Multithreading with OpenMP

Sistemi Digitali M

A.A. 2024/2025

Stefano Mattoccia

Università di Bologna

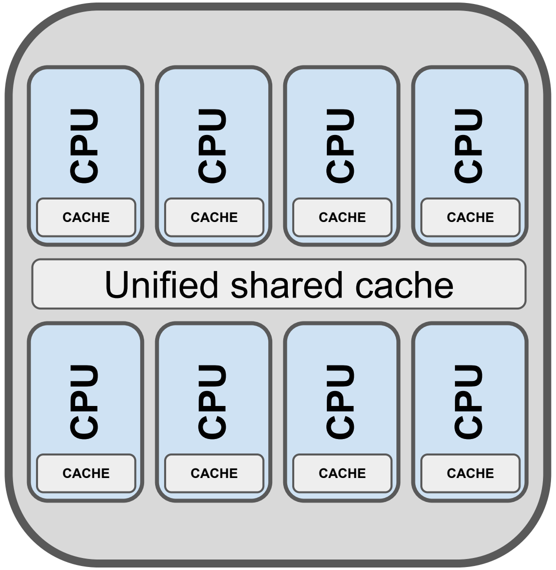
Introduction

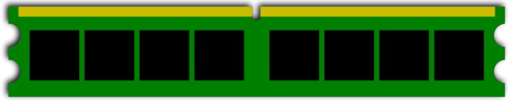
- Before moving forward, let’s briefly review different strategies to depart from a pure/conventional sequential computation:
  - **Instruction Level Parallelism** (**ILP**): executes more than one instruction at the same time and is typically put in place by CPUs equipped with multiple processing pipelines (ie, superscalar processors)
  - **Data Level Parallelism** (**DLP**): processes multiple data with a single execution stream, for instance, with SIMD instructions on CPUs
  - **Thread Level Parallelism** (**TLP**): executes multiple instruction streams, possibly on various processing units, for instance, exploiting multi-core CPU systems

The good thing is that all these paradigms are **NOT mutually** exclusive and can be synergically exploited to achieve the highest performance

Overview of a multicore CPU architecture

- Multicore architectures aim to improve performance with limited power consumption using reasonable operating frequencies
- A typical multicore processor consists of multiple CPUs, each one with its cache hierarchy (eg, L1 and L2) and often a shared unified cache memory (eg, L3)
- All the CPU cores have a unique and shared address space with conventional external memory devices like DDRs and ROMs
- Since the cores have the same addressing space, they can communicate through the shared memory
- However, this strategy also induces issues regarding race conditions and data consistency



