University of Pisa

Parallel and Distributed Systems: Paradigms and Models

Project: Parallelization of the Swarm Particle Optimization

Instructor: Prof. Marco Danelutto, Prof. Massimo Torquati

Student: Aytan Yagublu

Introduction

Swarm Particle Optimization is a computational method that optimizes a problem
by iteratively where particle's movement is influenced by its local best-known position localMin, but is also guided toward the best-known positions in the search-space globalMin, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions.
So, in each iteration, particles randomly evaluate their velocity as follow:

$$V_{t+1} = a*V_t + b*R_1*(Pos_t - \text{localMin}) + c*R_2*(Pos_t – \text{globalMin})$$

Where R_1 and R_2 are randomly distributed numbers in the interval (0…1), Pos_i and V_i are the positions and the velocity of the i^{th} particle respectively.

And at each iteration the new position is re-computed as:

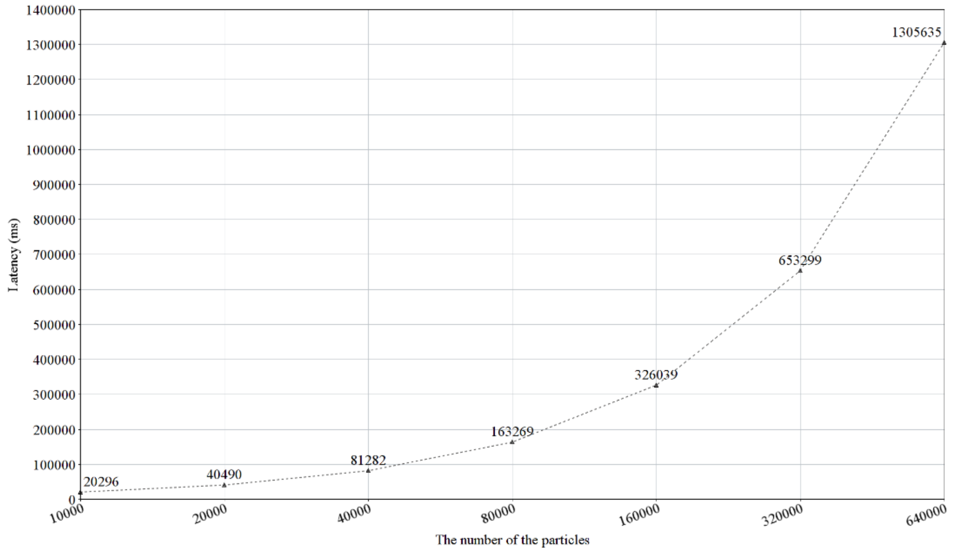
$$Pos_{t+1} = Pos_t + V_{t+1}$$

Sequential Implementation

```
1.  Initialize $Pos_i$ and $V_i$ for $i = 1…n$
2.  Compute localMin and GlobalMin
3.  for( iter=0;  iter<NIteration; iter++) do
4.          Define $V_{i+1} = a*V_i + b*R_1*(Pos_i - localMin) + c*R_2*(Post – globalMin)$  for $i=1…n$
5.          Define $Pos_{i+1} = Pos_i + V_{i+1}$
6.          Compute $function(Pos_i)$
7.          Compute localMin and GlobalMin
8.  end for
```

The cost of the sequential algorithm is O(n +NIteration*n)

While running this algorithm sequentially, it takes quite a long time for the big number of particles.



Graph 0. The graph illustrates the latency of the sequential algorithm with different number of particles

Therefore, it is a good idea to parallelize this algorithm, so I did this parallelization with 2 different tools as C++ thread and FastFlow.

Parallel Implementation

The algorithm has 2 main parts:

1. **Initialization part**
   First of all, we should assign the random positions and give initial velocity for each particle. I divided the number of particles between the number of threads equally So, that we can parallelly do our computations.

   In order to find the local and global minimum of the positions of the particles, we should wait for each thread to finish its work. There are 2 different possible approaches.
   - As one solution, we can use a barrier that will force the thread to wait until all other threads will finish their job. After that, we can update the local and global minimum, which is actually, a critical section, so we have to use the lock in order to synchronize our threads. However, this could cause a delay that could be solved by using the second approach.
   - As a second solution, we can take an array with the size of the number of threads. By doing this we can generate local minimum directly after the thread finishes updating the positions and assign this value to the array with the position of the thread. When all threads finish their work, we can compute the global minimum of the threads and update our localMin and globalMin.

