Parallelization of the Mean Shift Clustering with OpenMP

Emilio Cecchini

June 13, 2019

Università degli Studi di Firenze
emilio.cecchini@stud.unfi.it

Overview

The Mean Shift Clustering

Mean Shift key concepts

- Non-parametric technique to find the maxima of a density function.
- At each step, a *kernel function* is applied to each point that causes the points to shift in the direction of the local maxima determined by the kernel.

Gaussian Kernel

- There are many different types of kernel, the most used is the *Gaussian kernel*:

$$k(x) = e^{-\frac{x}{2\sigma^2}}$$

- The standard deviation σ is the bandwidth parameter, with a high bandwith value you will get a few large clusters and vice versa.

Mean Shift Clustering

Suppose x is a point to be shifted and $N(x)$ are the sets of points near to that point. Let $dist(x, x_i)$ be the distance from the point x to the point x_i. The new position x' where x has to be shifted is computed as follows:

$$x' = \frac{\sum_{x_i \in N(x)} k(dist(x, x_i)^2) x_i}{\sum_{x_i \in N(x)} k(dist(x, x_i)^2)}$$

The mean shift algorithm applies that formula to each point iteratively until they converge, that is until the position does not change.

Sequential implementation

Sequential implementation

Algorithm 1 Mean shift core

```
1: while allPointsHaveStoppedShifting() do
2:     for each point p do
3:         if hasStoppedShifting(p) then
4:             continue
5:         end if
6:         shift(p)
7:     end for
8: end while
```

OpenMP

Implicit threads

- With *implicit threads* there is no need to restructure the sequential program.
- It simplifies the process of parallelization of a program.

OpenMP

- OpenMP is an API for shared-memory programming.
- The program is parallelized with *compiler directives*
- It follows a *fork-join* thread model.

Parallelization of the Mean Shift with OpenMP

Parallelization of the Mean Shift with OpenMP

The mean shift algorithm is a embarrassingly parallel work: each point perform its shifting independently from the other points. This makes it the perfect case for using the OpenMP technology.

Sequential version

Algorithm 2 Mean shift core parallel

```
1: while allPointsHaveStoppedShifting() do
2:     #pragma omp parallel for schedule(dynamic)
3:     for each point p do
4:         if hasStoppedShifting(p) then
5:             continue
6:         end if
7:         shift(p)
8:     end for
9: end while
```

Differences with the sequential version

- The only difference from the sequential version is the `pragma` statement.
- The `pragma` statement is placed just before the `for` loop, in this way there is no need of any critical sections. If it had been placed before the `while` loop, then the parallel algorithm would have been more complex introducing an overhead due to the synchronization between threads.

Data structures

- The main data structures used by the algorithm are two lists of points.
- The first list contains the original points. It never changes during the algorithm, so it can be easly shared between threads.
- The second list contains the new positions of the points. That list is changed during the algorithm, but each point performs its shifting independently from all the other points, so even in this case no synchronization are needed.

The problem of static scheduling

- Not all the points need the same number of steps to reach the final position.
- By default, OpenMP uses a *static scheduling*, where the entire for loop is divided statically in chunks of equal size.
- It could be happen that a thread finishes very soon its iterations because all its points have stopped shifting and then it has to wait the other threads wasting computational resources.

Dynamic scheduling

- The best scheduling strategy for this algorithm is the *dynamic scheduling*, where the iterations are assigned to the threads while the loop is executing.
- To perform a dynamic scheduling we have to write in the `pragma` statement the clause `schedule(dynamic)`.

Dynamic vs static scheduling

Threads	**Static** (*seconds*)	**Dynamic** (*seconds*)
2	703.729	695.051
3	467.874	465.511
4	351.299	345.102
5	378.612	325.376
6	320.762	306.645
7	300.665	289.901
8	277.214	277.058

Speedup

Speedup: definition

To compare the performance of a sequential algorithm with a parallel version, the main measure used is the *speedup*, that is the ration between the execution time of the sequential version and the execution time of the parallel version with the same input.

