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 dividing the particles in smaller independent groups, that share their best values (the group knowledge) instead of the individual ones. This workflow is shown in Figure 1 and described below:

- **Initial step on group X:** Execute the initialization step sequentially for each particle of the group and keep track of the best initial values achieved. 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 and keep track of the best values achieved overall. 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 achieved (the global knowledge). This step is executed at the end of both the initial and iterative step and it is required before moving onto the next iteration.
- **END**: After a number of iterative steps, the swarm of groups returns the best evaluation achieved and terminate the search.

This sort of workflow can be seen as three data parallel computations in sequence: a Map phase, which is the application of the Swarm Particle's search logic on a set of particles, a Reduce phase ("local" reduction), which corresponds to the localization of the group knowledge among a set of particles, and another Reduce phase ("global" reduction), which corresponds to the localization of the overall best group knowledge, the global knowledge.

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, where the first controls the latters by issuing tasks and gathering responses. The only possible communication is between master and worker, as shown in Figure 2. In our particular case, each worker represents a group of particles and it is in charge of executing the first two data parallel computations, which together forms the group initial or iterative step, while the master is in charge of the global reduction.

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 issuing its group knowledge; the worker goes from waiting to running phase after receiving the swarm's global knowledge. The master is also in charge of notifying the termination of the algorithm.

Hence, to summarize the roles:

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

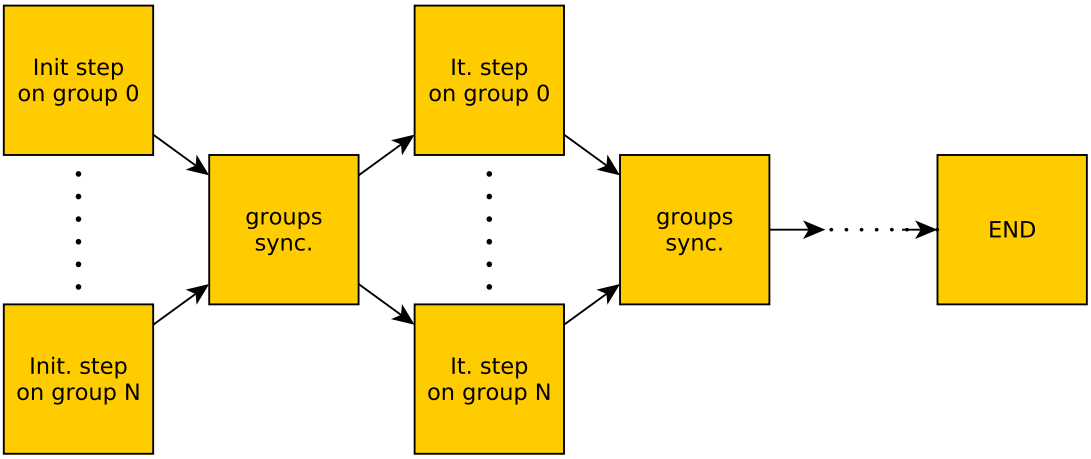

Figure 1: PSO Workflow in a parallel environment

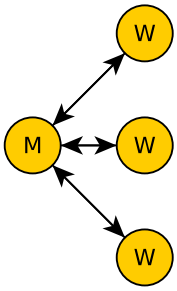

Figure 2: master-worker model

Performance modeling

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

$$T_s = max\{T_{master}, \frac{T_w}{N}\}$$

T_{master} is the service time of the master, which corresponds to the global reduction period, T_w is the service time of the workers, which corresponds to the group iteration period, and N is the parallel degree.

Since each worker has to wait all the others before starting the next iteration, the best possible scenario is ensuring that the workload is well balanced. Therefore:

$$\#particles\_per\_group = \frac{\#n\_particles}{\#groups}$$

This way every worker should have the same service time, assuming that all the processing elements have the same performances. The serial fraction that we are unable to parallelise is the master, which is the bottleneck of the model: indeed an increase on the number of workers causes additional communication overhead, which leads to longer waiting phases and to a loss in efficiency.

