Parallel Computing Mid Term: *k-means*

Giovanni Bindi - 7016072
giovanni.bindi@stud.unifi.it
https://github.com/w00zie/kmeans

Abstract

In this project I've developed two implementations of the k-means clustering algorithm. These implementations are one sequential version of the naive k-means (also known as Lloyd's algorithm) and its relative parallel version. A simple performance study has then been operated, in order to analyze and quantify the benefits of parallelism. The developed code comes as an header-only C++17 library, with simple test cases to verify the correct behaviour of the implementation on a synthetic data-set.

1. Brief Introduction

The k-means clustering algorithm is a popular and simple technique, developed in its first form during the 1950s[5]. It aims to partition N observations into k clusters. Formally, given a set of observations $\mathcal{S} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, where $\mathbf{x}_i \in \mathbb{R}^D$, the algorithm aims to partition them into k ($\leq N$) sets $\mathcal{C} = \{\mathcal{S}_1, \dots, \mathcal{S}_k\}$, as to minimize the within-cluster sum of squares:

$$\arg\min_{\mathcal{C}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in \mathcal{S}_i} \|\mathbf{x}_i - \boldsymbol{\mu}_i\|^2 \tag{1}$$

where $\boldsymbol{\mu}_i$ is the mean of the points in $\mathcal{S}_i$. This objective is equivalent to

$$\arg\min_{\mathcal{C}} \sum_{i=1}^{k} \frac{1}{2|\mathcal{S}_i|} \sum_{\mathbf{x}_j, \mathbf{y}_j \in \mathcal{S}_i^2} \|\mathbf{x}_j - \mathbf{y}_j\|^2 \tag{2}$$

i.e. minimizing the pairwise squared deviations of points in the same cluster. Despite being an NP-hard problem[1] with a time complexity of $O(N^{DN+1})$[4], there exist a lot of heuristics which are able to cut down the complexity. The simplest heuristic, known as Lloyd's algorithm, has a running time of $O(NkDI)$[3] where I is the number of iterations needed until convergence. From here on the terms *Lloyd's algorithm* and *k-means* will be used interchangeably.

2. Algorithm

After an initial sampling phase, where the first sets of k candidate centroids $\mathcal{S}_1^{(0)}, \dots, \mathcal{S}_k^{(0)}$ are selected from $\mathcal{S}$, the algorithm proceeds iteratively, alternating two steps, for $t = 1, \dots, I$:

1. **Assignment** step: where each data point is assigned to the cluster with the nearest mean, i.e. building $\mathcal{S}_i^{(t)}$:

$$\mathcal{S}_i^{(t)} = \left\{ \mathbf{x}_p : \|\mathbf{x}_p - \boldsymbol{\mu}_i\|^2 \leq \|\mathbf{x}_p - \boldsymbol{\mu}_j\|^2 \right\} \tag{3}$$

   for all $\mathbf{x}_p \in \mathcal{S}$ and for all $j \neq i$.

2. **Update** step: where the centroids are recalculated, based on the previous step calculations:

$$\boldsymbol{\mu}_i^{(t+1)} = \frac{1}{|\mathcal{S}_i^{(t)}|} \sum_{\mathbf{x}_j \in \mathcal{S}_i^{(t)}} \mathbf{x}_j \tag{4}$$

   As stated previously this algorithm runs for I iterations. An extra stopping solution could employ a criterion such as checking for convergence

$$\|\boldsymbol{\mu}_i^{(t+1)} - \boldsymbol{\mu}_i^{(t)}\|^2 \leq \epsilon \tag{5}$$

   for all $i = 1, \dots k$ simultaneously.

