Memórias Cache em Arquiteturas Multiprocessador e Multicore

Ricardo Rocha

Departamento de Ciência de Computadores
Faculdade de Ciências
Universidade do Porto

Computação Paralela 2015/2016

Memórias Cache

A principal motivação para utilizar o OpenMP é conseguir maximizar a utilização do poder computacional disponível de modo a reduzir o tempo de resolução de uma determinada aplicação.

Apesar das diferentes arquiteturas multiprocessador e multicore existentes, estas apresentam caraterísticas fundamentais que são idênticas. Em particular, utilizam **memórias cache** muito perto dos processadores/cores de modo a minimizar o tempo necessário para aceder aos dados.

Caches e Desempenho

Do ponto de vista conceptual, não existe qualquer diferença entre manipular uma variável `A` ou uma variável `B`. No entanto, aceder a `A` num determinado momento pode ser **mais ou menos dispendioso** do que aceder a `B` ou do que aceder novamente a `A` um momento mais tarde.

De entre os vários fatores que podem limitar o desempenho da programação paralela, existem dois que estão intrinsecamente ligados às arquiteturas multiprocessador/multicore baseadas em memórias cache:

- **Localidade**
- **Sincronização**

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

Caches e Desempenho

Parte do tempo necessário para fazer chegar os dados em memória até ao processador/core 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/core mais rapidamente poderão lá chegar. No entanto, existe um limite físico na quantidade de memória que pode estar a uma determinada distância do processador/core.

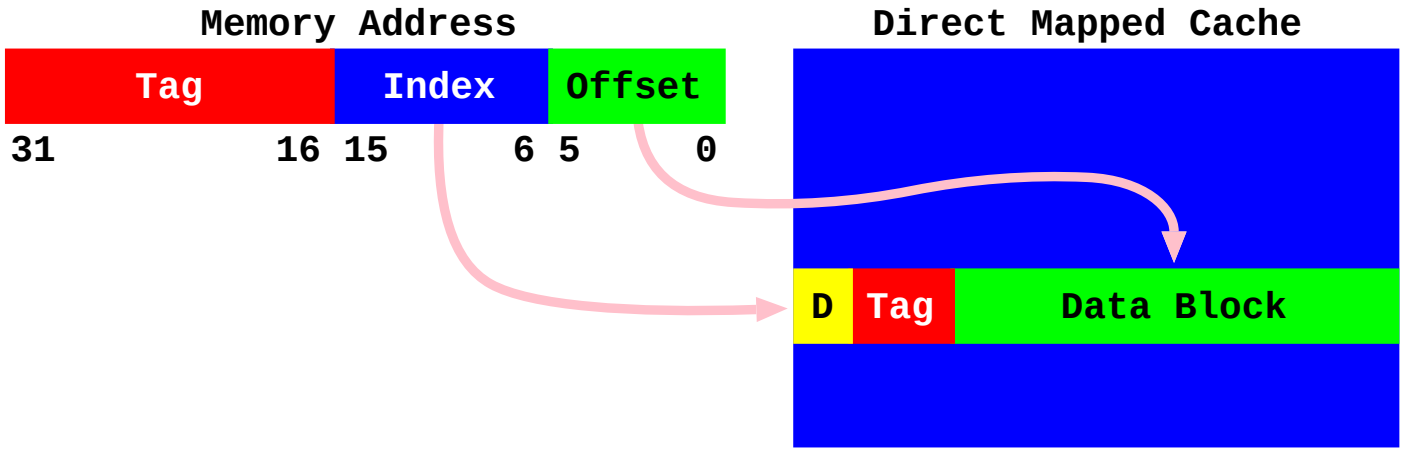
A motivação para usar memórias cache é conseguir maximizar as vezes que os dados necessários a uma determinada computação estão mais perto do processador numa memória mais pequena mas que seja bastante rápida de aceder, **minimizando assim o número de interacções com a memória principal do sistema**, que normalmente é mais lenta e está mais longe.

