J Glob Optim (2017) 69:607–627
DOI 10.1007/s10898-016-0468-7

Non-dominated sorting procedure for Pareto dominance ranking on multicore CPU and/or GPU

G. Ortega[1] · E. Filatovas[2] · E. M. Garzón[1] · L. G. Casado[1]

Received: 15 December 2015 / Accepted: 14 September 2016 / Published online: 20 September 2016

Abstract Evolutionary multi-objective optimization algorithms aim at finding an approximation of the Pareto set. For hard to solve problems with many conflicting objectives, the number of functions evaluations to represent the Pareto front can be large and time consuming. Parallel computing can reduce the wall-clock time of such algorithms. Previous studies tackled the parallelization of a particular evolutionary algorithm. In this research, we focus on improving one of the most time consuming procedures—the non-dominated sorting—, which is used in the state-of-the-art multi-objective genetic algorithms. Here, three parallel versions of the non-dominated sorting procedure are developed: (1) a multicore (based on Pthreads); (2) a Graphic Processing Unit (GPU) (based on CUDA interface); and (3) a hybrid (based on Pthreads and CUDA). The user can select the most suitable option to efficiently compute the non-dominated sorting procedure depending on the available hardware. Results show that the use of GPU computing provides a substantial improvement in terms of performance. The hybrid approach has the best performance when a good load balance is established among cores and GPU.

This work has been partially supported by the Spanish Ministry of Science throughout Project TIN15-66680, by J. Andalucía through Projects P12-TIC-301 and P11-TIC7176, and by the European Regional Development Fund (ERDF).

✉ G. Ortega
gloriaortega@ual.es

E. Filatovas
ernest.filatov@gmail.com

E. M. Garzón
gmartin@ual.es

L. G. Casado
leo@ual.es

1 Informatics Department., Agrifood Campus of International Excellence (ceiA3), University of Almería, Ctra. Sacramento s/n. La Cañada de San Urbano, Almería 04120, Spain

2 Institute of Mathematics and Informatics, Vilnius University, Akademijos str. 4, Vilnius, LT 08663, Lithuania

Keywords Multi-objective optimization · Evolutionary algorithms · Non-dominated sorting · High Performance Computing · GPU computing · NSGA-II

1 Introduction

Many real-world problems are multi-objective, where several conflicting objective functions have to be optimized. The main aim of Multi-Objective Optimization (MOO) is to provide the set of solutions that determine the Pareto front used by the Decision Maker (DM). The most popular approaches to solve MOO problems are evolutionary multi-objective optimization (EMO) and Multiple Criteria Decision Making (MCDM). In MCDM, the DM's preferences are important when we deal with multi-objective optimization problems, because the main goal is to find the most satisfactory solution for the DM without exploring the whole Pareto set. MCDM approaches usually are classified into a priori, posteriori and interactive methods, depending on when the preference information is requested to the DM [24]. EMO approaches do not require any DM's preference information — the main target is to find such elements of Pareto set which correspond to well-converged and well-distributed non-dominated objective vectors along the entire Pareto front. The set of achieved solutions by an algorithm is presented to the DM, who finally chooses one among them, according to his/her preferences. EMO approaches are popular because they do not require any characteristic of the objective functions and are easy to code. Some well-known EMO algorithms to approximate the Pareto front are: NSGA-II [7], PAES [20], MOAE/D [36], IBEA [39], SPEA2 [40], etc.

There exist many works where EMO algorithms are successfully applied to solve relatively small problems, i.e., problems with two or three objective functions, using population sizes from 100 to 500 individuals to compute the Pareto front. Usually, the population size do not exceed 1000 individuals for many instances of the problem which is not enough to generate the level of detail in the Pareto front needed by the DM in general. The reason of relative small populations is the significative increase in the computational burden with the population size. Parallel computing allows to handle larger populations in a reasonable amount of time. Only in few studies, relatively big populations were used for approximating the Pareto front. For instance, populations of sizes up to 10,000 individuals were used in NSGA-II and MOAE/D algorithms to solve 2–10 objectives knapsack problems [18]. The parallel GPU implementation of an EMO algorithm was experimentally tested with more that 16,000 individuals in [35]. An EMO algorithm was proposed and experimentally tested in [1] with population sizes varying from 100 to 20,000 individuals. A parallel GPU implementation of NSGA-II using ZDT and DTLZ benchmark problems with population up to 30,000 individuals was investigated in [15].

Usually, a multi-objective algorithm can be organized in several phases [2,21]: evaluation of an objective function, Pareto dominance ranking (non-dominated sorting) and genetic operations. Examples of EMO approaches based on Pareto dominance ranking are: PESA-II [5], NSGA-II [7], R-NSGA-II [8], Synchronous R-NSGA-II [13], MOGA [14], PAES [20], NSGA [32], SPEA2 [40], etc. The Pareto dominance ranking can be one of the most computationally expensive phases of the EMO algorithms, mainly when the computational cost of the evaluation of the objective functions is not high. The fast non-dominated sorting (FNDS) procedure, which is used in NSGA-II, has a complexity order of $O(vp^2)$, where v is the number of objective functions and p is the number of individuals [7]. Its complexity was reduced to $O(p\log^{v-1} p)$ by adopting divide-and-conquer strategy in [19]. However, as it was argued in [12,23], this algorithm is not applicable to many instances of the problem, e.g., when

ϵ-dominance is used for comparison of the individuals [6,22] or the population has duplicate individuals or strong dominance. In [34], non-dominated sorting approach based on Arena's Principle was proposed, which technically has the same complexity as the sorting approach used in NSGA-II. Nevertheless, a $O(vp\sqrt{p})$ can be achieved for some instances of the problem. Deductive sort was proposed in [23], where dominance relation between individuals is recorded, avoiding some unnecessary comparisons. Similar idea was presented in [37], where individuals to be assigned to the front are compared only with those that have already been assigned to the front. There are more proposals to reduce the computational burden of the non-dominated sorting procedure: Deb et al. proposed the omni-optimizer in [10], Shi et al. introduced a better non-dominated sorting in [30], Zheng et al. applied quick sort in [38], Du et al. proposed a sorting based algorithm in [11] and Fang et al. presented divide-and-conquer based non-dominated sorting algorithm in [12]. Summarizing, the reduction of the complexity order of the non-dominated sorting procedure has gained much interest from researchers. However, its computational burden have a complexity $O(vp^2)$ in the worst case for all the approaches. The time complexity of the studied algorithms in this work is also $O(vp^2)$. Therefore, parallel strategies have been used to accelerate the computation of the procedure.

Nowadays, CPUs are multicore. They usually include accelerators such as Graphics Processing Units (GPUs) [16], that provide a considerable computational power to the desktop, laptop and mobile platforms. GPUs are widely used in the High Performance Computing (HPC) field due to their performance/cost ratio [16]. In the last few years, the use of GPUs in general purpose applications has greatly increased thanks to the availability of application programming interfaces, such as Compute Unified Device Architecture (CUDA)[1] and OpenCL [25]. GPUs have hundreds of cores that can collectively run thousands of computing GPU-threads. However, only few attempts have been done to develop a parallel Pareto dominance ranking. For instance, parallel stochastic ranking operator was proposed in [28] and the solution of several two-objective benchmark problems are presented in [29]. Formal proofs to reduce the time complexity of these algorithms and an efficient parallel version of FNDS have been presented in [31], but their experimental results deserve a more detailed explanation. Recently, a novel NSGA-II parallel implementation on a GPU, focusing on NDS and achieving promising speed-ups, has been proposed in [15]. Although it is closely related to our proposal, the goal of our work is the exploitation of all the resources of modern heterogeneous platforms, not only the GPUs. So, we propose a new NDS parallel version which can be accelerated by using simultaneously the multicore and the GPU. Moreover, our proposal defines the NDS module which can be easily used, not only by NSGA-II but also by the NSGA family of algorithms.