This project's implementation can be described by the pseudo-code in Algorithm 1:

Algorithm 1 Sequential *k-means*

Input: $\mathcal{S} \subset \mathbb{R}^D$, $k, I \in \mathbb{N}$. Optional: $\epsilon \in \mathbb{R}$.
Output: Clusters $\mathcal{C}$ with centroids $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$

1: $\boldsymbol{\mu}^{(1)} = \{\boldsymbol{\mu}_1^{(1)}, \dots, \boldsymbol{\mu}_k^{(1)}\} \leftarrow \texttt{sample}(\mathcal{S})$
2: **for** $t = 1, \dots, I$ **do**
3: $\quad \boldsymbol{\tau} = \{\mathbf{0}, \dots, \mathbf{0}\}$
4: $\quad \eta = \{0, \dots, 0\}$
5: $\quad$ **for** $i = 1, \dots, N$ **do**
6: $\quad\quad c = \texttt{closest\_centroid}(\mathbf{x}[i], \boldsymbol{\mu}^{(t)})$
7: $\quad\quad \boldsymbol{\tau}[c] = \boldsymbol{\tau}[c] + \mathbf{x}[i]$
8: $\quad\quad \eta[c] = \eta[c] + 1$
9: $\quad$ **end for**
10: $\quad$ **for** $j = 1, \dots, k$ **do**
11: $\quad\quad \boldsymbol{\mu}^{(t+1)}[j] = \boldsymbol{\tau}[j]/\eta[j]$
12: $\quad$ **end for**
13: $\quad$ **if** $\texttt{dist}(\boldsymbol{\mu}^{(t+1)}, \boldsymbol{\mu}^{(t)}) \leq \epsilon$ **then**
14: $\quad\quad$ **return** $\boldsymbol{\mu}^{(t+1)}$
15: $\quad$ **end if**
16: **end for**
17: **return** $\boldsymbol{\mu}^{(I)}$

The sampling strategy implemented in this project is the simplest one possible, *i.e*. random sampling without replacement. Other and more fine sampling methods are possible, such as the one in *k-means++* for example, which gives a provable upper bound on the objective[2]. The uniform random sampling has been implemented via the Mersenne-Twister pseudo-random generator, available in the C++ standard library.

Variables $\boldsymbol{\tau}, \eta$ are temporary elements used for the calculation of the next centroids. The function `closest_centroid` computes the distances between the point $\mathbf{x}_i$ and the current centroids $\{\boldsymbol{\mu}_1^{(t)}, \dots, \boldsymbol{\mu}_k^{(t)}\}$ and returns the index of the closest one $c \in \{1, \dots, k\}$. This index is then used to access the accumulators $\boldsymbol{\tau}$ and η, respectively used to add up the elements of the same cluster and count them.

Once gone through every element of the data set, each future centroid is computed by dividing each temporary sum with the total count of points in that cluster, as written in line 11.

Optionally, if the user decides to employ a convergence-based stopping criterion, the shift from the previous centroids to the current ones is computed for each candidate cluster center. If these distances are all lower or equal to some user-specified $\epsilon \in \mathbb{R}$, then the convergence is reached and the current centroids are returned.

Before describing the parallel version of the code, it can be useful to notice how it is possible to parallelize the *assignment step*, which is the one that goes from line 5 to 9 of Algorithm 1. The loop could be entirely avoided if, in theory, one could employ N workers that execute the same job in parallel, one for each element of the dataset.

Since this implementation is CPU only, threads have been used in order to concurrently execute this part of the algorithm, splitting the work and, possibly, speeding up the computations.

3. Parallelization

A thread pool with a task queue has been used to manage the concurrent execution of the *assignment step*. The parallelization scheme is an hybrid between the producer-consumer model and the fork-join model: the user populates a thread pool of T threads, which executes tasks from the queue. The work balance between the threads has been equally divided, statically assigning to each thread the computation on a subset of (at most) $\lceil N/T \rceil$ elements. With this simple strategy, ideally, every spawned thread should execute one single task per iteration. It is also possible that a thread runs multiple tasks in one iteration, if the amount of computation needed does not require the action of the other threads.