2. **Iteration part**
   The same logic is followed by the iteration part as well. But this time we use our formulas for updating the velocity and position of the particles.

Random Number Generator

In our computation, we use a large number of random numbers. So, generating an independent sequence of random numbers is one of the most important and challenging parts of this algorithm. The best solution is to use the Mersenne Twister 199737 algorithm to generate these. This algorithm takes very little time to generate a 32-bit deviate and the period of the generator is $2^{19,937}$. Also, by assigning different seeds to each thread guarantees the unique sequence of random numbers per thread.

Performance of the algorithm

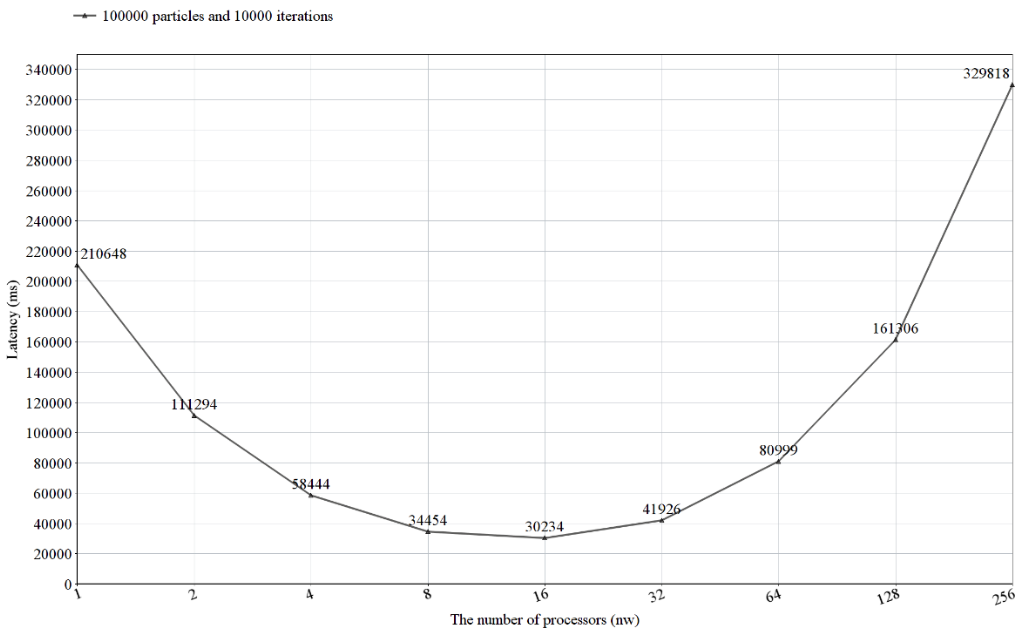
The cost of the parallel algorithm for initialization part is O(n/nw+ nw) and for iteration part is O(Niteration*(n/nw+nw))

