Int J Parallel Prog (2016) 44:1173–1199
DOI 10.1007/s10766-015-0368-3

Parallel Implementations of the Cooperative Particle Swarm Optimization on Many-core and Multi-core Architectures

Nadia Nedjah[1] · Rogério de M. Calazan[1] · Luiza de Macedo Mourelle[2] · Chao Wang[3]

Received: 5 December 2014 / Accepted: 11 May 2015 / Published online: 5 June 2015

Abstract Particle swarm optimization (PSO) is an evolutionary heuristics-based method used for continuous function optimization. PSO is stochastic yet very robust. Nevertheless, real-world optimizations require a high computational effort to converge to a good solution for the problem. In general, parallel PSO implementations provide good performance. However, this depends heavily on the parallelization strategy used as well as the number and characteristics of the exploited processors. In this paper, we propose a cooperative strategy, which consists of subdividing an optimization problem into many simpler sub-problems. Each of these focuses on a distinct subset of the problem dimensions. The optimization work for all the selected sub-problems is done in parallel. We map the work onto four different parallel high-performance multiprocessors, which are based on multi- and many-core architectures. The performance of the strategy thus implemented is evaluated for four well known benchmark functions with

✉ Nadia Nedjah
nadia@pq.cnpq.br

Rogério de M. Calazan
rogerio@eng.uerj.br

Luiza de Macedo Mourelle
ldmm@eng.uerj.br

Chao Wang
cswang@ustc.edu.cn

[1] Department of Electronics Engineering and Telecommunications, Faculty of Engineering, State University of Rio de Janeiro, Rio de Janeiro, Brazil

[2] Department of Systems Engineering and Computation, Faculty of Engineering, State University of Rio de Janeiro, Rio de Janeiro, Brazil

[3] Embedded System Lab, School of Computer Science, University of Science and Technology of China, Hefei, China

high-dimension and different complexity. The obtained speedups are compared to that yielded by a serial PSO implementation.

Keywords PSO · CPPSO · Parallel algorithm · OpenMP · MPI · CUDA · GPU

1 Introduction

Parallel processing is a strategy used in computing to solve complex computational problems faster by splitting them into sub-tasks that will be allocated on multiple processors to run concurrently [17]. These processors communicate so there is synchronization or information exchange. The methodology for designing parallel algorithms comprises four distinct stages [14]: partitioning, communication, aggregation and mapping.

During the partitioning step, the computation is decomposed into sub-tasks that are executed in parallel. This decomposition may be either of fine or coarse granularity. The decomposition may be functional or domain based, depending on whether it is computation or data driven. Subtasks generated by partitioning usually need to exchange information. The flow of data transfer between the parallel tasks is done via communication. This communication can either local or global or a combination of both. Moreover, the communication between tasks can be either synchronous or asynchronous. During the aggregation stage, the decision steps taken during partitioning and communication may be reviewed so as to improve the overall performance. At this stage, the granularity degree may be increased so as to decrease the communication between tasks. Finally, the mapping stage consists of identifying the resource where tasks will be executed. In general, independent tasks should be allocated on different processors so as they can be executed concurrently while tasks that communicate frequently should be allocated on the same processor to mitigate the communication overhead.

A multi-core processor is typically a single computing machine composed of up of 2–8 independent processor cores in the same silicon circuit die connected through an on-chip bus. All included cores communicate with each other, with the memory and I/O peripherals via this internal bus. The multi-core processor executes multiple threads concurrently, typically to boost performance in compute intensive processes. However as more cores are added to the processor, the information traffic that flows along the on-chip bus, increases as all the data must travel through the same path. This limits the benefits of a multi-core processor.

A many-core also known as a massively multi-core processors are simply multi-core processors with an especially high number of cores, ranging from 10 to 100 cores. Of course, in this multi-processors communication infrastructure between the included cores must be upgraded to a sophisticated interconnection network to cope with the amount of data exchanged by the cores. Furthermore, cores are coupled with private and local memories to reduce data traffic on the main interconnection network.

Particle swarm optimization (PSO) was introduced by Kennedy and Eberhart [16] and is based on collective behavior, social influence and learning. It imitates the social behavior of a flock of birds. If one element of the group discovers a way where there is easy to find food, the other group members tend instantly, to follow same way. Many successful applications of PSO have been reported, in which this algorithm has shown many advantages over other algorithms based on swarm intelligence, mainly due to its robustness, efficiency and simplicity. Moreover, it usually requires less computational effort when compared to other stochastic algorithms [12,20].

The PSO algorithm maintains a swarm of particles, each of which represents a potential solution. In analogy with evolutionary computation, a *swarm* can be identified as the population, while a *particle* with an individual. In general terms, the particle flows through a multidimensional search space, and the corresponding position is adjusted according to its own experience and that of its neighbors [12].

In recent years, the use of graphics processing units, or GPUs, has been exploited for some general purpose computing applications. GPUs have shown great advantages when the application requires intensive yet independent computations. Despite the fact that GPU-based architectures require an additional CPU time to assign tasks to the available units, the speedups obtained by the former in relation to simple CPU architectures are usually higher for applications wherein the processing effort is much more focused on floating-point and matrix handling operations.

The major benefit of a PSO implementation using a GPU-based architecture is the possibility of reducing the execution time. This is actually quite probable since the fitness evaluation and the update processes of the particles are somehow independent and so can be parallelized and executed simultaneously via different threads.

Several works show that PSO implementation on dedicated hardware [5,7] and GPUs provide a better performance than CPU-based implementations [2,25,28,31]. However, these implementations take advantage of the parallelization only within the loop of particles processing and also the stopping condition used in those works is always based on the total number of iterations. It is worth noting that the number of iterations required to reach a good solution is problem-dependent. Few iterations may terminate the search prematurely and large number of iterations has the consequence of unnecessary added computational effort. In contrast, the purpose of this paper is to implement a massively parallel algorithm of PSO in GPU and compare with a serial implementation using the stopping condition that depends on the acceptability of the solution that has been found so far. We investigate the impact of fine-grained [4] in contrast with coarse-grained [6] parallelism in high-dimension problems and also analyze the efficiency of the process to reach the solution. In the latter, we subdivide the search space into a uniform grid of search sub-spaces.

In order to take full advantage of the massively parallel of multi-cores architectures, in this paper, we explore another strategy for coarse-grained parallelism, which consists of subdividing the original optimization problem into many simpler sub-problems. Each of these sub-problems focuses on a distinct subset of the original problem dimensions. The optimization work for all the selected sub-problems is done in parallel. Note that we handle as many swarms of particles as the formed sub-problems. We implement the strategy using OpenMP, MPI, and OpenMP/MPI. The implementations are executed cluster of multi-core processors. Note that the implementation of this strategy on many-core architectures (GPUs) is detailed in [3], and its performance is used in the comparison presented in this paper.

As such, the CUDA implementation maps a sub-swarm to a block and a particle to a thread. The OpenMP implementation maps a sub-swarm to Hereafter, the strategy proposed is termed *Cooperative Parallel* PSO—CPPSO. This approach should favor optimization problems with very high dimensionality, in the order of 256 and more.

This paper is organized as follows: first, in Sect. 2, we comment on some existing related works. After that, in Sect. 3, we sketch briefly the PSO process and the sequential version of the algorithm; then, in Sect. 4, we describe the CPPSO strategy. Thereafter, in Sect. 5, we present some implementation issues for the implementation of the CPPSO on a shared memory multi-core processor architecture using OpenMP; subsequently, in Sect. 6, we do the same for the implementation of CPPSO on a distributed memory multi-core processor using MPI; after that, in Sect. 7, we report on the implementation of CPPSO on a cluster of multi-core shared memory processor combining the use of OpenMP and MPI; then, in Sect. 8, we present the implementation of CPPSO on a many-core GPU using CUDA. There follows, in Sect. 9, an analysis of the obtained results; finally, in Sect. 10, we draw some conclusions and point out some directions for future work.

2 Related Works

In [28], the authors report on an implementation of the particle swarm optimization algorithm in CUDA. The algorithm is tested on well-known benchmark optimization problems and the computing time is compared with the same algorithm implemented in C and Matlab. In [31], the authors present an approach to run standard particle swarm optimization on GPU. The results are compared with CPU and third party GPU implementations. The implementation on GPU shows special speed advantages on large swarm population applications and high-dimension problems. The ring topology was used.

An alternative GPGPU PSO implementation is presented in [2]. The authors show performance analysis for high-dimension problems. Unlike the work presented here, only the fitness evaluation is executed by the GPU, while the PSO main steps are performed in the host CPU. In [30], the authors present the so-called SPSO model that combines CPU and GPU codes on the particles best solution updates. The authors carried out the analysis for a ring topology. This topology is also used in our implementation for reasons that will be explained in the following section.

In [25], the authors implement the comprehensive learning particle swarm optimizer (CLPSO) with application to economic dispatch problem in the GPU. The results demonstrate that GPUs can be applied with success to speed up computationally intensive problems in electric energy systems. Unlike this paper, we will investigate the impact of fine-grained parallelism in high-dimensionality problems.

