About two data-parallel implementations of an algorithm that extimates the minimum of a function on a two-dimensional search space leveraging the *Particle Swarm Optimization* (PSO) method.

Francesco Tosoni[1,2]

[1] *Computer Science dept., Università degli Studi di Pisa, Pisa, I-56127*
[2] *TeCIP Institute, Sant'Anna School of Advanced Studies, Pisa, I-56124*
[*] *Corresponding author: f.tosoni@studenti.unipi.it*

Compiled January 27, 2020

These pages contain a set of subsidiary notes about the analysis, design and implementation of a multi-thread c++ data-parallel code which approximates the minimum of a floating-point function defined over a two-dimensional search space, leveraging the *Particle Swarm Optimazion* (PSO) computational method.

https://github.com/SuperNabla95/Particle_Swarm_Optimization

1. INTRODUCTION

According to Wikipedia the Particle Swarm Optimization (PSO) is "a computational method that optimizes a problem by iteratively trying to improve a *candidate solution* [...]. It solves a problem by having a population of candidate solutions, here dubbed *particles*, and moving these particles around in the search-space according to simple mathematical formulae over the particle's position and velocity."

In this set of notes (and in our implementation) we are going to discuss the parallelization of an algorithm that makes use of the PSO technique in order to determine – by mean of an iterative process – better and better *extimates for the minimum value a function on a two-dimensional (2D) search space*.

More formally, the problem and the related algorithm herein described are defined by the quadruple $(D, func, n_p, n_{iter})$, where (1) $D = [x_{min}, x_{max}] \times [y_{min}, y_{max}] \in \mathbb{R}^2$ is two-dimensional rectangular search space, (2) $func : D \rightarrow \mathbb{R}$ is a floating point function, (3) n_p is a positive integer that represents the number of candidate solutions computed at each algorithmic iteration, and (4) n_{iter} is a positive integer that specifies the number of iterations of the PSO method.

Initially (let's say for the simplicity of exposition: at iteration number 0), the PSO algorithm sets $gmin_0 = +\infty$ as the candidate minimum for $func$ in D, and sets the candidate position $xgmin_i \in D$ for that minimum to a dummy value. During this same iteration 0, the PSO algorithm generates also a set of n_p *particles* for the extimation of the n_p candidate solutions. Each particle p_i $(i = 1, 2, ..., n_{iter})$ is a quadruple $p_i = (x_i, v_i, lmin_i, xlmin_i)$, where:

- $x_i \in D$ and $v_i \in \mathbb{R}^2$ represent – respectively – the position and the velocity vectors of the particle p_i within the two-dimensional search space D. Initially (i.e. during iteration 0) x_i is chosen uniformely at random in D;
- the local minimum $lmin_i \in \mathbb{R}$ is a value initially set to $lmin_i = +\infty$; and
- the position of the local minimum $xlmin_i \in D$ is initially set to a dummy value.

Since the components of each particle p_i are time-varying during the iterations of the algorithm, in the following we will denote (when necessary) with $x_i(j)$, $x_i(j)$, $lmin_i(j)$ and $xlmin_i(j)$ the values assumed by position, velocity, local minimum value and local minimum position (respectively) during the j-th iteration of the algorithm.

During a generic j-th iteration $(j = 1, 2, ..., n_{iter})$, the PSO algorithm undergoes three different phases:

1. for each particle p_i updates the local minimum $lmin_i$ according to the rule $lmin_i \leftarrow \max(lmin_i, func(x_i)$, in order to obtain the set of candidate solutions $\{lmin_i | i = 1, 2, ..., n_p\}$ for the minimum of $func$ in D. If a new local minimum value is found, then the position $xlmin_i$ for the local minimum is updated accordingly;
2. computes the minimum $gmin_j$ among those candidate solutions and the minimum $gmin_{j-1}$ found at the previous step; and finally
3. updates the position and velocity of each particle p_i as in equations 1 and 2. The parameters $a \in [0.4, 1.4]$, $b \in [1.5, 2.0]$ and $c \in [2.0, 2.5]$ are global parameters of the algorithm (i.e. are the same for all the particles) and are known in the literature as *inertia factor*, *self confidence* and *swarm confidence* (respectively); R_1 and R_2 – instead – are two i.i.d. uniform random variables which assume values in the interval $[0.0, 1.0]$.

$$\begin{aligned} v_i(j+1) \leftarrow\ & a \cdot v_i(j) \\ & + b \cdot R_1 \cdot (lxmax_i(j) - x_i(j)) \\ & + c \cdot R_2 \cdot (gmin_i(j) - x_i(j) \end{aligned} \quad \textbf{(1)}$$

$$x_i(j+1) \leftarrow x_i(j) + v_i(j+1) \quad \textbf{(2)}$$

2. ANALYSIS AND C++ IMPLEMENTATION

It is easy to recognize a **map-reduce** pattern int he logic of the application.