Direct Mapped Caches

Considere uma arquitetura 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**:

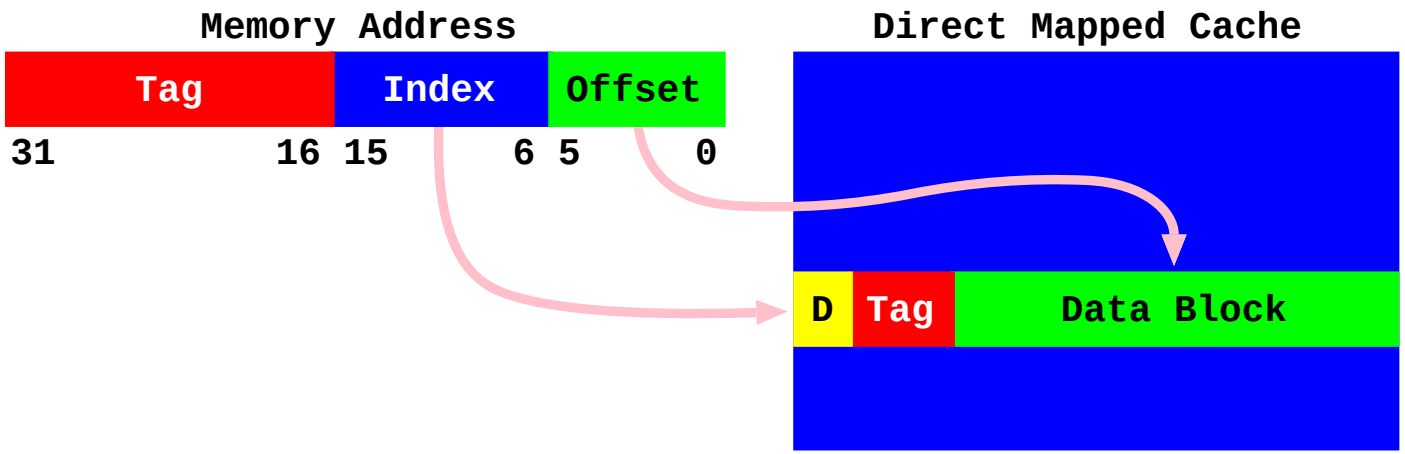
- Os bits 6–15 (2^{10}) indexam a linha de cache
- Os bits 16–31 (2^{16}) permitem verificar se o endereço está em cache
- Os bits 0–5 (2^6) indexam o byte respetivo dos 64 bytes de cada linha



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 6–15 são mapeados na mesma linha de cache.

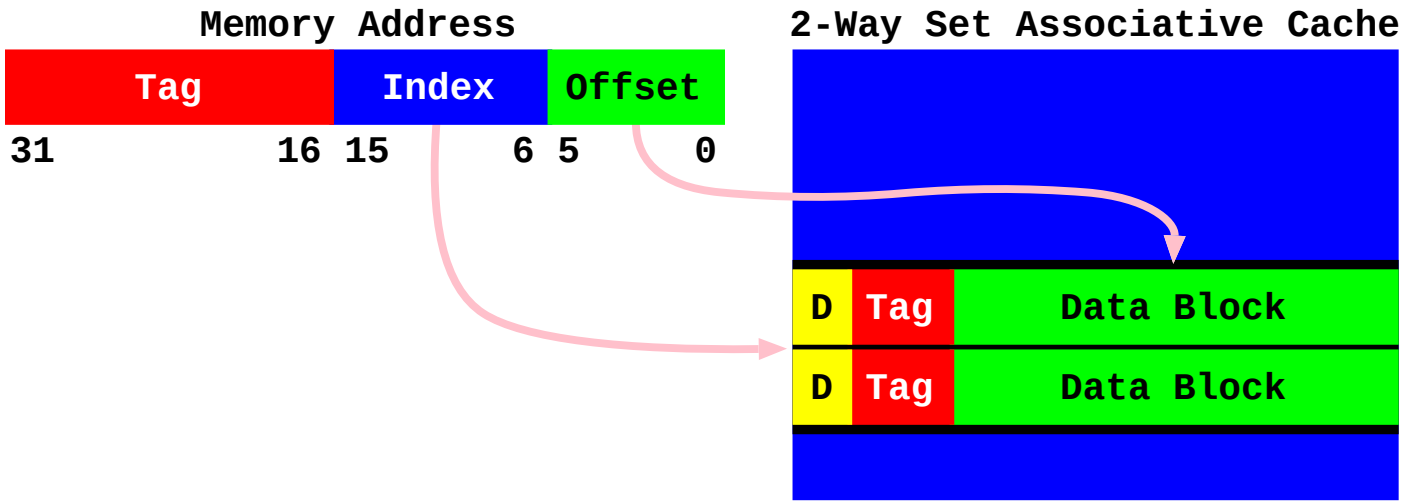
Quando ocorre um **cache miss** é necessário substituir a linha em cache pela linha que contém o endereço de memória pretendido. No caso de existirem alterações sobre os dados da linha a substituir, i.e., se o **dirty bit D** for 1, então essa linha deve ser primeiro escrita de volta para a memória principal. Caso contrário, a linha pode ser simplesmente substituída.



