SPM Application Project Report

Remo Andreoli

September 2019

Parallel architecture design

The Particle Swarm Optimization deploys a "swarm" of particles on a N dimensional space in order to find the global optimum. Each particles shares its results to a global knowledge, that represents the current best position and best evaluation achieved by the whole swarm. At the end of each iteration, the particles access this global knowledge in order to update their results to the best ones. Therefore, even if we are able to parallelise the iterative step of a particle, the latter has to wait the rest of the swarm before starting the next step, in order to avoid proceeding towards the wrong search space. Such a fine grained computation may lead to poor performance in large swarms due to communication overheads: so I opted for a coarser approach, by diving the particles in smaller independent groups, that share their "group" knowledge instead of the individual one. This workflow is shown in Figure 1, and its steps are described below:

- **Initial step on group X:** Execute the initialization step sequentially for each particle of the group. Each group is independent from one another and can run in parallel.
- **Iterative step on group X:** Execute sequentially the iterative step for each particle of the group. As for the initial step, each group is independent from one another and can run in parallel. This step is executed a defined number of times.
- **groups synchronization:** Synchronize every group on the overall best group knowledge (the group's best position and evaluation) achieved until that moment. This step is executed at the end of both the initial and iterative step.
- **END**: After a number of iterative steps, or if a given threshold is reached, the swarm of groups returns the best evaluation achieved and terminate the search.

A parallel design that fits in this context is the **master-worker model**. The model consists of a master and a number of identical workers. The first controls the latters by issuing tasks and gathering responses. The only possible communication is between master and worker. Figure 2 shows a graphical rappresentation of the model.

In our particular case, each worker represents a group of particles.
Depending on the situation, a worker waits for the next iteration to start (**waiting phase**), or it is free to execute the next iterative step of the group (**running phase**). The phase switch is issued by the master: the worker goes from running to waiting phase after sending its best values; the worker goes from waiting to running phase after receiving the current best values achieved by the swarm. The master is also in charge of notifying the termination of the algorithm, if the global optimum is reached.

Hence, to summarize the roles:

- The master acts as a barrier and makes sure that everyone is heading towards the correct search space. It does it by starting the computation, terminating the computation and synchronizing the workers' knowledge at the end of each iteration.
- The worker initializes the group of particles and executes the iterative step whenever the master authorises it.

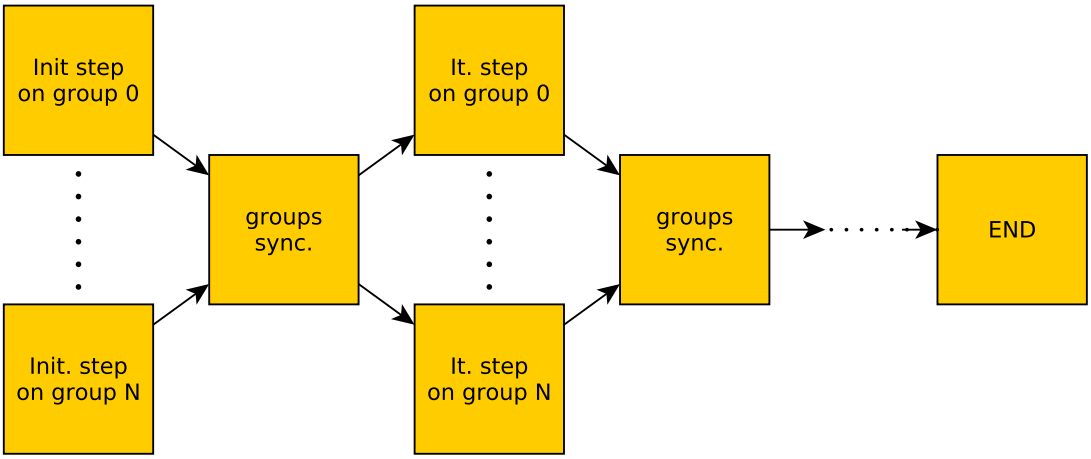


Figure 1: PSO Workflow in a parallel environment

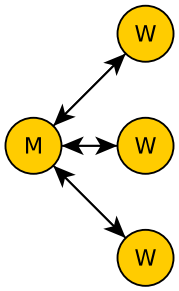


Figure 2: master-worker model

Performance modeling

An abstract performance model that define the service time of the master-worker design is the following:

$$\begin{cases} T_s = max\{T_{master}, T_w\} \\ \\ T_w = max\{T_{w0}, ..., T_{wN}\} \end{cases}$$