Parallel Simulation Performance

1. Parallel application written using C++ threads

a. Latency

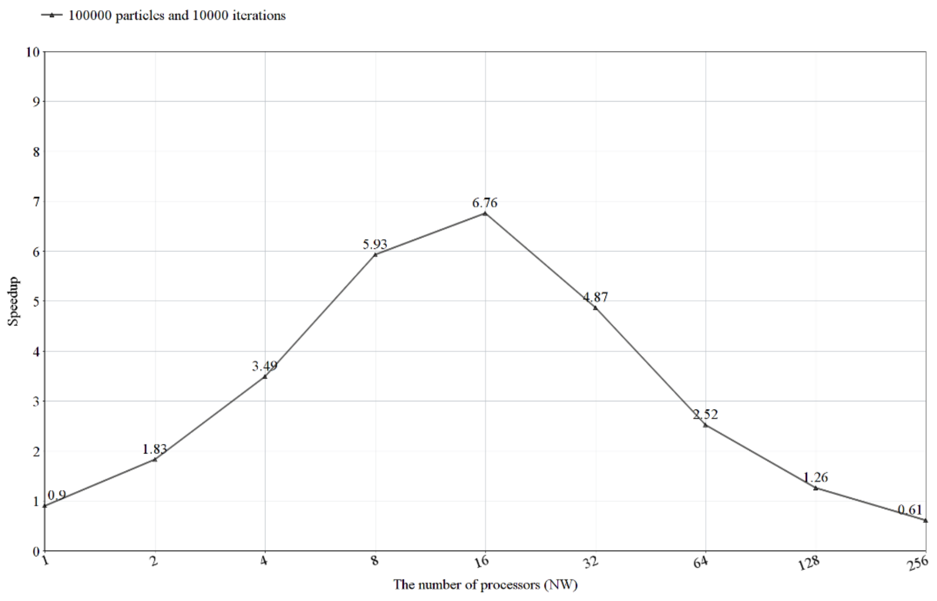
As you may observe from the *Graph 1,* the latency is consistently decreasing while running the program at most 16 threads, however, after 16 threads is starting increasing, furthermore, when we use the whole sources the latency of the parallel program even gets greater than the sequential program's latency.



Graph 1. The graph illustrates the changes in the latency by increasing the number of threads while the size of the program stays even.

b. Speedup

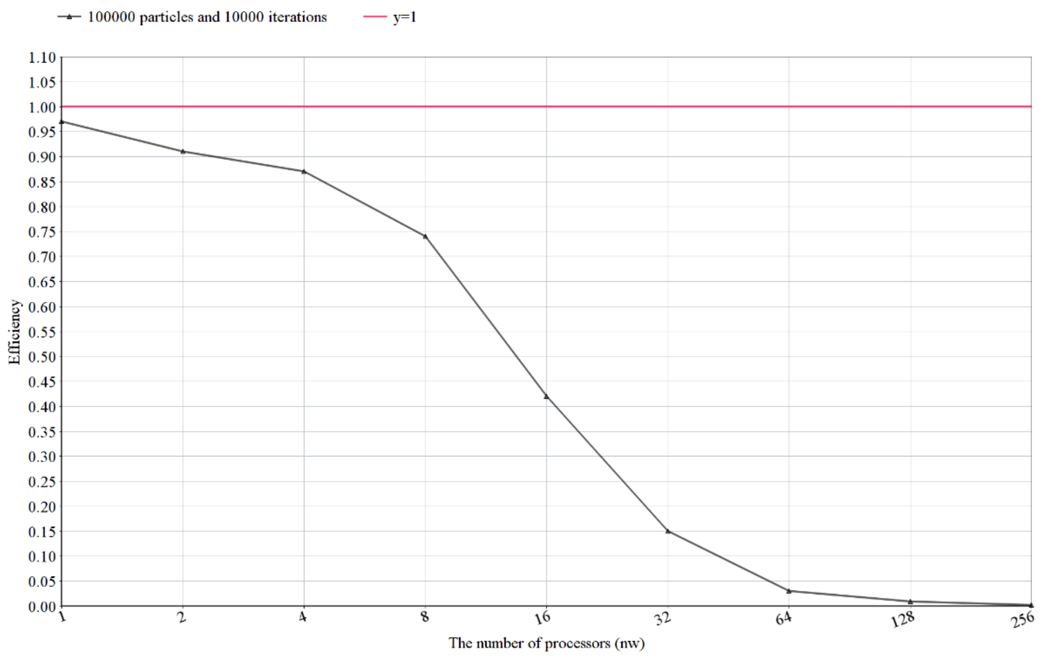
From *Graph 2* we can clearly see that the maximum speedup that we could achieve from the program that has 100000 particles and 10000 iterations is 6.8 with 16 threads.



