# Chapter 3: Multithreading with OpenMP



## Hyper-threading

So far we have discussed the simple case of each CPU core running only one thread. However, some high-end CPUs are able to run multiple threads per core. This is known as hyper-threading.

From the perspective of the operating system, a 4-core CPU with hyper-threading looks a lot like an 8-core CPU. The operating system can all the time keep 8 threads running on the CPU. However, from the perspective of the programmer, things get more complicated if we are interested in the performance.

### Hyper-threading helps to keep the CPU busy

Here is a table that summarizes the key points:

| CPU type | CPU sees... | CPU has... |
|---|---|---|
| 4 cores, no hyper-threading | 4 threads | 4 copies of all execution units |
| 4 cores, with hyper-threading | 8 threads | 4 copies of all execution units |
| 8 cores, no hyper-threading | 8 threads | 8 copies of all execution units |

It is important to note that hyper-threading does not help with the maximum throughput of the execution units. For example:

- a 4-core Intel Skylake CPU without hyper-threading can do 64 floating-point additions per clock cycle
- a 4-core Intel Skylake CPU with hyper-threading can do 64 floating-point additions per clock cycle.

However, hyper-threading may help to keep the CPU busy. When we are fully utilizing the maximum performance of the CPU, each core has to have e.g. 8 vector additions in progress at the same time. If we do not have hyper-threading, it means that in each thread of execution, we must have lots of opportunities for instruction-level parallelism; at any point of time each CPU core has to be able to see at least 8 independent vector additions that are ready for execution in order to keep its pipelines busy.

With hyper-threading, each CPU core has access to two threads of execution and hence two instruction streams. Even if each thread has only fairly limited opportunities for parallelism, if the CPU can look at both of the threads at the same time, it can more easily find something that is ready to be scheduled for execution.

### When it might help

Hyper-threading is helpful in applications in which it is difficult to find opportunities for parallelism in the small scale (vectorization, instruction-level parallelism) but easy to find opportunities for parallelism in the large scale (multithreading).

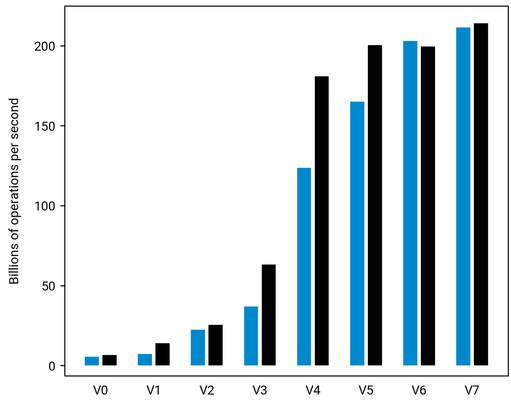
A simple example is sorting. For example, in a parallel quicksort implementation it is easy to do multiple "partition" operations for different parts of the input simultaneously in parallel. However, it is more difficult to figure out how to parallelize each individual "partition" operation with the help of vector instructions and instruction-level parallelism. It is also quite complicated to try to do multiple "partition" operations in an interleaved manner with the help of vector instructions or instruction-level parallelism. Here hyper-threading may give a nice performance boost.

Here are examples of the performance of sequential and parallel `std::sort` implementations in the GNU C++ library (libstdc++); we are sorting 100 million 64-bit integers on a 4-core computer with hyper-threading:

| Function | Threads | Time |
|---|---|---|
| `std::sort` | 1 | 8.1s |
| `__gnu_parallel::sort` | 1 | 8.2s |
| `__gnu_parallel::sort` | 4 | 2.3s |
| `__gnu_parallel::sort` | 8 | 1.7s |

### When it might not help

Let us now look at the application that we developed and optimized in Chapter 2. Here are benchmarks of our solutions using Intel Xeon E3-1230v5, a 4-core processor that supports hyper-threading:




The blue bars are without hyper-threading (4 threads), while the black bars are with hyper-threading (8 threads).

Some of the first versions that we developed indeed benefit from hyperthreading significantly. For example, in V1 we had a sequential dependency chain that prevented us from doing more than one "min" operation at a time in each core. Now we can almost double the throughput, as we have got two threads per core, each of which has always one "min" operation ready for execution.

However, for the fastest version, V7, the benefit that we get from hyper-threading is negligible — and this is to be expected, as we are already very close to the theoretical maximum performance of the arithmetic units of the CPU.