$$S = \frac{t_S}{t_P} \tag{1}$$

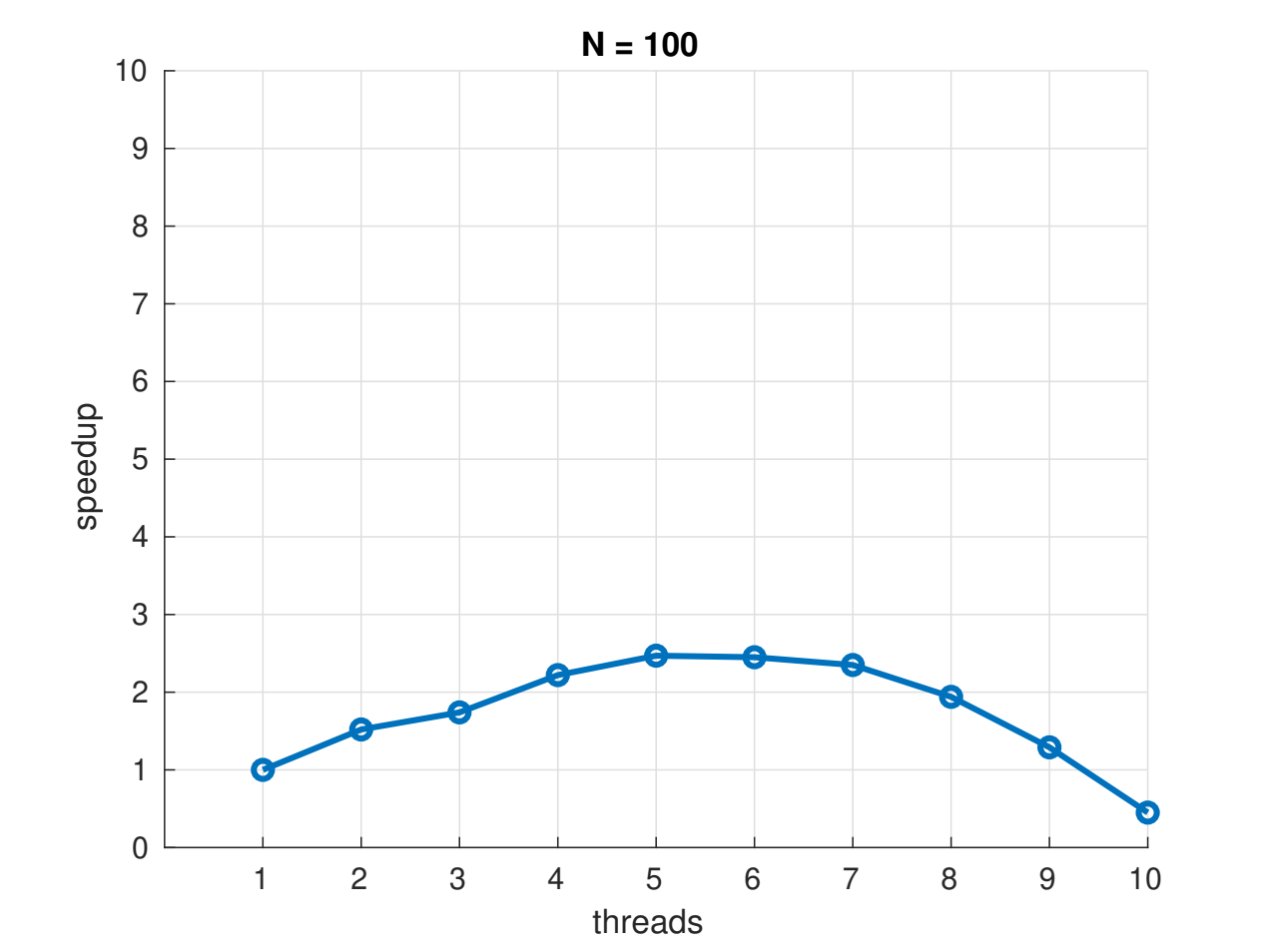
Ideally, the speedup should be the number of processor used to perform the parallel version.