Graph 2. The graph illustrates the gained speedup by using different numbers of threads.

c. Efficiency

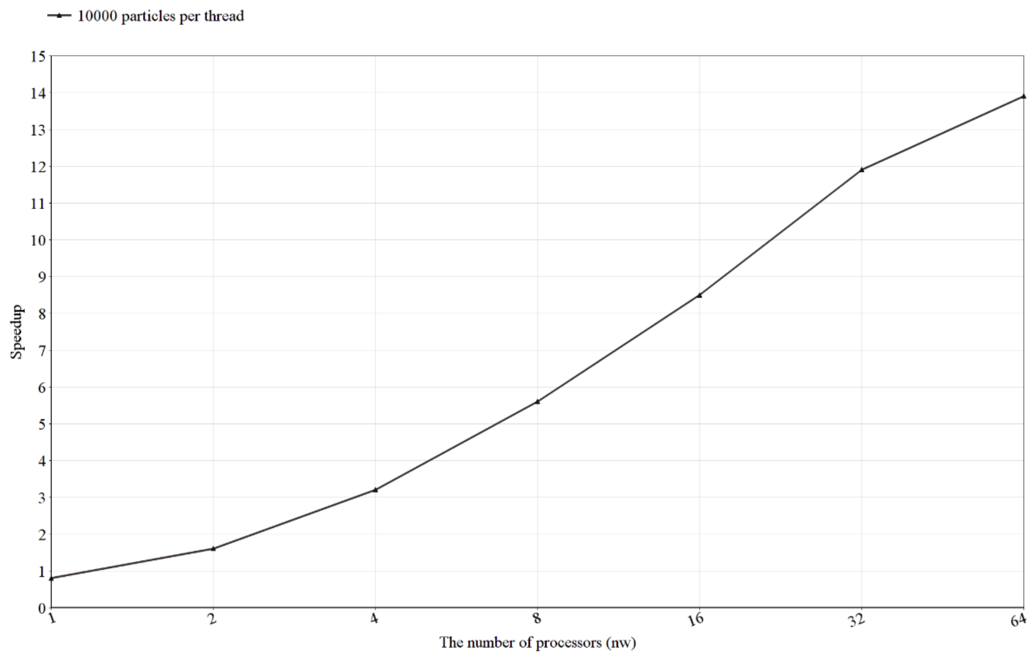
By observing the *Graph 3* we can see that using approximately 4 threads we could gain good efficiency.



Graph 3. The graph illustrates the efficiency of using different numbers of threads.

d. Scalability

From *Graph 4* we can see that by keeping the total number of the particles per thread and increasing the number of threads we could get much better speedup, rather than trying to run the same amount of particles with different numbers of threads.

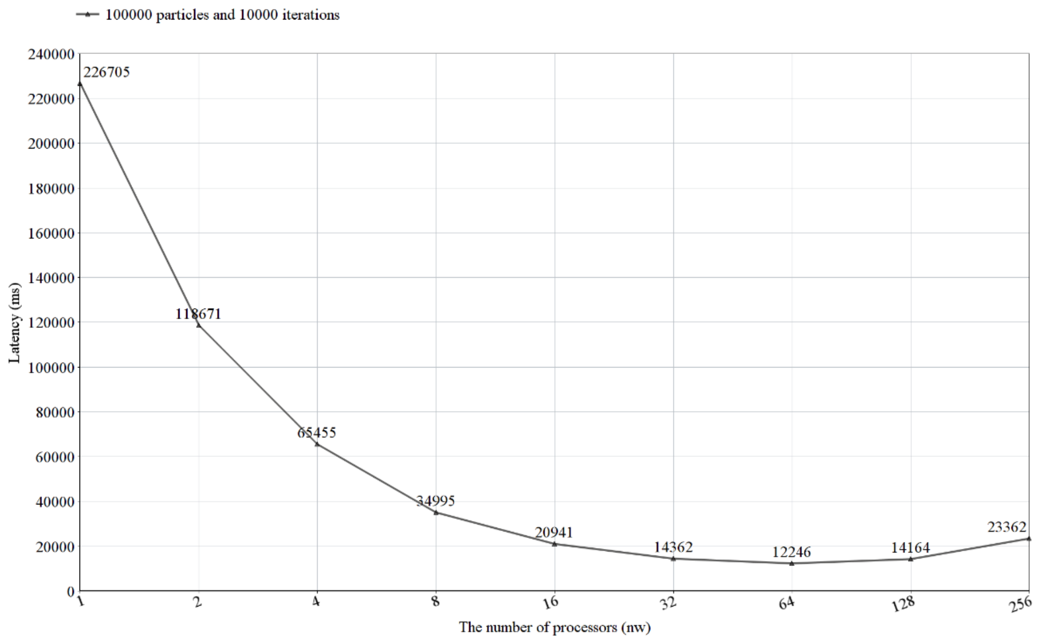


Graph 4. The graph illustrates the weak scaling of the program where we keep constant the total number of particles per thread.

