PMC Lecture 15

Gianmaria Romano

9 December 2025

Chapter 5

Shared memory systems

5.1 Caching in single-core architectures

A cache is a small and fast memory component that stores copies of data from slower memory resources, allowing these data to be accessed more quickly.
Caches exploit the locality of reference, which comes in two main forms:

- **Spatial locality:** if a memory address is accessed, nearby memory locations are likely to be accessed soon.
- **Temporal locality:** if a memory address is accessed, it is likely to be accessed again in the near future.

To improve performance, data is transferred between memory and the cache in fixed-size blocks or cache lines.
While this approach introduces some overhead, it benefits performance because accessing data in larger contiguous chunks makes better use of spatial locality and reduces the number of slow memory accesses.

5.1.1 Cache levels

Generally speaking, a cache is organized as a hierarchy consisting of multiple levels, typically known as L1, L2 and L3 cache structures.
Whenever the processor needs to access some data, it searches the cache hierarchy starting from the closest and fastest section, which corresponds to the L1 cache, and going down until the resource is found.
However, if the resource is not present in the cache, the processor will need to fetch it from the main memory, which is significantly slower.

5.1.2 Cache consistency

To avoid the risk of inconsistency between cache and memory, caches make use of policies that determine how and when updates are propagated to the main

memory.
Generally speaking, it is possible to handle consistency by implementing a write-through cache, which updates data directly in the main memory, or a write-back cache, which marks data in the cache until they are updated.

5.2 Caching for multicore architectures

When dealing with multicore architectures, different cores my hold copies of the same memory addresses in their caches, requiring the system to ensure that these copies stay consistent over time.

- Programmers have no control over caches and when they get updated.

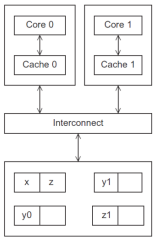


Figure 2.17
A shared memory system with two cores and two caches

Generally speaking, cache coherence in multicore architectures can be handled using one of the following approaches:

- **Snooping:** all processors share a bus and, whenever a processor updates a variable, it will broadcast the information to other processors to allow the to update or invalidate their copies.
- **Directory-based cache coherence:** a directory keeps track of which processors hold each cache line and, whenever a processor updates a cache line, the directory notifies the intended processors to update or invalidate their copies.

However, since invalidating a variable requires invalidating its entire cache line, cache coherence may lead to false sharing, where multiple processors share cache lines without actually accessing data.
A common solution consists in forcing these variables in different cache lines, often by applying additional padding.

- Original

```
double x[N];
#pragma omp parallel for schedule(static, 1)
  for( int i = 0; i < N; i++ )
     x[ i ] = someFunc( x [ i ] );
```

- Padded:

```
double x[N][8];
#pragma omp parallel for schedule(static, 1)
  for( int i = 0; i < N; i++ )
     x[ i ][ 0 ] = someFunc( x [ i ][ 0 ] );
```

Overall, however, applying padding requires using additional memory, often reducing cache efficiency.
For this reason, it might be more convenient to have each processor work on a local copy of the variable and merge their results at the end, allowing to significantly mitigate the issue of false sharing.
Example: The following code provides a PThreads implementation of matrix–vector multiplication:

```c
void *Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m / thread_count;
    int my_first_row = my_rank * local_m;
    int my_last_row = (my_rank + 1) * local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++) {
            y[i] += A[i][j] * x[j];
        }
    }

    return NULL;
}
```

Run the code using a single thread on vectors and matrices of various sizes:

Matrix size denoted as # Rows x # Cols

	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000

Notice, however, that, for larger input data, performance is dominated by memory-access costs.
In fact, the input vector might be too large to fit in the cache, requiring frequent transfers from the main memory, which increase the overall execution time.
Now, try running the same code using multiple threads:

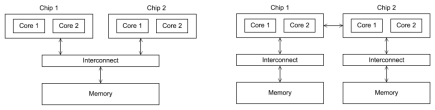
Matrix size denoted as # Rows x # Cols

	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

While all the versions experience reduced efficiency, the last configuration shows the largest performance drop as it is the most subjected to false sharing due to having a smaller output vector that could fit within a single cache line.

5.3 Memory organization for multicore architectures

Generally speaking, memory organization for multicore architectures can follow a uniform memory access model, in which all processors share a single memory space with the same access latency, or a non-uniform memory access model, in which processors may access different memory regions, each with their access cost depending on where the data is allocated.



UMA
(Uniform Memory Access)

NUMA
(Non-Uniform Memory Access)