Programação Paralela e Distribuída

Memórias Cache e Arquitecturas Multi-Processador

Memórias Cache e Arquitecturas Multi-Processador

- A principal motivação para utilizar o OpenMP é conseguir maximizar a utilização do poder computacional disponível numa máquina multi-processador de modo a reduzir o tempo de resolução de um determinado problema.
- Apesar das diferentes arquitecturas multi-processador existentes actualmente, uma grande percentagem deste tipo de arquitecturas apresenta características fundamentais que são idênticas. Em particular, elas utilizam processadores comuns ligados entre si por uma espécie de rede e possuem **memórias cache** muito perto dos processadores de modo a minimizar o tempo necessário para aceder aos dados.
- De entre os vários factores que podem limitar o desempenho da programação paralela, existem dois que estão intrinsecamente ligados às arquitecturas multi-processador baseadas em memórias *cache*:
  - **Localidade**
  - **Sincronização**

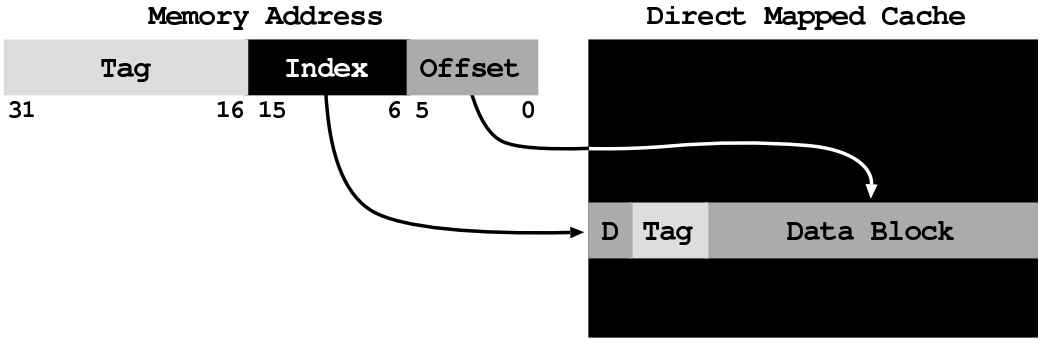
  O efeito destes dois factores é frequentemente mais surpreendente e difícil de entender que os restantes factores, e o seu impacto pode ser enorme.

Caches e Desempenho

- Do ponto de vista conceptual, não existe nenhuma diferença entre manipular uma variável `var1` ou uma variável `var2`. No entanto, do ponto de vista do desempenho, aceder à variável `var1` num determinado momento pode ser mais ou menos dispendioso do que aceder à variável `var2`, e aceder à variável `var1` pode ser mais ou menos dispendioso do que aceder novamente a `var1` um momento mais tarde.
- Parte do tempo necessário para fazer chegar os dados em memória até ao processador é despendido em **encontrar os dados**, enquanto outra parte é despendida em **mover os dados**. Isso significa que quanto mais perto os dados estiverem do processador mais rapidamente poderão lá chegar. No entanto, existe um limite físico na quantidade de memória que pode estar a uma determinada distância do processador.
- A motivação para usar memórias *cache* é conseguir maximizar as vezes que os dados necessários a uma determinada computação estão mais perto do processador numa memória mais pequena mas que seja bastante rápida de aceder, minimizando assim o número de interacções com a memória principal do sistema, que normalmente é mais lenta e está mais longe do processador.