2. Parallel application written using FastFlow

a. Latency

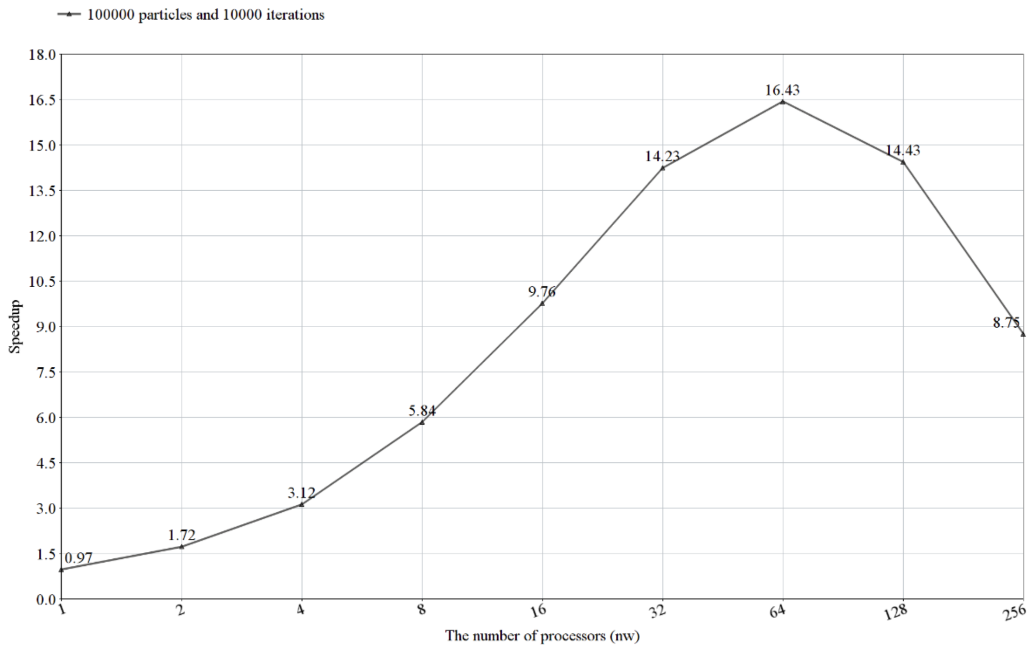
In contrast, what we have seen in *Graph 1,* the latency of the application that is written by using Fastflow is frequently decreasing while increasing the number of threads.



Graph 5. The graph illustrates the changes in the latency by increasing the number of threads while the size of the program stays even.

b. Speedup

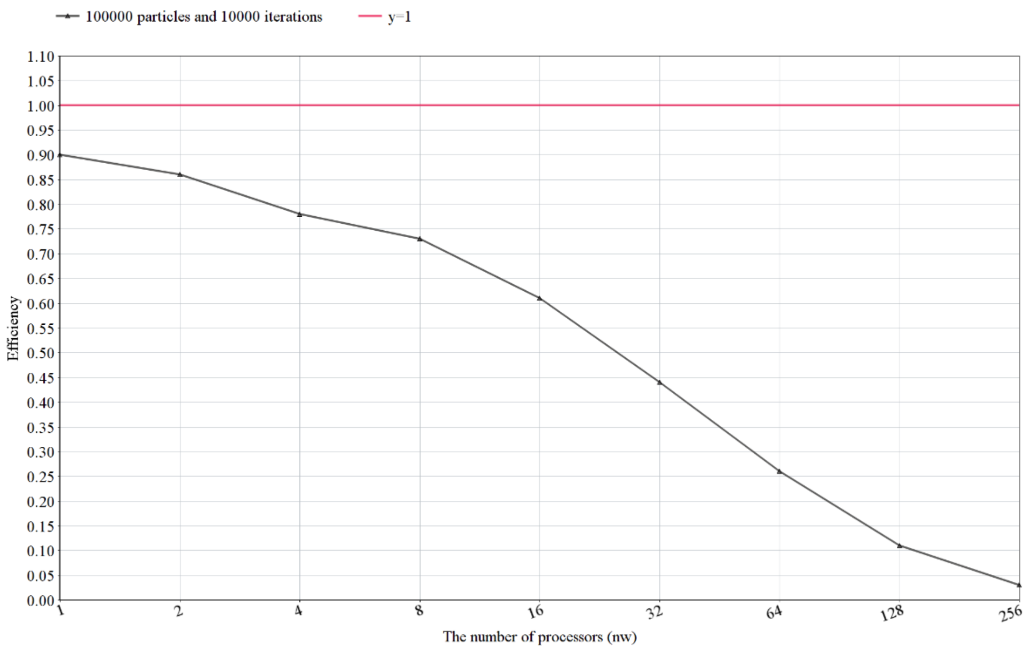
Obviously, the speedup is much higher in this application as well. Hence, by using at most 64 threads we could get our maximum speed up which is approximately 16.5.