N-Way Set Associative Caches

Considere novamente uma arquitetura com endereços de memória de 32 bits, mas agora com uma **N-way set associative cache**, em que cada endereço de memória é **mapeado em N linhas de cache diferentes**.

Se considerarmos novamente uma cache com blocos de dados de 64 bytes por linha e os bits 6–15 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 6–15).



Localidade Espacial e 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 curto 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 curto espaço de tempo.

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

Localidade Espacial e Temporal

Em geral, a **localidade espacial é mais fácil de conseguir do que a localidade temporal**. No entanto, para obter boa localidade espacial 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 = 0; j < N; j++)
    a[i][j] = scale * a[i][j];
```

Localidade Espacial e Temporal

Em arquiteturas multiprocessador e multicore, para além de uma boa localidade espacial e temporal por processador/core, é necessário restringir essa localidade a cada processador/core, i.e., **evitar que mais do que um processador/core aceda às mesmas linhas de cache no mesmo espaço de tempo**.

Aceder a dados que estão na cache de outro processador/core, **pode ser pior do que aceder a dados na memória principal**. Vamos analisar 3 situações diferentes que evidenciam este tipo de comportamento:

- Escalonamento de ciclos consecutivos
- Falsa partilha
- Paralelização inconsistente

Escalonamento de Ciclos Consecutivos

Considere o código que se segue onde uma determinada matriz é percorrida por duas vezes.

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

Se executarmos o código com um diferente número de threads, será que o speedup obtido será diferente se utilizarmos um escalonamento estático **schedule(static)** ou um escalonamento dinâmico **schedule(dynamic)**?

Escalonamento de Ciclos Consecutivos

Consideremos o caso de matrizes de tamanho 400x400, 1000x1000 e 4000x4000 e uma máquina 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.

Consideremos ainda a execução com 1 e 8 threads utilizando escalonamento estático e escalonamento dinâmico.

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

Escalonamento de Ciclos 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 conseguem albergar as matrizes por completo, o escalonamento estático é cerca de 10x mais rápido do que o dinâmico. Isto acontece porque ao percorrermos a matriz pela segunda vez, o escalonamento estático **atribuí as mesmas porções da matriz a cada thread** 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 (18.3).

Com escalonamento dinâmico, essa localidade é perdida e o custo de aceder a dados que estão em outras caches revela-se bastante elevado.

Escalonamento de Ciclos 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

Existem situações em que, apesar do balanceamento de carga ser perfeito, uma boa localidade nem sempre é possível. Considere o caso em que se pretende calcular o número de elementos pares e ímpares de um vetor `a[]`.

```
int count[NTHREADS][2]; // shared between all 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]++; // each thread updates different positions
  }
  #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 vetor `count[]` é representado. Apesar de cada thread escrever em posições diferentes do vetor, essas posições correspondem a **posições contíguas de memória**.

Como uma linha de cache diz respeito a várias posições contíguas de memória, quando um thread carrega o seu índice do vetor `count[]` para a sua cache, está também a carregar as posições contíguas do vetor que dizem respeito a índices de outros threads.