The implementations carried out in this work squeeze out the performance of the multicore and GPU platforms. Additionally, to facilitate the use of the new codes, they have been encapsulated in Matlab[2] thanks to MEX-files [33]. Thus, our developed routines can be called from Matlab codes when solving MOO problems by EMO approaches, allowing users to chose between multicore, GPU or hybrid implementations, according to the features of their computers and MOO problems.

The main contributions of this research are:

– The development of new parallel versions of the FNDS procedure on multicore and/or GPU, that allow a significant reduction of the wall-clock time of EMO algorithms by the exploitation of the resources of the current desktop computers. They are: a multicore

[1] https://developer.nvidia.com/cuda-toolkit.

[2] http://www.mathworks.com/help/pdf_doc/matlab/getstart.

version (based on Pthreads[3]); a GPU version (based on CUDA); and a hybrid version (based on Pthreads and CUDA).
- The evaluation of the parallel algorithms on several multicore and/or GPU configurations with several test cases by varying the number of objective functions and populations size, in order to identify which parallel version results in a better performance for each test case.
- The provision of a training procedure to help the user to select the most suitable FNDS version according to the particular characteristics of the problem and the computational platform.

Additionally, the codes of the developed algorithms are open source[4] and they can be called from Matlab or C.

The rest of this paper is organized as follows. In Sect. 2, the definition of the multi-objective problem is provide, as well as the description of the fast non-dominated sorting procedure (FNDS). Section 3 shows the data structures considered for the implementations. Section 4 discusses the improvement of the Dominance Comparison, which is one of the most time consuming phases of the FNDS procedure. The parallel implementations of non-dominated sorting (multicore, GPU and hybrid) are presented in Sect. 5. Experimental results of the parallel implementations are discussed in Sect. 6. Finally, Sect. 7 shows the conclusions of this work.

2 Background

Multi-objective optimization as well as description of the fast non-dominated sorting procedure are summarized in next subsections.

2.1 Multi-objective optimization problem

Let us have $v \geqslant 2$ conflicting objectives, described by the functions $f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_v(\mathbf{x})$, where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is a vector of variables (*decision vector*) and n is the number of variables or dimension of the problem. A multi-objective minimization problem is formulated as follows [24]:

$$\min_{\mathbf{x}\in\mathbf{S}} \mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_v(\mathbf{x})]^T \tag{1}$$

where $\mathbf{z} = \mathbf{f}(\mathbf{x})$ is an *objective vector*, defining the values for all objective functions, $f_i\colon \mathbf{S} \subseteq \mathbb{R}^n \rightarrow \mathbb{R}$, $i \in \{1, 2, \ldots, v\}$, where v is the number of objective functions, and $\mathbf{S} \subseteq \mathbb{R}^n$ is an n-dimensional euclidean space, called *search space*, which defines all feasible decision vectors. When solving a multi-objective optimization problem a feasible solution that minimizes all objective functions simultaneously does not exist. Therefore, Pareto optimal solutions are obtained and provided to the DM.

A decision vector $\mathbf{x}' \in \mathbf{S}$ is a *Pareto optimal solution* if $f_i(\mathbf{x}') \leqslant f_i(\mathbf{x})$ for all $\mathbf{x} \in \mathbf{S}$ and $f_j(\mathbf{x}') < f_j(\mathbf{x})$ for at most one j. Objective vectors are defined as optimal if none of their elements can be improved without worsen at least one of the other elements. An objective vector $z' = \mathbf{f}(\mathbf{x}')$ is Pareto optimal if the corresponding decision vector $\mathbf{x}'$ is Pareto optimal. The set of all the Pareto optimal decision vectors is called the *Pareto set*. The region defined by all the objective function values for the Pareto set points is called the *Pareto front*.

[3] https://computing.llnl.gov/tutorials/pthreads/.

[4] https://sites.google.com/site/hpcoptimizationproblems/FNDS.

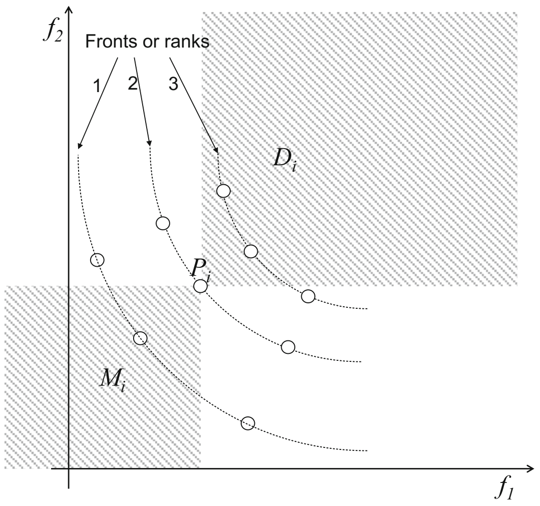

Fig. 1 Two dimensional illustration of fronts and dominance among individuals

We say that an objective vector $\mathbf{z}' \in \mathbb{R}^v$ dominates another objective vector $\mathbf{z} \in \mathbb{R}^v$ (or $z' \succ z$) if $z'_i \leqslant z_i$ for all $i = 1, \ldots, v$ and there exists at least one j such that $z'_j < z_j$.

In EMO algorithms, the subset of solutions in a population whose objective vectors are not dominated by any other objective vector is called the *non-dominated set*, and the objective vectors are called the *non-dominated objective vectors*. The main aim of the EMO algorithms is to generate well-distributed non-dominated objective vectors as close as possible to the Pareto front.

In the following subsection, one of the most time consuming procedures used in most of the state-of-the-art EMO algorithms to calculate the Pareto dominance ranking is analysed.

2.2 Fast non-dominated sorting procedure

The FNDS procedure assigns ranks to the individuals, and classifies the population into several non-dominated levels (so-called fronts) (see the Fig. 1). According to Pareto dominance, the individuals with the same rank are non-dominated among them and they can be only dominated by a solution in a lower rank. Dominance comparisons between the individuals is the most repeated operation in non-dominated sorting with a high computational burden, which determines its efficiency.

Algorithm 1 (FNDS-UDC) shows a pseudocode of the FNDS procedure. The presented procedure can be divided into three phases. *Phase 1—Unidirectional Dominance Comparison* –, where comparisons between individuals of initial population P are computed and the information about their dominance is collected. *Phase 2*, where the individuals of the first front are classified, and *Phase 3—Front assignment*—, where the initial population P is classified in several fronts by the usage of dominance information. The FNDS-UDC sorts the population (P) in ranks (or fronts) (R_i) starting from the front 1. The following notations are used:

- n : the dimension of the search region $\mathbf{S} \in \mathbb{R}^n$.
- p : the number of individuals.
- v : the number of objective functions.
- $X_{(p\times n)}$: the matrix of individuals' decision vectors.