As already seen in section 1, each iteration of the algorithm herein analyzed consists of *three* different phases. Let's assume to have at our disposal n_p virtual processors. Virtual processor $VP[i]$, $i = 1, 2, ..., n_p$, takes care of the processing procedure for the i-th particle during all the iterations of the algorithm (i.e. $VP[i]$ is the owner of particle p_i). Using the terminology of parallel programming, the operations performed in these three phases can be identified as `map` operations (phases 1 and 3) and `reduce` operations (phase 2). Even the initialization operations (iteration 0) can be modelled as `map operations`. It's important at this point to go deeply into the analysis of the `reduce` phase (phase 2), in order to ensure (a) the correctness and (b) the optimality of the performances of our data-parallel solution for the PSO problem. In particular one should notice that the reduce phase, in which the candidate new value for the global minimum $gmin_i$ is determined must be executed *synchronously* with respect to the subsequent `map` phase 3; by other words, any possible solution that we are going to propose must prevent all virtual processor $VP[\cdot]$ to enter in phase 3 until there is at least a (slower) virtual processor that is still involved in the operations of phases 1 or 2 *of the same iteration*. Indeed, looking back to the semantics of phases 2 and 3, it appears straightforward to say that it is not possible to update space and velocity of the particles, until the global value for the minimum during that iteration is yet to be computed (i.e. until there are still some particles for which the reduce phase is yet to be completed). On the other hand, one should also notice that phases 1 and 2 can instead be overlapped within a certain algorithmic iteration, that is: one can allow some virtual processors to start computing phase 2 (i.e. the reduce phase) regardless of the fact that there are still virtual processors in phase 1 or not. Similarly, one should also notice that it is possible to allow some (fast) virtual processor to enter in phase 1 (or even 2) of a certain iteration j, even if there are still other virtual processors performing some computation in phase 3 `of the previous iteration` $j - 1$.

A. job partitioning

Let's image that we want to implement our PSO application utilizing nt different threads. We need to define a strategy to map virtual processors onto threads. In the implementation herein discussed, we have assigned to the n-th of those threads ($n = 1, 2, ..., nt$) the computation performed by $VP[i]$ for all $i \in [(n-1) \cdot \frac{n_p}{nt} + 1, n \cdot \frac{n_p}{nt}]$, so that each threads owns a partition of $\frac{n_p}{nt}$ different particles[1].

[1]For sake of simplicity we consider here n_p to be a multiple of nt. In our implementation we actually assign $\lfloor \frac{np}{nt} \rfloor$ particles to some threads, and one particle more (i.e. $\lfloor \frac{np}{nt} \rfloor + 1$) to the remaining ones, in order to ensure that: (a) every of the n_p particles is assigned to some thread, (b) we are still able to guarantee *load balancing* among threads.

B. initialization

The initialization (i.e. iteration 0) has an execution cost that is proportional to the number of particles assigned to each thread. Hence, the time T_{init} for the initialization can be written as $T_{init} \approx \delta \cdot (n_p/nt)$. Some performance evaluation analysis have suggested for *delta* the value $\delta = 3.72$ nanoseconds.

C. map operations

The `map` operations (namely the one of phases 1 and 3) do not require any *inter* thread communication nor synchronization (i.e. they are *emberassingly parallel*), so that they turn out particularly easy to parallelize. Furthermore, as already seen in the introductory part of this section, there is no need for some kind of synchronization between phase 3 of iteration $j - 1$ and phase 1 of iteration j; this suggests to *aggregate map operations of phases 3 and 1* in just one larger `map`. The times T_{m1} and T_{m3} needed by each thread to perform those `map` operation are proportional to the number of particles to be processed.