The optimal parallel degree is the one that minimizes the service time, which is:

$$\hat{N} = \frac{T_w}{T_{master}}$$

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: the only difference is that the native implementation couples together functional and non functional code.

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, and the workload is well balanced. Therefore, the `ff::farm` building block is used only as a "template" to fulfill the master-worker design's requirements.

An instance of `PSOWorker` represents a group of particles, while an instance of `PSOMaster` represents the master. Each worker has its own communication queue: this "queue individuality" should decrease the number of cache misses due to the coherency mechanism (**Note:** the native version uses the blocking queue's implementation provided in class). The actual communication happens using `SyncMessage`s, which are used by the workers to send their group's knowledge at the end of an iteration, and to receive the global knowledge by the master 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 computation is started by the workers.

The next page shows a general workflow applicable to both implementations, and it should be enough to better understand the whole logic. The source files are available in the folder and are named respectively `ff_parallel.cpp` and `native_parallel.cpp`.

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. If the search is not started yet, the worker executes the initial step, else it executes the iterative step for each particle. After that, it issues its group knowledge to the master via a `SyncMessage` and waits for a response, putting the thread to sleep.

2. The master waits by idling on a random queue. When everyone has issued its message, the master finds the best group knowledge, updates the content of all the received messages with the best values and sends them back to the workers, in a round-robin fashion.

3. The worker updates its group knowledge to the new values, then it repeats Step 1 and Step 2.

4. After a certain amount of iterations, the master forwards an End-of-Stream message that terminates the computation.

Note: The messages are reused in order to reduce the number of memory allocation/deallocation.

Experimental validation

Description

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

- Run the tests with the following syntax: `./prog n_particles nw`. The particles will be automatically distributed among the groups in order to guarantee the balanced workload mentioned in the Performance section. **Note:** For simplicity sake, fractions are always rounded up.
- The cost function is the same for all the experiments:

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

- 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.

- At the end of the report are shown the speedup and efficiency plot of the average completion time, using different swarm sizes. Each test have been executed 15 times. Note that 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.

- At the end of the report are shown the average execution times of the workers and the average execution time of the master (in milliseconds), using different swarm sizes.
- 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 both implementations, using different parallel degrees and different swarm's sizes. The plots are shown per swarm size from Figure 3 to Figure 8: "medium size" refers to a swarm of 10000 particles, a "large size" refers to swarm of 100000 particles, and a "extra large size" refers to a swarm of 1000000 particles.

Both implementation shows similar results and seems to suffer the global reduction phase, which is the serial fraction of the computation, when the swarm's size is small. This is particularly evident with the medium size, where the execution time of the master raises quickly (Figure 4), lowering the possible optimal parallel degree. For this reason I ditched the idea of showing the results on smaller sizes, since the synchronization overhead was too noticeable. Therefore, both implementations are not able to fully exploit the whole machine in these scenarios. This visible deterioration of performance affects especially the native version: as a matter of fact, the gathering and scheduling logic are not as optimized as in the FastFlow's farm template. The native's master simply waits onto a random queue, because (in theory) all the queues should fill roughly at the same time, then it processes the received messages and sends them back in a round-robin fashion. As discussed in the Performance section, this leads to poor performance when dealing with lots of workers. The poor way the messages are handled is further aggravated by the Xeon Phi KNL's architecture, that arranges the cores in a grid: since the threads are assigned in a round-robin fashion in both the implementations, the data routing might be inefficient, causing additional communication overhead.

However, the master's execution time does not increase proportionally to the number of particles, mitigating the serial fraction's burden on larger sizes. This is well shown by the extra large swarm size's plots (Figure 7 and Figure 8), where the speedup is almost always linear and the master's execution

time remains relatively constant, allowing for an higher optimal parallel degree. Therefore, the parallel model performs well if the overall computation is coarse grained.

The only time the native implementation is able to beat FastFlow is when the parallel degree is equal to the number of physical cores and the swarm particle is extra large. This is probably due to the fact that FastFlow has to handle internal subroutines and background tasks that, in this scenario, have to share the contexts with the computing nodes.