T_{master} is the service time of the master, which corresponds to the synchronization period, N is the parallel degree and T_w is the service time of the slowest worker. Therefore, T_{wi} corresponds to the group iteration period of worker i plus the idling period. Indeed, each worker has to wait all the other ones before starting the next iteration. In the best case scenario, the time spent executing the iterative step is the same for everyone: therefore, there is no idling phase, because everyone is working and issuing results in parallel. Assuming no communication overheads:

$$T_{wi} = \frac{T_{group\_iteration}}{N}$$

The bottleneck in the master-worker model is the master: a large number of workers implies a complex communication mechanism, that necessarily leads to longer waiting phases and a loss in efficiency.
According to a simulation cited by [1][1], which briefly describes the master-worker paradigm and its problems in chapter 7, the bottleneck appears with a number of worker between 8 and 16.

[1]Unfortunately, I was unable to retrieve the article itself

Implementation structure

The FastFlow implementation is based on the `ff::farm` building block without the collector, while the native tries to loosely mimic it, by taking inspiration from the `ff::ff_node`'s life-cycle logic. Note that the business code is exactly the same in both version; Although, the native implementation integrates the functional code with non functional one. Note that in our context the farm is not used as a proper stream parallel skeleton: indeed there is no need for a scheduling or gathering strategy, since every worker is supposed to receive at once the same message. Therefore, the `ff::farm` building block is used only as a "template" to fulfill the master-worker design's requirements.

The group of particles' code is defined in the `PSOWorker` class. The master's code is defined in the `PSOMaster` class. The only difference between the native and FastFlow version of the two classes is that in the latter the underlying thread is handled by the framework itself. The communications between this two classes occurs through queues and `SyncMessage`'s objects. Note that the native version uses the blocking queue's implementation provided in class. The purpose of `SyncMessage`s is to allow the workers to communicate their group's knowledge at the end of an iteration, and to allow the master to communicate the updated values at the start of the next one. Therefore, the barrier is "implicit": the master simply does not communicate to the workers that already finished the iteration, leaving them waiting on the queue.

The next page shows a general workflow applicable to both implementations. Hand in hand with the source code, it should be enough to better understand the whole logic.

Implementation's Workflow

In the following, all the activities and communications happening between a `PSOWorker` and `PSOMaster` node, right after calling the `run_and_wait_end()` method:

1. The worker executes the initial step of the algorithm, the master prepares N `SyncMessage`s storing the highest float number possible, and issues them to the workers in a round-robin fashion.

2. The worker retrieves the message from its queue and chekc if its group knowledge is better than the one stored in the message. If so, it appends its values, then it sends the message back to the master. The latter waits an answer from every worker.

3. The worker is idling on the `pop()` method of the queue. The master updates all the messages with the best group knowledge achieved and then sends back the results to the workers.

4. The worker wakes up and repeats step 2 and step 3. After a certain amount of iterations, the master forwards an End-of-Stream message that terminates the computation.

Experimental validation

Description

Here is a list of notes that describe the configuration and the structure of the experimentation phase:

- All the tests have been performed by varying the parallel degree, which corresponds to the number of groups, and the total number of particles. All the other parameters have been fixed and stored as macro in `pso_utils.hpp`:

PSO parameter	Value
a	0.5
b	0.8
c	0.9
max iteration	150
search-space bounds	[-10000, 10000]

  Note that, for simplicity sake, the boundaries are the same for both dimensions.

- The cost function is the same for all the experiments:

$$f(x, y) = x^2 + y^2 + 1$$

- To achieve the best performance possible, the workloads must be balanced. For this reason, the particles are equally distributed among the groups:

$$\#particles\_per\_group = \frac{\#swarm}{\#groups}$$

  Note: For simplicity sake, fractions are rounded up in any case.

- The next section will show the speedup and efficiency plot of the average completion time. Each test have been executed 15 times. The scalability plots are not shown, due to the fact that the parallel time with parallel degree 1 resulted equal to the sequential time in all the experiments.
- Each thread have been sticked to a different core using a round-robin strategy, in order to reduce cache misses caused by unfortunate thread migrations and to increase the efficiency.

The experiments

The objective of the experiments was to assess the performance of the implementations on two different scenarios: a "medium" complexity scenario, using a swarm made of 10000 particles, and a "high" complexity scenario, using a swarm made of 100000 particles. I ditched the idea of showing the results on lighter workloads, since the overheads were too noticeable. The plots are shown respectively in Figure 3 and Figure 4.