D. reduce operations and barrier

Once that we applied the partition described in subsection A, we need to subdivide phase 2 (i.e. the reduce phase) into two sub-phases. Hence, we can think to implement during each iteration a *local reduce* among particles *within the same partition*, followed by a *global reduce* among different threads. We will also refer to this two sub-operations as *intra*-thread reduce (for the local reduce) and *inter*-thread reduce (for the global).

local reduce sub-phase. The local reduce procedure clearly requires a time T_{lred} that is proportional to the number of particles assigned to every thread; it is straightforward to say that a *local* reduce procedure does not require any *inter*-thread synchronization and hence can be considered *emberassignly parallel*. Moreover, it is possible to define the operations in this `reduce` sub-phase in terms of a `map` operation, as for phases 1 and 3. As seen at the beginning of section 2, there is no need for a barrier (or similar synchronization primitives) between phases 1 and 2; hence, bringing argumentations similar to the ones of subsection C, we can figure out to aggregate the *local reduce subphase* performed in iteration j with operations permformed in phase 3 (of iterations $j - 1$) and phase 1 (of iteration j). This contributes to a further increment of the granularity of the computation that we have at hand. In this way, we finally obtain the *mapping scheme* `map(sequential(phase 3, phase 1, local reduce))` · `(global reduce)` utilized for the `c++` (and also the `fastflow`) implementations.

global reduce sub-phase. Let's dig now into the details of the global reduce procedure. With regard to what stated at the beginning of this section, one can think to implement some kind of barrier among those nt threads between phases 2 and 3 of a particular iteration. The task of the barrier is ensuring that all threads terminate the reduce phase (i.e. phase 2) before allowing any other thread to enter in phase 3; as already seen, this is important to preserve the semantics of the algorithm. In the following, we will see through a series of transformations how to design a better and better implementation for such a barrier. One can think to implement this barrier leveraging classical synchronization primitives such as `locks`, `atomics`, `condition variables` and/or `wait()` and `notify()` procedures, that are made available by the `c++ standard library`.

For instance: one can implement a barrier using an `atomic integer` that is set to nt at the start of each iteration, and is decremented by one unit every time a thread completes its reduce phase, up to the point that the last (i.e. slower) thread sets this atomic integer to zero and `notify` all the other `waiting` threads that it's now possible for them to enter into the next phase (i.e. phase 3); unfortunately these solution would be characterized by a *high contention parameter* ($p = nt$) among threads for the acquisition of the atomic integer that is at the core of this solution.

An alternative idea to reduce the contention phenomena is that of structuring the gloabl reduce sub-phase in a *tree-like fashion*, with the *nt* threads mapped both to the internal nodes and to the leaves of the binary three. In this case we can assign a an `atomic integer` set to *two* at the beginning of each iteration and assigned to each of the internal nodes of the tree: the contention parameter for each `atomic` is now reduced to *three*, since at most three threads can have access to each `atomic integer`. Those threads are the one mapped to that internal node and the threads mapped to each of its (at most) two children nodes. The reduce phase proceeds bootom-up from the leaf threads towards the root thread, in such a way that each internal node keep track of the best (i.e. highest) value for *gmin* computed by any thread mapped in any of its descendent nodes in the tree. In this way the root node is finally able to determine the best value for *gmin* computed over all the tree (and so, all the threads), and `notify` this new value to all the threads in the tree.

Even if now we have at hand a much better implementation idea, there is still some room for further improvement. One can realize that at this point it's possible to avoid at all `atomic`, `mutex` and `lock` by simply using a set of well designed `flags`. In our implementation we have utilized two `bool` variables per internal node that we have called `isDone` (see figure 1). At the beginning

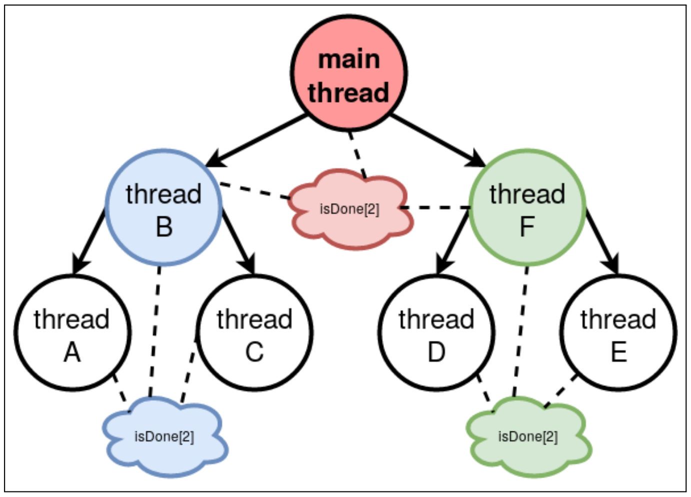


Fig. 1. Threads are organized in a tree like fashion, where each edge represents the relation "is parent of".