Experiments setup

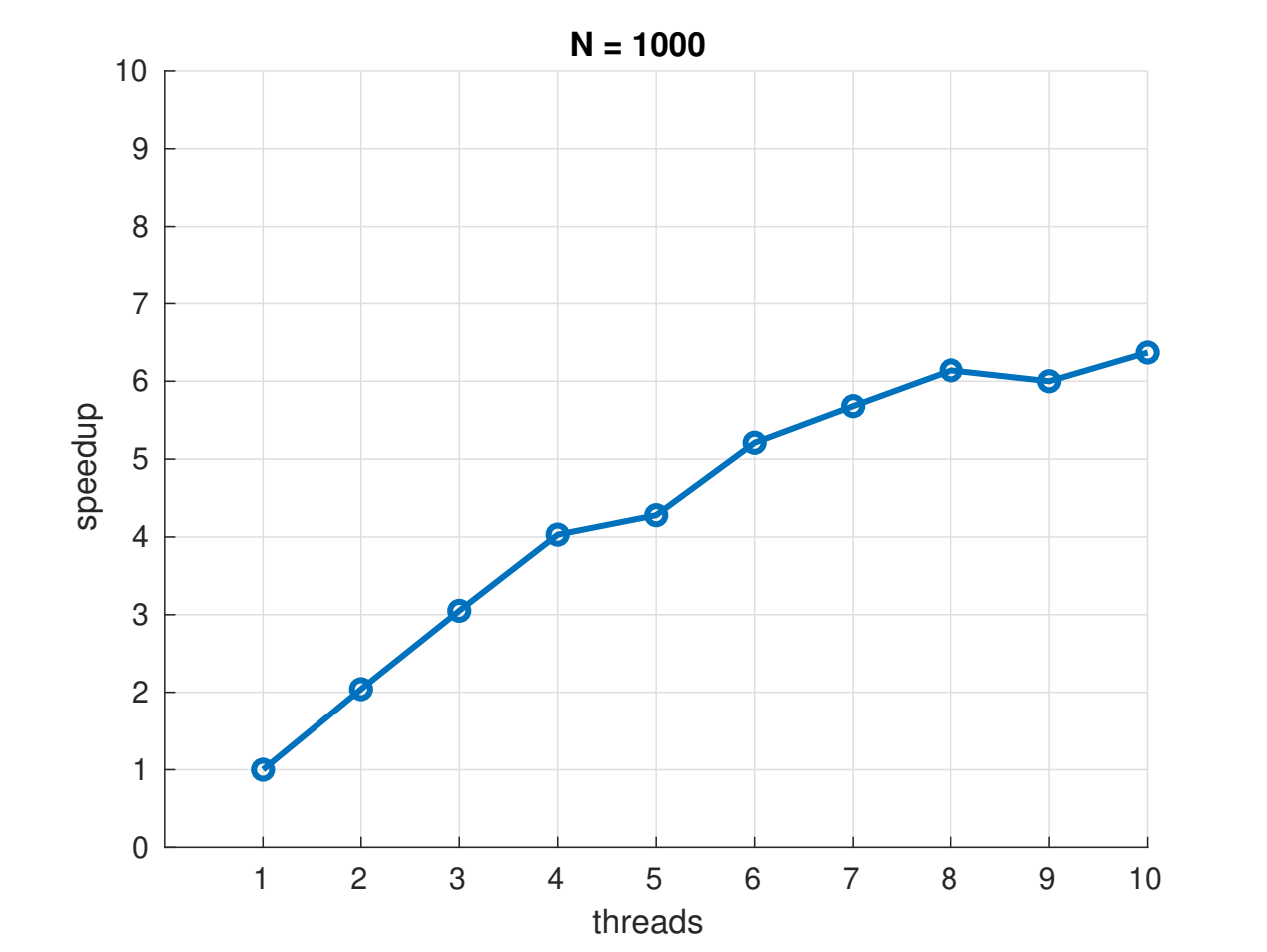
The speedup is computed with a script that first executes a sequential version, then it executes on the same data set differents parallel versions using a different number of threads. In this section are reported some results obtained with datasets of different sizes.

The tests were performed on a machine with an Intel Xeon CPU 2.00 *GHz* with eight cores.