Unlike these existing implementations, our goal is to increase the convergence using the ring global topology, while maximizing the execution time within the GPU. We expect an improvement of the overall performance due to the work division and mapping of sub-swarms into blocks and particles into threads.

3 Particle Swarm Optimization

The main steps of the PSO algorithm are described in Algorithm 1. Note that, in this specification, the computations are planned to be executed sequentially. In this algorithm, each particle has a *velocity* and an *adaptive direction* [16] that determine its next movement within the search space. The particle is also endowed with a memory that makes it able to remember the best previous position it passed by. In Algorithm 1, as well as in the remainder of this paper, we denote by $Pbest[i]$ the best fitness particle i has achieved so far and $Pbestx[i]$ the coordinates of the position that yielded it. In the same way, we denote $Sbest[i]$ the swarm best fitness particle i and its neighbors have achieved so far and $Sbestx[i]$ the coordinates of the corresponding position in the search space.

Algorithm 1 Local Best PSO

```
for $i = 1$ $to$ $\eta$ do
  initialize the information of particle $i$
  initialize the position and velocity of particle $i$
end for
repeat
  for $i = 1 \rightarrow \eta$ do
    compute $fitness[i]$
    if $fitness[i] \leq Pbest[i]$ then
      update $Pbestx[i]$ using position of particle $i$
    end if
    if $Pbest[i] \leq Sbest[i]$ then
      update $Sbestx[i]$ using the $Pbestx[i]$
    end if
  end for
  for $i = 1$ $to$ $\eta$ do
    update velocity and position of particle $i$
  end for
until stopping criterion
return $Sbest[i]$ and $Sbestx[i]$
```

The PSO uses a set of particles, where each is a potential solution to the problem, having position coordinates in a space of d-dimensional search. Thus, each particle has a position vector with the corresponding fitness, a vector keeping the coordinates of the best position reached by the particle so far and one field to fitness and another to better fitness. To update the position of each particle i of the PSO algorithm is a set velocity for each dimension j of this position. The velocity is the element that promotes the ability of movement of the particles., and can be calculated according to Eqs. 1 and 2:

$$v_{i,j}^{(t+1)} = \omega v_{i,j}^{(t)} + \phi_1 r_{1,j}^{(t)} \left(y_{i,j} - x_{i,j}^{(t)}\right) + \phi_2 r_{2,j}^{(t)} \left(z_{i,j} - x_{i,j}^{(t)}\right) \quad (1)$$

$$x_{i,j}^{(t+1)} = v_{i,j}^{(t+1)} + x_{i,j}^{(t)} \quad (2)$$

wherein $x_{i,j}$, $y_{i,j}$ and $z_{i,j}$ represent the coordinate regarding dimension j of the actual, particle best and swarm best positions, respectively. Thus, we have $y_{i,j} = Pbestx[i][j]$, $z_{i,j} = SBest[i][j]$.

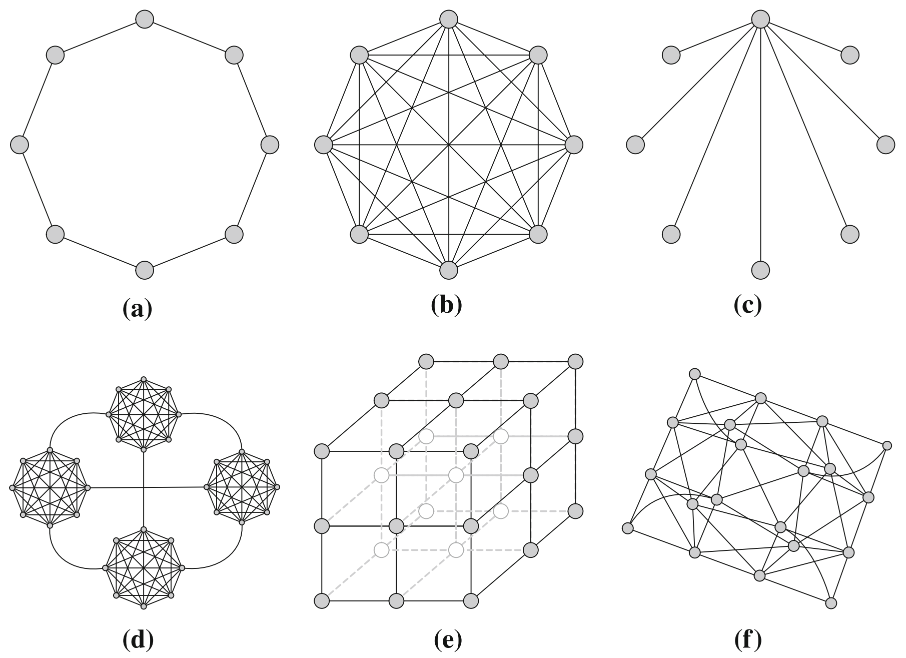

Fig. 1 PSO neighborhood common topologies. **a** Ring. **b** Star. **c** Circle. **d** Cluster. **e** Von Neumann. **f** Pyramid

The selection of the swarm best particle depends on the neighborhood topology of the PSO. Some of possible topologies are illustrated in Fig. 1.

In the implemented variation of the PSO algorithm, a ring topology is used as a social network topology where smaller neighborhoods are defined for each particle. The social component, denominated *local best*, reflects the information exchanged within the neighborhood of the particle [12]. The Local Best PSO is less susceptible to being trapped into a local minimum and also the ring topology used improves performance. The velocity is the component that promotes the capacity of particle locomotion and can be computed as described in (1) [12,16], wherein ω is called *inertia weight*, r_1 and r_2 are random numbers in [0,1], ϕ_1 and ϕ_2 are positive constants, y_{ij} is the particle best position *Pbest* found by particle i so far, regarding dimension j, and l_{ij} is the local best position *Lbest* found by all the particles in the neighborhood of particle i, regarding dimension j. The position of each particle is updated as described in (2). Note that $x_{i,j}^{(t+1)}$ is the current position and $x_{i,j}^{(t)}$ is the previous position.

The velocity component drives the optimization process, reflecting both the experience of the particle and the exchange of information between the particles. The particle's experimental knowledge is referred to as the *cognitive behavior*, which is proportional to the distance between the particle and its best position found, regarding its first iteration [12]. The maximum velocity $v_{j,max}$ is defined for each dimension j of the search space. It can be expressed as a percentage of the search space, according to (3), wherein $x_{j,max}$ and $x_{j,min}$ are the maximum and minimum limits of the search space explored, regarding dimension j, respectively and $\alpha \in [0, 1]$.

$$v_{j,max} = \alpha(x_{j,max} - x_{j,min}) \tag{3}$$

4 The Cooperative Parallel PSO

Many improved version of PSO were proposed used to yields efficient solutions for hard problems [9,10]. The CPPSO (*Cooperative Parallel* PSO) algorithm is based on CPSO-S$_\sigma$ (*Cooperative Split* PSO) developed by Van den Bergh in [1]. The main underlying idea of this approach is to view the optimization of a problem that involves δ dimensions as σ sub-problems of δ/σ dimensions each. Each sub-problem is optimized by a group of particles that form a sub-swarm of the overall acting swarm. A sub-swarm is responsible of optimizing the original problem only with respect to the corresponding δ/σ dimensions. Within a given sub-swarms, the computation done by the particles is performed in parallel. The number of sub-problems σ is called the *partition factor*. This factor is a natural number that may vary from 1 to δ. A partition factor of 1 implies that there are δ sub-problems wherein each one is taking care of a single dimension. Note that a partition factor that coincides with the number of dimensions yields the original non-cooperative parallel PSO.

A drawback of the CPPSO strategy lies in computing the value of the objective function, a δ-dimension input is required. As each sub-swarm disposes only of a part of the total number of dimensions, it is not possible to perform whole computation, and thus the particles of the sub-swarm cannot proceed with the optimization process. In order to remedy to this situation, a so-called *context vector* is used to provide the sub-swarm with the results obtained by the other cooperating sub-swarms via some kind of communication.

It is clear that the partition factor must be chosen carefully so as to maximize the cooperative nature, yielding an acceleration of the optimization process, but not to cause too much fragmentation of the original problem, which generates too much communication to compose the original objective function. This said, however, in this work, we do not analyze the impact of the partition factor on the performance. Instead, we chose some specific factors that are suitable in the case of the hardware resources in the GPU used. Further analysis will be done and published when the corresponding results are conclusive.

5 CPPSO on Shared Memory Multi-core Processors

A shared memory multi-core processor (SMMCP) is a parallel multiprocessor in which all cores share a single addressable physical space [26], as illustrated in Fig. 2. The cores communicate through shared variables that are stored in the shared memory, that accessible to all cores via load and store instructions. Also, independent computational work can be performed using a virtual address space.