of every iteration all the `isDone flags` in all the internal nodes of the tree are set to `False`; then, when a child node terminates the calculation of the best *gmin* candidate in its subtree, it sets the corresponding `flag isDone` in the parent node to `True`. When an internal node has both its `isDone flags` set to `True`, it (1) sets back them to `False`, and (2) sets to `True` its corresponding `isDone flag` in its parent node. With reference to figure 1, we can imagine that thread A and C, after the termination of their phase 2, set their corresponding `isDone flags` owned by its parent thread B to `True`. Now Thread B sets back those flags to `False` and sets its corresponding `isDone flag` in its parent `main` thread to `True`. The process keeps going on, up to the point that the root node has both its owned `isDone flags` set to `True`. Now the root thread is able to (1) determine the best *gmin* candidate over all the tree, (2) check if the candidate value for *gmin* is better than the previous extimate, and eventually update the new value of *gmin* and its position, and (3) notify to all the threads that they can now leave phase 2 and enter into phase 3 of the current iteration. Under the hypothesis that there are at least as many threads as the available physical (or logical) cores available in the machine, it turns out to be a good idea to leverage `busy waiting` techniques for an alternative implementation of the `wait()` and `notify()` primitives. There are a set of good reasons that suggest the usage of `busy waiting` techniques in this context:

- Let's assume that a thread performs a standard `wait()` operation on some `condition variable` and it is immediatly `notified` by some other thread. Even in this lucky case, the overall `wait-notify` procedure will typically require a non-negligible positive amount of time (~10s of mcirosec-onds, typically), whilst potentially `busy waiting` can allow to implement the same primitives with a time wasting that is at least an order of magnitude below (in the order of the microseconds)

- The computation of steps 1,2,3 is very *fine-grained*: even with ~10^4 particles per thread, we expect on a commodity PC a processing time that is in the order of a fraction of the millisecond. In particular, the sub-phase in which the global reduce among threads is performed is *extremely* fine-grained; indeed, the reduce phase involves just few bytes (namely, few words) per thread. Therefore it is possible to bound the time spent in the `for-loops` that implement `busy waiting` to a very low time value.

- Since all threads/workers are assigned the same number of particles, the load on `map` is *perfectly balanced*, and hence it is unlikely that some worker has to wait for the completion of phase 1 of some other worker before starting phase 2. We indeed expect that all the worker will be characterized by the *service time*.

The usage of *spin loops* and *busy waiting techniques* can result, in cases like the one we have at hand, in a overall reduction of the extra-time typically introduced by multithread schemes for the implementation of synchronization primitives. This explains why these mechanims have been introduced in the code of our implementation. Nevertheless, the usage of *spin loops* in the *system code* certainly also contributes to an increase in the *energy consumption* of the target machine; hence, under the *perspective of a more green parallel computing* code [1], a good parallel programmer should always consider to carefully ponderate and minimize (when not avoiding at all) the usage of such techniques.

Summarizing, we have designed a barrier implemented in the form of a binary tree, which is mapped on the same workers/threads of the `map`, and that exploits `busy waiting` technques for the *inter* thread synchronization.

A careful reader could have also noticed that – nevertheless – since the global reduce sub-phase is very fine-grained, it would have been enough to implement it using instead just a plain naïve `sequential reduce`, performed by a single thread (e.g. the `main` thread).

E. fork and join operations

Let's denote with T_f and T_j the times needed to perform a `fork` and a `join` operation, respectively. Typical values for T_f and T_j on commodity PCs are in the order of few hundreds of microseconds. In order to fork all the threads using few time, we can think to avoid a sequential fork of all the threads performed by the `main` thread, and rather structure (again) this operation in a tree-like fashion. In the `c++` implementation attached to these notes, we have proposed to structiure the fork operation using a *binary tree* model. Each thread is in charge to fork *at most* two other threads, that will correspond to the children nodes in the reduce phase described in the subsection D. In the following we will focus on the model for the time spent during the `fork` operation; for the time spent during the `join` operation the same arguments apply as well. Now:

- at time $t = 0$ we have just one active thread (i.e. the `main` thread);
- after T_f seconds (namely at time $t = T_f$) the `main` thread manages to activate another thread, so that we have now two active threads;
- after other T_f seconds (namely at time $t = 2 \cdot T_f$), the two active threads have activated other two threads, so that we have 4 active threads now;
- at time $t = 3 \cdot T_f$ other three threads have been activated[2], so that we have now a total of 7 active threads.