This idea is illustrated in the pseudo-code of Algorithm 2: after the initialization of the thread pool (and its internal queue), the algorithm proceeds as the previous one. For each available thread a subset of the dataset is selected (through the calculation of the slicing indices s and e) and the computations are executed by a thread on this portion of the data.

Once every thread has terminated its computations (or, equivalently, the task queue is empty), it is possible to proceed to the update step, computing the divisions and obtaining the current cen-

troids.

Since every thread writes to the shared variables $\boldsymbol{\tau}$ and η, a locking mechanism has been employed to assure the correct behaviour and avoid data-races.

In Algorithm 2 blue text represents code that is executed in parallel.

Algorithm 2 Multi-threaded *k-means*

Input: $\mathcal{S} \subset \mathbb{R}^D$, $k, I \in \mathbb{N}$. $T \in \mathbb{N}$. Optional: $\epsilon \in \mathbb{R}$.
Output: Clusters $\mathcal{C}$ with centroids $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$

$\boldsymbol{\mu}^{(1)} = \{\mu_1^{(1)}, \dots, \boldsymbol{\mu}_k^{(1)}\} \leftarrow$ `sample`$(\mathcal{S})$
$\mathcal{P} \leftarrow$ `init_pool`(T)
for $t = 1, \dots, I$ **do**
  $\boldsymbol{\tau} = \{\mathbf{0}, \dots, \mathbf{0}\}$
  $\eta = \{0, \dots, 0\}$
  for each thread $p \in \mathcal{P}$ **do**
    $(s, e) \leftarrow$ `compute_start_end`(p, N)
    for $i = s, \dots, e$ **do**
      $c =$ `closest_centroid`$(\mathbf{x}[i], \boldsymbol{\mu}^{(t)})$
      $\boldsymbol{\tau}[c] = \boldsymbol{\tau}[c] + \mathbf{x}[i]$
      $\eta[c] = \eta[c] + 1$
    end for
  end for
  `wait_all_threads()`
  for $j = 1, \dots, k$ **do**
    $\boldsymbol{\mu}^{(t+1)}[j] = \boldsymbol{\tau}[j]/\eta[j]$
  end for
  if `dist`$(\boldsymbol{\mu}^{(t+1)}, \boldsymbol{\mu}^{(t)}) \leq \epsilon$ **then**
    return $\boldsymbol{\mu}^{(t+1)}$
  end if
end for
return $\boldsymbol{\mu}^{(I)}$

3.1. Thread Pool

The thread pool is mainly composed of a *vector of threads* and a *double-ended queue* of tasks. These tasks are forced to be `void` functions. It uses a *mutex* for locking/unlocking critical sections and two condition variables for synchronization. It also uses a *counter* for currently-running tasks and a *boolean* variable indicating whether it is time to stop the execution and join the threads.

This class has three main methods: `enqueue`, `run_task` and `wait_all_threads`. The `enqueue` methods enqueues a function `f` into the deque and notifies the task condition variable that a new task is scheduled. Method `run_task` loops endlessly waiting for the task condition variable to notify that at least one task is scheduled. If this condition is verified it pops the first task from the queue and executes it. Finally it signals the "global" condition variable that a task has been executed. Lastly, method `wait_all_threads` waits for the "global" condition variable to notify that no other tasks are scheduled nor running. Code can be visioned in Appendix E.

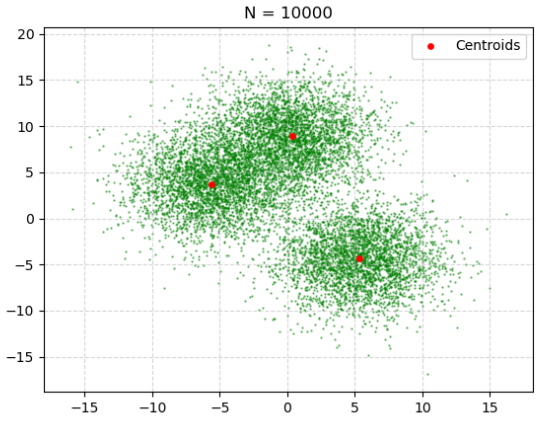

Figure 1. Example of a dataset: $N = 10000, D = 2$.