5.1 OpenMP Concepts

The OpenMP [8] is an API that is designed to enable and facilitate programming in parallel environment using a shared memory. It has three basic components: com-

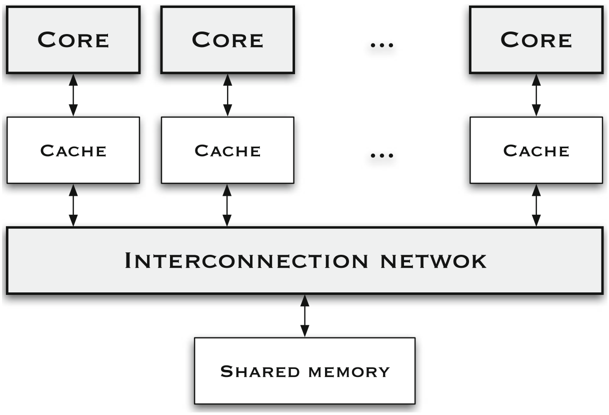

Fig. 2 Typical SMMCP architecture

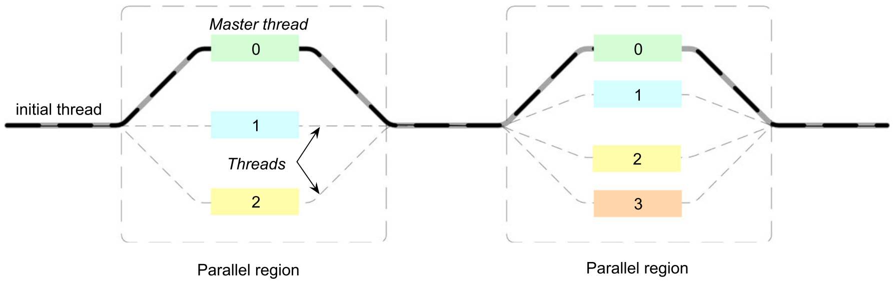

Fig. 3 Fork–Join model as supported by OpenMP

pilation directives, runtime libraries and environment variables. This API provides necessary means for the programmer to create a set of parallel threads, specify how the data will be shared between threads, declare shared and private variables and synchronize threads. The thread is an implementation structure that is able to run a stream of independent instructions when scheduled by the operating system [8]. If multiple threads cooperate to run a program, these will share the same address space of the process. However, each thread has its own register file, program counter and instruction register.

The OpenMP uses the Fork–Join programming model [11], shown in Fig. 3. In this approach, the program starts in a similar way to a serial program, identified as the *initial thread*. When this thread meets a parallel feature of OpenMP, then a group of independent threads is created and the initial thread will be identified as the *Master thread*. The group of parallel threads start cooperating to execute the planned code until the end of the parallel region. In parallel region, the threads may share information or use the data available in their private memory. When the parallel region execution is completed, the parallel threads are synchronized and finalized. Hence, only the master thread remains active.

The OpenMP features provide a simple way to inform the compiler which instructions should be executed in parallel and how to distribute the computational work among the planned threads. Each OpenMP directive begins with *#pragma omp*, following the convention of the C/C++ compiler. A parallel region always starts with the directive *parallel*. As it can be seen in Algorithm 2, line 6 creates a parallel region, the variables a and b are private and the variable c is shared by all threads.

In order to set the number of threads (nt) that will run a parallel region function *omp_set_num_threads*(nt) is used (lines 3 and 4). Within the parallel region, function *omp_get_thread_num*() generates an identifier for each thread (tid). The initial thread is identified by $tid = 0$, ranking it as the master thread. The other threads are numbered sequentially starting from 1. The data parallelism is implemented by the use of feature *for* (line 9). This constructor specifies that the iterations will be distributed and executed in parallel by the group of threads. The loop iterations are divided as instructed by the directive $schedule(static)$. This type of scheduling divides the work continuously in equal parts based on to the number of threads assigned to the parallel region. The static schedule imposes a low overhead when compared to different types of schedules provided by OpenMP. At the end of the loop, the threads are synchronized before continuing the program execution, unless the clause *nowait* is specified.

Algorithm 2 Creating a parallel region in OpenMP

```
1: Begin
2:     /* sequential region */
3:     Let $nt$ be the number of threads
4:     omp_set_num_threads($nt$)
5:     int $a$,$b$,$c$
6:     #pragma omp parallel private($a$,$b$) shared($c$)
7:     Begin
8:         /* parallel region */
9:         #pragma omp for schedule(static)
10:         for $i = 0 \rightarrow p$ do
11:             /* shared computational work */
12:         End for
13:     End
14:     /* sequential region */
15: End
```

5.2 OpenMP-Oriented CPPSO Implementation

Algorithm 3 presents an overview of the implementation of CPPSO using OpenMP. The original problem with δ dimensions is divided into σ sub-swarms dedicated to a subset of dimensions $\epsilon = \delta/\sigma$. Similarly, the original swarm, composed of η particles is divided into σ sub-swarms, each of which with $\ell = \eta/\sigma$ particles, wherein each sub-swarms optimizes one sub-problem. In this approach, a sub-swarm is mapped into a thread. Vector $Best$ of size σ and context array $Bestx$ of size δ are both stored in shared memory.

After the beginning of the parallel region, the threads initialize the velocity and coordinates of the particles of their respective sub-swarms. Then, each thread initializes the context of vector, at the position indicated by the thread index, using the coordinated of particle 0 of the respective sub-swarm. After this procedure, the threads

Algorithm 3 CPPSO in OpenMP

```
1: Let $\sigma$ = number of threads (or sub-swarms)
2: omp_set_num_threads($\sigma$)
3: #pragma omp parallel
4: Begin parallel region
5: for $i := 0 \rightarrow \ell - 1$ do
6:   initialize particles for sub-swarm $\sigma$
7:   if $i = 0$ then
8:     $Bestx[tid * \epsilon + j] := x_{ij}$
9:   end if
10: end for
11: synchronize threads
12: repeat
13:   for $j = 0 \rightarrow \delta - 1$ do
14:     $Tbestx[j]$:=$Bestx[j]$
15:   end for
16:   for $i := 0 \rightarrow \ell - 1$ do
17:     for $j := 0 \rightarrow \epsilon - 1$ do
18:       $Tbestx[tid * \epsilon + j]$:=$x_{ij}$
19:     end for
20:     compute $fitness_i$;
21:     update $Pbest_i$ and $Lbest_i$
22:     if ($Lbest_i < Best[tid]$) then
23:       update $Best[tid]$
24:       for $j := 0 \rightarrow \epsilon - 1$ do
25:         $Bestx[tid * \epsilon + j]$:=$Lbestx_{ij}$
26:       end for
27:     end if
28:   end for
29:   for $i := 0 \rightarrow \ell - 1$ do
30:     update $Lbest_i$;
31:     update $v_{ij}$ and $x_{ij}$
32:   end for
33:   synchronize threads
34:   if $tid = 0$ then
35:     find the minimum in $Best[tid]$
36:   end if
37: until stopping condition
38: End parallel region
39: return $Best[0]$ and $Bestx[i]$
```

are synchronized in order to prevent copying of uninitialized context vector for the local memory of the thread.

Once synchronized, the threads perform the copy of the entire context vector to the respective local memory $Tbestx$ (line 12). Before performing the calculation of fitness, the thread replaces, in the context of vector, at the position corresponding to its sub-swarm, stored in local memory, the coordinates of particle i (lines 14–16). The other elements of the context vector are kept unchanged. Thus, the computation of fitness value can be performed using the δ dimensions, as well as updating $Pbest$ and $Lbest$. The first update of $Lbest$ (line 19) refers to the value of $Pbest$ the particle itself. Then, the vector $Best$ is updated with the lowest value of $Lbest$ obtained by the particles of the sub-swarm. After that, the context vector is updated at position corresponding to the thread, with the coordinates of the selected $Lbest$.

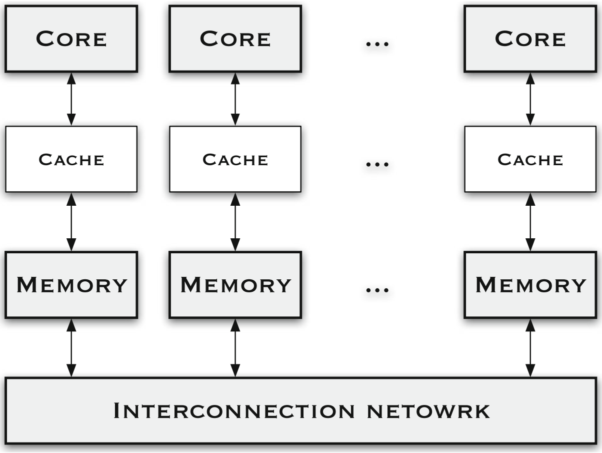

Fig. 4 Typical DMMCP architecture

Here we sue the ring topology. So, the neighbors of particle i are particles $(i + 1) \bmod \eta$ and $(i - 1) \bmod \eta$. Thus, *Lbest* is obtained according to this neighborhood setting. This exchange of information about the best location happens only within each sub-swarm. There is no exchange of information between sub-swarms. Then, the particle velocity and position are updated. At this point, the threads are synchronized and the master thread performs a sequential search in vector *Best* in order to obtain the lowest value and store it in the first position thereof. The other threads will access this position when the stop condition needs to be verified.