Generalizing, one can say that at time $t = n \cdot T_f$, we have (possibly up to) F_{n+1} threads that are activated with this procedure, were F_n is the n-th number of the *Fibonacci series*. The total number of active threads at time $t = n \cdot T_f$ is then bounded to:

$$\sum_{i=1}^{n} F_{n+1} = F_{n+3} - 1 \qquad (3)$$

as one can also verify in table 1, that reports the number of active threads over time. The table can be read as follows. The first row reports the number of active threads at time $t = 0$, and each following row corresponds to a forward advance in time of T_f seconds, so that the n-th row of the table reports the number of active threads at time $t = n \cdot T_f$. Each row is organized in the following way: the first colomn reports the time passed since the instant $t = 0$ (i.e. the time passed since the application has been launched); the second colomn reports the number of new threads that have been created at that time instant and that, hence, have still to perform (up to) two fork operations; in the third column we have reported the number of threads activated at the previous iteration (they have already forked one thread, and so have still eventually to fork another one); in the fourth colomn we have reported the number of threads that have already forked both their children threads, and are just waiting for the termination of the `fork` procedure; finally in the fifth (i.e. the right-most) column we have reported the total number of active threads.

Now, since F_n can be approximated as:

$$F_n \approx \frac{\phi^n}{\sqrt{5}} \qquad (4)$$

[2]Notice that the `main` thread has already forked two threads, and hence it does not perform any other `fork` operation.

time	two forks	one fork	done	TOTAL
0	1	0	0	1
T_f	1	1	0	2
$2 \cdot T_f$	2	1	1	4
$3 \cdot T_f$	3	2	2	7
$4 \cdot T_f$	5	2	4	12
$5 \cdot T_f$	8	5	7	20
...	...	...	...	...
$n \cdot T_f$	F_{n+1}	F_n	$F_{n+1} - 1$	$F_{n+3} - 1$

Table 1. Number of active threads at time intervals multiple of T_f. The central columns report the number of threads that have still to perform two, one and zero fork operations respectively.

where the *golden ratio* ϕ is:

$$\phi = \frac{1 + \sqrt{5}}{2} = 1.6180339887.... \qquad (5)$$

We can finally state that the number of *active threads* at time $t = n \cdot T_f$ can be approximated as:

$$\frac{\phi^{n+3}}{\sqrt{5}} - 1 \qquad (6)$$

From equation 6, with simple algebra calculations, we can finally state that the time needed to activate nt threads is:

$$\Delta t = T_f \cdot (\lceil \log_\phi((nt + 1)\sqrt{5}) \rceil - 3) \qquad (7)$$

Similarly, we can conclude that the time needed to execute the `join` procedure is:

$$\Delta t = T_j \cdot (\lceil \log_\phi((nt + 1)\sqrt{5}) \rceil - 3) \qquad (8)$$

Finally, figure 2 reports a graph of the fork time in equation 7 as a function of the number of threads nt. One can notice that $\Delta t = 8 \cdot T_f$ is sufficient to `fork` more than 80 threads. The time taken by this forking scheme is *asymtotically optimum*; indeed, the fastest forking scheme one can design is able to reach in t seconds a minimum of $2^{\lfloor t/T_f \rfloor}$ active threads, and hence needs again a time $t = \Theta(\log nt)$ to perform the `fork` operations.

F. some pseudo-code

An high level pseudo-code of the operations executed by each thread is provided by the algorithmic procedure below.

G. ideal completion time

Gathering the results of the previous subsections, we can finally propose a formula that predicts the completion time of the algorithm. We notice that it is not possible to have overlapping between the `fork` phase and the processing phases, since the completion of even the first algorithmic iteration of any thread requires that all the other threads have already been activated. Considering that the time for the `global reduce` sub-phase is negligible ($T_{gred} \approx 0$) even when considering tens of threads, we can state that the *ideal completion time* T_c of our application, with

Algorithm 1. the pseudo-code for a generic thread

```
1: procedure MAXIMUM THROUGH PSO($D, func, n_p, n_{iter}, nt$)
2:     if the thread is mapped to an internal node then
3:         fork its child thread(s)
4:     Initialize $(n_p/nt)$ particles
5:     for $i = 1$ to $n_{iter}$ do
6:         perform phase 1
7:         perform the local reduce
8:         wait until all child threads set isDone to True
9:         set isDone flags of child threads to False
10:        set isDone to True in parent node
11:        if the thread is the root thread then
12:            perform the last reduce and notify the others
13:        else
14:            wait for the completion of the global reduce
15:        perform phase 3
16:    if the thread is mapped to an internal node then
17:        join its child thread(s)
```