- $V_{(p\times v)}$: the matrix of the individuals' objective vectors.
- $rank(P_i)$ is the rank of P_i.
- $T_{(p\times 1)}$: the column vector with the rank of each individual, i.e., $T_i = rank(P_i)$, $i = 1, \ldots, p$.
- $P = [X|V|T]_{(p\times(n+v+1))}$: the matrix of individuals, as the result of the concatenation of the columns of X, V and T.
- $F_{(p\times p)}$: the front matrix. It stores the individuals at each front or rank. Each row F_i of F stores the set of indices of individuals at front i, i.e., $F_i = \{j : rank(P_j) = i\}$.
- $\#F_{(p\times 1)}$: the column vector with the number of individuals at each front. $\#F_i = |\{j : rank(P_j) = i\}|$.
- t_{max}: the number of fronts, i.e. the number of non zero rows of $\#F$.
- $R = [\#F|F]_{(p\times(p+1))}$: the rank matrix, as the result of the concatenation of the columns of $\#F$ and F.
- $\#M_{(p\times 1)}$: the column vector with the number of dominators (masters) of each individual.
- $\#D_{(p\times 1)}$: the column vector with $\#D_i = |\{j : P_j \prec P_i\}|$, i.e, the element i stores the number of individuals dominated by P_i.
- $D_{(p\times p)}$: the dominated matrix. The row i stores the indexes of individuals dominated by P_i, i.e., $D_i = \{j : P_j \prec P_i\}$.
- $MD = [\#M|\#D|D]_{(p\times(p+2))}$: the masters-dominated matrix, as the result of the concatenation of the columns of $\#M$, $\#D$ and D.

Each ith iteration (or individual processing) of the algorithm is irregular because it contains conditional instructions that can change the program trace. However, if we assume that the population to be sorted by FNDS is large enough, the workload for several sets of iterations can be considered to be near to their average value.

3 Data structures in Fast non-dominated sorting

Due to the importance of the exploitation of the memory hierarchy in the performance of the algorithms, the data structures in FNDS has been designed to preserve as much as possible the locality of the data in memory. Figure 2 shows the bi-dimensional data structures used in FNDS-UDC.

The input of the FNDS is the population P and the output is the rank matrix R and the update of T in the P structure. FNDS generates the dominance matrix D which is very sparse to calculate R. Because the number of fronts (or ranks) is unknown when FNDS starts, the number of rows of R is set to p. P, MD and R matrices have been defined as static data structures of size $(p \times (n + v + 1))$, $(p \times (p + 2))$ and $(p \times (p + 1))$, respectively. It means that the sizes of these matrices are invariant throughout their lifetime. They are set to their maximum size. The elements of a static structure are held in contiguous memory words to improve the spatial locality of the memory access. Focusing on P and MD, they have been stored column-wise, which presents a better spatial locality in memory accesses than the row-wise approach. On one hand, the dominance checking between every individual and the population is computed with a high spatial locality in the access to P. On the other hand, first columns of MD are frequently acceded to update the number of dominators and dominated of an individual. Storing MD in column order increases the probability of maintaining first columns in cache memory and the number of cache misses is consequently reduced. This has a strong impact on the performance of FNDS when the population is very large, since most of the running time consumed by the algorithm is due to the memory access performed in the dominance comparison.

Algorithm 1 FNDS-UDC. Fast non-dominated sorting procedure - Unidirectional Dominance Comparison.

```
Input:
    $n$: the dimension of the search space,
    $p$: the number of individuals,
    $v$: the number of objective functions,
    $P_{(p\times(n+v+1))}$: population.
Output:
    $F$: the front matrix,
    $T$: the rank of each individual.
    Phase 1: Unidirectional Dominance Comparison (UDC)
1:  for $i \leftarrow 1$ to $p$ do
2:      $D_i \leftarrow \{\emptyset\}$
3:      $\#M_i \leftarrow 0$
4:      for $j \leftarrow 1$ to $p$ do
5:          for $k \leftarrow 1$ to $v$ do
                Check dominance between individuals $P_i$ and $P_j$ for the objective $k$
6:          if $P_i \succ P_j$ then
7:              $D_i \leftarrow D_i \cup \{j\}$          ▶ Add $P_j$ to the set of individuals dominated by $P_i$
8:          else if $P_j \succ P_i$ then
9:              $\#M_i \leftarrow \#M_i + 1$          ▶ Increase dominators counter for $P_i$
    Phase 2: Assign front to the individuals of the $1^{st}$ front
10: for $i \leftarrow 1$ to $p$ do
11:     if $\#M_i = 0$ then          ▶ If individual $P_i$ belongs to the $1^{st}$ front
12:         $T_i = 1$          ▶ Set individual $i$ to the $1^{st}$ front
13:         $F_1 \leftarrow F_1 \cup \{i\}$
    Phase 3: Front Assignment
14: $t = 1$          ▶ Initialize the front counter
15: while $F_t \neq \{\emptyset\}$ do
16:     $Q \leftarrow \{\emptyset\}$          ▶ Temporary set used to store individuals of the next front
17:     for each $i \in F_t$ do
18:         for each $j \in D_i$ do
19:             $\#M_j \leftarrow \#M_j - 1$
20:             if $\#M_j = 0$ then          ▶ If individuals $P_j$ belongs to the next front
21:                 $T_j \leftarrow t + 1$          ▶ Set individual $j$ to the $t + 1$ front
22:                 $Q \leftarrow Q \cup \{j\}$
23:     $t \leftarrow t + 1$          ▶ Increase the front counter
24:     $F_t \leftarrow Q$
25: $t_{max} \leftarrow t$
26: return $R, T$          ▶ Returns the Pareto Dominance Ranking
```

Static allocations have the drawback of oversizing *MD* and *R* to guarantee enough memory for them due to the unknown dominance patterns. However, the memory requirements for *R* could be greatly reduced if the number of fronts is a priori known.

4 The Dominance Comparison Phase

Experimental results of the execution of the sequential implementation have shown that Dominance Comparison phase (see Phase 1 in Algorithms 1 and 2) consumes most of the computational burden of the FNDS procedure (see Sect. 6, Table 2) due to the intensive memory accesses and the large number of performed comparisons.

The number of comparisons in the UDC phase of FNDS-UDC procedure (see Alg. 1) is reduced in FNDS-BDC (see Alg. 2) by using a BDC (Bidirectional Dominance Comparison)

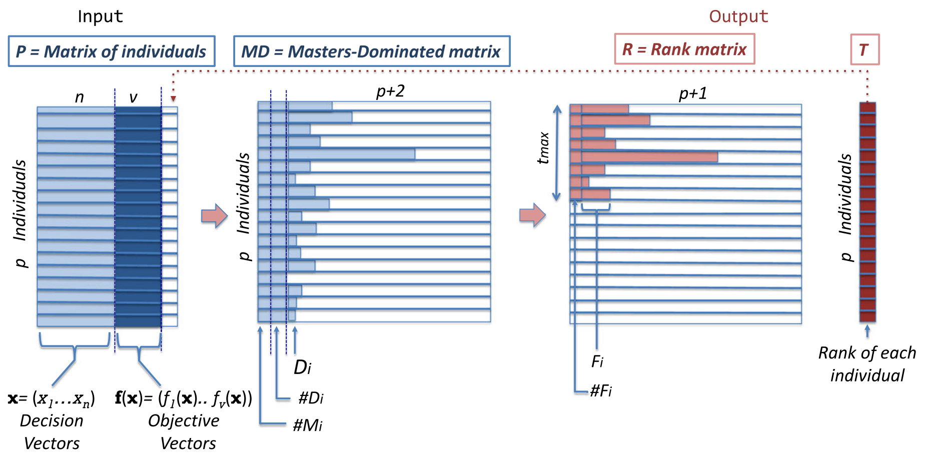
Input
P = Matrix of individuals
MD = Masters-Dominated matrix
Output
R = Rank matrix
T
n
v
p Individuals
p+2
p+1
tmax
Di
#Di
#Mi
Fi
#Fi
Rank of each individual
x= (x1…xn) Decision Vectors
f(x)= (f1(x).. fv(x)) Objective Vectors

Fig. 2 Data structures defined in the implementations of FNDS

instead. In the later, when two individuals (P_i and P_j) are compared, the information about the dominance of both is updated (MD_i and MD_j). The main difference between UDC and BDC is the modification of the *for* nested loop (Alg. 2, line 5).