6 CPPSO on Distributed Memory Multi-core Processors

In distributed memory multi-core processors (DMMCP), communication is done via message exchange [26]. Moreover, each processor has its own private memory. The processors are interconnected via a high-speed network used for message exchange. Figure 4 illustrates the organization of a this type of multiprocessor. It is possible to note that, unlike the SMP multiprocessors, the interconnection network is not between the cache and memory, but instead it is inserted between the memories associated with each core processor. In this environment, the programs make use of libraries of message passing, which enable communication between processors. Note that a core cannot access the memory of another processor using the load and store instructions.

6.1 MPI Concepts

The Message passing interface (MPI) [29] is a specification for a standard messaging library, which was defined in the MPI Forum [18]. The MPICH [15] is a complete implementation of the MPI standard, designed to be portable and efficient in high-performance environments. When using MPI, the processes that run in parallel have private address spaces. Communication occurs when a process A sends some data stored in its address space the system buffer of a process B, that resides in another

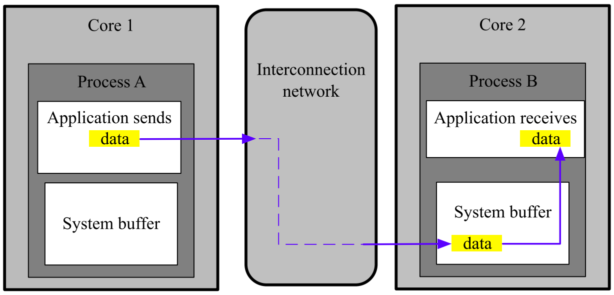

Fig. 5 Process communication via messages

processor, as shown in Fig. 5. The system buffer is a memory address reserved by the system to store incoming messages. This operation is cooperative and occurs only when process *A* performs a *send* operation and process *B* performs a *receive* operation.

The primitives of sending and receiving messages can be *blocking* or it no-blocking. In the case of blocking primitives, the processes involved in the operation would stop the program execution until the send/receive operation is completed. Otherwise, the program execution continues immediately after scheduling the communication. The MPI lists the communicating processes in groups, wherein the processes are identified by their classification within that group.

This classification within the group is called *ranking*. Thus, a process in MPI is identified by a group number and its rank within this group. As a process may belong to more than one group, it may have different ranks. A group uses a specific communicator that describes the universe of communication between processes. The MPI_COMM_WORLD is the default communicator. It includes all processes, defined by the user, in an MPI application. Algorithm 4 displays some of the main MPI primitives. The command at line 4 defines and starts the environment required to run the MPI. The statement at line 5 identifies the process within a group of parallel processes. The function at line 6 returns the number of processes within a group. From that point on, each process runs in parallel computing specified in the parallel region and the may processes cooperate with each other via message passing. The routine at line 9 ends MPI processes.

Algorithm 4 Examples of MPI primitives as implemented in MPICH

```
1: Begin
2:     /* Sequential region */
3:     int rank
4:     MPI_Init
5:     MPI_Comm_rank(MPI_COMM_WORLD,&rank)
6:     MPI_Comm_size(MPI_COMM_WORLD,&size)
7:     Begin
8:         /* Parallel region */
9:     MPI_Finalize()
10:     End
11:     /* Sequential region */
12: End
```

6.2 MPI-Oriented CPPSO Implementation

Algorithm 5 presents an overview of CPPSO implemented in MPICH. In this approach the sub-swarms is mapped as a process. The context vector $Bestx$ is stored in the local memory of each process. To exchange information of the context vector between processes, we used the vector $Kbestx$ and size ϵ, containing only the items of the context vector corresponding to the sub-swarm.

Algorithm 5 CPPSO in MPICH

```
1: Let ]$sigma$ = number of processes (sub-swarms)
2: MPI_Init()
3: srand(seed + $rank$)
4: initialize particles and $Kbestx$
5: send message to all processes with $Kbestx$
6: assemble $Bestx$ using $Kbestx$ of other processes
7: repeat
8:    $Tbestx$:=$Bestx$
9:    for $i := 0 \rightarrow l$ do
10:       $Bestx[rank * \epsilon + j] := x_{ij}$
11:       compute $fitness_i$
12:       update $Pbest_i$
13:       update $Lbest_i$
14:       if $Lbest_i < Best$ then
15:          $Best$:=$Lbest_i$
16:          $Kbestx[j] := Lbest_{ij}$
17:       end if
18:    end for
19:    for $i := 0 \rightarrow \ell$ do
20:       update $Lbest$
21:       update $x_{ij}$ and $v_{ij}$
22:    end for
23:    $Kbestx[j]$:= $bestx[rank * \epsilon + j]$
24:    send message to all processes with $Kbestx$
25:    assemble $Bestx$ using $Kbestx$ of other processes
26:    send message with $Best$ to the Master process
27:    if $rank = Master$ then
28:       if $Best \leq error$ then
29:          set exit flag
30:       end if
31:    end if
32:    $Master$ sends message with exit flag to all processes
33: until flag enabled
34: MPI_Finalize()
35: return $Best[Master]$ and $Bestx[Master]$
```

At the beginning of an MPI environment, processes initialize the information of the ℓ particles and $KBestx$ with the coordinates of the first particle of the corresponding sub-swarm. Then each process sends a message to all other processes with $KBestx$ yield for its sub-swarm. When a process receives $KBestx$ of other sub-swarms, it puts together the context of vector stored in $Kbestx$ at position $rank$ of the process that sent the message. Before performing the fitness computation, each process replaces,

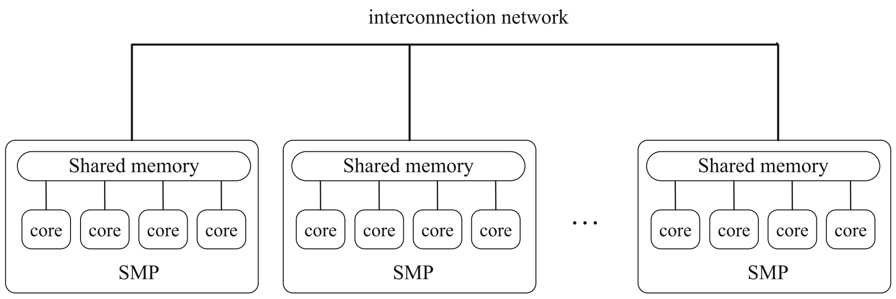

Fig. 6 SMMCP cluster typical architecture

in the context of vector position of its sub-swarm, which is stored in the local memory, the relative coordinates of particle i (line 8 of Algorithm 5) . The remaining entries of the context vector are kept unchanged. Hence, the of fitness computation is performed using the δ dimensions, as well as updating $Pbest$ and $Lbest$.

After updating $Lbest$, $Kbestx$ is updated using the coordinates of $Lbest$, which was selected as the best position obtained by sub-swarm so far. Then, $Lbest$ is updated (line 18) using the value obtained as the neighborhood setting $(i + 1) \bmod \eta$ and $(i - 1) \bmod \eta$. This exchange of information about the best location happens only within the sub-swarm. There is no exchange of information between the running sub-swarms. After that, the particle next velocity and position are computed.

Each process sends a message with $Kbestx$ to the remaining processes. In order to receive these messages, each process updates its context vector $Bestx$. Then, the master process receives the lowest fitness found together with the corresponding position by each sub-swarm. The master process then checks whether any of the received results satisfies the stopping condition. If so, the master process activates the exit flag and sends a message to all running processes. As soon as the processes receive the enabled flag, they suspend the optimization execution, while the master process returns the found best fitness and the corresponding position.

7 CPPSO on Clusters of Multi-core Shared Memory Processors

The combination of SMP + DMP multi-cores provides an hierarchical approach to exploit the inherent parallelism of an application and/or the processor architecture more efficiently [8]. Thus, the combination fits perfectly architectures based on a cluster multiprocessor because it reduces the number of message passing between cores and increases the performance of each core. This combination is illustrated in Fig. 6.

7.1 MPI + OpenMP Concepts

The hybrid programming style is most effective when the MPI processes run at a coarser granularity level, taking advantage of the data distribution and task scheduling, while the parallelization based on OpenMP is used in the multiprocessor architecture, providing the running processes with a finer granularity parallelization. This combi-

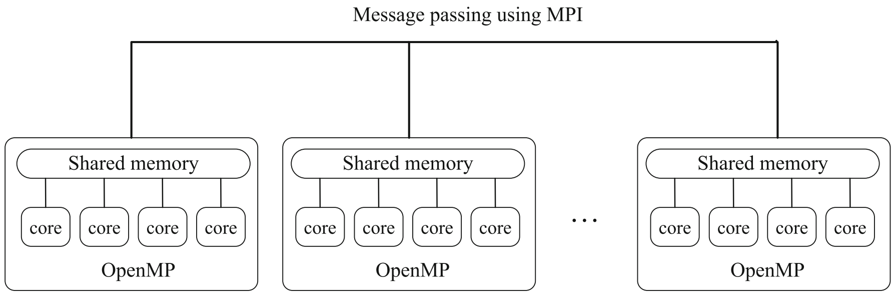

Fig. 7 OpenMP and MPI hybrid programming combination

nation is illustrated in Fig. 7. The hybrid coding also offers the advantage of reducing communication latency between MPI message, migrating some of the code to OpenMP. Additionally, this approach increases the scalability of the parallel implementation. In implementations proposed in this section, parallel regions OpenMP and MPI are created and maintained throughout the execution of optimization in order to reduce the overhead of generation at the beginning of each iteration.