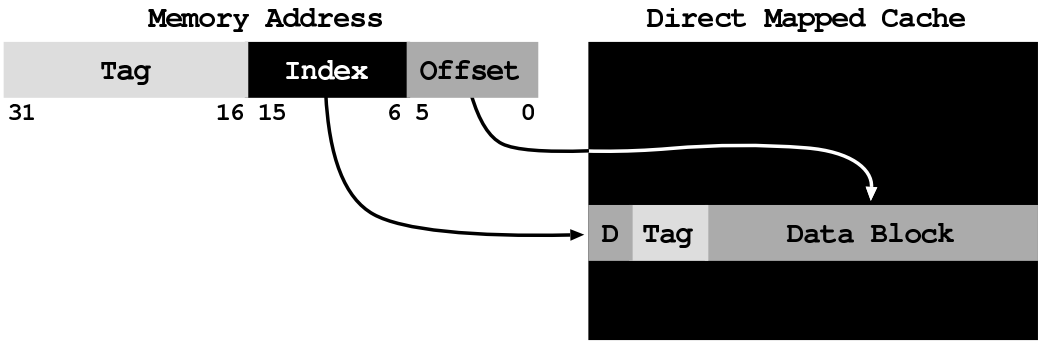
Direct Mapped Caches

- Considere uma arquitectura com endereços de memória de 32 bits e uma *cache* de 64 Kbytes organizada em 1024 (2^{10}) entradas (linhas de *cache*) com blocos de dados de 64 (2^6) *bytes* por linha.
- Se essa *cache* for do tipo *direct mapped cache*, isso significa que cada endereço de memória é **mapeado numa única linha de cache**:
  - Os *bits* de 6 a 15 (2^{10}) indexam a linha de *cache*.
  - Os *bits* de 16 a 31 (2^{16}) permitem verificar se o endereço está em *cache*.
  - Os *bits* de 0 a 5 (2^6) indexam o *byte* respectivo dos 64 *bytes* de cada linha.



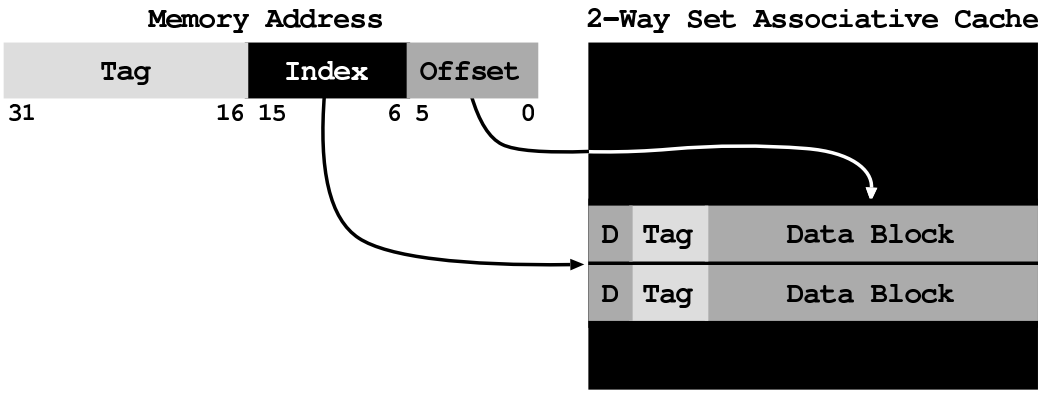
Direct Mapped Caches

- Como cada endereço de memória é mapeado numa única linha de *cache*, todos os endereços de memória com os mesmos valores nos *bits* de 6 a 15 são mapeados na mesma linha de *cache*.
- Quando ocorre um **cache miss** é necessário substituir a linha em *cache* pela linha que contém o endereço de memória pretendido. No caso de existirem alterações sobre os dados da linha a substituir (i.e., se o **dirty bit D** for 1), então essa linha deve ser primeiro escrita de volta para a memória principal. Caso contrário, a linha pode ser simplesmente substituída.



N-Way Set Associative Caches

- Considere novamente uma arquitectura com endereços de memória de 32 bits, mas agora com uma *cache* em que cada endereço de memória pode ser **mapeado em N linhas de cache diferentes** (*N-way set associative cache*).
- Se considerarmos novamente uma *cache* com blocos de dados de 64 *bytes* por linha e os *bits* de 6 a 15 como índices das linhas de *cache*, então uma *2-way set associative cache* ocuparia 128 Kbytes para um total de 2048 ($2 * 2^{10}$) linhas de *cache* (2 linhas por cada endereço com os mesmos valores nos *bits* de 6 a 15).