Medium swarm (10000 particles)

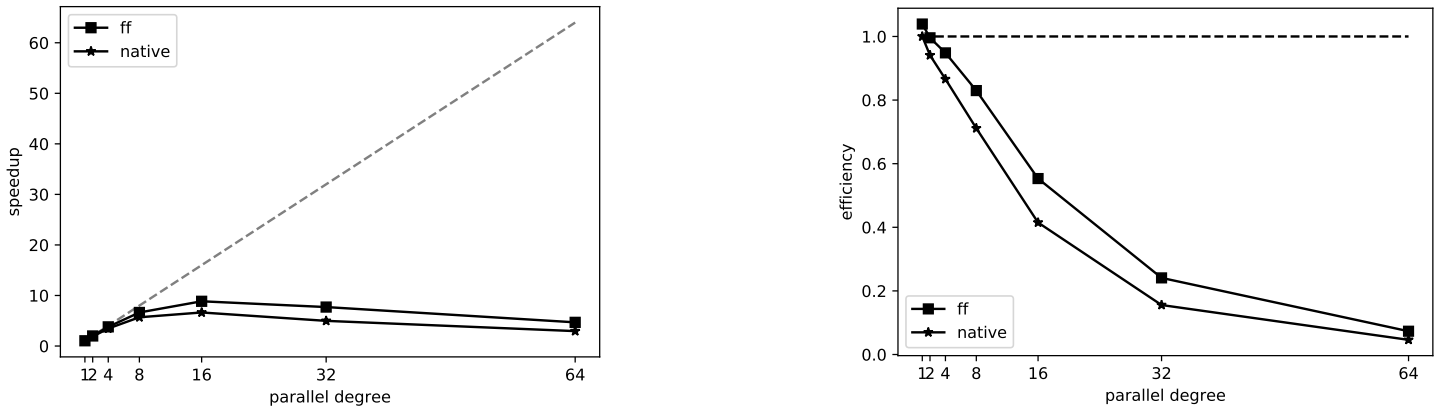

Figure 3: Medium swarm size: speedup (Left), efficiency (Right)

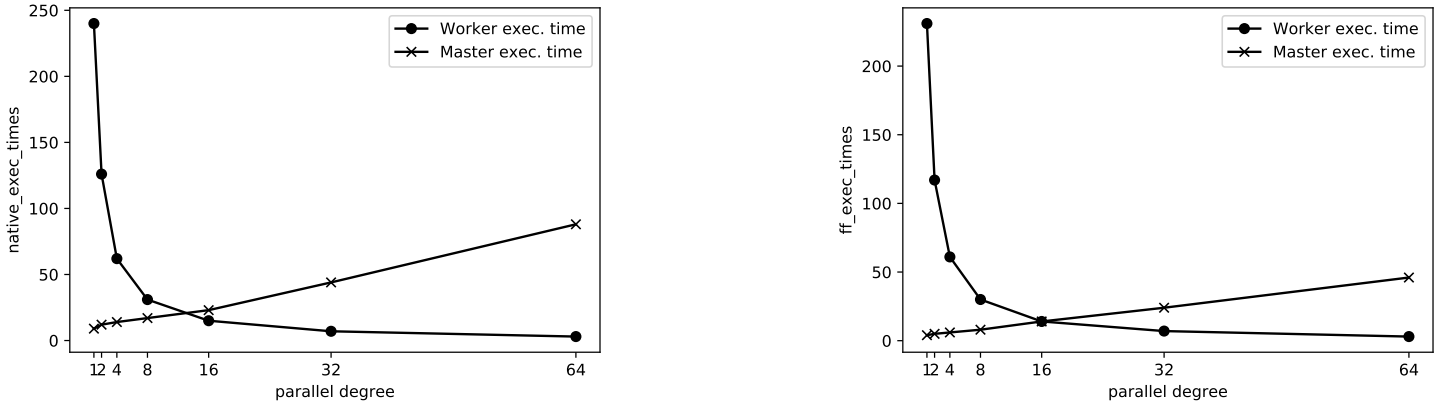

Figure 4: Execution times with medium swarm size: native (Left), FastFlow (Right)