7.2 MPI + OpenMP-Oriented Implementation for CPPSO

Algorithm 6 presents an overview of CPPSO implementation in OpenMP with MPICH (OpenMPI). The implementation is similar to that in MPICH. However, each MPI process that is mapped now as a sub-swarm creates an OpenMP parallel region. Thus, the loops that correspond to particle computations are divided and executed by parallel threads of the region.

The implementation is very similar to algorithm proposed for CPPSO in MPICH (see Algorithm 5). However, each MPI process, mapped as a sub-swarm, generates an OpenMP parallel region. Thus, the loops of the particles comprised in a sub-swarm are divided and executed by threads that compose the parallel region. In each particle loop, the constructor for is used along with the static scheduling clause. The context vector $Bestx$ of size δ is stored in the local memory of each process. For context vector information exchanging ($Bestx$) between processes, we used the vector $Kbestx$ of size ϵ, containing only the context of vector components related to sub-swarm. Only the Master thread performs the update $Kbestx$ (lines 16–21).

In order to parallelize the optimization process, we use vector $Best$, whose size is equal to the number of threads. Once $Lbest$ updated, each thread checks for the smallest result obtained by the corresponding particles and stores the found value in vector $Best$ at the position that is associated with its identifier tid. Then, the master thread performs a sequential search in vector $Best$ and stores the found result in the first position of this vector. The master thread Master accesses this position and sends a message with $Best$ to the master process. The latter then checks whether, among the obtained results, there is any that satisfies the stopping condition. If so, the master process enables the exit flag and sends a message to all running processes. As soon as these processes receive the enabled flag, they stop the execution and the master process returns the best result value together with the corresponding found position.

Algorithm 6 CPPSO in OpenMP with MPICH (OpenMPI)

```
1: let $\sigma$ = number of processes (sub-swarms)
2: MPI_Init()
3: #pragma omp parallel
4: Begin OpenMP parallel region
5: srand(seed + $rank$)
6: initialize particles and $Kbestx$
7: if $tid = Master$ then
8:   send message to all processes with $Kbestx$
9:   assemble $Bestx$ using $Kbestx$ of other processes
10: end if
11: repeat
12:   $Tbestx$:=$Bestx$
13:   #pragma omp for schedule(static)
14:   for $i := 0 \rightarrow \ell$ do
15:     $Bestx[rank * \epsilon + j] := x_{ij}$
16:     compute $fitness_i$
17:     update $Pbest_i$ and $Lbest_i$
18:   end for
19:   if $tid$ = Master then
20:     if $Lbest_i < Best$ then
21:       $Best := Lbest_i$
22:       $Kbestx[j] := Lbest_{ij}$
23:     end if
24:   end if
25:   #pragma omp for schedule(static)
26:   for $i := 0 \rightarrow \ell$ do
27:     update $Lbest$;
28:     update $x_{ij}$ and $v_{ij}$
29:   end for
30:   $Kbestx[j]$:= $bestx[rank * \epsilon + j]$
31:   if $tid$ = Master then
32:     send message to all processes with $Kbestx$
33:     assemble $Bestx$ using $Kbestx$ of other processes
34:   end if
35:   if $tid = Master$ then
36:     Find minimum in $Best$
37:     send message with $Best$ of $rank$ to process Master
38:     if $rank$ = Master and $Best \leq error$ then
39:       set exit $flag$
40:     end if
41:     Master sends message with exit flag to all processes
42:   end if
43:   synchronize threads
44: until $flag$ enabled
45: End OpenMP parallel region
46: MPI_Finalize()
47: return $Best[Master]$ and $Bestx[Master]$
```

8 CPPSO on Many-core Processors

Because of a growing market demand for real-time applications for high-definition 3D graphics, the programmable graphical processing units (GPUs) have evolved into a highly parallel, multi-threaded, many-core processors with a tremendous computational potential together with very high memory bandwidth. The GPU is especially well suited to address problems that can be expressed as data-parallel computations. Hence, the same program, which requires intensive arithmetic computation, is executed on many processing elements in parallel. The larger the ratio of arithmetic operations to memory operations is, the more efficient the implementation is supposed to be. Because the same program is executed on each of the processing elements, there is a lower requirement for sophisticated control flow. Moreover, since it is executed on many data elements and has high arithmetic intensity, the memory access latency can be overcome with parallel computations instead of big data caches [17].

A GPU implements a number of streaming multiprocessors (SMs). An SM is able to create, schedule, manage, and execute concurrently many threads in groups, termed as *warps*. The execution of thousands of threads by an SM aims at mitigating the high latency of global memory access, support parallel computing with fine granularity.

Access to global memory may require hundreds of cycles. Moreover, GPUs typically have a smaller cache memory than that of the main CPU. Multhi-threading mitigates the latency required by memory access. So, while a thread is waiting for to have access to data which, are in the memory, another thread is running on the processor. The parallel fine granularity programming model provides thousands of independent threads that can keep the processor busy while other threads are waiting for data from the memory [6, 13].

Figure 8 shows a GTX460 GPU, FERMI architecture [24], used in this work. This GPU has 7 SM, each of which is composed of 48 processing cores SIMT (single instruction, multiple threads), 32K registers, 8 special function units (SFUs), 4 units of multi-threaded instructions, an instruction cache and a shared memory. The thread blocks are assigned to run by a given SM. In the SM, the threads are divided into warps, each of up to 32 threads. Thus, the scheduler can select a warp that is ready to run in SM cores. Each SM SIMT is a multi-threading hardware component that includes and arithmetic and logic unit and a floating-point unit. The SIMT core can execute concurrently more threads that use few registers or fewer threads that use more registers. The LD/ST units compute the source and destination address for up to 16 threads simultaneously. The SFUs execute instructions such as sine, cosine and square root. These instructions are executed concurrently with the instructions of the SIMT cores.

The SIMT architecture is similar to the SIMD one, that applies an instruction to multiple data, but in the former each thread has its program counter and also can still run an independent datapath of other threads. Even though the SM executes a warp more efficiently when the datapath of the threads of this warp is the same, this requirement is not necessary.

An SM, as shown in Fig. 8, schedules the threads per warp. Each SM has two schedulers, allowing two warps to run at the same time. Each scheduled warp runs by a group of 16 SIMT cores, 16 LD/ST units or 8 SFU. Thus, only half of the warp threads are executed at a time. An SM also has 4 instruction units to send instructions, i.e. 2 per scheduler. These units allow the SM to perform superscalar operations. Thus, a single thread within a scheduled warp can execute two different directions simultaneously.

While a warp is waiting to access the global memory, the warp scheduler selects another warp to run [26]. The execution context (registers, program counter, etc.) for

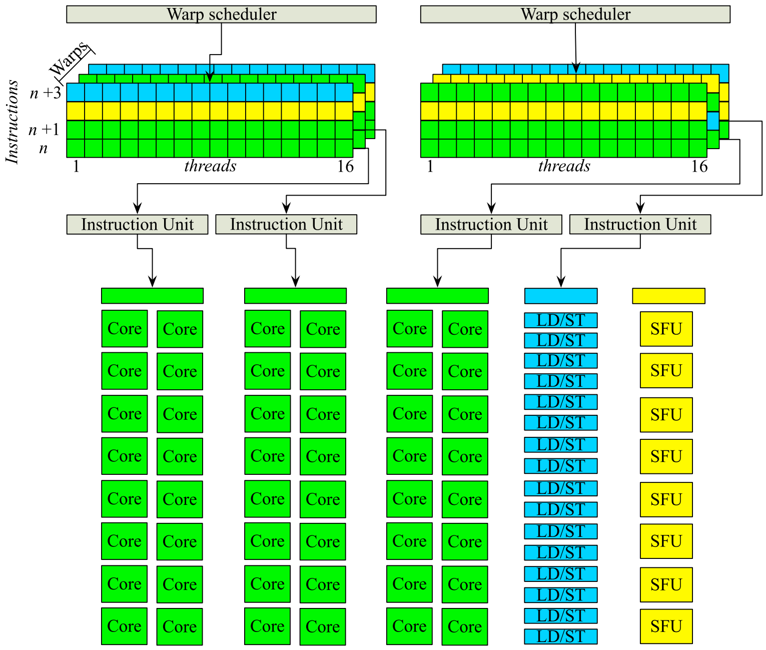

Fig. 8 SM fo the GTX460 GPU

each warp is kept on-chip throughout the whole execution time. Thus, changing the execution from one warp to another yields no additional cost as to load the registers. This change permits that the latency due to global memory access overlaps with the runtime of other threads in an available warp.

8.1 CUDA Concepts

NVIDIA CUDA technology is a C language environment that enables programmers to develop software that solve computationally complex problems by investing into the many-core parallel processing power of GPUs. In the CUDA programming model, all threads in a grid execute the same kernel function. Thus, to each thread is assigned a unique identifier. Besides, groups of threads are organized into blocks and, hence, have access to a fast local shared memory and can be synchronized using a barrier synchronization function.