Algorithm 2 FNDS-BDC. Fast Non-Dominated Sorting Procedure - Bidirectional Dominance Comparison.

```
Input:
    $n$: the dimension of the search space,
    $p$: the number of individuals,
    $v$: the number of objective functions,
    $P_{(p\times(n+v+1))}$: population.
Output:
    $F$: the front matrix,
    $T$: the rank of each individual.
 1: for $i \leftarrow 1$ to $p$ do
 2:     $S_i \leftarrow \{\emptyset\}$
 3:     $\#M_i \leftarrow 0$
    Phase 1: Bidirectional Dominance Comparison (BDC)
 4: for $i \leftarrow 1$ to $p$ do
 5:     for $j \leftarrow i+1$ to $p$ do
 6:         for $k \leftarrow 1$ to $v$ do
 7:             Check dominance between individuals $P_i$ and $P_j$ for the objective $k$
 8:         if $P_i \succ P_j$ then
 9:             $D_i \leftarrow D_i \cup \{j\}$        ▶ Add $P_j$ to the set of individuals dominated by $P_i$
10:             $\#M_j \leftarrow \#M_j + 1$        ▶ Increase the dominators counter for $P_j$
11:         else if $P_j \succ P_i$ then
12:             $D_j \leftarrow D_j \cup \{i\}$        ▶ Add $P_i$ to the set of individuals dominated by $P_j$
13:             $\#M_i \leftarrow \#M_i + 1$        ▶ Increase dominator counter for $P_i$
    Phase 2 (see Alg. 1)
    Phase 3 (see Alg. 1)
14: return $R$, $T$        ▶ Returns the Pareto Dominance Ranking
```

As explained in next section, the drawback of the BDC in comparison with UDC phase is the existence of more conflicting concurrent writes in the parallel versions produced in the former than in the later, which reduces the efficiency of the parallel algorithm. However, sequential BDC codes are widely used[5] due to their better performance on a CPU-core.

5 Parallel implementations of the non-dominated sorting

As aforementioned, experimental results of the execution of the sequential implementation have shown that Unidirectional Dominance Comparison phase consumes most of the computational burden of FNDS (see Sect. 6, Table 2).

In previous sections, unidirectional (UDC) and bidirectional (BDC) dominance comparison phases have been shown. Their parallel versions have been experimentally evaluated on the computer architectures described in Sect. 6. The experimental evaluation shows relevant execution time differences between these parallel versions. Parallel BDC achieves a worse performance than parallel UDC as the number of parallel processes increases, in spite of the smaller number of comparisons performed by BDC. This result is a consequence of the parallelization of the outer for-loop of BDC, because the dominance information of one individual can be updated by several processes at the same time. Therefore, to preserve the correctness of parallel BDC, it is necessary to update the dominance of every individual (row of *MD*) by atomic write operations, which produces the serialization of the concurrent accesses. However, for the parallel UDC version, the comparisons of several individuals can be executed in parallel and they can update their dominance information in different rows of *MD*, without conflicting writings.

According to the previous considerations, we only address the following parallelizations of the FNDS-UDC algorithm: (1) multicore based on Pthreads; (2) GPU based on CUDA; and (3) hybrid multicore-GPU, which combines both parallel architectures. These implementations will be chosen by researchers according to the dimensions of their particular problems and/or their computational platforms.

5.1 Multicore implementation

The proposed multicore implementation is based on Pthread (Posix-Threads) interface [3] and ANSI C to exploit the parallelism of the cores available on the platform. The multicore version of FNDS-UDC is described in Algorithm 3.

At the beginning, a number of threads (less than or equal to the number of CPU cores) is initialized. Thus, each thread runs on a CPU core. Each thread computes the dominance comparison of a subset of individuals in parallel and its output is saved in the corresponding rows of *MD*, without writing conflicts among threads and synchronization points. P is the input for all threads to compute the dominance of the corresponding subpopulation. The information in the cache memory is useful for all threads. Therefore, the accesses of threads to P are optimized.

The total workload is proportional to p (population size), since it determines the number of comparisons and memory accesses that consume the most of the FNDS runtime. A static load distribution among threads has been performed. Thus, each thread classifies a sub-population of similar size because the outer for-loop of UDC has been homogeneously distributed among the threads.

[5] http://www.mathworks.com/matlabcentral/fileexchange/31166-ngpm-a-nsga-ii-program-in-matlab-v1-4.

When the parallel Phase 1 of UDC finishes a synchronization point among all threads is necessary to guarantee that the following phases start after all rows of *MD* were computed. Then, the threads join and only one is maintained active to continue the computation of Phases 2 and 3. The serialization of these phases has not a high impact in the performance, according to the profiling analysed in Sect. 6, Table 2, since they have a low computational burden.

Algorithm 3 Multicore implementation of FNDS-UDC.

```
Input:
    $n$: the dimension of the search space,
    $p$: the number of individuals,
    $v$: the number of objective functions,
    $P_{(p\times(n+v+1))}$: population,
    $nTh$: number of threads.
Output:
    $F$: the front matrix,
    $T$: the rank of each individual.
1: for $id \leftarrow 1$ to $nTh$ do
2:     Create $id$ thread
3:     $init_{id} = id * ceil(p/nTh)$
4:     $end_{id} = init_{id} + ceil(p/nTh)$
   Parallel Phase 1: Unidirectional Dominance Comparison (UDC)
5: for $i \leftarrow init_{id}$ to $end_{id}$ do        ▶ Every thread computes a subset of MD rows
6:     $D_i \leftarrow \{\emptyset\}$
7:     $\#M_i \leftarrow 0$
8:     for $j \leftarrow 1$ to $p$ do
9:         for $k \leftarrow 1$ to $v$ do
10:            Check dominance between individuals $P_i$ and $P_j$ for the objective $k$
11:        if $P_i \succ P_j$ then
12:            $D_i \leftarrow D_i \cup \{j\}$        ▶ Add $P_j$ to the set of individuals dominated by $P_i$
13:        else if $P_j \succ P_i$ then
14:            $\#M_i \leftarrow \#M_i + 1$        ▶ Increase dominators counter for $P_i$
15: Synchronization and join of threads        ▶ Only one thread is computing
   Sequential Phase 2: Assign front to the individuals of the $1^{st}$ front (see Alg. 1)
   Sequential Phase 3: Front assignment (see Alg. 1)
16: return $R$, $T$        ▶ Return the Pareto Dominance Ranking
```

5.2 GPU implementation

The keys of the GPU computing are described in this section for a better understanding of the proposed FNDS version on a GPU, which is detailed in Sect. 5.2.2.

5.2.1 CUDA programming model

GPU platforms are appropriated to compute procedures that involve launching large number of threads in parallel with the same sequence of instructions over several input data. Thus, the programmer can consider the GPU as a set of SIMT (Single Instruction, Multiple Threads) multiprocessors [4].

The parallel code executed on a GPU is called kernel [26]. GPUs are composed of hundreds of cores that can collectively run thousands of GPU-threads. Each core, called Scalar Processor (SP), belongs to a set of multiprocessors units called Streaming Multiprocessors

(SM or SMX) that compose the device. The SPs in a SMX share resources such as registers and memory. The on-chip shared memory allows the parallel tasks running on these cores to access the data without using the system memory bus [26].