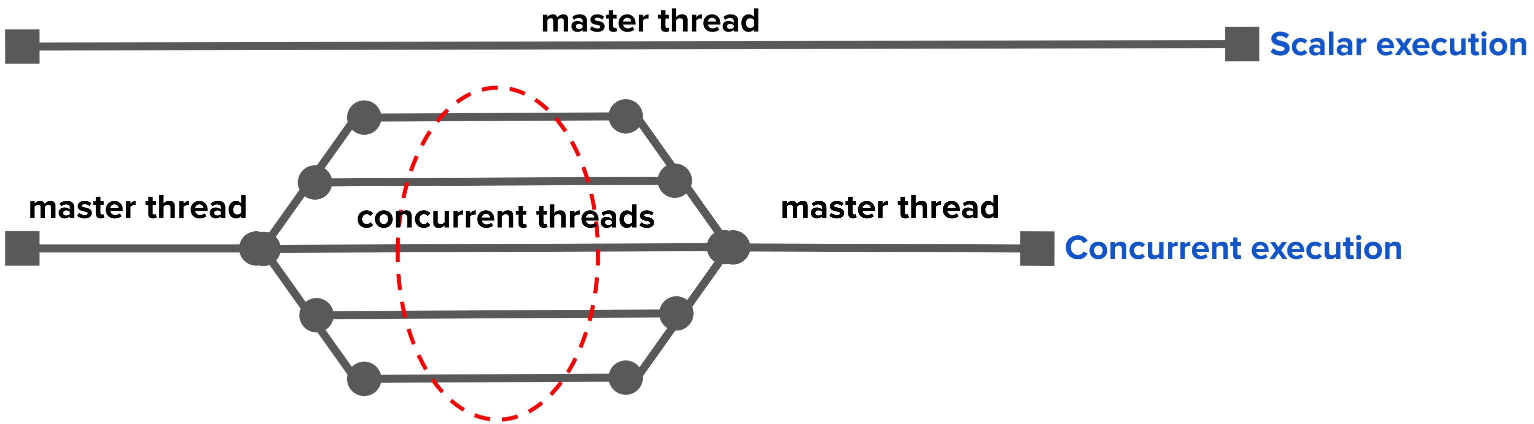


Threads

- A thread is a lightweight process created to run concurrently with others on the available processing resources
- When more than one CPU is available, multiple threads can concurrently run on each CPU, communicating through the shared address space
- Distributing the workload across multiple cores/threads can significantly speed up the computation
- However, excluding *embarrassingly parallel* problems, such a parallel execution paradigm is prone to challenging issues not present with a scalar execution
- For instance, in a scalar execution, access to data structures occurs according to a specific order, and the same should happen when the same code is spread across multiple threads to avoid *race conditions* and, as a consequence, obtaining different results due to a different order of events

Concurrent threads and issues 1/2

- Since multiple threads run concurrently, their behaviour needs to be properly managed to obtain the desired result (ie, the same as the conventional scalar code)



- During the concurrent execution, each thread can autonomously read or write shared data, even at the same time (ie, resulting in a *data race*)
- Moreover, as in the scalar code, the order of operations performed by each thread should be preserved to avoid *race conditions*
-

Concurrent threads and issues 2/2

- Unfortunately, the outcome could not be deterministic due to the concurrent operations yielding non-identical outcomes across different executions of the same code
- Moreover, such a non-deterministic behaviour might show only sporadically, making this kind of error complicated to spot
- This evidence might occur regardless of the number of CPU cores since even with a single CPU, the scheduler of the operating systems might induce undesired behaviour across different executions
- Of course, increasing the number of threads and cores is more likely to occur.
- Although it is much more challenging to write parallel/concurrent code than a scalar version, solutions exist to write safe equivalent code, which can be much more efficient by taking advantage of multiple cores

OpenMP

- OpenMP (https://www.openmp.org/)is a library providing an API for writing multithreaded applications
- It is available for almost any Operating System and computing device (including GPUs and other hardware accelerators with the most recent versions)
- Widely used for multicore CPUs, it supports C/C++ (and Fortran)
- It enables the execution of code across multiple threads (and hopefully physical CPU cores) with minimal or null modification to the code, primarily using compiler directives
- It allows to manage synchronization among threads, methods to access shared data safely, plus many other features not discussed in the remainder
- An alternative yet much less popular solution is Intel TBB

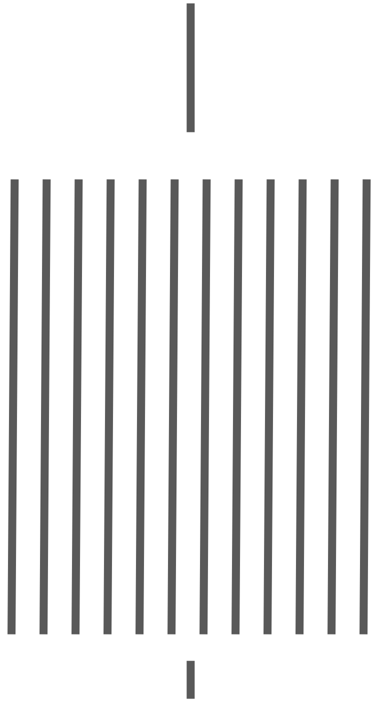
Hello World with OpenMP 1/2

```
#include <stdio.h>
#include <omp.h>

int main()
{  omp_set_num_threads(12);
   printf("Thre is %d thread running now\n", omp_get_num_threads());

#pragma omp parallel
{
   printf("There are %d threads running now \n", omp_get_num_threads());
   int ID = omp_get_thread_num();
   printf("This ");
   printf("is ");
   printf("Thread %d: ", ID);
   printf("Hello ");
   printf("World ");
   printf("with ");
   printf("OpenMP\n");
}

}
```