4. Performance Analysis

In order to understand if this parallel implementation could improve the performances with respect to the sequential version, a simple speedup analysis has been carried out. This analysis has been performed on synthetically-generated datasets of variable size and dimensionality. In order to avoid the curse of dimensionality, only datasets in $\mathbb{R}^2$ and $\mathbb{R}^3$ have been generated. Specifically, for each $D \in \{2, 3\}$, many datasets have been built around 3 centroids, each of different size N. This analysis involves the ones spanning from $N = 10000$ to $N = 500000$. The generation has been done via Scikit-Learn's API. One example can be seen in Figure 1.

For each one of the datasets involved, both the sequential and the multi-threaded executions have been timed 10 times, in order to obtain a better estimate. For each dataset of size N and dimension-

ality D, the average running times for the sequential and multi-thread versions, $\bar{t}_s^{(N,D)}$ and $\bar{t}_p^{(N,D)}$ are obtained as:

$$\begin{cases} \bar{t}_s^{(N,D)} = \frac{1}{10}\sum_{i=1}^{10} t_s^{(i,N,D)} \\ \bar{t}_p^{(N,D)} = \frac{1}{10}\sum_{i=1}^{10} t_p^{(i,N,D)} \end{cases}, \tag{6}$$

where $t_*^{(i,N,D)}$ represents a single run. The speedup is then obtained as:

$$S^{(N,D)} = \frac{\bar{t}_s^{(N,D)}}{\bar{t}_p^{(N,D)}} \tag{7}$$

This calculation has been performed for various cases, varying the number of threads: $T = \{1, 2, 3, 4, 5, 6, 7, 8\}$. Results are plotted in Figure 2. The plotted error bars are the standard deviations obtained from the variances propagation of Equation 7 (see Appendix A).

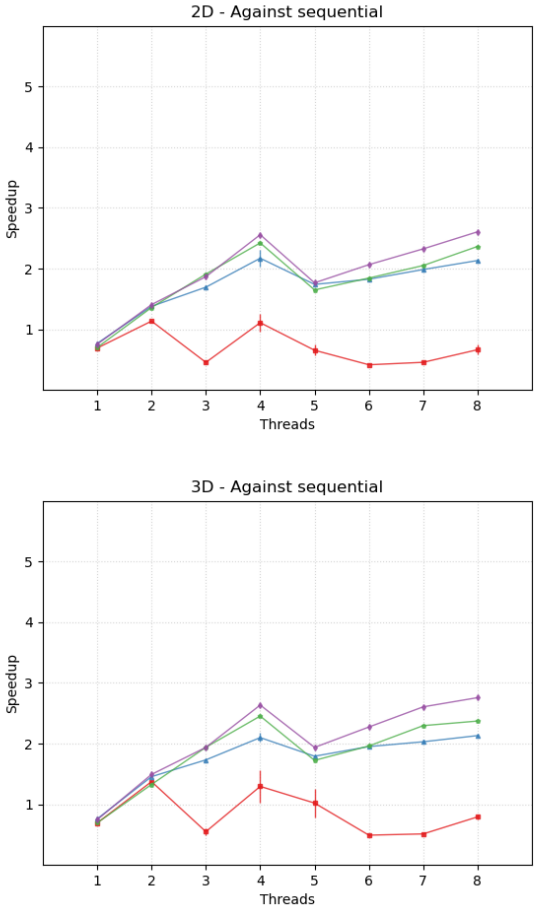

Figure 2. Speedup against sequential version for $D = \{2, 3\}$.

This same calculation has been performed comparing the case of $T = 1$ against the other 7 cases $T = \{2, \ldots, 8\}$. Results for this case are plotted in Figure 3. For both cases the tables containing the results data can be seen in Appendices C and D.