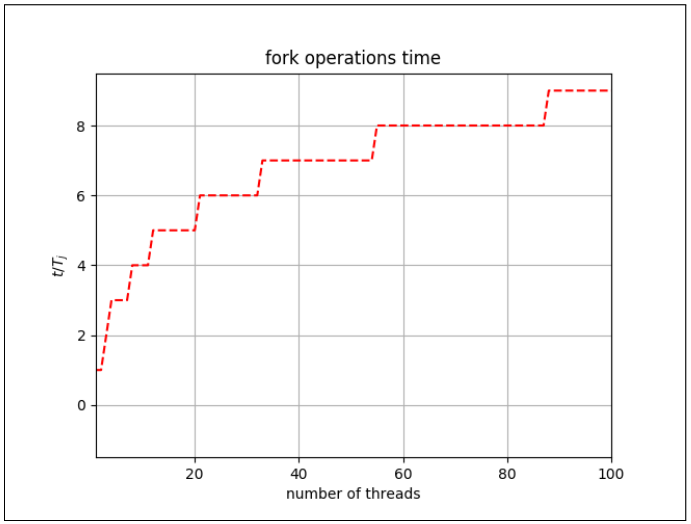


Fig. 2. Time needed to fork threads.

a parallelism degree equal to nt, should be well-approximated by:

$$\begin{aligned} T_c(nt) = {} & (T_f + T_j) \cdot (\lceil \log_\phi((nt+1)\sqrt{5}) \rceil - 3) \\ & + T_{init} \\ & + n_{iter} \cdot (T_{m1} + T_{lred} + T_{m3}) \end{aligned} \tag{9}$$

When n_{iter} is sufficiently high, we can neglect the `fork` and `join` procedures time and simply state that :

$$T_c(nt) \approx n_{iter} \cdot (T_{m1} + T_{lred} + T_{m3}) \tag{10}$$

H. ideal parallelism degree

Let T_{m1}, T_{lred} and T_{m3} be $T_{m1} = \alpha \cdot (n_p/nt)$, $T_{lred} = \beta \cdot (n_p/nt)$ and $T_{m3} = \gamma \cdot (n_p/nt)$. Let also be

$$k_1 = \frac{T_f + T_i}{\ln \phi} \tag{11}$$

and

$$k_2 = n_p \cdot [\delta + n_{iter} \cdot (\alpha + \beta + \gamma)] \tag{12}$$

Let's denote then with k the ration $k = k_2/k_1$. Then the derivative of the ideal completion time w.r.t. the number of threads nt is:

$$\frac{d}{d(nt)} T_c(nt) = \frac{k_1}{nt+1} - \frac{k_2}{nt^2} \tag{13}$$

Imposing equation 13 to be equal to zero, we obtain the (ideal) *optimal parallelism degree* for our application:

$$nt_{opt} = \frac{1}{2}\left(k + \sqrt{k(k+4)}\right) \tag{14}$$

The actual values of parameter α is strictly dependent on the complexity of the function $func$ to be evaluated. In the performance evaluation analysis that we have conducted, we have tested the sequential version of the application and registered the value $\beta + \gamma \approx 200$ nanoseconds (notice that the actual value of α depends only on the choice of the function $func$ for which we compute the minumum). In the (interesting) case in which $func$ takes at least some hundreds of microseconds to be evaluated, we can neglect the cost for the local reduce and the update operations since we have $\alpha >> \beta, \gamma$. This last consideration hold for many practical cases.

I. Proposals for alternative implementations

speculative execution. In particular scenarios it can happen that it's difficult/unlikely to find many good candidate for the minimum $func$ input function in D. For example:

- If the application executes a huge number of iterations, one can imagine that – iteration after iteration – it becomes more and more difficult to find a better and better candidate for $gmin$;
- It is also possible that the *true* value for the minimum is found at some iteration: clearly from that iteration going on no other better candidate for the minimum of $func$ in D will be found.

In this cases *specultive execution* models can come at hand. In these models it is *not necessary to implement any rigid barrier* among threads to perform the PSO computation. Simply, every thread performs calculations independently one w.r.t. another, assuming that the global value of the minimum will not change from an iteration to another. When a thread/worker finds a better value for $gmin$, that worker notifies this new value to all the others. Notice that, as always happen in speculative execution models, the threads can be possibly required to *undo* some part of their calculation. This procedure, nevertheless, requires the maintanance in each map worker/thread of a representation of the previous states of the computation during the previous iterations, and can possibly lead to memory explosion.