To develop codes for NVIDIA GPUs with CUDA, the programmer has to take into account several architectural characteristics, such as the topology of the multiprocessors and the management of the memory hierarchy. For the execution of the program, the CPU (called host in CUDA) performs a succession of kernel invocations to the device. The input/output data to/from the GPU kernels are communicated between CPU and GPU memories by the PCI Express bus. These communications reduce the performance.

Each kernel is executed as a batch of threads organized as a grid of thread blocks. The execution of every thread block is assigned to every SMX. Moreover, every block is composed by several groups of 32 threads, called warps. The occupancy is defined as the ratio between the number of active warps and the theoretical maximum number of active warps. Therefore, the higher the occupancy, the higher the exploitation. The occupancy depends on the threads block size (BS), which is defined by the programmer. The maximum instruction throughput is achieved when all threads of the same warp execute the same instruction sequence. Threads of the same warp diverge when they follow different execution paths due to control flow statements. If this occurs, the different executions paths have to be serialized, increasing the total number of instructions executed by the warp [26].

There are several kinds of memory available on GPUs, with different access time and sizes that constitute the memory hierarchy. The exploited memory bandwidth can vary by an order of magnitude depending on the access pattern for each type of memory. There is a parallel memory interface between the global memory and every SMX of the GPU. It is necessary to maintain the coalescent and aligned memory access by the threads in order to improve the bandwidth. Hence, the data structure and the ordering of the chosen data accesses in an algorithm may affect significantly to the performance.

Summarising, in order to optimise the performance of the GPU, several good practices have to be considered by the CUDA programmer: (1) to balance the computation among the sets of threads, (2) to optimise the data access through the memory hierarchy, (3) to define enough number of threads to maximize the occupancy of the GPU, (4) to avoid the serialization of the kernel due to control instructions, and (5) to minimize the data transfers between CPU and GPU.

5.2.2 GPU implementation of FNDS

Algorithms 4 and 5 show the GPU version of FNDS-UDC. They are CUDA pseudocodes for the CPU host and GPU (kernel), respectively. Only UDC is parallelized on GPU and Phases 2 and 3 are sequentially computed on the CPU, as it was done for the multicore version. Algorithm 4 starts its execution on CPU and activates the GPU (denoted by device d). Then, it allocates memory for the input ($d\_V_{(p\times v)}$) and output ($d\_MD_{(p\times(p+2))}$) data on GPU (Alg. 4, line 3). The input data for GPU kernel is the matrix of objective vectors, V (see Fig. 3). Then, GPU_UDC kernel is launched (Alg. 5) to compute UDC, where p threads are activated. Each GPU thread computes one individual (tid), i.e., the tid^{th} row ($d\_\#M_{tid}$, $d\_\#D_{tid}$ and d_D_{tid}) of the masters-dominated (d_MD) matrix. When all GPU threads finish, d_MD stores the output data which will be copied from the GPU to CPU.

The values of $d\_\#M$ and $d\_\#D$ will be copied from GPU to $\#M$ and $\#D$ on CPU. However, in order to communicate only the useful information of d_D, $d\_\#D_{max} = \max\{d\_\#D\}$ is

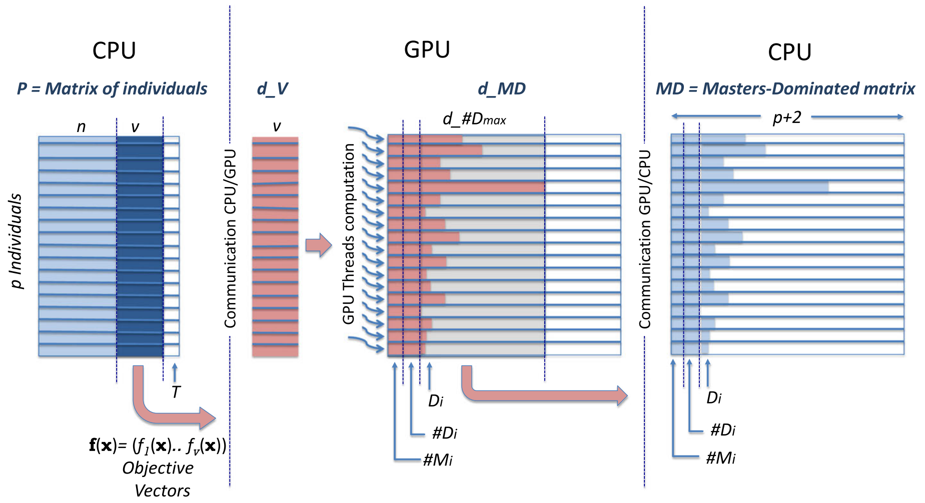

Fig. 3 Data structures which are involved in the GPU kernel to compute the individual dominance

computed in parallel on GPU with a CUDA Thrust routine[6]. Only the first $d\_\#D_{max}$ columns of d_D store useful information and will be copied from d_D on GPU to D on CPU (Alg. 4, line 7). Therefore, the selected columns of d_MD are copied to MD in just one step. After that, the computation of FNDS-UDC continues (Phases 2 and 3) on the CPU. Figure 3 shows a sketch of the data structures and communications between CPU/GPU, which are involved in the UDC kernel.

Giving an outline of the GPU version of FNDS-UDC, the GPU is in charge of the UDC computation because this process usually exhibits enough parallelism to squeeze such kind of architecture. UDC kernel has been optimized since we have considered the good practices of GPU programming, mentioned in Sect. 5.2.1. The load balance is achieved because each GPU thread revises the dominance of an individual and therefore, the work load is similar for every GPU warp of threads. The memory access to the input data is coalescent, since the structure d_V is stored in column-major order. This way, consecutive threads, i and $i+1$, access to consecutive data in memory to revise the dominance related to the k objective, $d\_V[\ ,k+i]$ and $d\_V[\ ,k+i+1]$. The coalescent reading of d_V has a strong impact in the kernel performance since it is dominated by the access to the objective vectors. It is very relevant to supply enough workload to maintain active the GPU resources. Therefore, the UDC kernel will be accelerated on GPU only if the population size p is large enough. The minimal population size to optimize UDC on GPU mainly depends on the number of SMX of the specific GPU. UDC computation is usually expressed using control instructions, which can serialize the threads execution on GPU. To help the CUDA compiler to generate object code that avoids the serialization of threads, the dominance checking between individuals has been coded using Boolean operators instead of control instructions (see Alg. 5, lines 6–8). For instance, line 6 actually computes for every objective i: $ObjEq \leftarrow ObjEq + (d\_V(tid,i) == d\_V(j,i))$ instead of $if\ (d\_V(tid,i) == d\_V(j,i))\ then\ ObjEq \leftarrow ObjEq + 1$. The communications between CPU and GPU play a relevant role in the GPU version of FNDS-UDC since only UDC process is computed on GPU and its input and output data have to be communicated

[6] http://docs.nvidia.com/cuda/thrust/index.html.

between CPU and GPU. As described above, the size of input/output data is large if the MOO problem deals with large populations and many objectives. Thus, only the sub-matrices that are useful for UDC computation are communicated.