The parallel computing CUDA (compute unified device architecture) platform [23] is a scalable parallel programming model and a platform for GPU programming software that allows the programmer to use an extension of the C language as a programming interface without the need of an application programming interfaces. The CUDA programming model has a style of a single instruction multiple data (SIMD), in which the designer writes a program for a single thread that will be instantiated and executed by multiple threads in parallel by the GPU multiple processors [17].

CUDA provides three key abstractions: hierarchy in thread groups, shared memory access and synchronization barriers, which add a parallel structure to the C language. In the hierarchy of thread groups, the programmer organizes the threads in *blocks* and *grids*. A grid is a set of blocks that can be run independently and in parallel. A block is formed by a set of concurrent threads that can cooperate via synchronizing barriers and using data stored in the shared memory. The shared memory is a memory area of the block private memory. It is visible to all the active threads of the block. A synchronization barrier is a point where the threads of a block that completed the required work until that point, wait for the remaining still running threads of the same block to reach the barrier [17].

Unlike threads used in CPU architecture, it is desirable that the CUDA threads are extremely light, i.e. the individual tasks should be relatively small in code size and required runtime, thus facilitating a fast exchange of context. This is important to reduce the overhead of thread generation and scheduling on the GPU [26].

The CUDA C extends the C language allowing the programmer to define functions called *kernels*. The kernel is executed by an array of threads that run the same code. When calling a kernel, the programmer specifies the number of threads per block and the number of blocks that compose the grid. Each thread has a unique identifier *Thread ID*, within the block it is inserted in and each block has a unique identifier *Block ID* in the grid it composes, as shown in Fig. 9. A kernel function is defined by the declaration __global__ and CUDA programs call the parallel kernels using $kernel \lll \sigma, \theta \ggg (\ldots p \ldots)$, where σ and θ specify the number of blocks of the grid and threads of the block respectively and p specifies the values of the required parameters. During execution, threads can access data that come from several areas of the memory. Each thread has its own private memory area, called *local memory*. Each block of threads has its own private memory area, visible to all threads in the block, called *shared memory*. Finally, all the threads, even though they are run on different grids, they can access the same memory, called *global memory*, as shown in Fig. 10.

8.2 CUDA-Oriented CPPSO Implementation

The CPPSO algorithm, expressed in a CUDA-based pseudo-code, is given in Algorithm 7, wherein there are σ blocks, which are associated to the sub-swarm and θ threads per block, which are associated to the particles in the sub-swarm corresponding to the block in consideration. Hence, there are $\sigma \times \theta$ particles operating in parallel. The CPPSO CUDA implementation uses four kernel functions: The first one launches $\sigma \times \theta$ random number generators, which are associated with the particle dimension [22], and initializes the basic information of the particles, such as position and velocity; Then, the second kernel generates $\sigma \times \theta$ threads that run the computational work to update the velocities and positions of the next iteration; After that, the third kernel also generates $\sigma \times \theta$ thread, which will rank the particle, computing the corresponding fitness. Furthermore, this kernel allows the selection of $Pbest$ of the particle. Each thread performs a copy of the entire context array, stored in the global memory, to its private memory. Then, the threads proceed with the replacement of the entries related to their sub-swarms with the coordinates of the respective best particle.

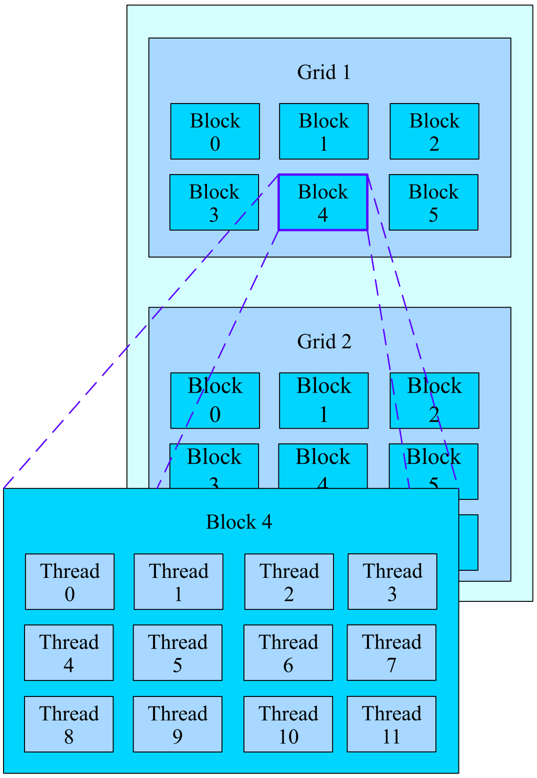

Fig. 9 Decomposition of a grid into blocks and blocks into threads

Thus, the computation of the fitness is executed taking into account all the problem dimensions.

Algorithm 7 CUDA Pseudo-code for CPPSO

```
let $\sigma$ = number of blocks (sub-swarms)
let $\theta$ = number of threads per block (particles/sub-swarm)
kernel$\lll \sigma, \theta \ggg$ initialization
repeat
  kernel$\lll \sigma, \theta \ggg$ particle movement
  kernel$\lll \sigma, \theta \ggg$ particle ranking
  kernel$\lll \sigma, \theta \ggg$ $Lbest$ selection
  transfer $Lbest$ to CPU
until $Lbest \leq error$
return the best particle
```

We use distinct kernels so we would be able to synchronize execution of threads that reside in different blocks. Such synchronization is needed to guarantee a correct simulation of the PSO core idea, allowing the exchange of the best particle details so as to select the best swarm particle.

During the execution of the optimization process, the algorithm requires $2 \times \sigma$ random numbers generators for velocity updates. In order to generate these numbers

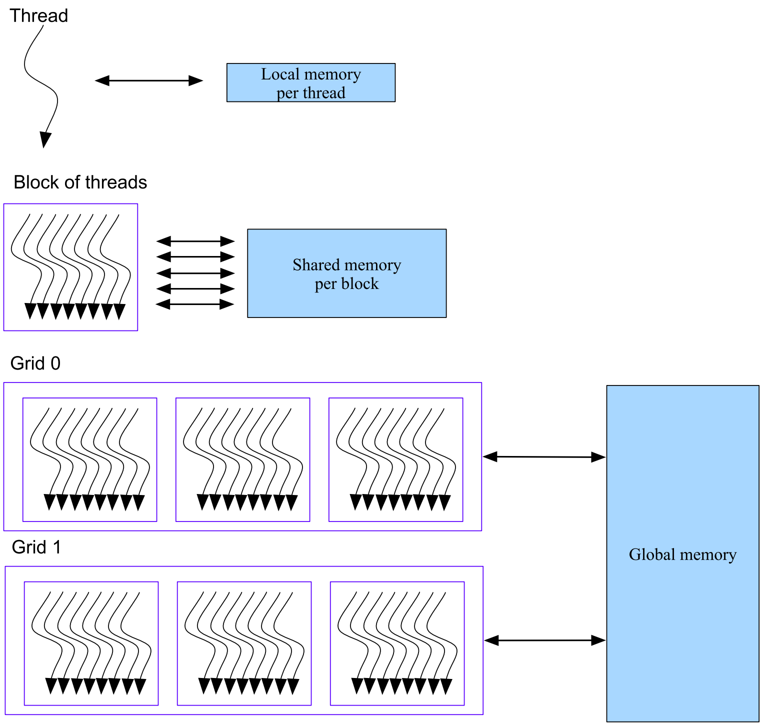

Fig. 10 GPU Memory hierarchy

inside the GPU, we use the CURAND library. This library includes functions to set up the generator seed as well as generate sequences of pseudo-random and quasi-random numbers. Thus, this implementation allows for the generated random numbers to be used directly by the kernel functions without access to the global memory. More details about the CUDA-oriented pseudo code of this implementation together with other parallelization strategies can be found in [19].

9 Performance Results

In order to evaluate the performance of the alternative implementations, we used an SGI Octane III cluster. The host CPU is an Intel I7 core of 3.07 GHz. The cluster has two 2.4 GHz Intel Xeon processors, which include 4 HT cores each, hence a total of 16 cores. The GPU solution uses an NVIDIA GeForce GTX 460 GPU to run the CUDA implementation. The GPU includes 7 SMs with 48 CUDA cores of 1.3 GHz each, hence a total of 336 cores. Four classical benchmark functions, as listed in Table 1, were used to evaluate the performance of the proposed implementations.