We have chosen to avoid these kind of solutions since in the majority of practical cases the $gmin$ extiomate is updated very frequently, so that it seems not justified to augment the complexity of the parallel pattern, and consume extra memory (and computation!) resources for keeping track of the previous states of the computation in each thread.

dynamic partiotioner. In our analysis we have always supposed that each thread spends the same time in the same portions of code, so that we can consider our map-reduce perfectly balanced. Nevertheless, if there are other active processes on the execution machine, it can be the case that some threads have to share their core with some other process. This can lead to a situation in which some workers appear to be "slower" w.r.t. the other. This can suggest the usage os some kind of *feedback controller mechanism* that dynamically re-assigns after every global reduce sub-phase the particles to be computed by each thread in order to have a better balance for the map-reduce[3].

In our project we have not considered this solution since we always guarantee: (a) that the partitions need always the same time to be processed, and (b) that no other process is running on the target machine.

pool evolution pattern. The minimum search performed by the PSO can be alternatively modelled by mean of the *pool evolution* pattern, which is also supported by `fastflow`[4].

3. THE FASTFLOW IMPLEMENTATION

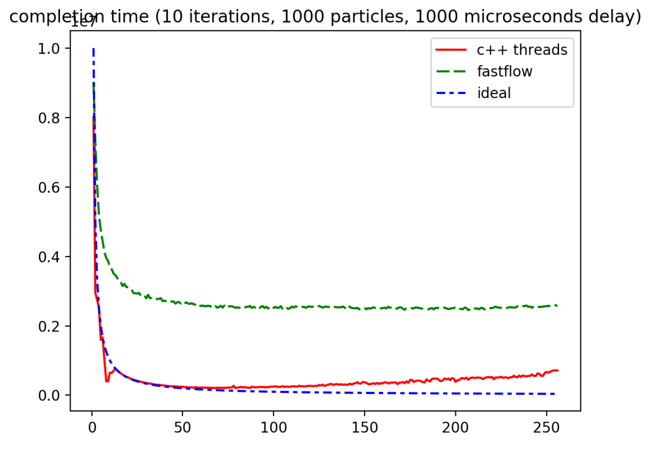
For the FastFlow [2] implementation, we have encapsulated the business logic code and the reduce function in a single `ParallelForReduce` object. The implementation has required just one `cpp` file and few lines of code[5].

4. PERFORMANCE EVALUATION

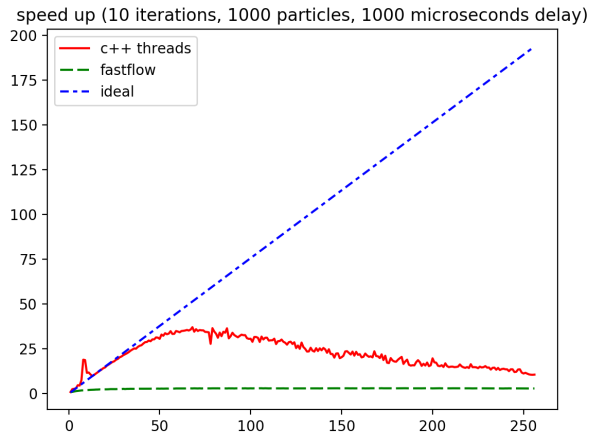
We have tested the time performances of our parallel implementation (both the `c++ threads` and the `fastflow` ones). More specifically, we have launched the `c++ threads` and the `fastflow` versions of our code on a `Xeon Phi KNL` machine, using parallelism degree values in the range 1 to 256. We have then measured the resulting completion times, compared it against the sequential version, and derived the corresponding speed-up and efficiency. In order to gurantee acuuracy in the sampled data, we have runned the experiment *9 times*, and each time considered as an accurate measure the *median* value within the set of the 9 sampled values. As input we have considered a problem with 10 iterations, 1k particles, and $T_f = 1$ millisecond. FIGURES 4 (A),(B),(C) report – respectively – the plots for: the completion time, the speed-up, and the efficiency for both our parallel solutions. It is interesting to notice that the performances that we achieved are similar to the ones achieved by the authors of [3], where a similar experiment executed over an input with the same dimension resulted in a completion time of 426 milliseconds, leveraging a parallelism degree equal to 48.