Algorithm 4 GPU implementation of FNDS-UDC.

```
Input:
    $d$: device to use,
    $n$: the dimension of the search space,
    $p$: the number of individuals,
    $v$: the number of objective functions,
    $P_{(p\times(n+v+1))}$: population.
Output:
    $F$: the front matrix,
    $T$: the rank of each individual.
 1: Initialize the GPU device, $d$
 2: Allocate memory on GPU for $d\_V_{(p\times v)}$ and $d\_MD_{(p\times(p+2))}$
 3: Copy objective vectors from CPU to GPU                     ▶ $d\_V \leftarrow V \in P$
 4: Launch the GPU kernel to compute $d\_MD$                   ▶ See Alg. 5
 5: Synchronization point
 6: Launch the computation of $d\_\#D_{max}$ on GPU.
 7: Copy $d\_MD$ on GPU to $MD$ on CPU     ▶ $d\_\#M$, $d\_\#D$ and $d\_\#D_{max}$ columns of $d\_D$
 8: Deallocate memory for $d\_MD$ and $d\_V$
    CPU Sequential Phase 2: Assign front to the individuals of the $1^{st}$ front (see Alg. 1)
    CPU Sequential Phase 3: Front assignment (see Alg. 1)
 9: return $R$, $T$                             ▶ Return the Pareto Dominance Ranking
```

Algorithm 5 GPU_UDC kernel.

```
Input:
    $d$: device to use,
    $p$: the number of individuals,
    $v$: the number of objective functions,
    $d\_V$: the individuals' objective matrix.
Output: Updating masters-dominated matrix $d\_MD = [\#M|\#D|D]_{(p\times(p+2))}$.
 1: $tid \leftarrow blockDim.x \cdot blockIdx.x + threadIdx.x$     ▶ Identifier of every thread.
 2: if $tid < p$ then
 3:     $D_{tid} \leftarrow \{\emptyset\}$                  ▶ Set of individuals dominated by $P_{tid}$.
 4:     $\#M_{tid} \leftarrow 0$                            ▶ The number of masters of $P_{tid}$.
 5:     for $j \leftarrow 1$ to $p$ do
 6:         $ObjEq = |\{i : d\_V(tid,i) = d\_V(j,i),\ i = 1,\dots,\ v\}|$
 7:         $ObjLt = |\{i : d\_V(tid,i) < d\_V(j,i),\ i = 1,\dots,\ v\}|$
 8:         $ObjGt = |\{i : d\_V(tid,i) > d\_V(j,i),\ i = 1,\dots,\ v\}|$
 9:         if $ObjLt = 0$ and $ObjEq \neq v$ then                 ▶ $P_{tid} \succ P_j$
10:             $D_{tid} \leftarrow D_{tid} \cup \{j\}$
11:         else if $ObjGT = 0$ and $ObjEq \neq v$ then            ▶ $P_j \succ P_{tid}$
12:             $\#M_{tid} \leftarrow \#M_{tid} + 1$
    return $d\_MD$                 ▶ Return the matrix of masters-dominated individuals.
```

5.3 Hybrid GPU-Multicore implementation

Hybrid GPU-Multicore implementation has been designed to exploit CPU cores and one GPU device to accelerate the UDC phase in the FNDS-UDC algorithm (see Algorithm 1).

This implementation is based on Pthreads and CUDA. The workload balance of the UDC phase between GPU and CPU is essential to achieve a good performance. The goal of the load balancing is to achieve a similar runtime in CPU and GPU.

The hybrid GPU-Multicore code launches several CPU-threads, each one running on a CPU-core. The p iterations of UDC (i-loop of Alg. 1, line 1) are statically distributed among CPU-threads and GPU. Each CPU-thread and GPU computes a different subset of iterations of the loop ($chunk$).

One CPU-thread has additional work: to start the GPU device, to call the CUDA kernel and to handle the communication between CPU and GPU. Memory allocation for d_V and d_MD at GPU is done in the same way as in the GPU implementation. However, d_MD has only $|chunck|$ rows, since the GPU only computes the masters-dominated of a $chunk$ of individuals.

At the end of the UDM phase, the CPU and the GPU threads are synchronized to join the results. Then, Phases 2 and 3 are executed by just one CPU-thread.

The number of individuals to evaluate can be distributed (partition of the i-loop) among CPU and GPU according to their computational power [27]. This static distribution of workload is based on a previous estimation of the relative performance (r) of both devices which is estimated as follows: $r = t_{CPU}(w)/t_{GPU}(w)$, where $t_{CPU}(w)$ and $t_{GPU}(w)$ represent the runtime of the CPU and the GPU to compute a given large enough workload of size w, respectively. In this way, the GPU computes $rp/(r+1)$ individuals (iterations of i-loop) and the CPU $p/(r+1)$ ones. When $r \approx 1$, the ratio of computational workload and size of the problem for both computational devices is similar and the maximum relative speed-up of the hybrid implementation over the fastest (GPU or multicore) implementation is 2.

Therefore, a benchmarking to determine a good r value has to be executed before the FNDS-UDC algorithm.

5.4 Support routine for determining the fastest FNDS version

FNDS is included in iterative procedures for solving MOO problems by EMO algorithms. The performance achieved by every FNDS implementation depends on the problem size (p and v) and the resources of the computational platform. Therefore, we provide an off-line benchmarking routine for helping users to select the fastest version, according to their particular MOO problems and platforms. Algorithm 6 shows this benchmarking routine. Firstly, it explores the runtime of sequential, multicore and GPU executions. Secondly, the fastest CPU version is considered to compute the r parameter, in order to configure the hybrid version. Then, the runtime for the hybrid version is also evaluated. Finally, the routine identifies the fastest implementation according to the application context. Additionally, the configuration parameter r is also returned, since it can be useful to configure the hybrid version, when it is the fastest.

6 Experimental Results

In this section, the performance of the parallel implementations of the FNDS-UDC algorithm (see Alg. 1) is experimentally investigated.

Parallel implementations of the FNDS-UDC algorithm are useful for EMO algorithms with large number of objectives. DTLZ2 test problem has been considered, because the number of objective functions can be defined by the user [9,17]. DTLZ2 problem was specially designed for evaluating multi-objective algorithms in this context. Moreover, it can be configured to

Algorithm 6 Routine to determine the fastest FNDS version.

```
Input:
    $P$: population,
    $v$: number of objective functions,
    $d$: GPU device to use,
    $nc$: number of CPU cores.
Output:
    $ci$: configuration index
      – $ci \in [1, nc]$: CPU with $ci$ threads,
      – $ci = nc + 1$: GPU,
      – $ci = nc + 2$: hybrid,
    $r$: the relative performance,
    $nTh$: number of CPU-threads in the hybrid version.
1: $t_1$:= time of sequential Alg. 2                                  ▶ Wall-clock time
2: for $c \leftarrow 2$ to $nc$ do
3:     $t_c$:= time of Alg. 3 with $c$ threads
4: $t_{CPU} \leftarrow \min_{i\in[1,nc]} \{t_i\}$
5: $nTh \leftarrow \arg\min_{i\in[1,nc]}\{t_i\}$
6: $t_{GPU} \leftarrow t_{c+1} \leftarrow$ time of GPU Alg. 4
7: $r = t_{CPU}/t_{GPU}$
8: $T_{hybrid} \leftarrow t_{c+2} \leftarrow$ time of hybrid with $nTh$ CPU-Threads and GPU        ▶ See Sect. 5.3
9: $ci \leftarrow \arg\min_{i\in[1,nc+2]} \{t_i\}$
10: return $ci$, $r$ and $nTh$
```

have high computational demands by establishing a large number of objectives and variables. The formulation of the problem is as follows:

$$
\begin{aligned}
&\min f_1(\mathbf{x}) = (1+g(x))\prod_{i=1}^{v-1} cos\left(\tfrac{x_i\pi}{2}\right)\\
&\min f_2(\mathbf{x}) = (1+g(x))sin\left(\tfrac{x_{v-1}\pi}{2}\right)\prod_{i=1}^{v-2} cos\left(\tfrac{x_i\pi}{2}\right)\\
&\dots\\
&\min f_l(\mathbf{x}) = (1+g(x))sin\left(\tfrac{x_{v-l+1}\pi}{2}\right)\prod_{i=1}^{v-l} cos\left(\tfrac{x_i\pi}{2}\right)\\
&\dots\\
&\min f_v(\mathbf{x}) = (1+g(x))sin\left(\tfrac{x_1\pi}{2}\right)
\end{aligned}
\tag{2}
$$

where $g(\mathbf{x}) = \sum_{i=v}^{n}(x_i - 0.5)^2$, $x_i \in [0, 1]$. The number of variables n is selected according to the equation $n = v + k - 1$, with a suggested value of $k = 10$.