Large swarm (100000 particles)

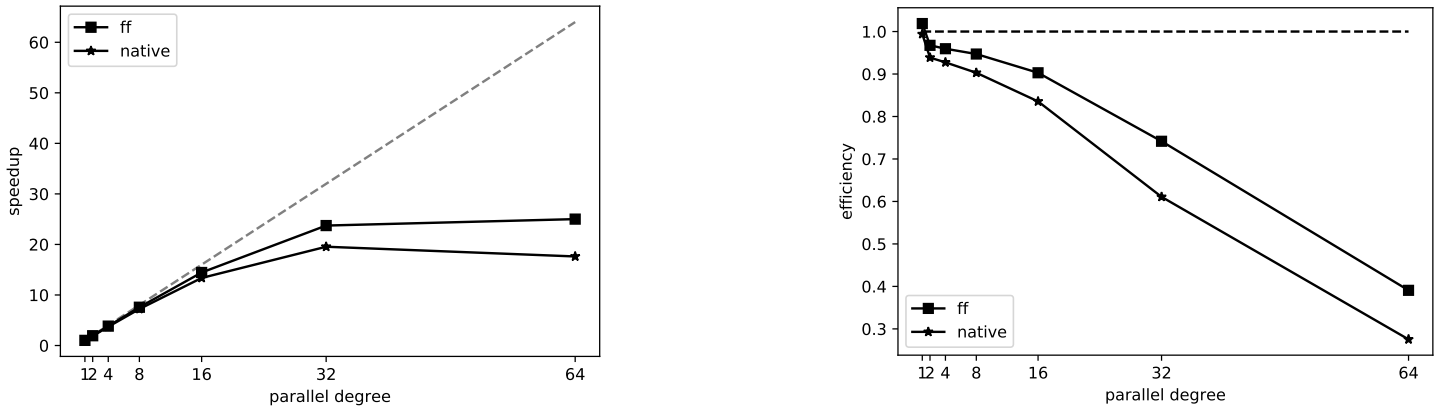

Figure 5: Large swarm size: speedup (Left), efficiency (Right)

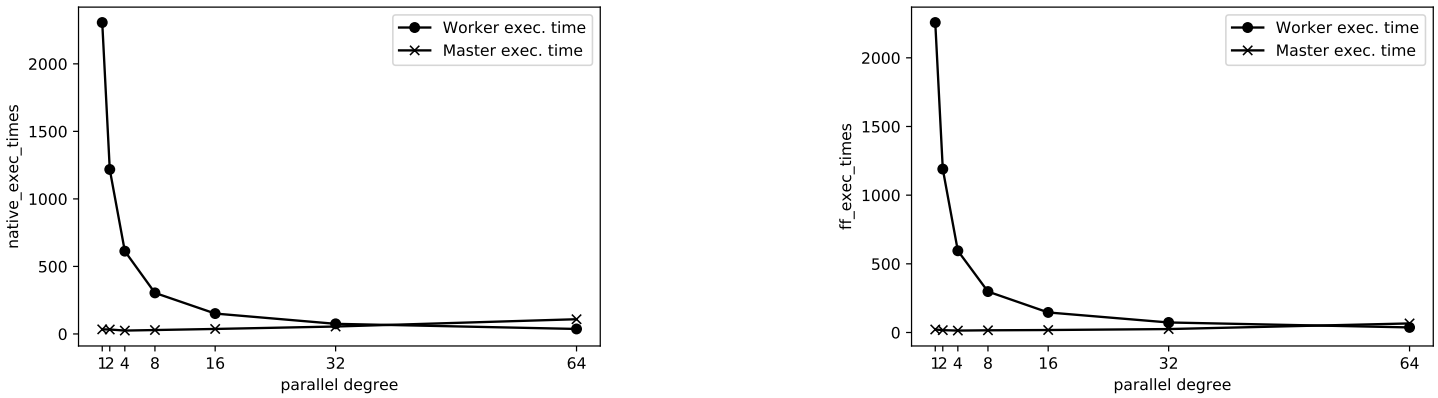

Figure 6: Execution times with large swarm size: native (Left), FastFlow (Right)