Both implementation shows similar results and a visible deterioration of performances after more or less 16 workers. Interestingly, it is the same threshold traced by [1]. One of the major cause of this bottleneck should be the gathering strategy. This is even more noticeable on the native version, due to the fact that it is not as optimized as the FastFlow's farm template. The native's master constantly polls the queues to check if they are empty in a round-robin fashion. As discussed in the Performance section, this leads to poor performance when dealing with lots of workers. However, If we fix the number of groups and increase the particles, the bottleneck becomes less noticeable, due to the coarser computation.

Out of curiosity, I came up with several options that should dampen the bottleneck:

- Change the gathering strategy. This might not be possible, considering that even FastFlow is not performing much better.
- Remove the lock from the `is_empty()` method, which is as trivial as irrelevant.
- Removes the queues and use a different mechanism.
- Give up on the model and use a decentralized one.

I designed a different native version, that mixes the third and fourth option: the worker owns a pointer to a synchronization entity, that acts as the global knowledge and as an "explicit" barrier; The last worker to finish triggers the phase switch, that wakes up everyone. The performance plots are shown in Figure 5 and Figure 6. The results are interesting, considering that the Achille's heel of this implementation should be the increase of cache misses due to the coherency mechanism. Indeed, each time a worker invokes this synchronization entity, it causes a chain of cache entry updates on the cores where the other workers reside. This problem should be also highlighted by the Xeon Phi KNL's architecture, that arranges the core in a grid, and by the thread affinity strategy that I chose: indeed, assigning a core to a thread in a round-robin fashion may not be the best solution for routing data in a grid.

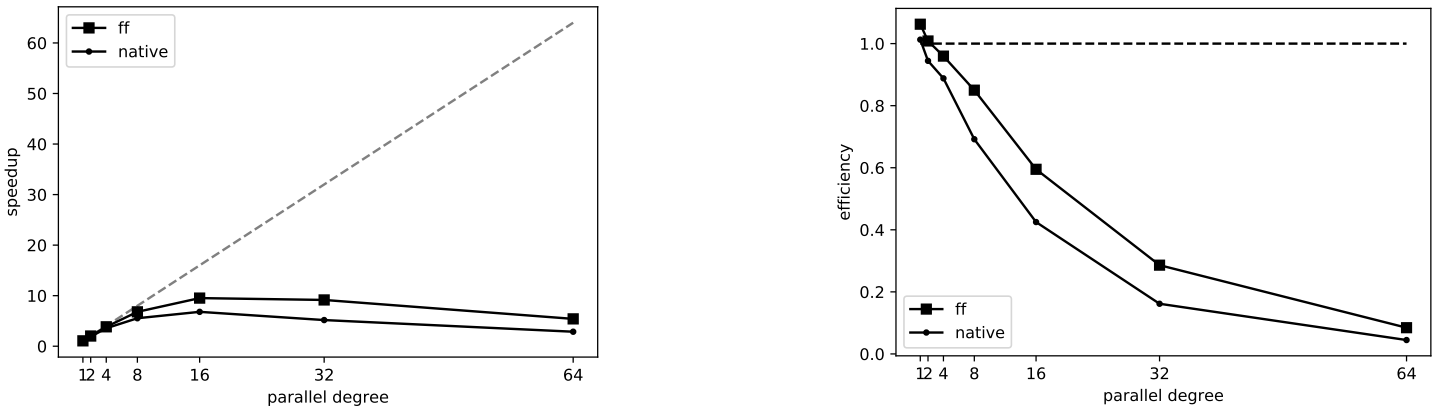


Figure 3: Medium complexity: speedup (Left), efficiency (Right)