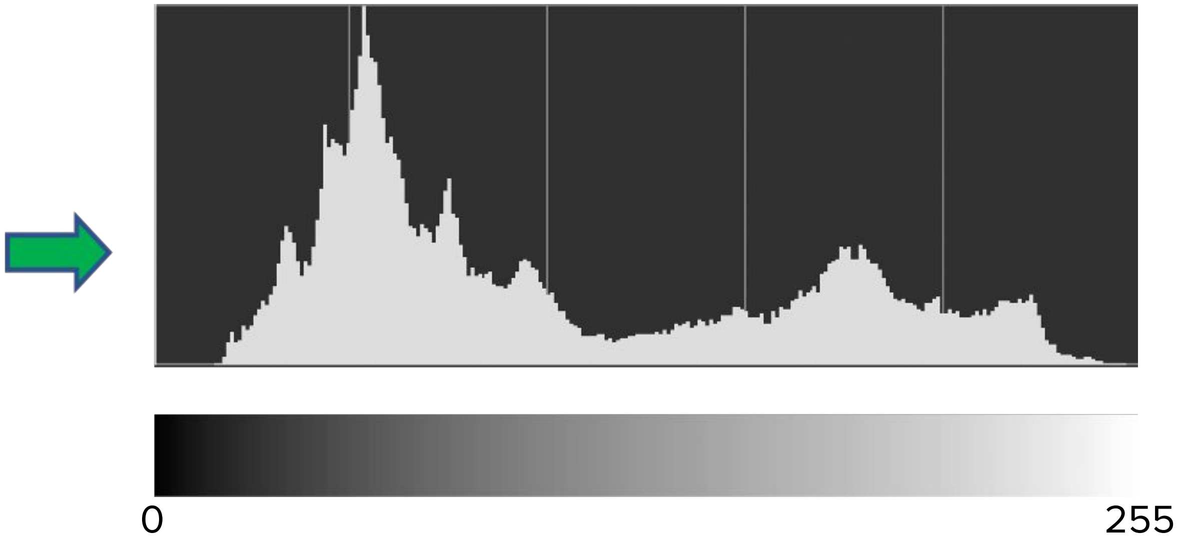


Hello World with OpenMP 2/2

- In the previous example, a single master thread is initially executed, and then, in the parallel section, 12 threads run concurrently until each of them ends its execution
- The parallel section is defined with the OpenMP directive: **`#pragma omp parallel`**
- Of course, thread execution is managed by the OS scheduler according to the resources available and the workload
- Although not at each code execution, weird behaviour might occur in the standard output (a shared resource among all threads)
- This evidence already highlights a major problem regarding the fact that all threads in the example run concurrently without caring at all about what others do (eg, if another thread uses the standard output) in this specific case
- Let’s consider now a more insightful example

Use case: histogram computation

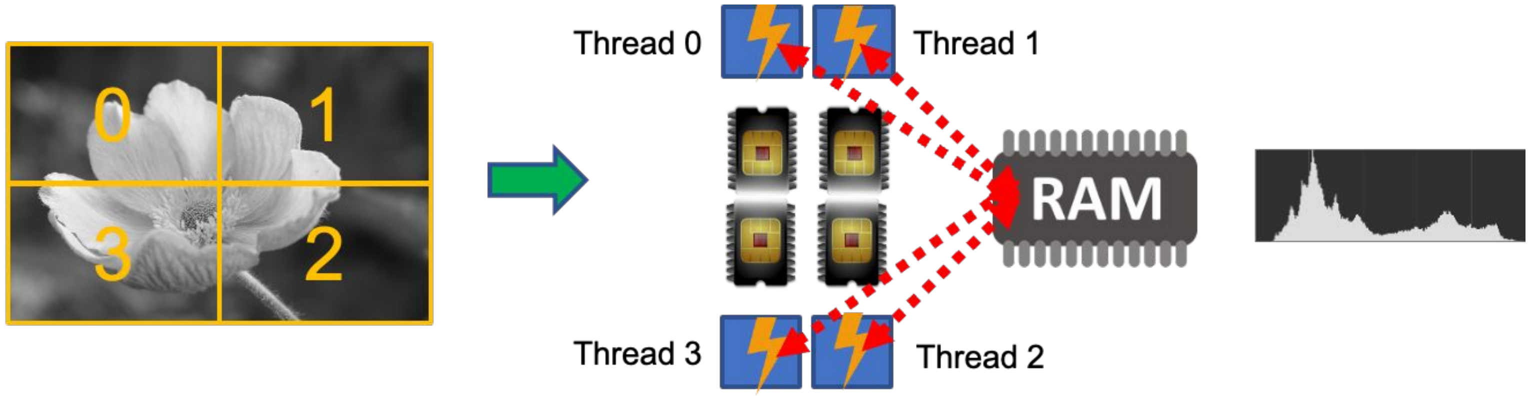
- Let’s consider as a simple use case histogram computation
- It aims at measuring the number of occurrences of each element (eg, color/grayscale image) value within a set (eg, an image)
- A scalar implementation is trivial (see file `histogram_with_openMP.c`)



Histogram computation using parallel for

- Since OpenMP allows spreading the computation across multiple threads, we could use the strategy below using the **`#pragma omp parallel for`** directive as follows:

```
#pragma omp parallel for
   for (int i=0; i<LENGHT; i++)
   {
       histogram_parallel[Vector[i]] = histogram_parallel[Vector[i]] + 1;
   }
```