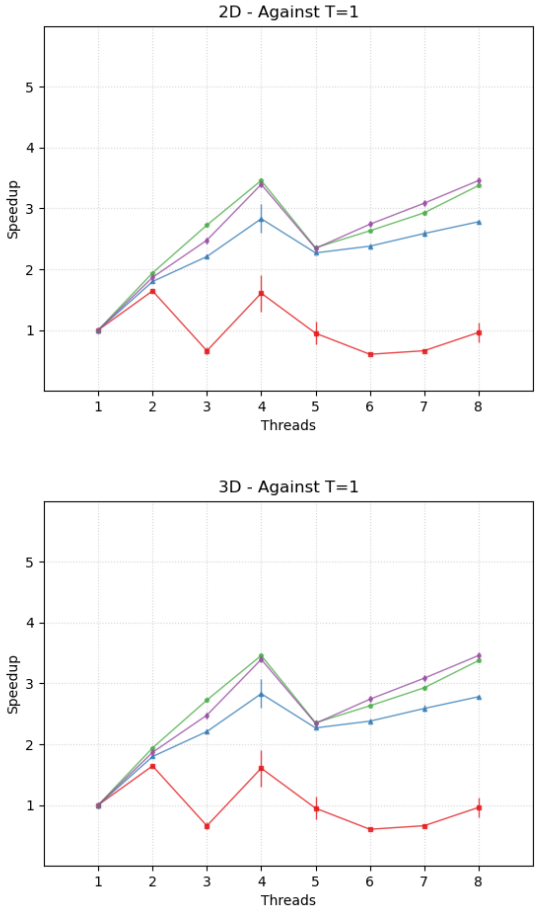

Figure 3. Speedup against $T = 1$ for $D = \{2, 3\}$.

The CPU used for this analysis is a 4 cores / 8 threads Intel i7-8550U, running at 1.8GHz. The operating system was Debian/Linux 5.9. All the timings were obtained while maintaining the lowest possible load average on the machine.

For bench-marking purposes no ϵ-convergence strategy has been employed: every run is composed of $I = 1000$ iterations.

5. Conclusions

It's clear from Figures 2 and 3 that the performance gains are far from the theoretical optima. Results are slightly better for the comparison against $T = 1$, obtaining a maximum speedup of 3.69 for $D = 3, N = 500000$ and $T = 8$. The best speedup obtained comparing against the sequential version was 2.76 for $D = 3, N = 500000$ and $T = 8$.

It is worth noticing that the worst speedup behaviour was obtained for $N = 10000$, for every D and comparison case. This may be partially explained by the fact that this dataset size is too small: thread managing is expensive, and paying this overhead may not be worth when dealing with "small" datasets.

This statement may be confirmed by the fact that, almost in every case, for both $N = 100000$ and $N = 500000$ the benefice of parallelism is more prominent, with respect to other sizes.

An interesting phenomenon is the performance drop at $T = 5$. This worsening is evident for $N = \{50000, 100000, 500000\}$ and a little less sharp for $N = 10000$. For this latter case, instead, a huge performance drop is manifested at $T = 3$.

Generally speaking the best speedups were obtained for both $T = 4$ and $T = 8$, giving comparable results. The used processor has 4 cores and 8 threads and it seems that, when the computations requirements are high enough (*e.g.* $N >= 10^5$), this simple parallelization mechanism is able to give a sensible (yet not huge) speedup.

References

[1] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. Np-hardness of euclidean sum-of-squares clustering. *Mach. Learn.*, 75(2):245–248, May 2009.

[2] D. Arthur and S. Vassilvitskii. K-means++: The advantages of careful seeding. volume 8, pages 1027–1035, 01 2007.

[3] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[4] M. Inaba, N. Katoh, and H. Imai. Applications of weighted voronoi diagrams and randomization to variance-based k-clustering: (extended abstract). In *Proceedings of the Tenth Annual Symposium on Computational Geometry*, SCG '94, page 332–339, New York, NY, USA, 1994. Association for Computing Machinery.