We run the compared implementations using the same configuration parameters as defined in Table 2. These are the number of dimensions in the original problem δ, the total number of particles η, the number of blocks or sub-swarms σ, the number of dimensions by sub-problem ϵ, which is common to all sub-swarms and the number of

Table 1 Objective functions used as benchmarks

Function	Domain	f_{min}
$f_1(x) = \sum_{i=1}^{\delta} x_i^2$	$[-100, 100]^\delta$	0.0
$f_2(x) = 100\left(x_{i-1} - x_i^2\right)^2 + (x_{i-1} - 1)^2$	$[-16, 16]^\delta$	0.0
$f_3(x) = 418.9829\delta - \sum_{i=1}^{\delta} x_i sin\left(\sqrt{\lvert x_i \rvert}\right)$	$[-500, 500]^\delta$	0.0
$f_4(x) = \sum_{i=1}^{\delta} (x_i^2 - 10cos(2\pi x_i) + 10)$	$[-5.12, 5.12]^\delta$	0.0

Table 2 Arrangements of particles and dimensions for the evaluations

Case	C1	C2	C3	C4	C5	C6	C7	C8
δ	2	4	8	16	32	64	128	256
η	8	16	32	64	128	256	512	1024
θ	4	4	8	16	32	64	128	256
σ	1	1	2	4	8	16	32	64
$\eta \times \delta$	16	64	256	1024	4096	16,384	65,536	262,144

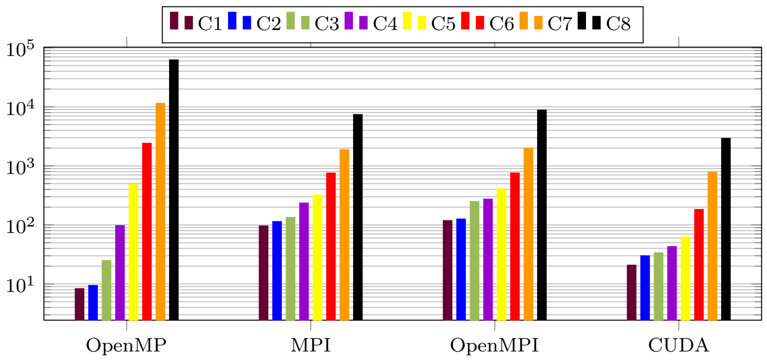

Fig. 11 Optimization time comparison for function f_1

threads per block or that of particles per sub-swarm, θ. All charts horizontal axes are represented in terms of $\eta \times \delta$ as computed in the last row of Table 2. The optimization processes were repeated 50 times with different seeds to guarantee the robustness of the obtained results. The PSO basic parameters were set up as follows: inertial coefficient ω initialized at 0.99 and decreased linearly as far as 0.2; $vmax$ computed using δ of 0.2 and the stopping condition adopted was either achieving at most an error of 10^{-4} or 6000 iterations.

The charts of Figs. 11, 12, 13 and 14 show the convergence times of benchmark functions, f_1–f_4, using the four proposed implementations of CPPSO.

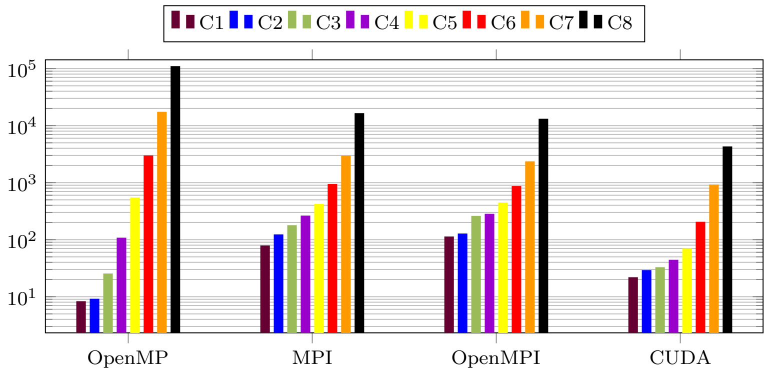

Fig. 12 Optimization time comparison for function f_2

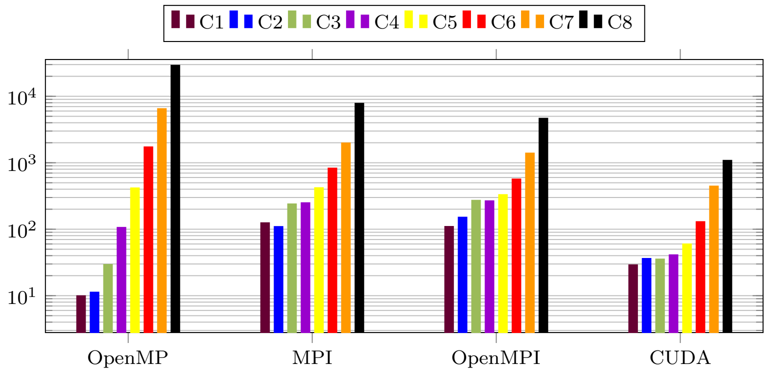

Fig. 13 Optimization time comparison for function f_3

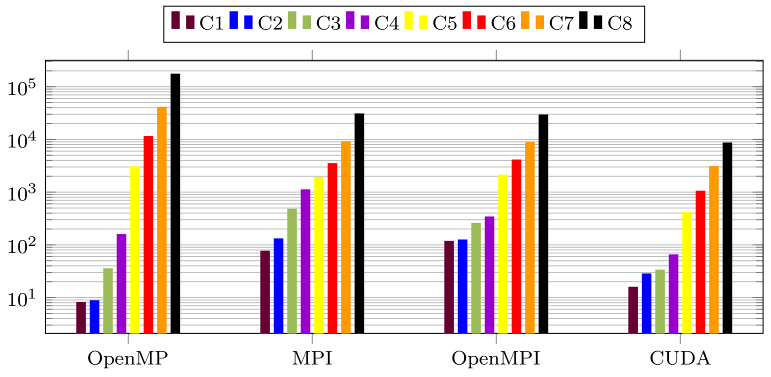

Fig. 14 Optimization time comparison for function f_4

In order to make an assessment of the impact of the CPPSO parallelization strategy and the four alternative implementations on different parallel architecture, we compare the execution time of the proposed implementation to that obtained by a the sequential implementation of the PSO. The charts of Figs. 15, 16, 17 and 18 show the

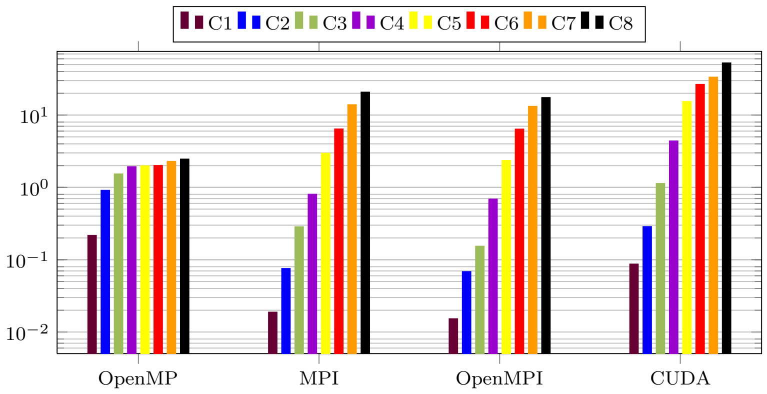

Fig. 15 Speedup comparison for function f_1

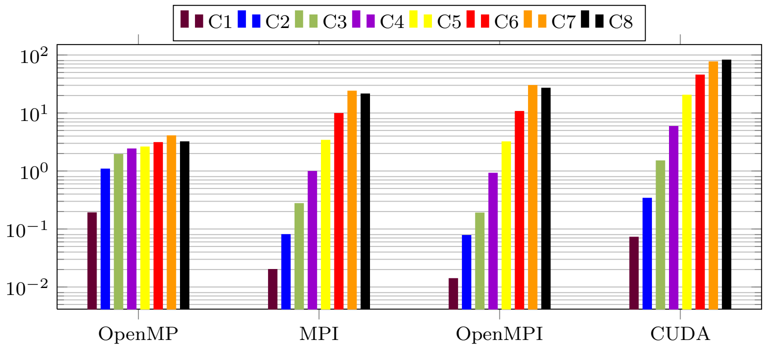

Fig. 16 Speedup comparison for function f_2

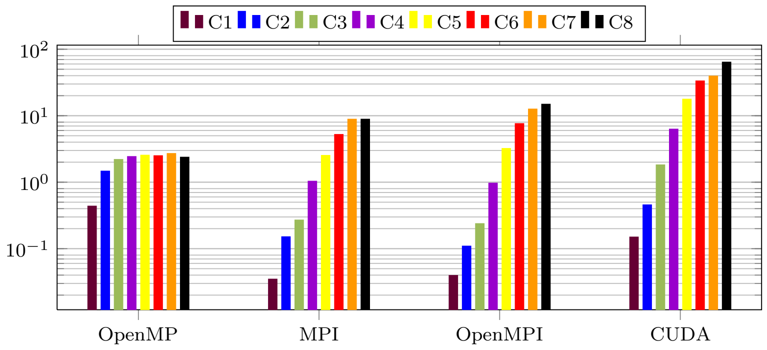

Fig. 17 Speedup comparison for function f_3

speedups achieved during the minimization of benchmark functions f_1–f_4 using the four proposed implementations of CPPSO. In all cases, the speedup is computed with respect to the sequential implementation of the PSO running on a single core of the Xeon processor.