Instances have been created by generating p decision vectors randomly over the search space. The size of p, v and n vary for every test case (see Table 1(a)). The notation for each instance of the problem uses a, b, c or d for 5000, 10,000, 20,000 or 30,000 population sizes, respectively, followed by $\_v$. Additionally, Table 1(a) also includes the memory requirements on CPU and GPU (in GBytes) for the proposed FNDS instances of the problem.

Two computational architectures have been considered in the experiments:

A_1 : Bullx R424-E3 Intel Xeon E5 2650 (16 CPU-cores and 8GB RAM) with one Tesla M2070 (Fermi) GPU.

A_2 : Bullx R421-E4 Intel Xeon E5 2620v2 (12 CPU-cores and 64GB RAM) with two NVIDIA K80 (Kepler GK210) GPUs.

Table 1 Characteristics of the test functions and their corresponding memory requirements on CPU and GPU (in GB) (a); and the GPU devices (b)

Test case	p	v	n	CPU (GB)	GPU (GB)		M2070 (A_1)	K80 (A_2)
(a)						(b)		
a_05	5000	5	14	0.14	0.09	Peak performance	0.51	1.87
a_10	5000	10	19	0.14	0.09	(double prec.) (TFlops)		
a_15	5000	15	24	0.14	0.09	Peak performance	1.03	5.6
b_05	10,000	5	14	0.56	0.37	(simple prec.) (TFlops)		
b_10	10,000	10	19	0.56	0.37	Device memory (GB)	5.2	11.2
b_15	10,000	15	24	0.56	0.37	Clock rate (GHz)	1.2	0.82
c_05	20,000	5	14	2.24	1.49	Memory bandwidth	150	480
c_10	20,000	10	19	2.24	1.49	(GBytes/s)		
c_15	20,000	15	24	2.24	1.49	Multiprocessors	14	13
d_10	30,000	10	19	5.04	3.36	CUDA cores	448	4992
d_15	30,000	15	24	5.04	3.36	Compute Capability	2.0	3.7

The characteristics of the GPU devices are given in Table 1(b). The Table shows that the required memory for the instances of the problem on CPU and GPU fit loosely in the memory of the probed architectures.

FNDS-UDC (see Alg. 1) has been considered for all parallel implementations (multicore, GPU and hybrid). The use of the data structures described in Sect. 3 improves the memory management for FNDS. In order to analyze the impact of the improvement in the performance, two sequential versions of FNDS-BDC have been analysed: a FNDS-BDC Matlab version without optimized data structures (SeqMatlab), which is widely used[7], and a C routine to compute FNDS-BDC procedure with the improved data structures (SeqC).

SeqMatlab has been coded and evaluated in Matlab 2016a. The other versions have been coded C++, Pthreads and CUDA 6.5, as well as *gcc* and *nvcc* compilers with $-O2$ optimization option. Since the decision vectors are randomly generated for each execution, measurements have been repeated until the sample mean has a confidence level of 95%. The average executions time is given as result.

A profiling of the SeqC FNDS-BDC (see Alg. 2) on A_1 is presented in Table 2. This table shows the runtime (in seconds) on a single CPU-thread for the phases of FNDS-BDC: "Phase 1: BDC", "Phase 2" and "Phase 3" columns. "Phase 1: BDC" running time has been used as the best sequential time to calculate the parallel speed-up. The percentage of runtime of Phase 1 with respect to the total runtime is shown between parentheses. "Others" column shows the time invested by Matlab to call to the SeqC FNDS-BDC implementation. It is shown that most of the runtime of FNDS-BDC is related to the bidirectional dominance comparison. The results presented in Table 2 can be extrapolated to FNDS-UDC, where the sequential UDC consumes more time than BDC. This profiling justifies the decision to parallelize only the dominance comparison phase (Phase 1 of Algorithm 1). Table 2 also shows the strong increment of the runtime as the size of the problem increases, in terms of the values p and v. An analogous profiling was carried out on A_2 with similar conclusions.

[7] http://www.mathworks.com/matlabcentral/fileexchange/31166-ngpm-a-nsga-ii-program-in-matlab-v1-4.

Table 2 Profiling of the SeqC FNDS-BDC implementation on A_1

	Phase 1: BDC	Phase 2	Phase 3	Others
a_05	1.21E+00 (95.17 %)	6.40E−05	5.09E−02	1.02E−02
a_10	2.24E+00 (98.37 %)	1.56E−04	1.91E−02	1.77E−02
a_15	3.29E+00 (98.84 %)	6.62E−04	1.43E−02	2.37E−02
b_05	4.87E+00 (94.33 %)	9.80E−05	2.89E−01	3.42E−03
b_10	8.96E+00 (98.32 %)	2.54E−04	9.78E−02	5.50E−02
b_15	1.33E+01 (98.72 %)	3.01E−04	6.34E−02	1.08E−01
c_05	1.94E+01 (92.59 %)	1.96E−04	1.48E−01	7.68E−02
c_10	3.55E+01 (98.19 %)	4.27E−04	4.66E−01	1.89E−01
c_15	5.25E+01 (98.85 %)	4.99E−04	3.06E−01	3.06E−01
d_10	8.35E+01 (98.39 %)	5.95E−04	5.12E−01	1.97E−01
d_15	1.25E+02 (98.70 %)	6.98E−04	8.02E−01	8.45E−01

Wall-clock time in seconds for all implementations of FNDS procedures on A_1 and A_2 architectures are shown in Table 3 and represented in Figure 4. There are two sequential versions (referred as SeqMatlab and SeqC and executed on a CPU-core), parallel versions with 2–16 or 2–12 CPU-cores, GPU and hybrid (which exploit both CPU-cores and the GPU). In SeqMatlab implementation on A_1 (A_2), executions time varies from 10 s (5 s), for $p = 5000$, to more than 469 s (224 s), for $p = 30,000$. The set of execution time of SeqC outperforms SeqMatlab in a factor which ranges from 3.4× (3.1×) to 7.9× (8.7×) on A_1 (A_2). The improvement is not only due to the well known better performance of C codes than Matlab ones but also to the use of structures defined in Sect. 3.

All the multicore implementations outperform the SeqC implementation. In fact, the efficiency of the multicore implementation improves when the size of the problem increases in terms of population size and number of objectives. The runtime for SeqC and 2CPU is very close since the BDC phase used in SeqC (FNDS-BDC) computes nearly the half of comparisons of the UDC phase in FNDS-UDC.

