_WORLD, &Stat);
        }

        else if (rank == 1) {
                dest = 0;
                source = 0;
```

```
                rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
MPI_COMM_WORLD, &Stat);
                rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
MPI_COMM_WORLD);
        }

      rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
      printf("Tarefa %d: Recebida %d char(s) da tarefa %d com tag
%d \n", rank,count,
      Stat.MPI_SOURCE, Stat.MPI_TAG);

        MPI_Finalize();
}
```

Envio não bloqueante de mensagens implica que o fluxo de execução continua após ter chamado a rotina de envio, sem se importar se a mensagem chegou ao *buffer* da fonte ou do destino. Analogamente podemos dizer o mesmo das rotinas de recebimento não bloqueante.

Abaixo segue as rotinas de envio não bloqueantes mais usadas:

- MPI_Isend: identifica uma área na memória que sirva como buffer de envio. O processamento continua sem que a aplicação espere por uma confirmação de que a mensagem foi copiada com sucesso.

```
MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
```

- MPI_Irecv: identifica uma área na memória que sirva como buffer de recebimento. O processamento continua sem que a aplicação espere por uma confirmação de que a mensagem foi recebida e copiada para o buffer com sucesso.

```
MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
```

- MPI_Issend: indica quando o processo destino recebeu a mensagem.

```
MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)
```

- MPI_Ibsend: indica quando o processo destino recebeu a mensagem. Deve ser usado com a rotina.

```
MPI_Buffer_attach.
MPI_Ibsend (&buf,count,datatype,dest,tag,comm,&request)
```

- MPI_Irsend: indica quando o processo destino recebeu a mensagem. Deve ser usado apenas se o programador tem certeza que o receive que casa está pronto.

```
MPI_Irsend (&buf,count,datatype,dest,tag,comm,&request)
```

- MPI_Test

- MPI_Testany
- MPI_Testall
- MPI_Testsome: MPI_Test verifica o status de um envoi não bloqueante específico ou operação de recebimento. Retorna (1) caso a operação está concluída ou (0) caso contrário. Para múltiplas operações não bloqueantes o programador pode especificar qualquer, todos ou algumas conclusões.

```
MPI_Test (&request,&flag,&status)
MPI_Testany (count,&array_of_requests,&index,&flag,&status)
MPI_Testall (count,&array_of_requests,&flag,&array_of_statuses)
MPI_Testsome (incount,&array_of_requests,&outcount,
               &array_of_offsets, &array_of_statuses)
```

- MPI_Iprobe: executa um teste não bloqueante para uma mensagem.

```
MPI_Iprobe (source,tag,comm,&flag,&status)
```

Segue abaixo um exemplo em C de uso dessas retinas não bloqueantes.

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
        int argc;
        char *argv[];  {
        int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
        MPI_Request reqs[4];
        MPI_Status stats[4];

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        prev = rank-1;
        next = rank+1;
        if (rank == 0)  prev = numtasks - 1;
        if (rank == (numtasks - 1))  next = 0;

        MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD,
&reqs[0]);
        MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD,
&reqs[1]);

        MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD,
&reqs[2]);
        MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD,
&reqs[3]);

      {  do some work  }

        MPI_Waitall(4, reqs, stats);

        MPI_Finalize();
}
```

Cuda

CUDA (anteriormente conhecido como Compute Unified Device Architecture ou Arquitetura de Dispositivo de Computação Unificada) é uma API destinada a computação paralela, GPGPU, e computação heterogênea, criada pela Nvidia.[1] destinada a placas gráficas que suportem a API (normalmente placas gráficas com chipset da Nvidia). A plataforma CUDA dá acesso ao conjunto de instruções virtuais da GPU e a elementos de computação paralela, para a execução de núcleos de computação.[2]

A API inclui um conjunto de instruções CUDA ISA's (Instruction Set Architecture) e o mecanismo de computação paralela na GPU. Ele expõe os diferentes tipos de memória da placa e obriga que o desenvolvedor configure os acessos da memória global, a cache, a quantidade e a disposição das threads. O desenvolvedor também será responsável por escalonar as atividades entre a GPU e o CPU.