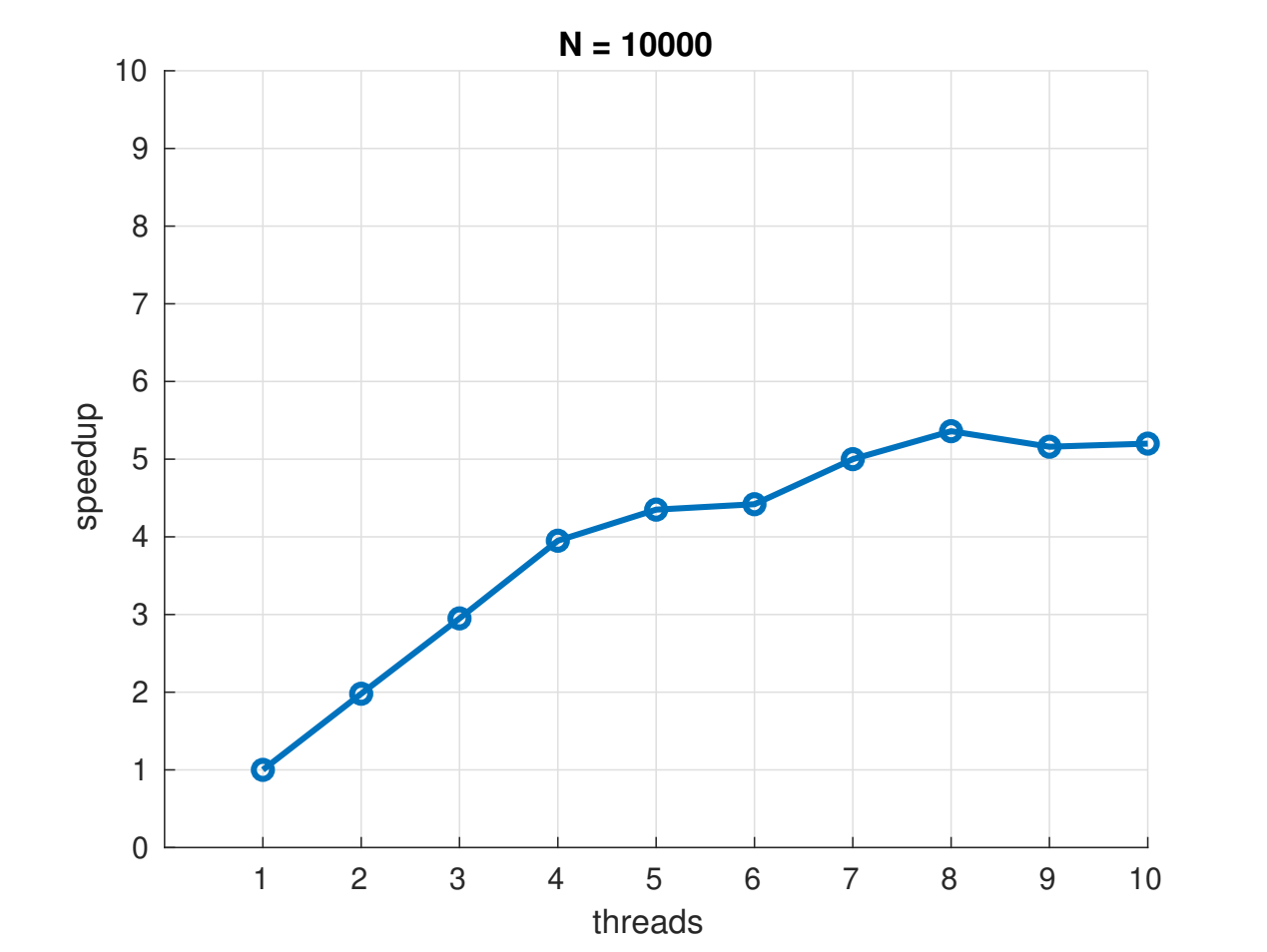
N = 100



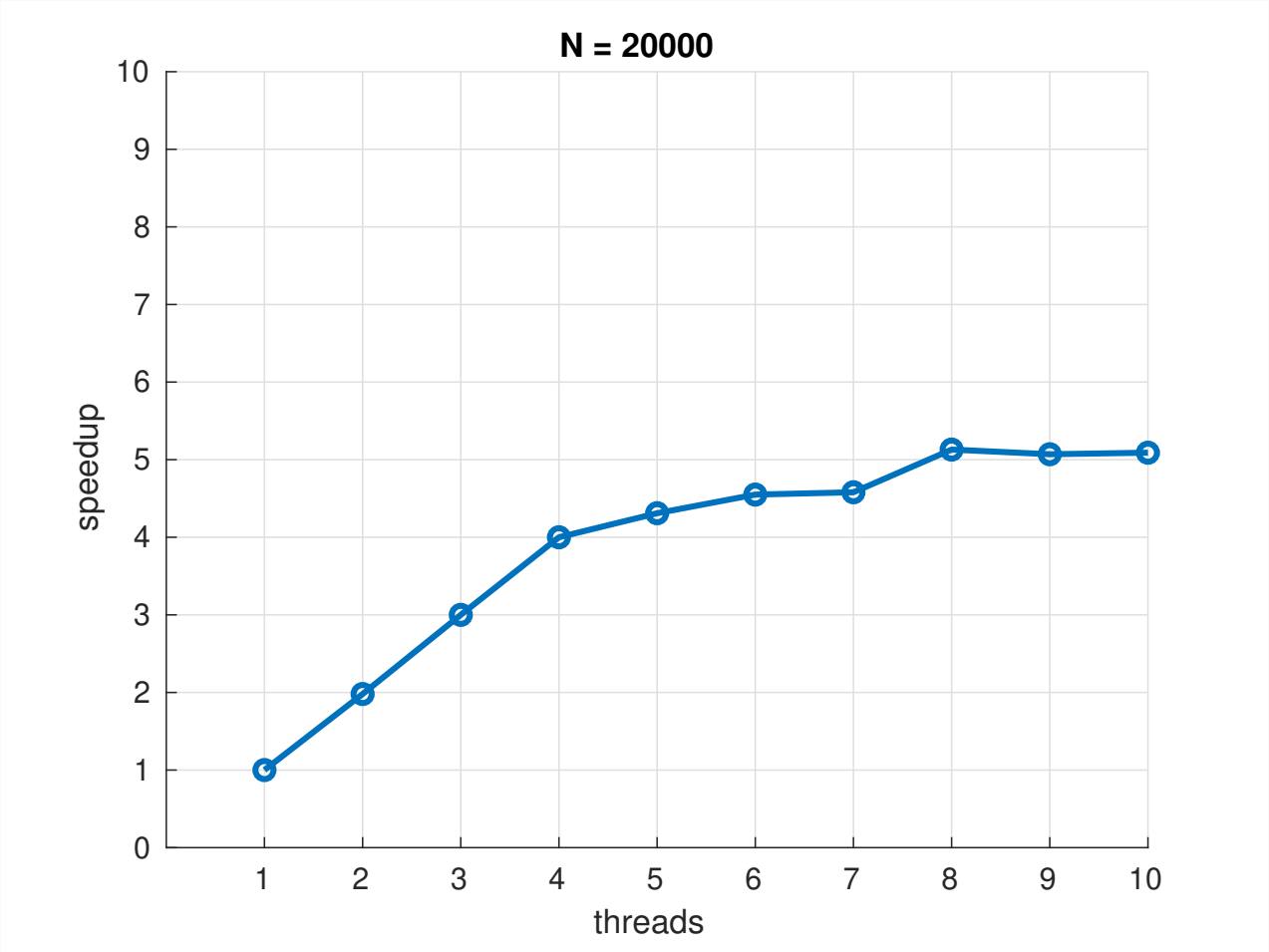
N = 1000