[5] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

A. Variance calculation

Dropping the indices (D, N), let $\bar{t}_s$ and $\bar{t}_p$ the average execution times for the sequential and parallel version, respectively. Let σ_s^2 and σ_p^2 be their sample variances (over 10 runs). The speedup is computed, as written in Equation 7, as

$$S = \frac{\bar{t}_s}{\bar{t}_p}$$

Variance σ^2 is then estimated via

$$\sigma^2 \approx S^2 \left[\frac{\sigma_s^2}{\bar{t}_s^2} + \frac{\sigma_p^2}{\bar{t}_p^2} \right]$$

Tables in Appendices C and D show the average speedup S and relative standard deviation σ for both comparison cases, for each D, N and T.

B. A note on implementation

The code, which is available on Git-Hub[1], is entirely based upon the C++17 standard library with an exception: the choice for the data structure employed for centroids has been the `unordered_map`. Since the standard library's implementation is not performance-oriented, a custom implementation has been used: `Tessil's robin map`[2]. It is still possible to switch back to the STL implementation, by applying a tiny modification in the `include/container.h` file. This action, although, will have a negative effect on performance.

B.1. Sanitizers

Running Cppcheck 2.3[3] should display a syntax error coming from Tessil's robin map (on a here-unused feature). Running Valgrind 3.16.1[4] should not display errors nor warnings.

[1] https://github.com/w00zie/kmeans
[2] https://github.com/Tessil/robin-map
[3] http://cppcheck.sourceforge.net/
[4] https://valgrind.org/

C. Against sequential version

C.1. $D = 2$

D	N	T	S	σ
2	10000	1	0.69	0.04
		2	**1.13**	0.08
		3	0.45	0.17
		4	1.11	0.38
		5	0.65	0.30
		6	0.42	0.03
		7	0.46	0.11
		8	0.66	0.28

D	N	T	S	σ
2	50000	1	0.77	0.04
		2	1.38	0.07
		3	1.69	0.16
		4	**2.17**	0.38
		5	1.74	0.11
		6	1.82	0.17
		7	1.98	0.20
		8	2.13	0.14

D	N	T	S	σ
2	100000	1	0.7	0.04
		2	1.36	0.07
		3	1.91	0.11
		4	**2.42**	0.14
		5	1.65	0.08
		6	1.84	0.10
		7	2.05	0.12
		8	2.36	0.13

D	N	T	S	σ
2	500000	1	0.75	0.03
		2	1.41	0.05
		3	1.87	0.10
		4	**2.56**	0.09
		5	1.77	0.06
		6	2.06	0.07
		7	2.32	0.07
		8	2.61	0.07

C.2. $D = 3$

D	N	T	S	σ
3	10000	1	0.69	0.02
		2	**1.37**	0.09
		3	0.55	0.23
		4	1.29	0.52
		5	1.02	0.48
		6	0.49	0.13
		7	0.51	0.13
		8	0.79	0.19

D	N	T	S	σ
3	50000	1	0.76	0.01
		2	1.46	0.03
		3	1.73	0.10
		4	2.10	0.27
		5	1.79	0.12
		6	1.95	0.17
		7	2.03	0.17
		8	**2.13**	0.14

D	N	T	S	σ
3	100000	1	0.69	0.01
		2	1.33	0.02
		3	1.94	0.06
		4	**2.45**	0.03
		5	1.72	0.04
		6	1.96	0.04
		7	2.30	0.05
		8	2.37	0.06

D	N	T	S	σ
3	500000	1	0.75	0.03
		2	1.49	0.03
		3	1.94	0.07
		4	2.63	0.05
		5	1.94	0.04
		6	2.27	0.04
		7	2.61	0.04
		8	**2.76**	0.05