Potencial

A placa gráfica (GPU), como unidade de processamento lógico, é capaz de calcular instrucções paralelas, com uma grande velocidade de largura de banda, como consequência das suas capacidades para gráficos em tempo real de alta resolução gráficos 3D, que são normalmente tarefas bastante exigentes. Por volta de 2012, os GPUs tinham evoluído para dispositivos de processamento paralelo altamente capazes, permitindo uma manipulação muito eficiente de blocos de informação de grande dimensão. O design é mais eficiente em instruções de cálculo paralelo para fins comuns, como por exemplo:

- push-relabel maximum flow algorithm.
- fast sort algorithms of large lists.
- two-dimensional fast wavelet transform.
- molecular dynamics simulations.
- FFT transforms.

Benefícios

- Leitura paralela - o código pode ler de endereços arbitrários na memória;
- Memória compartilhada - CUDA expõe uma região de memória compartilhada rápida (16KB em tamanho) que podem ser compartilhados entre threads. Isso pode ser usado como um cache de usuário, permitindo maior largura de banda do que é possível utilizando textura lookups;
- Downloads mais rápidos e readbacks para a GPU;
- Suporte completo para operações de números inteiros e operações de bitwise.

Limitações

- A renderização de texturas não é suportado;
- As cópias realizadas entre uma memória e outra podem gerar algum problema na performance das aplicações;
- Ao contrário do OpenCL, o CUDA está disponível apenas para placas de vídeo fabricadas pela própria NVIDIA. Caso seja usado em outro tipo de placa, o CUDA funcionará corretamente, entretanto a performance será bem limitada.

Exercícios resolvidos no slides

1.Calcule o tempo de execução do programa abaixo numa máquina com um relógio de 2 GHz e CPI=1.5 18

```
  movl 10, %eax
  movl 0, %ecx
ciclo:
  addl %eax, %ecx
  decl %eax
  jnz ciclo
```

#I = 32

NOTA: número de instruções executadas.

Texec = 32 * 1.5 / 2 E9 = 24 E-9 s = 24 ns

2.Considere uma máquina com uma miss rate de 4% para instruções, 5% para dados e uma miss penalty de 50 ciclos. Assuma ainda que 40% das instruções são loads ou stores, e que o CPI_{CPU}é 1. Qual o CPI total?

$$CPI = CPI_{CPU} + CPI_{MEM} = CPI_{CPU} + (mr_I + \%Mem * mr_D) * mp$$

$$CPI = 1 + (0.04 + 0.4 * 0.05) * 50 = 1 + 3 = 4$$

Se a frequência do relógio for de 1 GHz e o programa executar 109 instruções qual o tempo de execução?

$$T_{exec} = \#I * CPI * T_{CC} = 10^9 * 4 * \frac{1}{10^9} = 4s$$

3. Considere um programa com as características apresentadas na tabela, a executar numa máquina com memória de tempo de acesso 0. Se a frequência do processador for 1 GHz, qual o CPI médio e o tempo de execução?

Instrução	Nº Instruções	CPI
Cálculo	$3*10^8$	1,1
Acesso à Mem.	$6*10^8$	2,5
Salto	$1*10^8$	1,7
TOTAL:	10^9	

$$CPI = CPI_{CPU} + CPI_{MEM} = \frac{3 * 1.1 + 6 * 2.5 + 1 * 1.7}{10} + 0 = 2$$

$$T_{exec} = \#I * CPI * T_{CC} = 10^9 * 2 * \frac{1}{10^9} = 2s$$

4. Considere o mesmo programa e máquina do acetato anterior, mas agora com um tempo de acesso à memória de 10 ns (por palavra ou instrução). Suponha ainda que esta máquina não tem cache. Qual o CPI efectivo e Texec?

$$CPI = CPI_{CPU} + CPI_{MEM} = CPI_{CPU} + (mr_I + \%Mem * mr_D) * mp$$

Se a máquina não tem cache, então $mr_I = mr_D = 100\%$.