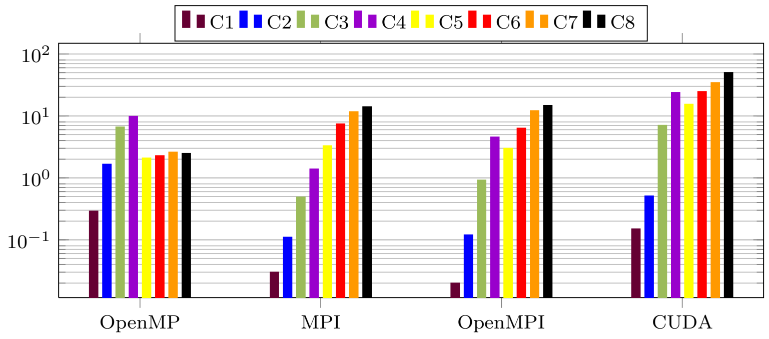

Fig. 18 Speedup comparison for function f_4

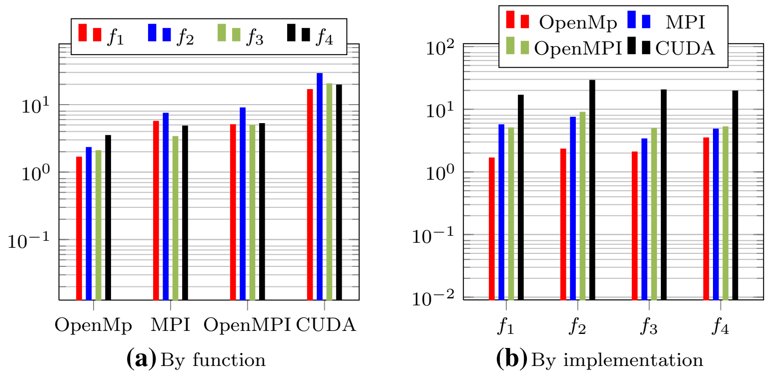

Fig. 19 Compared average speedups grouped by implementation and by function

The OpenMP implementation achieved speedups of up to 2.5× during the optimization of function f_1, 4.1× for f_2, 2.7× for function f_3 and 9.8 for function f_4. The MPI based implementation introduced speedups of up to 20.7× for function f_1, 23.8× for function f_2, 8.8× for function f_3 and 14.1× for function f_4. The hybrid MPI+OpenMP implementation achieved speedups of up to 17.4× for function f_1, 29.8× for function f_2, 14.8× for function f_3 and 14.7× for function f_4. Finally, the CUDA implementation occasioned speedups of up to 52.6× during the optimization of function f_1, 81.6× for function f_2, 63.8× for function f_3 and 50.2× for function f_4. It can easily be observed that, in general, the speedup yielded by the alternative implementations increases with the complexity of the optimization process. Average speedups as achieved by the four alternative parallel implementation of CPPSO regarding the four benchmark functions are compared in Fig. 19, grouped by function and by implementation.

10 Conclusion

This paper investigates the implementation of a cooperative parallel PSO on multi-core and many-core architectures. The parallelization strategy divides the original problem into many simpler sub-problems, each of which is optimized by a sub-swarm of particles. The sub-problems are similar to the original problem but handle fewer dimensions. Each sub-swarm is mapped into a block of threads while each of its composing particles is mapped into one thread of the corresponding block. This allows the distribution of the computational load at a coarse degree of granularity, which is up to one block per problem set of dimensions.

The strategy behind CPPSO was implemented in OpenM, MPICH, OpenMP with MPI and CUDA. The investigated implementations showed a positive impact on the performance on the optimization process. They achieved significant speedups when compared to the sequential implementation, especially in the case of complex problems that explore large dimension search space. The results show that the implementation on many-core GPGPU-based parallel architecture is the most efficient among the proposed implementations.

Future work includes the analysis of the partition factor on the speedup achieved and the implementation of other partitioning strategies on different parallel architectures.

References

1. Bergh, F.V., Engelbrecht, A.P.: Cooperative Learning in Neural Networks using Particle Swarm Optimizers. S. Afr. Comput. J. **26**, 84–90 (2000)
2. Cádenas-Montes, M., Vega-Rodríguez, M.A., Rodríguez-Vázquez, J.J., Gómez-Iglesias, A.: Accelerating particle swarm algorithm with GPGPU. In: Proceedings of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 560–564. IEEE Press (2011)
3. Calazan, R.M., Nedjah, N., Mourelle, L.M.: A cooperative parallel particle swarm optimization for high-dimension problems on GPUs. In: Proceedings of the BRICS Conference on Computational Intelligence, Porto de Galinhas, PE, Brazil, IEEE Press (2013)
4. Calazan, R.M., Nedjah, N., Mourelle, L.M.: Parallel GPU-based implementation of high dimension particle swarm optimizations. In: Proceedings of the Computational Science and Its Applications (ICCSA 2012), LNCS 7333, pp. 148–160 (2013)
5. Calazan, R.M., Nedjah, N., Mourelle, L.M.: A massively parallel reconfigurable co-processor for computationally demanding particle swarm optimization. IN: Proceedings of the 3rd International Symposium of IEEE Circuits and Systems in Latin America (LASCAS 2012), Cancun, Mexico. IEEE Computer Press, Los Alamitos, CA (2012)
6. Calazan, R.M., Nedjah, N., Mourelle, L.M.: Swarm grid: a proposal for high performance of parallel particle swarm optimization using GPGPU. In: Proceedings of the 4th International Symposium of IEEE Circuits and Systems in Latin America (LASCAS 2013), Cuzco, Peru, IEEE Computer Press, Los Alamitos, CA (2012)
7. Calazan, R.M., Nedjah, N., Mourelle, L.M.: Parallel co-processor for PSO. Int. J. High Perform. Syst. Archit. **3**(4), 233–240 (2011)
8. Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming, vol. 10. MIT Press, London (2008)
9. Cui, Z., Cai, X., Shi, Z.: Using fitness landscape to improve the performance of particle swarm optimization. J. Comput. Theor. Nanosci. **9**(2), 258–266 (2012)
10. Cui, Z., Cai, X., Zeng, J., Sun, G.: Particle swarm optimization with FUSS and RWS for high dimensional functions. Appl. Math. Comput. **205**(1), 98–108 (2008)
11. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Commun. ACM **9**(3), 143–155 (1966)
12. Engelbrecht, A.P.: Fundamentals of Computational Swarm Intelligence. Wiley, New Jersey (2005)
13. Farber, R.: CUDA Application Design and Development. Morgan Kaufmann, Waltham (2011)
14. Foster, I.: Designing and Building Parallel Programs, vol. 95. Addison-Wesley, Reading (1995)
15. Gropp, W., Smith, B.: Users Manual for the Chameleon Parallel Programming Tools. Mathematics and Computer Science, Argonne National Laboratory, Argonne (1993)

16. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings of the IEEE International Conference on Neural Network, pp. 1942–1948. IEEE Press, Australia (1995)
17. Kirk, D.J., Hwu, W.: Programming Massively Parallel Processors. Morgan Kaufmann, San Francisco (2010)
18. MPI_Forum: Message Passing Interface Forum, rev. MPI-2, http://www.mpiforum.org (2012)
19. Nedjah, N., Calazan, R.M., Mourelle, L.M.: Particle, dimension and cooperation-oriented PSO parallelization strategies for efficient high-dimension problem optimizations on graphics processing units. Comput. J. Sect. C Comput. Intell. Mach. Learn. Data Anal. (2015). doi:10.1093/comjnl/bxu153
20. Nedjah, N., Coelho, L.S., Mourelle, L.M.: Multi-Objective Swarm Intelligent Systems—Theory & Experiences. Springer, Berlin (2010)
21. NVIDIA: NVIDIA CUDA C Programming Guide, Version 4.0 NVIDA Corporation (2011)
22. NVIDIA: CURAND Library, Version 1.0, NVIDA Corporation (2010)
23. NVIDIA: CUDA C Programming Guide, rev. 3.2, NVIDA Corporation, http://developer.nvidia.com/object/cuda_3_2_downloads (2010)
24. NVIDIA: NVIDIA Next Generation CUDA Compute Architecture: Fermi, NVIDIA Corporation, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper (2009)
25. Papadakis, S.E., Bakrtzis, A.G.: A GPU accelerated PSO with application to economic dispatch problem. In: 16th International Conference on Intelligent System Application to Power Systems (ISAP), pp. 1–6. IEEE Press (2011)
26. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, Waltham (2011)
27. Sanders, J., Kandrot, E.: CUDA by Example, An Introduction to General-Purpose GPU Programing. Addison-Wesley, San Francisco (2010)
28. Veronese, L., Krohling, R.A.: Swarm's flight: accelerating the particles using C-CUDA. In: 11th IEEE Congress on Evolutionary Computation, pp. 3264–3270. IEEE Press, Trondheim (2009)
29. Walker, D.W., Dongarra, J.J.: MPI: a standard message passing interface. Supercomputer **12**, 56–68 (1996)
30. Weihang, Z., Curry, J.: Particle swarm with graphics hardware acceleration and local pattern search on bound constrained problems. In: IEEE Swarm Intelligence Symposium (SIS 2009), pp. 1–8. IEEE Press (2009)
31. Zhou, Y., Tan, Y: GPU-based parallel particle swarm optimization. In: 11th IEEE Congress on Evolutionary Computation (CEC 2009), pp. 1493–1500. IEEE Press, Trondheim (2009)