D. Against $T = 1$

D.1. $D = 2$

D	N	T	S	σ
2	10000	1	1.0	0.04
		2	**1.64**	0.11
		3	0.66	0.25
		4	1.61	0.54
		5	0.95	0.43
		6	0.60	0.04
		7	0.66	0.16
		8	0.96	0.41

D	N	T	S	σ
2	50000	1	1.0	0.04
		2	1.80	0.06
		3	2.21	0.19
		4	**2.83**	0.49
		5	2.27	0.11
		6	2.38	0.20
		7	2.59	0.24
		8	2.78	0.15

D	N	T	S	σ
2	100000	1	1.0	0.06
		2	1.94	0.09
		3	2.73	0.15
		4	**3.46**	0.20
		5	2.35	0.11
		6	2.63	0.14
		7	2.93	0.17
		8	3.38	0.19

D	N	T	S	σ
2	500000	1	1.0	0.04
		2	1.87	0.07
		3	2.48	0.13
		4	3.39	0.12
		5	2.34	0.07
		6	2.74	0.09
		7	3.09	0.10
		8	**3.46**	0.10

D.2. $D = 3$

D	N	T	S	σ
3	10000	1	1.0	0.01
		2	**1.98**	0.11
		3	0.79	0.32
		4	1.87	0.75
		5	1.47	0.70
		6	0.71	0.19
		7	0.74	0.18
		8	1.14	0.28

D	N	T	S	σ
3	50000	1	1.0	0.02
		2	1.92	0.03
		3	2.28	0.14
		4	2.76	0.35
		5	2.36	0.16
		6	2.57	0.22
		7	2.67	0.22
		8	**2.81**	0.18

D	N	T	S	σ
3	100000	1	1.0	0.02
		2	1.91	0.04
		3	2.80	0.09
		4	**3.54**	0.06
		5	2.48	0.07
		6	2.83	0.07
		7	3.31	0.08
		8	3.42	0.10

D	N	T	S	σ
3	500000	1	1.0	0.06
		2	2.00	0.09
		3	2.60	0.14
		4	3.53	0.16
		5	2.60	0.11
		6	3.05	0.13
		7	3.49	0.15
		8	**3.69**	0.16

E. Thread Pool Code

```cpp
#include <condition_variable>
#include <deque>
#include <functional>
#include <mutex>
#include <thread>
#include <vector>

class thread_pool {

        public:

            thread_pool(const unsigned int T = std::thread::hardware_concurrency()) :
                running_tasks(),
                stop() {
                    for (unsigned int i = 0; i < T; ++i)
                        workers.emplace_back(std::bind(&thread_pool::run_task, this));
                }

            ~thread_pool() {
                std::unique_lock<std::mutex> lock(mtx);
                stop = true;
                task_done.notify_all();
                lock.unlock();
                for (auto& w : workers)
                    w.join();
            }

            template<class F>
            void enqueue(F&& f) {
                std::unique_lock<std::mutex> lock(mtx);
                tasks.emplace_back(std::forward<F>(f));
                task_done.notify_one();
                return;
            }

            void wait_all_threads() {
                std::unique_lock<std::mutex> lock(mtx);
                all_done.wait(lock, [this](){ return tasks.empty() && (running_tasks == 0); });
                return;
            }

        private:

            std::vector<std::thread> workers;
            std::deque<std::function<void()>> tasks;
            std::mutex mtx;
            std::condition_variable task_done;
            std::condition_variable all_done;

            unsigned int running_tasks;
            bool stop;

            void run_task() {
                while (true) {
                    std::unique_lock<std::mutex> lock(mtx);
                    task_done.wait(lock, [this](){ return stop || !tasks.empty(); });
                    if (!tasks.empty()) {
                        ++running_tasks;
                        auto func = tasks.front();
                        tasks.pop_front();
                        lock.unlock();
                        func();
                        lock.lock();
                        --running_tasks;
                        all_done.notify_one();
                    }
                    else if (stop)
                        break;
                }
                return;
            }
    };
```