Parallel histogram computation issue

- The parallel for directive splits the loop across multiple threads with the directive **`#pragma omp parallel for`**
- The number of threads is set with the function **`void omp_set_num_threads(int num_threads)`**
- All threads are implicitly synchronized at the end of the loop and all variables defined outside the parallel section are shared by default, while variables defined within the parallel section and loop indexes are private to each thread
- However, splitting the histogram computation across multiple concurrent threads raises a major issue since each thread reads and updates the shared data structure histogram (i.e., hist[i] = hist[i] +1)
- This fact results in a data race yielding wrong results because hist[i] is written by one or more threads and read by others concurrently during its update

Parallel histogram computation with atomic operations

- The first solution consists of performing the histogram update with atomic operation to avoid data races
- OpenMP provides support for this operation through the **`#pragma omp atomic`**

```
#pragma omp parallel for
   for (int i=0; i<LENGHT; i++)
   {
#pragma omp atomic  // check the critical directive too
       histogram_parallel[Vector[i]] = histogram_parallel[Vector[i]] + 1;
   }
```

- The **`#pragma omp critical`** allows a single thread to access the data structure, enabling atomic operations even when the threads access the data structure with different indexes, but yields poor performance than **`#pragma omp atomic`**

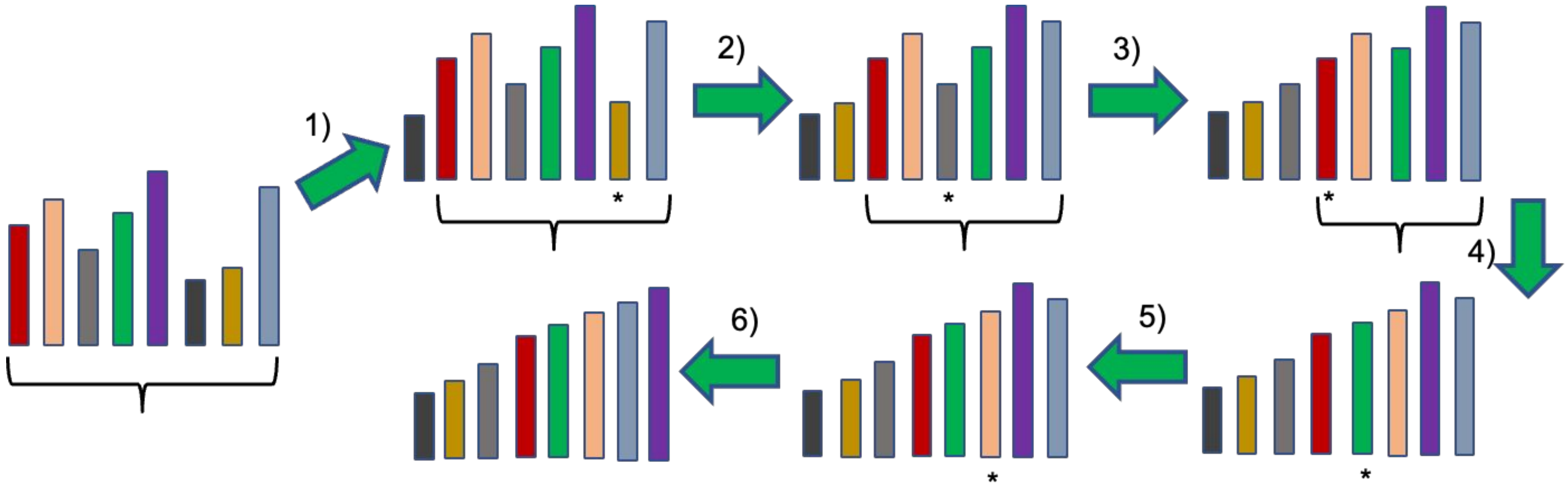
Parallel histogram computation with duplicated data structures

- The second solution consists of performing the histogram update within a private data structure (ie, a private histogram) for each thread, avoiding at all data races
- However, this strategy requires:
  - duplicated data structures (ie, one private histogram for each thread)
  - code modifications
  - an additional final (parallel) loop to merge all the histograms
- Nonetheless, it is more efficient than the previous solution in most cases
- See file `histogram_with_openMP.c` for details

Exercise

Exercise

Consider the Selection sorting algorithm and devise parallel strategies to speed up its computation by comparing the results with the scalar version.



References

Timothy G. Mattson, Yun (Helen) He, Alice E. Koniges, The OpenMP Common Core: Making OpenMP Simple Again, MIT Press, 2019