REFERENCES

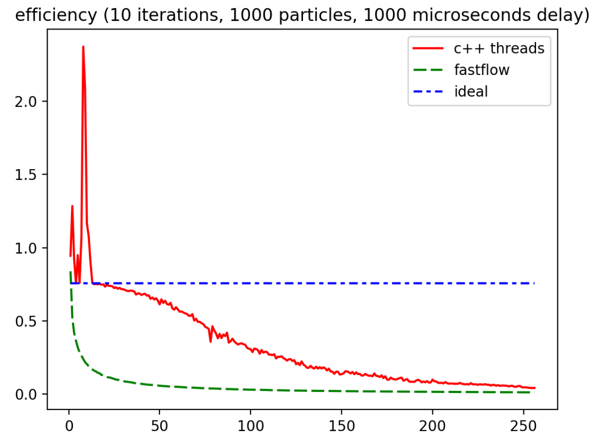
1. M. Danelutto, M. Torquati, and P. Kilpatrick, "A green perspective on structured parallel programming," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing,* (2015), pp. 430–437.
2. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: High-Level and Efficient Streaming on Multicore* (John Wiley & Sons, Ltd, 2017), chap. 13, pp. 261–280.
3. M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, Int. J. Parallel Program. **44**, 531–551 (2016).



(a) completion time plot



(b) speed-up plot



(c) efficiency plot

Fig. 3. Time performances of our code for different parallelism degree.

[3]This can be regarded as a *job stealing* procedure.

[4]Indeed the minimum search code is available as an example code in the fastflow documentation. You can find it at the link: http://calvados.di.unipi.it/storage/refman/doc/html/funcmin_8cpp-example.html#_a3

[5]We are excluding from the counting: (a) the mechanisms to measure times for performance evaluations, and (b) the configuration file

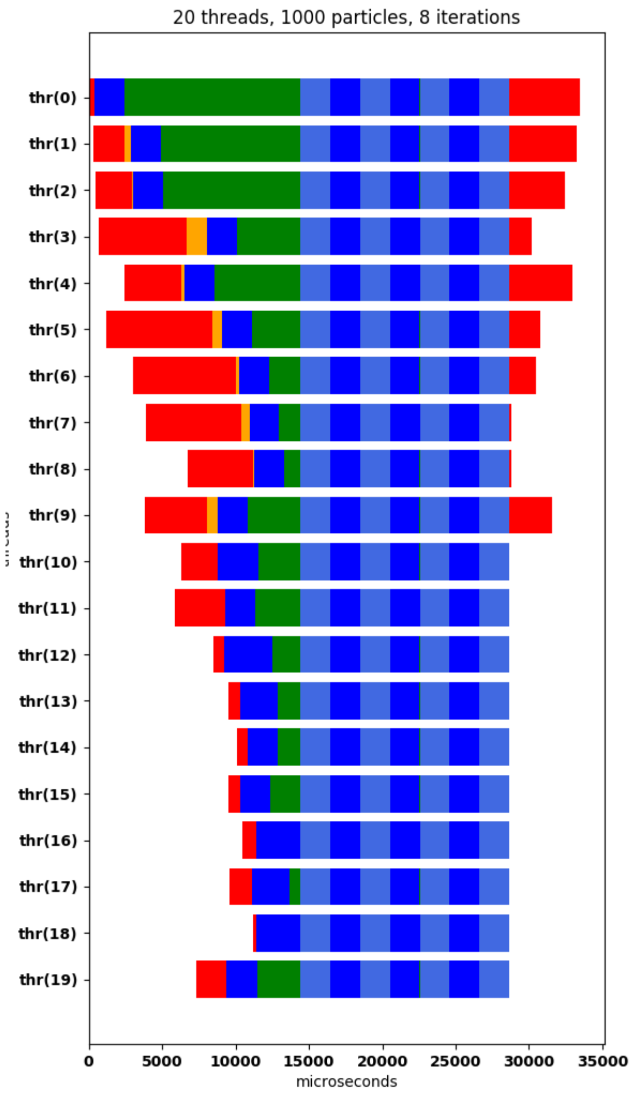


Fig. 4. A timeline of the execution of our `c++ threads` implementation. The red bars corresponds to `fork` and `join` operations; the yellow bars corresponds to data `initialization`; dark and light blue bars correspond to local computations; green bars correspond to synchronization operations. The plot refers to an execution on the `Xeon Phi KNL` machine with: 8 iterations, 1k particles, $T_{func} = 40\mu s$ and parallelism degree 20.