## 1.4 Multicore architectures

Two processors on the same die, L2 or L3 caches are shared between two cores, every core has its own L1 caches.

To make different cores work coherently, a cache line in the main memory with respect to a data has a specific state: scratch, valid, reserved, dirty, invalid.

Two basic mechanisms for realizing cache coherence

1. snooping

Any request for data is sent to all caches, and the data is returned if it is present anywhere; otherwise it is retrieved from memory.

2. directory-based schemes

A central directory that contains the information on what data is present in some cache, and what cache it is in specifically.

Multicore processor/chips

MPI library (used to communicate between processors that are connected through a network, can also be used in a single multicore processor.)

Shared memory and shared caches and program using threaded systems such as OpenMP.

## 1.5 Node architecture and sockets

The cluster node can have more than one socket.

## 1.6 Locality and data reuse

Arithmetic intensity, if n is the number of data items that an algorithm operates on, and f(n) the number of operations it takes, then the arithmetic intensity is f(n)/n.

Roofline model, the peak performance and the number of operations per second, bound the performance.

Locality:

1. temporal locality

If in between the two references less data has been referenced than the cache size, the element will still be in cache and therefore be quick accessible.

2. spatial locality

If it references memory that is close to memory it already referenced in physical address, the program is said to exhibit spatial locality.

## 1.7 Programming strategies for high performance

1. Peak performance

2. Pipelining

3. Cache size

4. Cache lines

5. TLB

6. Cache associativity

7. Loop nests

8. Loop tiling

9. Optimization Strategies

10. Cache aware and cache oblivious programming

**1.8 Further topic**

1. Power consumption

2. Operating system effects

**Questions**

1. How to solve false sharing?

2. How does the associativity effect the code performance?

Exercise 1.12

```
for i = 1, l
    for j = 1, m
        for k = 1, n
            $c_{i,j} += a_{i,k} * b_{k,j}$
```

In loop k, $c_{i,j}$ can be kept in register and only written back to the memory at the end.

In loop j, $a_{i,k}$ ($a_{i,1} \sim a_{i,n}$) can be kept in the register until the loop goes to end.

The length of cache compared to the space that $a_{i,k}$ ($a_{i,1} \sim a_{i,n}$) should occupied. If the length of cache is larger, the data can be reused; if the length of cache is small, the data can not be reused.

Exercise 1.15

1. The reusing of data is poor which makes the performance low.

2. As long as the cache size is small to the length scale, it does not matter with the performance.

3. Lower associativity makes the performance better.

The following code would perform better. Since nvectors is small compared to the cache size, array c can be stored into the register which can be reused for every length-loop.

Exercise 1.19

Exe 1.15 is an example of doubly-nested loop where loops can be exchanged.

Exe 1.14 is an example where loops can not be exchanged.