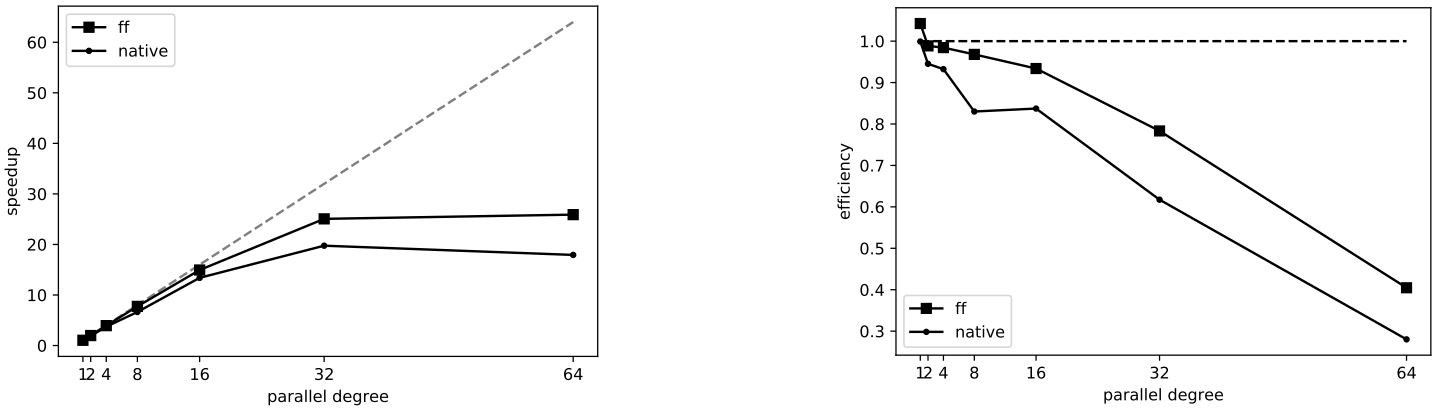
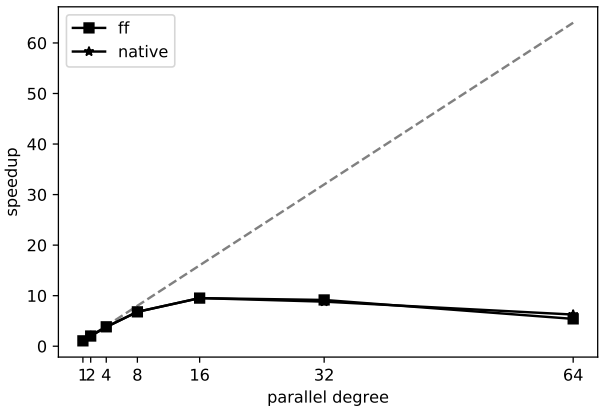


Figure 4: High complexity: speedup (Left), efficiency (Right)



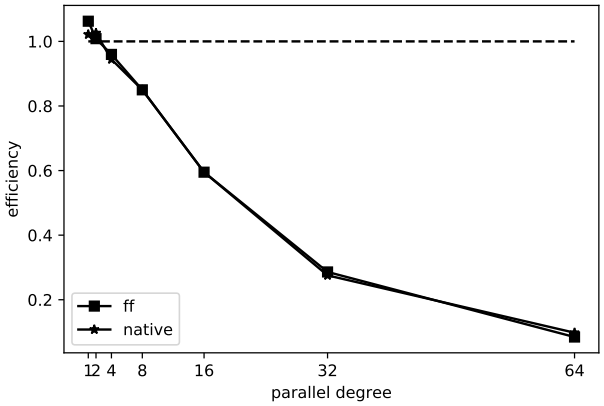
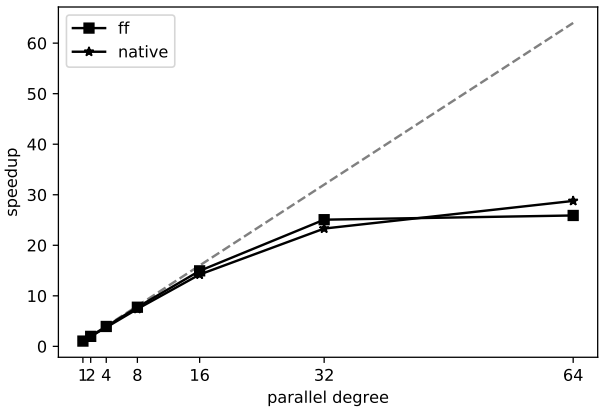


Figure 5: Medium complexity: speedup (Left), efficiency (Right)



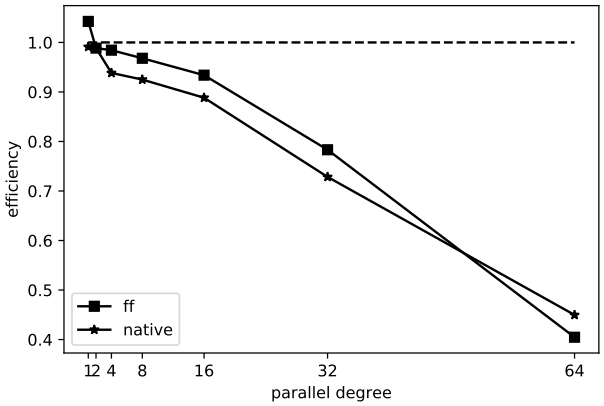


Figure 6: Medium complexity: speedup (Left), efficiency (Right)

Bibliography

[1] A. Migdalas, Panos M. Pardalos, and Sverre Storoy. *Parallel Computing in Optimization*. 1st. Springer Publishing Company, Incorporated, 2012. ISBN: 1461334020, 9781461334026.