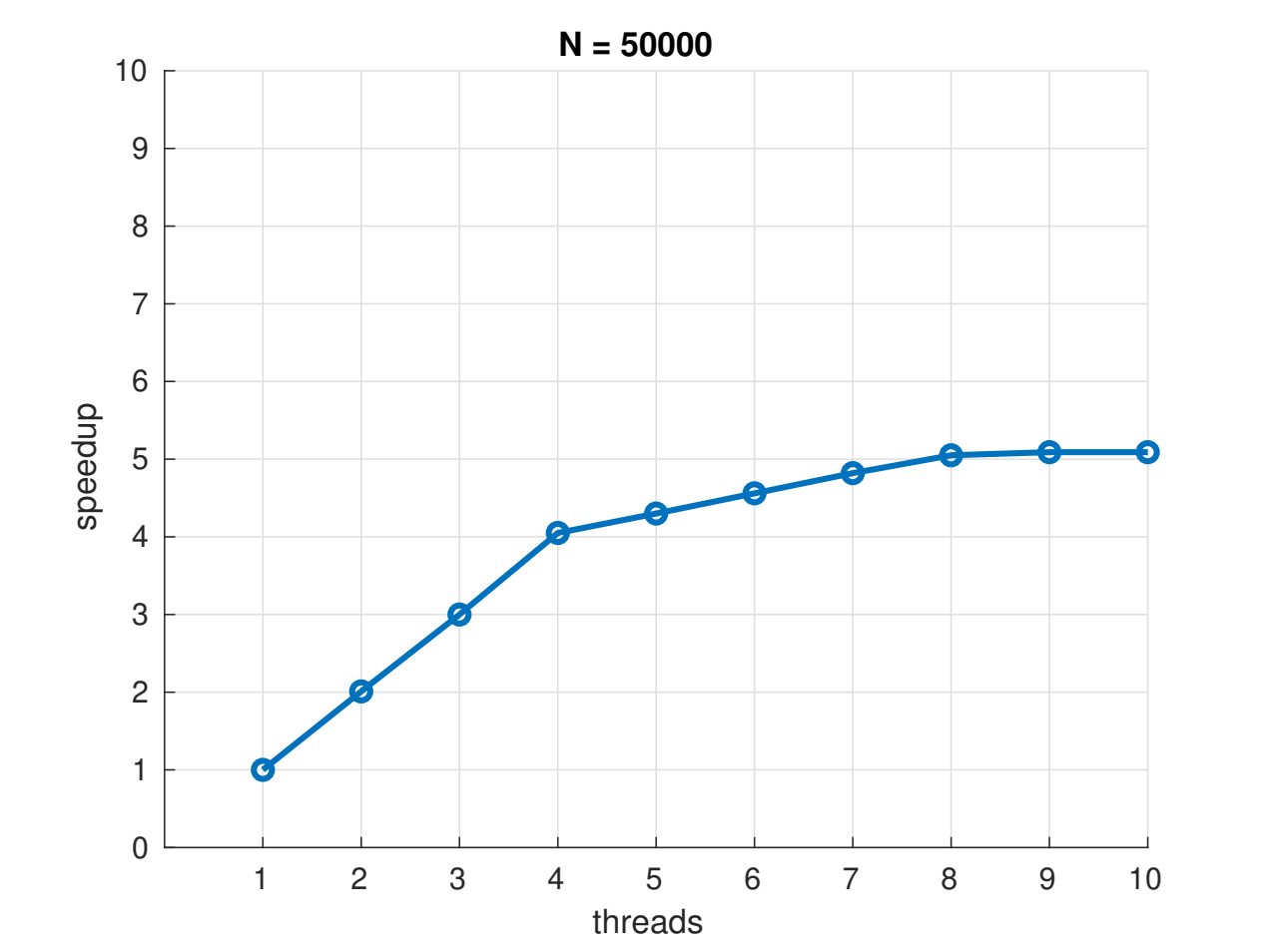
N = 10000



N = 20000

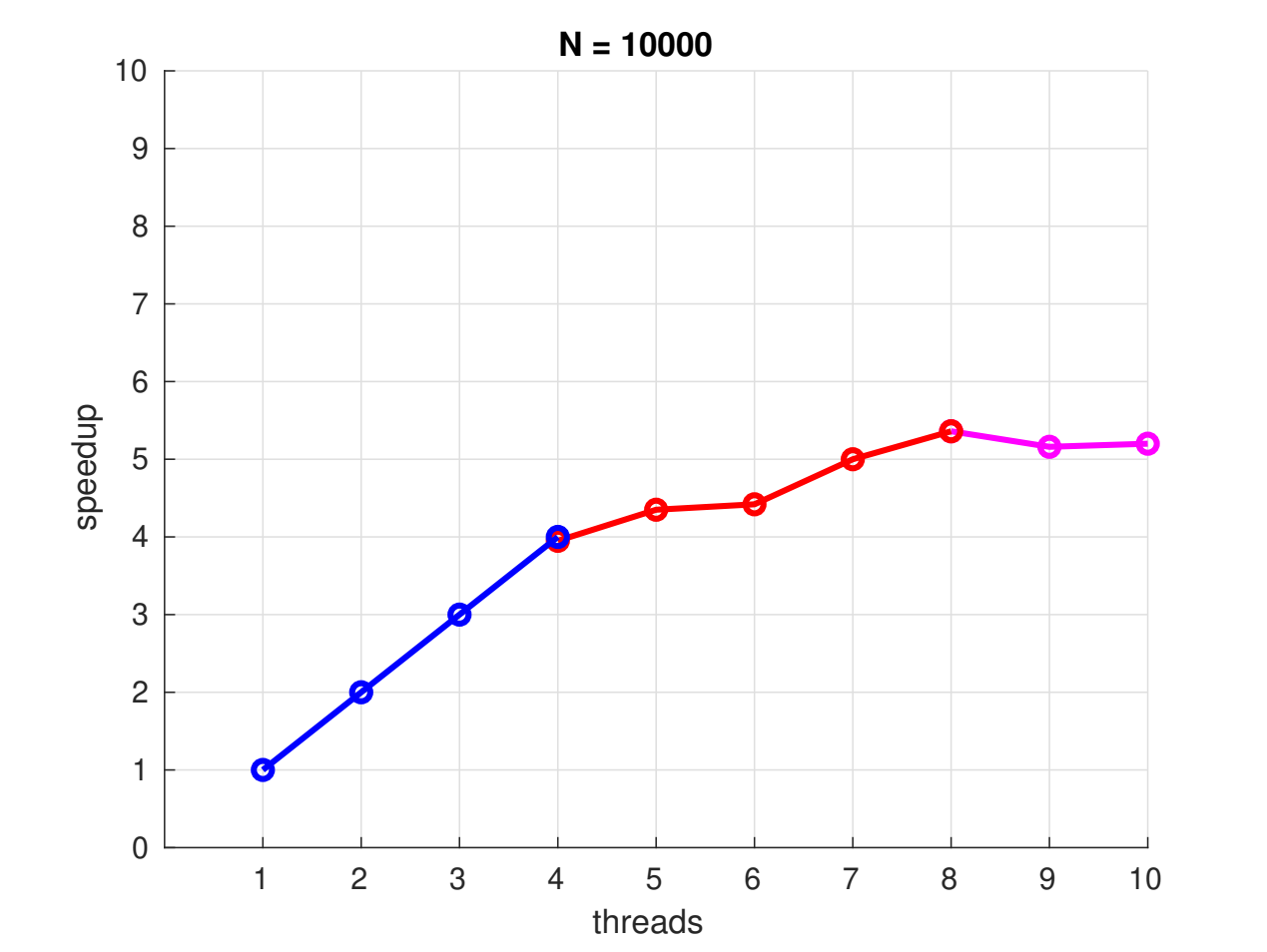