Graph 6. The graph illustrates the gained speedup by using different numbers of threads.

c. Efficiency

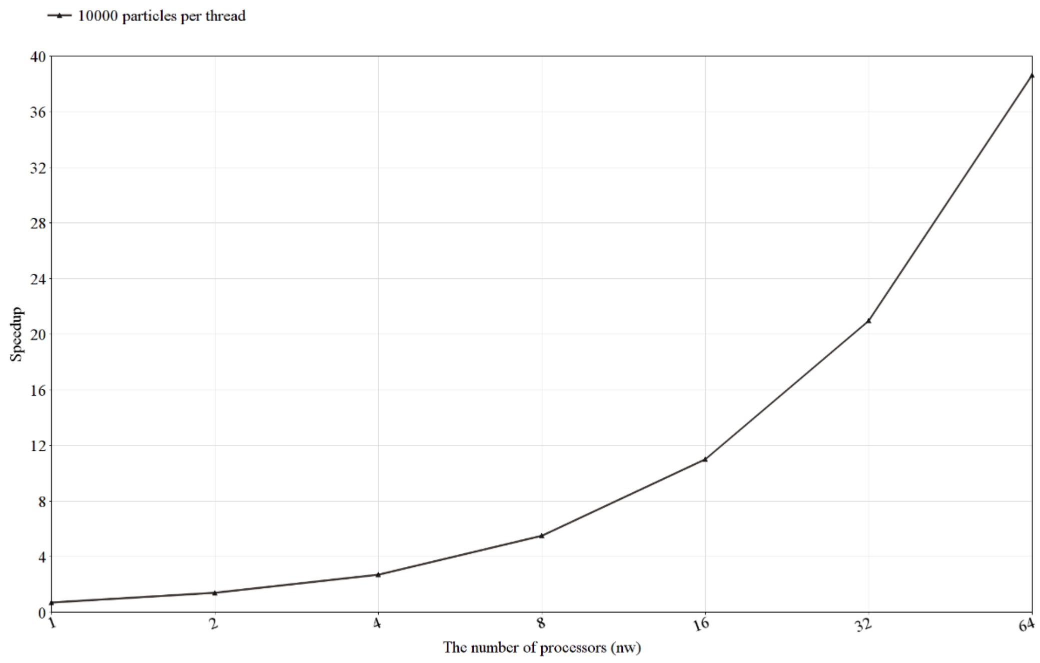
However, as you may see from *Graph 7, the efficiency is still not so good.*



Graph 7. The graph illustrates the efficiency of using different numbers of threads.

d. Scalability

Below *Graph 8* shows the expected results. Hence, by keeping the total number of the particles per thread and increasing the number of threads we could get much better speedup, which is noticeably higher than the previous version of the application that was written using C++ threads.



Graph 8. The graph illustrates the weak scaling of the program where we keep constant the total number of particles per thread.

Conclusion

Taking into account that our application is not 100% parallel, getting less speedup by using a higher number of threads was an expected result. Interestingly, the parallel application written in the FastFlow showed better performance against the one written by using C++ threads. Although, both versions have almost the same code, in the one that was written by using C++ threads, assigning the intervals per thread, forking, and joining the threads are more time-consuming.