Localidade Espacial e Localidade Temporal

- **Localidade espacial** é a propriedade de que quando um programa acede a uma posição de memória, então existe uma probabilidade maior de que ele aceda a posições de memória contíguas num breve espaço de tempo.

```
for (i = 0; i < N; i++)
  a[i] = 0;
```

- **Localidade temporal** é a propriedade de que quando um programa acede a uma posição de memória, então existe uma probabilidade maior de que ele aceda novamente à mesma posição de memória num breve espaço de tempo.

```
for (i = 1; i < N - 1; i++)
  for (j = 1; j < N; j++)
    a[i][j] = func(a[i-1][j],a[i+1][j],a[i][j-1],a[i][j+1]);
```


Localidade Espacial e Localidade Temporal

- Em geral, a localidade espacial é mais fácil de conseguir do que a localidade temporal. No entanto, para conseguir obter boa localidade quando percorremos posições contíguas duma determinada estrutura de dados, é necessário conhecer como essa estrutura de dados é representada em memória pelo compilador da linguagem de programação em causa.
- Por exemplo, o código abaixo exibe boa localidade espacial com um compilador de C (em C, as matrizes são guardadas por ordem de linha), mas o mesmo não acontece com um compilador de Fortran (em Fortran, as matrizes são guardadas por ordem de coluna).

```
for (i = 0; i < N; i++)
  for (j = 1; j < N; j++)
    a[i][j] = scale * a[i][j];
```

Localidade Espacial e Localidade Temporal

- Em arquitecturas multi-processador, para além de uma boa localidade espacial e temporal por processador, é necessário restringir essa localidade a cada processador, i.e., evitar que haja mais do que um processador a aceder às mesmas linhas de *cache* no mesmo espaço de tempo.
- Aceder a dados que estão na *cache* de outro processador, normalmente **é pior do que aceder a dados na memória principal**. De seguida vamos analisar 3 situações diferentes que evidenciam este tipo de comportamento:
  - Escalonamento de ciclos paralelos consecutivos.
  - Falsa partilha.
  - Paralelização inconsistente.

Escalonamento de Ciclos Paralelos Consecutivos

- Considere o código abaixo onde uma determinada matriz é percorrida por 2 vezes. Considere ainda um máquina multi-processador com 8 processadores em que a *cache* de cada processador consegue albergar matrizes de 400x400 e em que as *caches* agregadas dos 8 processadores conseguem albergar matrizes de 1000x1000 mas não conseguem albergar matrizes de 4000x4000.

```
scale1 = get_scale_value();
#pragma omp parallel for private(i,j)  // schedule(static/dynamic) ???
for (i = 0; i < N; i++)
  for (j = 1; j < N; j++)
    a[i][j] = scale1 * a[i][j];
scale2 = get_scale_value();
#pragma omp parallel for private(i,j)  // schedule(static/dynamic) ???
for (i = 0; i < N; i++)
  for (j = 1; j < N; j++)
    a[i][j] = scale2 * a[i][j];
```

- Se executarmos o código acima com 1 e 8 processadores, será que o *speedup* obtido será diferente se utilizarmos um escalonamento estático (**schedule(static)**) ou um escalonamento dinâmico (**schedule(dynamic)**) ?

Escalonamento de Ciclos Paralelos Consecutivos

Dimensão	*Speedup static*	*Speedup dynamic*	Relação *static/dynamic*
400 x 400	6.2	0.6	9.9
1000 x 1000	18.3	1.8	10.3
4000 x 4000	7.5	3.9	1.9

- Nos casos em que as *caches* agregadas dos 8 processadores conseguem albergar as matrizes por completo, o escalonamento estático é cerca de 10 vezes mais rápido do que o escalonamento dinâmico. Isto acontece porque ao percorrermos a matriz pela segunda vez, o escalonamento estático atribuí as mesmas porções da matriz a cada processador e como essas porções já se encontram totalmente em *cache* (localidade temporal), essa computação é bastante rápida. Em particular, no caso de matrizes 1000x1000, o *speedup* é superlinear.
- Ao utilizarmos um escalonamento dinâmico, essa localidade é perdida e o custo de aceder a dados que estão na *cache* de outros processadores revela-se bastante elevado.