Da tabela verificamos que %Mem=60%(6x10^8/10^9).

Mp expresso em ciclos de relógio é 10/1=10 ciclos (f=1 GHz).

$$CPI = CPI_{CPU} + CPI_{MEM} = 2 + (1 + 0.6 * 1) * 10 = 2 + 16 = 18$$

$$T_{exec} = \#I * CPI * T_{CC} = 10^9 * 18 * \frac{1}{10^9} = 18s$$

5.Considere agora que existe uma cache com linhas de 4 palavras; a miss rate de acesso às instruções é de 6% e de acesso aos dados é de 10%; o tempo de acesso à memória central é constituído por uma latência de 40 ns mais 10 ns por palavra. Qual o CPI médio e o tempo de execução?

mp=40+10*4=80ns; em ciclos mp=80/1=80 ciclos.

$$CPI = CPI_{CPU} + CPI_{MEM} = 2 + (0.06 + 0.6 * 1) * 80 = 2 + 9.6 = 11.6$$

$$T_{exec} = \#I * CPI * T_{CC} = 10^9 * 11.6 * \frac{1}{10^9} = 11.6s$$

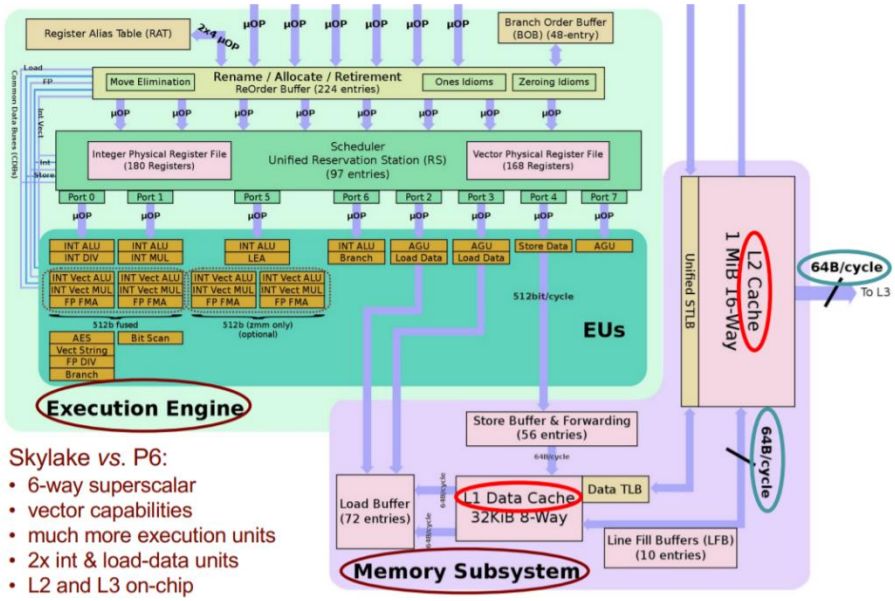
6.Consider the following case study:

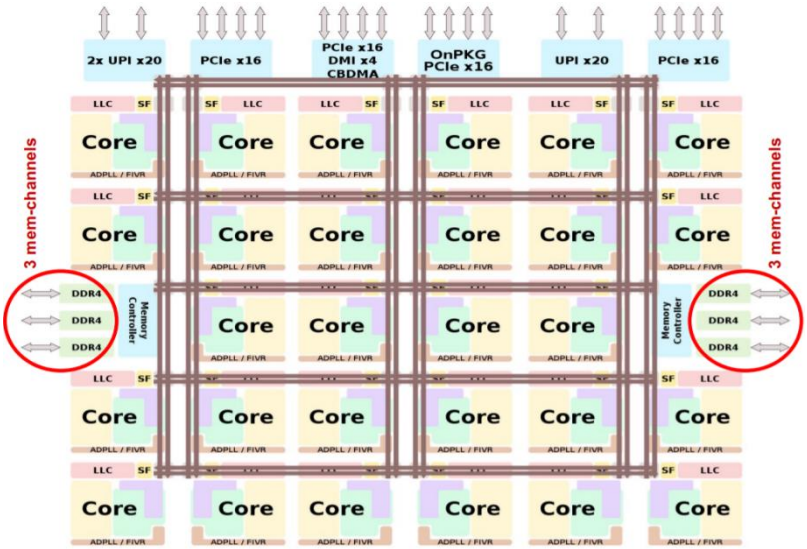
- … code in the SeARCH node with the Xeon Skylake …
- … same 2 instructions … in all cores of a single chip…
- … cores 6-way superscalar … 2 load units/core … cold data cache.

Compute:

a) The max required bandwidth to access the external RAM …

b) The aggregate peak bandwidth … DRAM-4 (w/ all memory channels).





Cada ciclo de clock precisa de 2 acessos à memória para ir buscar dois doubles.

A largura de banda máxima necessária para ir buscar uma linha de cache para cada double (a cache é fria e os doubles estão longe):

(16 cores * 2 linhas * 64 B/linha) * frequência de relógio = 2048 B * 2GHz = 4096 GB/s

RAM em cada Skylake Gold 6130: 6x DDR4-2666 (6x8 GiB).

Pico de largura de banda de 6x DDR4-2666 em 6 canais de memória:

6 mem_chan * 2.666 GT/s * 64 b/chan = 128 GB/s

7.Given

- I-cache miss rate = 2%
- D-cache miss rate = 4%
- Miss penalty = 100 cycles
- Base CPI (ideal cache) = 2
- Load & stores are 36% of instructions

Miss cycles per instruction:

- I-cache: 0.02*100=2
- D-cache: 0.36*0.04*100=1.44

Actual CPI = 2 + 2 + 1.44 = 5.44

8.Given

- CPU base CPI=1, clock rate=4GHz
- Miss rate/instruction=2%
- Main memory acess time=100ns

With just primary cache

- Miss penalty=100ns/0.25ns=400cycles
- Effective CPI=1+0.02*400=9

L1 cache: L1 miss rate/instrucion = 2%

L2 cache: access time = 5ns, L2 miss rate/instruction = 25%,

Global miss rate = 2%*25%=0.5%

Main memory: access time=100ns

With L1 and L2 cache

- L1 miss penalty, L2 hit = 99.5%*5ns/0.25ns=20 cycles
- L2 miss penalty = 100ns/0.25=400 cycles
- CPI=1+ 2%*20cycles + 0.5%*400cycles = 3.4

Performance ratio = 9/3.4=2.6

9.Similar to problem 1 (same node/chip in the cluster, code), but consider now:

- execution of scalar code in a 2 GHz single-core (already in L1 I-cache);
- code already takes advantage of all data cache levels (L1, L2 & L3), where 50% of data is placed on the RAM modules in the memory channels of the other PU chip (NUMA architecture);
- remember: the Skylake cores are 6-way superscalar and 2-way MT, and each core supports 2 simultaneous loads;

- cache latency time on hit: take the average of the specified values;
- memory latency: 80 nsec (NUMA local), 120 nsec (NUMA remote);
- miss rate per instruction : –at L1: 2%; at L2: 50%; at L3: 80% (these are not global values!).

Compute/estimate:

1. The miss penalty per instruction at each cache level.

2. The average memory stall cycles per instruction that degrades CPI.

PU: base CPI=1, clock rate=2GHz

L1 cache: L1 miss rate/instruction=2%

L2 cache: access time=14cycles, global miss rate= 2%*50%=1%

L3 cache: access time=60cycles, global miss rate= 1%*80%=0.8%

Main memory: NUMA local access time=80ns, NUMA remote=120ns

Average memory access=((80ns+120ns)/2)/0.5ns=200 cycles.