Extra large swarm (1000000 particles)

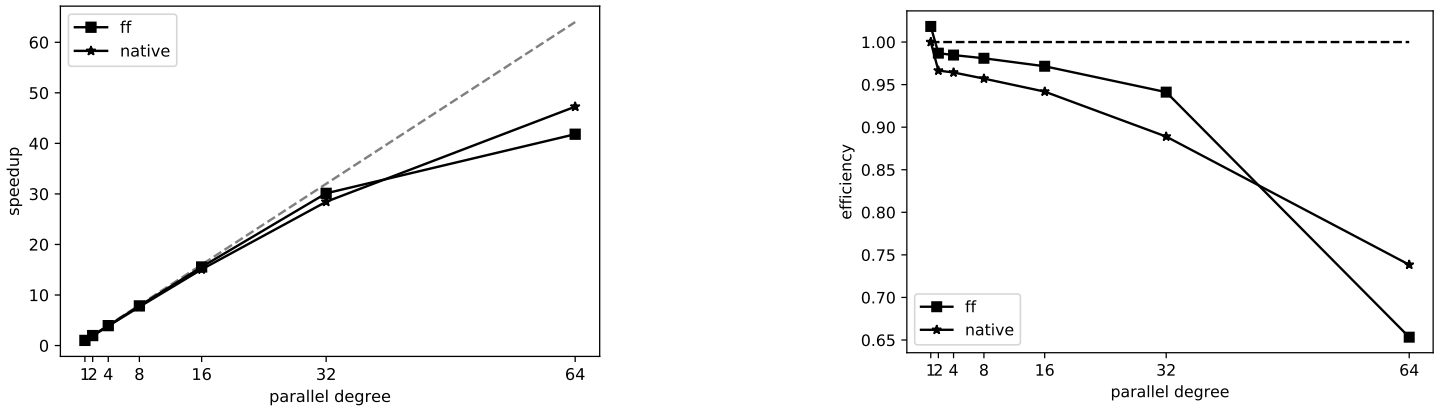

Figure 7: Extra large swarm size: speedup (Left), efficiency (Right)

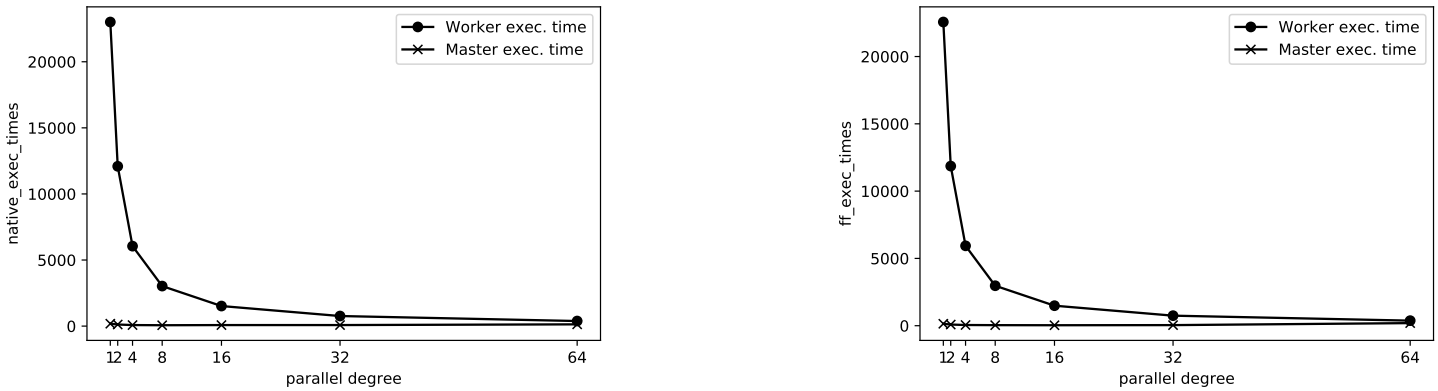

Figure 8: Execution times with extra large swarm size: native (Left), FastFlow (Right)