N = 50000

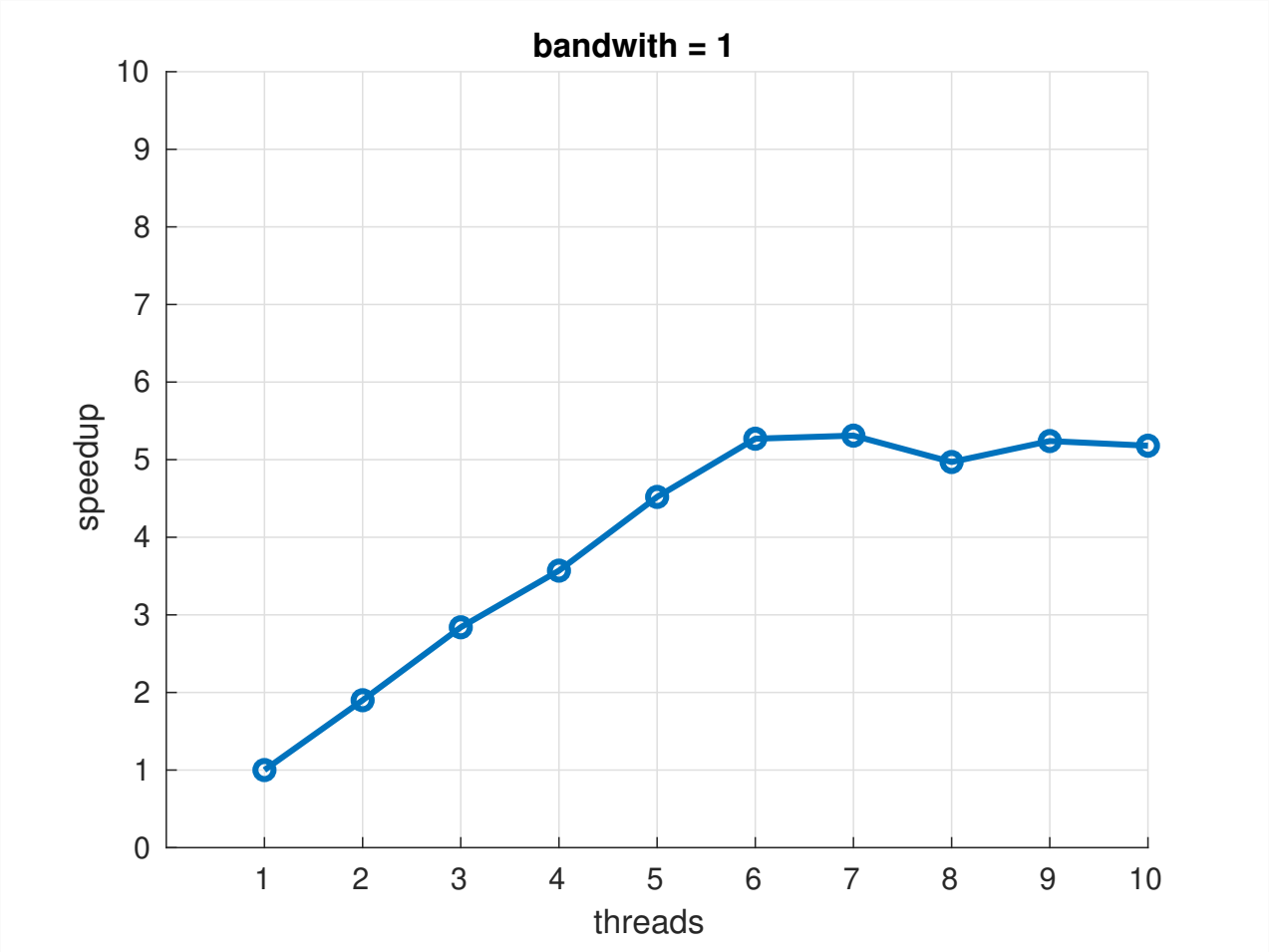


Speedup growth

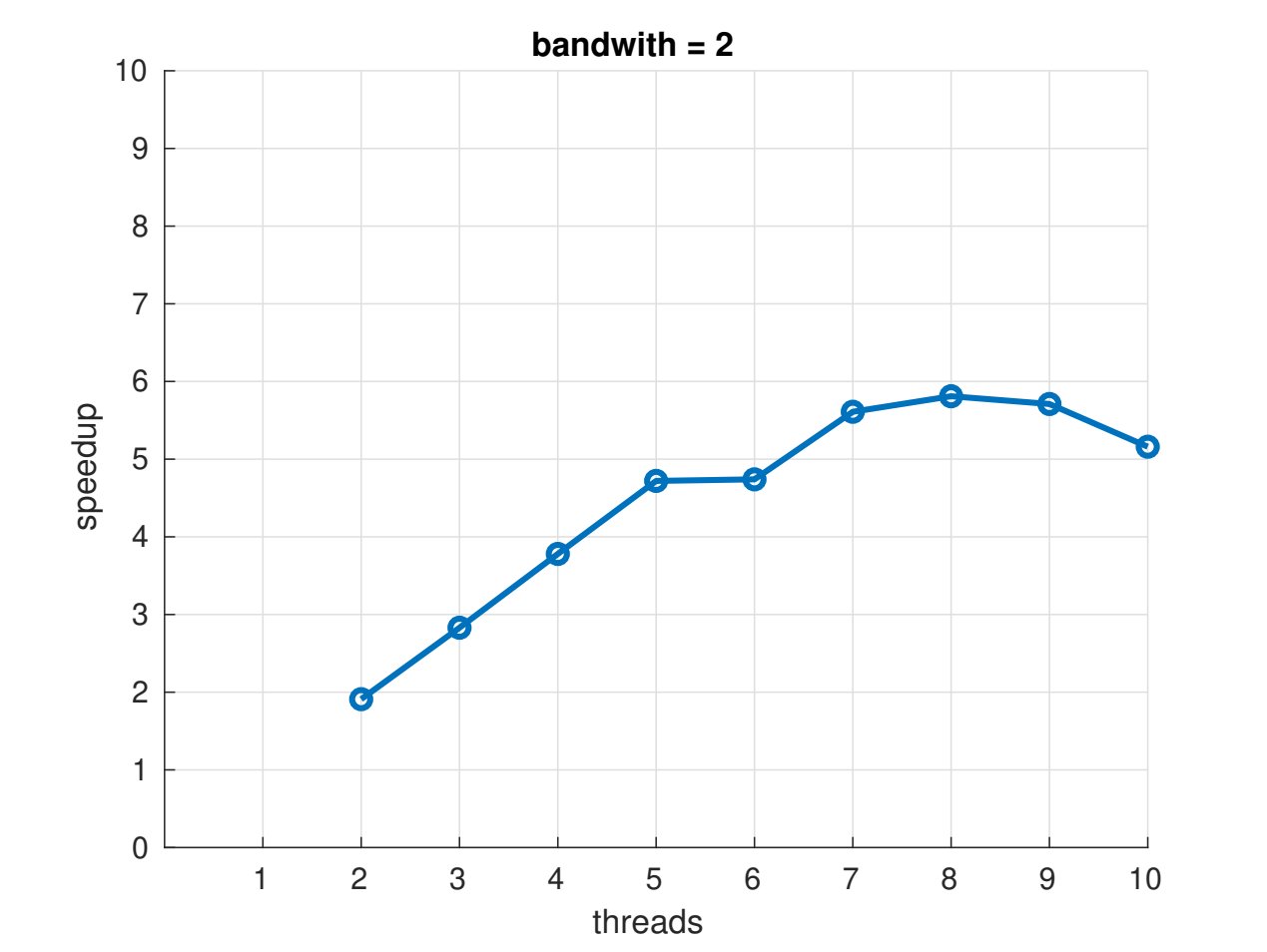
The speedup growths with three different patterns due to the *hyper-threading* technology.