Escalonamento de Ciclos Paralelos Consecutivos

Dimensão	*Speedup static*	*Speedup dynamic*	Relação *static/dynamic*
400 x 400	6.2	0.6	9.9
1000 x 1000	18.3	1.8	10.3
4000 x 4000	7.5	3.9	1.9

- No caso de matrizes 4000x4000, o escalonamento estático é ainda o mais rápido, mas a diferença revela-se bastante inferior. A influência da interacção entre localidade temporal e o tipo de escalonamento tende a diminuir à medida que o tamanho dos dados aumenta.
- Podemos então concluir que nos casos em que o balanceamento de carga é perfeito, é preferível utilizar um escalonamento estático. O escalonamento dinâmico revela-se uma melhor alternativa apenas quando o balanceamento de carga é um problema.

Falsa Partilha

- No entanto, existem situações em que apesar do balanceamento de carga ser perfeito, uma boa localidade nem sempre é conseguida. Considere, por exemplo, o código abaixo onde se pretende calcular o número de elementos pares e ímpares de um determinado vector `a[]`.

```
int count[NTHREADS][2];  // o vector é partilhado por todas as threads
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  count[tid][0] = count[tid][1] = 0;
  #pragma omp for private(i,index) schedule(static)
  for (i = 0; i < N; i++) {
    index = a[i] % 2;
    count[tid][index]++;  // mas cada thread escreve em posições diferentes
  }
  #pragma omp atomic
  even += count[tid][0];
  #pragma omp atomic
  odd += count[tid][1];
}
```


Falsa Partilha

- O problema do exemplo anterior está no modo como o vector `count[]` é criado. Apesar de cada *thread* escrever em posições diferentes do vector `count[]`, essas posições correspondem a posições contíguas de memória.
- Por outro lado, como uma linha de *cache* diz respeito a várias posições contíguas de memória, quando um processador carrega o seu índice do vector `count[]` para a sua *cache*, está também a carregar as posições contíguas do vector que dizem respeito a índices de outros processadores.
- Sempre que um processador escreve para o seu índice do vector `count[]`, todos os outros índices na mesma linha de *cache* têm que ser invalidados nas *caches* dos restantes processadores que partilhem essa linha. A linha de *cache* irá então saltar entre as *caches* dos diferentes processadores, originando a perca de localidade temporal, o que impossibilitará qualquer eventual ganho de desempenho.

Falsa Partilha

- O problema do exemplo anterior poderia ser solucionado do seguinte modo.

```
int count[2];
#pragma omp parallel private(count)  // o vector é privado a cada thread
{
  count[0] = count[1] = 0;
  #pragma omp for private(i,index) schedule(static)
  for (i = 0; i < N; i++) {
    index = a[i] % 2;
    count[index]++;  // e passa a estar em linhas de cache diferentes
  }
  #pragma omp atomic
  even += count[0];
  #pragma omp atomic
  odd += count[1];
}
```


Paralelização Inconsistente

- Outra situação em que pode não ser possível conseguir uma boa localidade, acontece quando existem vários ciclos sobre determinados dados e nem todos os ciclos são susceptíveis de serem paralelizados. Considere, por exemplo, o código abaixo onde se paraleliza um primeiro ciclo sobre o vector `a[]` e não se consegue paralelizar um segundo ciclo sobre o mesmo vector.

```
#pragma omp parallel for private(i)
for (i = 0; i < N; i++)
  a[i] = func(i);
// o segundo ciclo não pode ser paralelizado
for (i = 0; i < N; i++)
  a[i] = a[i] + a[i-1];
```

- Se as *caches* agregadas dos processadores disponíveis conseguirem albergar por completo o vector `a[]`, então a paralelização do primeiro ciclo pode ser uma má decisão. Ao paralelizar o primeiro ciclo, o vector `a[]` fica dividido pelas *caches* dos diferentes processadores. Ao executar o segundo ciclo, o *master thread* tem que recolher os dados a partir de todas essas *caches*, o que poderá ter um custo superior ao ganho conseguido com a execução do primeiro ciclo em paralelo.