Sempre que um thread escreve para o seu índice do vetor `count[]`, todos os outros índices na mesma linha de cache têm que ser invalidados nas caches dos restantes threads que partilhem essa linha. A **linha de cache irá então saltar entre as caches dos diferentes threads**, originando a perca de localidade temporal (**falsa partilha**), o que impossibilitará qualquer eventual ganho de desempenho.

Falsa Partilha

A falsa partilha do exemplo de calcular o número de elementos pares e ímpares de um vetor poderia ser solucionado do seguinte modo.

```
int count[2];
#pragma omp parallel private(count) // count is private to each 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]++; // ... and is now at different cache lines
  }
  #pragma omp atomic
  even += count[0];
  #pragma omp atomic
  odd += count[1];
}
```

Paralelização Inconsistente

Outra situação em que nem sempre é possível conseguir uma boa localidade acontece quando nem todos os ciclos sobre determinados dados são suscetíveis de serem paralelizados. Considere o caso em que não se consegue paralelizar um segundo ciclo sobre um vetor `a[]`.

```
#pragma omp parallel for private(i)
for (i = 0; i < N; i++)
  a[i] = func(i);
// cannot parallelize this cycle
for (i = 0; i < N; i++)
  a[i] = a[i] + a[i-1];
```

Se as caches agregadas conseguirem albergar por completo o vetor `a[]`, então o vetor `a[]` fica dividido pelas caches dos vários threads. 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** e como tal o seu uso indiscriminado pode revelar-se bastante dispendioso. Sempre que possível devemos evitar a sincronização com barreiras, sejam elas **barreiras explícitas ou barreiras implícitas**.

```
#pragma omp parallel for private(i)
for (i = 0; i < N; i++)
  a[i] += func_a(i); // implicit barrier at exit
#pragma omp parallel for private(i)
for (i = 0; i < N; i++)
  b[i] += func_b(i); // implicit barrier at exit
#pragma omp parallel for private(i) reduction(+:sum)
for (i = 0; i < N; i++)
  sum += a[i] + b[i]; // implicit barrier at exit
```

Ao completar uma região paralela, o master thread sincroniza numa barreira implícita com o team of threads. No caso de N regiões paralelas consecutivas, o master thread sincroniza N vezes com o team of threads.

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 master thread e o team of threads.

```
#pragma omp parallel private(i)
{
  #pragma omp for
  for (i = 0; i < N; i++)
    a[i] += func_a(i); // implicit barrier at exit
  #pragma omp for
  for (i = 0; i < N; i++)
    b[i] += func_b(i); // implicit barrier at exit
  #pragma omp for reduction(+:sum)
  for (i = 0; i < N; i++)
    sum += a[i] + b[i]; // implicit barrier at exit
}
```

Mas ao completar uma região delimitada por um construtor work-sharing, todos os threads sincronizam igualmente 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 // same as not adding 'nowait' to last 'omp for'
  #pragma omp for reduction(+:sum) nowait
  for (i = 0; i < N; i++)
    sum += a[i] + b[i];
}
```

De notar ainda que, em termos de localidade espacial, ter os dois primeiros ciclos `for` separados é potencialmente melhor do que ter um único ciclo `for` em que os vetores `a[]` e `b[]` são atualizados na mesma iteração.

Sincronização com Exclusão Mútua

A exclusão mútua permite **isolar a execução sobre regiões críticas de código**. O custo de isolar a execução de uma região crítica pode revelar-se igualmente bastante dispendioso quando existe um número elevado de threads a tentar aceder simultaneamente à mesma região crítica.

Por exemplo, no caso de utilizarmos um spinlock para restringir o acesso a uma região crítica, a linha de cache que contém o spinlock **andará a saltar entre as caches dos diferentes threads** 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 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 threads a diferentes partes da árvore.

Sincronização com Exclusão Mútua

Para evitar ter um spinlock por estrutura de dados a isolar, podemos ter um vetor 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]);
  ... // critical section
  omp_unset_lock(&lock[index]);
  ...
}
```