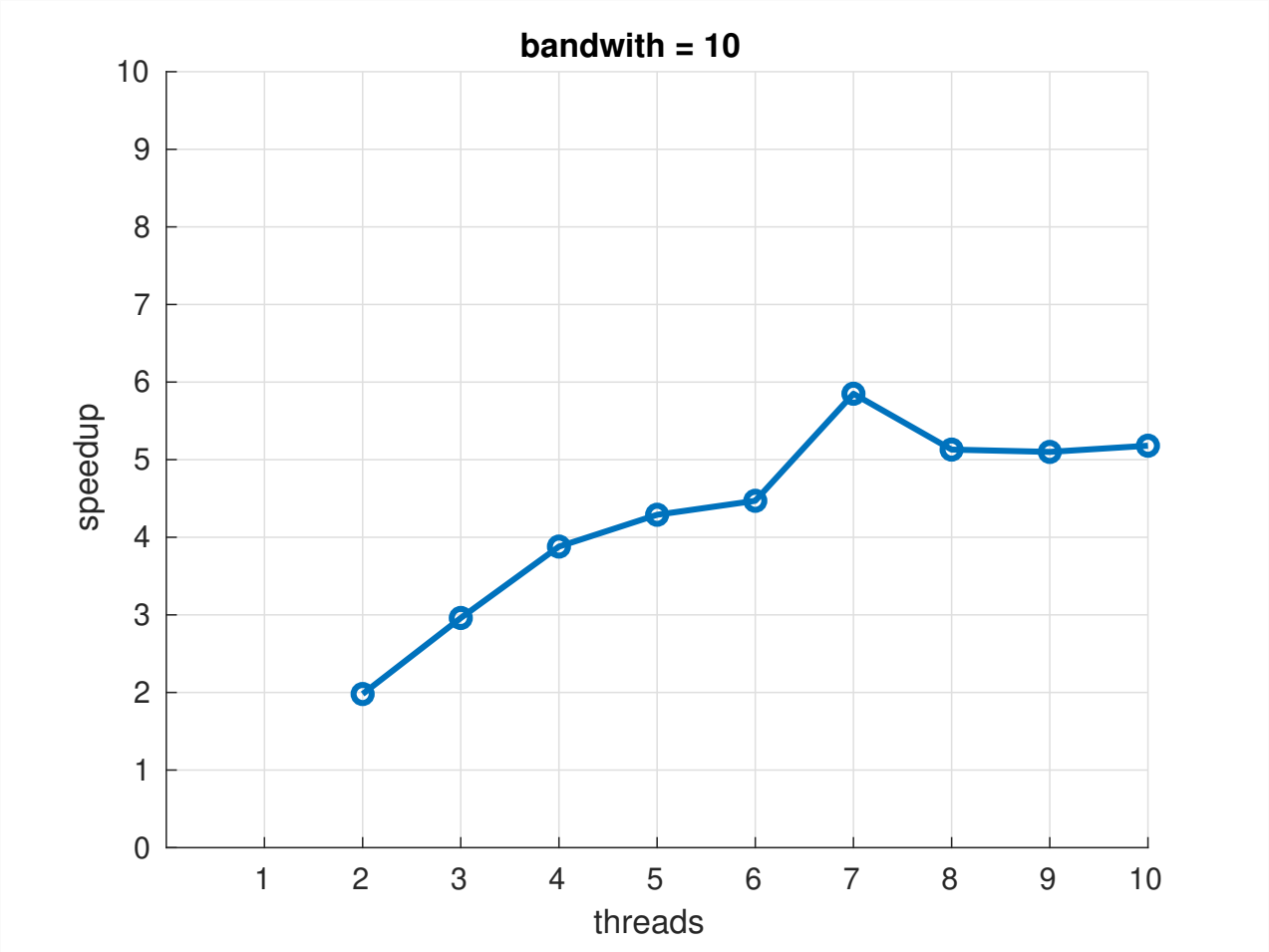
$\sigma = 1$



$\sigma = 2$



$\sigma = 10$



Bandwith influence on the speedup

The growing trends from one to four cores seem to be almost all the same for every value of the bandwith parameter. So we can conclude that if the dataset is big enough, the choice of the bandwith parameter will not influence the speedup.