Sincronização com Barreiras

- As barreiras permitem implementar pontos de sincronização globais. No entanto, o seu uso indiscriminado e pouco cuidado pode revelar-se bastante dispendioso. Sempre que possível devemos evitar a sincronização com barreiras, sejam elas **barreiras explícitas ou implícitas**.

```
#pragma omp parallel for private(i)
for (i = 0; i < N; i++)
  a[i] += func_a(i);
// barreira implícita no final da região paralela
#pragma omp parallel for private(i)
for (i = 0; i < N; i++)
  b[i] += func_b(i);
// barreira implícita no final da região paralela
#pragma omp parallel for private(i) reduction(+:sum)
for (i = 0; i < N; i++)
  sum += a[i] + b[i];
// barreira implícita no final da região paralela
```

- Ao completarem uma região paralela, o *team of threads* sincroniza numa barreira implícita com o *master thread*. Caso existam N regiões paralelas consecutivas, o *team of threads* necessita de sincronizar N vezes com o *master thread*.

Sincronização com Barreiras

- Sempre que essas N regiões paralelas não apresentem dependências críticas entre os dados, deve-se **juntá-las numa só região** minimizando assim o número de sincronizações entre o *team of threads* e o *master thread*.

```
#pragma omp parallel private(i)
{
  #pragma omp for
  for (i = 0; i < N; i++)
    a[i] += func_a(i);
  // barreira implícita no final do construtor de work-sharing
  #pragma omp for
  for (i = 0; i < N; i++)
    b[i] += func_b(i);
  // barreira implícita no final do construtor de work-sharing
  #pragma omp for reduction(+:sum)
  for (i = 0; i < N; i++)
    sum += a[i] + b[i];
  // barreira implícita no final do construtor de work-sharing
}
```

- No entanto, ao completarem uma região delimitada por um construtor de *work-sharing*, todos os *threads* sincronizam novamente numa barreira implícita.

Sincronização com Barreiras

- Sempre que for seguro eliminar barreiras implícitas em construtores de *work-sharing*, deve-se utilizar a **cláusula nowait**.

```
#pragma omp parallel private(i)
{
  #pragma omp for nowait
  for (i = 0; i < N; i++)
    a[i] += func_a(i);
  #pragma omp for nowait
  for (i = 0; i < N; i++)
    b[i] += func_b(i);
  #pragma omp barrier  // igual a não colocar 'nowait' no último 'omp for'
  #pragma omp for reduction(+:sum) nowait
  for (i = 0; i < N; i++)
    sum += a[i] + b[i];
}
```

- De notar ainda que ter os dois primeiros ciclos `for` em separado é potencialmente melhor (em termos de localidade espacial) do que ter um único ciclo `for` em que ambos os vectores `a[]` e `b[]` são actualizados na mesma iteração.

Sincronização com Exclusão Mútua

- A exclusão mútua permite isolar a execução sobre regiões críticas de código. No entanto, o seu uso indiscriminado e pouco cuidado pode revelar-se ainda mais dispendioso do que a sincronização com barreiras.
- O custo de isolar a execução de uma região crítica pode revelar-se extremamente dispendioso quando existe um número elevado de processadores a tentar aceder simultaneamente à mesma região crítica. Nesse caso, a linha de *cache* que contém o *spinlock* utilizado para restringir o acesso à região crítica, **andará a saltar entre os diferentes processadores** à medida que estes obtêm o acesso ao *spinlock*.
- Para minimizar a contenção no acesso a uma região crítica devemos **minimizar o número de estruturas de dados protegidas pela mesma região crítica**. Por exemplo, ao manipularmos uma estrutura de dados em árvore, em lugar de isolar toda a árvore, podemos isolar apenas regiões ou nós específicos da árvore, o que permitirá o acesso simultâneo de vários processadores a diferentes partes da árvore.

Sincronização com Exclusão Mútua

- Para evitar ter um *spinlock* por estrutura de dados a isolar, podemos ter um **vector de** `spinlocks[]` e utilizar uma **função de *hashing*** como forma de indexar o *spinlock* a utilizar.

```
omp_lock_t locks[NLOCKS];
...
#pragma omp parallel private(index)
{
  ...
  index = hash_function(data);
  omp_set_lock(&lock[index]);
  ...  // região crítica
  omp_unset_lock(&lock[index]);
  ...
}
```