In the GPU implementation, one key parameter is the occupancy of the GPU. High occupancy levels are reached if an optimal value of threads Block Size (BS) is used. Thus, an experimental investigation has been carried out to analyze the influence of the BS on the performance of the algorithm on the GPU. The possible values of BS are powers of two, and vary from 16 to 1024. The experimental results have shown that only for $BS = 256$ and 512 the GPU kernels reach values close to 100 % of occupancy of the GPU. Executions time of the GPU implementation are presented in Fig. 4 and Table 3 when the optimal BS (256 or 512) is used. Additionally, the percentage of the GPU-CPU data transfer time with respect to the runtime of FNDS on a GPU is shown between parenthesis in the “GPU” column of Table 3. This implementation can appropriately exploit the GPU when the number of individuals is large enough, according to the available number of Stream Multiprocessors and cores in the GPU. Firstly, focusing our attention on A_1 of Table 3, results point out that the performance of the GPU version can be competitive to the multicore version if the number of objectives is large enough ($v = 10, 15$) and/or the CPUs number is reduced (1–8). Secondly, focusing our attention on A_2, results point out that the performance of the GPU version is better than the multicore version even for 12 CPU-cores. It is due to the fact that the difference between the computational power of the GPU and the CPU is smaller on A_1 than on A_2. The costs

Table 3 Wall-clock time in seconds on architectures A_1 and A_2 for implementations: SeqMatlab (SeqM column), SeqC, multicore (with 2, 4, 8 and 16(12) CPUs), GPU and hybrid multicore-GPU (Hyb column)

	A_1								
	SeqM	SeqC	2CPU	4CPU	8CPU	16CPU	GPU	Hyb	r
a_05	10.02	1.27	1.12	0.58	0.31	0.18	0.25 (15.7 %)	0.18	1.1
a_10	10.15	2.27	2.03	1.02	0.51	0.27	0.23 (15.4 %)	0.19	1.2
a_15	11.57	3.33	3.00	1.51	0.76	0.39	0.23 (10.1 %)	0.20	3.0
b_05	40.59	5.16	4.53	2.40	1.32	0.78	0.91 (28.0 %)	0.67	1.1
b_10	41.66	9.12	8.18	4.12	2.11	1.10	0.77 (19.8 %)	0.69	1.8
b_15	53.10	13.44	12.08	6.07	3.05	1.56	0.89 (14.6 %)	0.76	4.4
c_05	167.11	20.97	18.43	9.83	5.51	3.37	3.77 (25.2 %)	2.74	1.1
c_10	163.56	36.19	32.54	16.47	8.44	4.42	2.95 (21.9 %)	2.55	1.8
c_15	205.25	53.14	47.97	24.10	12.20	6.25	3.34 (16.4 %)	2.89	4.5
d_10	381.29	84.88	76.42	38.48	19.62	10.40	7.71 (18.9 %)	5.16	2.2
d_15	469.05	127.07	114.49	57.68	29.18	14.96	7.94 (15.4 %)	6.81	4.8
	A_2								
	SeqM	SeqC	2CPU	4CPU	8CPU	12CPU	GPU	Hyb	r
a_05	5.87	0.78	0.93	0.54	0.30	0.22	0.09 (31.1 %)	0.07	3.0
a_10	4.78	1.41	1.48	0.90	0.51	0.33	0.08 (30.5 %)	0.10	4.4
a_15	6.09	1.94	2.04	1.18	0.69	0.53	0.08 (18.4 %)	0.07	16.0
b_05	24.05	2.91	3.10	1.96	1.14	0.76	0.35 (39.0 %)	0.23	2.3
b_10	19.43	4.93	5.14	2.80	1.70	1.19	0.23 (32.9 %)	0.19	12.0
b_15	24.39	7.04	7.38	4.64	2.30	1.50	0.23 (24.6 %)	0.22	14.0
c_05	96.62	11.12	11.15	6.12	3.76	2.86	1.57 (26.3 %)	0.78	2.1
c_10	78.46	18.79	20.54	11.20	6.32	4.36	1.01 (25.9 %)	0.93	4.7
c_15	98.28	27.29	29.37	15.09	8.74	6.16	0.94 (23.2 %)	0.88	10.0
d_10	178.07	42.52	45.04	23.55	13.42	9.61	2.20 (22.7 %)	1.84	7.0
d_15	224.13	62.04	65.64	34.31	19.45	13.90	2.17 (22.4 %)	1.96	10.5

The percentage of the GPU-CPU data transfer time with respect to the runtime of FNDS on a GPU is shown between parenthesis. The value of the relative performance GPU-CPU, r, is also shown

of the data transfer between GPU and CPU (parenthesis in the "GPU" column of Table 3) are relevant with respect to the total wall-clock time of the FNDS on a GPU. These costs reach their maximum values when $v = 5$ on A_1 and A_2 (28% and 39%, respectively), as the smaller the number of objective functions, the higher the possibility of few individuals dominate large subpopulations. It means that $\#D_{max}$ (see Sect. 5.2.2) has a high value for $v = 5$. Therefore, the data transfer between GPU and CPU results in a longer wall-clock time, as can be seen in Fig. 4.

The Hybrid versions on architectures A_1 and A_2 are shown in the Fig. 4 and the Table 3. These implementations are based on a previous benchmarking process to determine a static workload balancing according to the performance of 15 (11) CPU-cores on A_1 (A_2) and the GPU. "r" column shows the estimation of the relative performance r for each instance. In all the experiments of the hybrid version, the total number of CPU-threads was obtained using the training procedure described in Alg. 6, obtaining 16 CPU-cores on A_1 and 12 CPU-cores

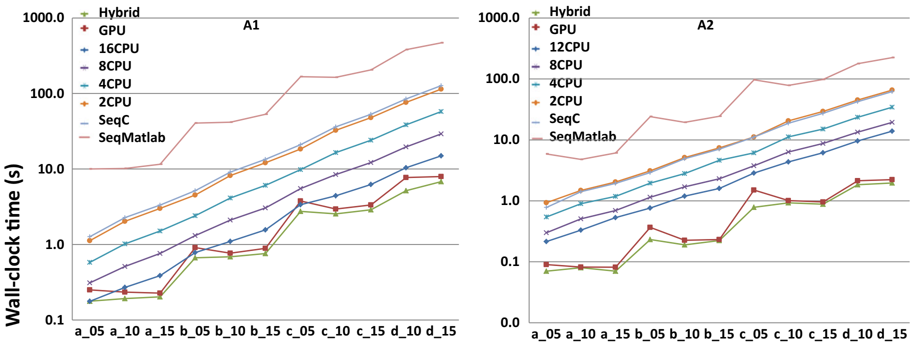

Fig. 4 Wall-clock time in seconds on platforms A_1 and A_2 for implementations: SeqMatlab, SeqC, multicore (with 2, 4, 8 and 16(12) CPUs), GPU and hybrid multicore-GPU

on A_2. Hybrid implementation in A_1 (A_2) can take advantage of the parallel computation of 15 (11) CPU-cores, in combination with a GPU architecture to enhance the computational efficiency of the parallelization of Algorithm 1. According to last considerations of Sect. 5.3, the hybrid version always obtains similar or better performance than the fastest multicore or GPU version. The advantage of the hybrid version versus the GPU implementation decreases as the value of v increases, since an increment in v causes a negligible variation on the performance of the hybrid version. However, for the GPU version, when v increases the data transfer penalties have less impact which results in a better performance. It can be observed for instances of the problem with $v = 15$ ($r \gg 1$), the performance of the GPU is similar to the hybrid version.

Summarizing, the obtained results show that the values of the runtime of FNDS algorithms strongly increase with the population (p) and/or the number of objectives (v). For the evaluated problem instances, the sequential runtime on A_1 (A_2) ranges from 1.3 (0.8) sec. to 127.1 (62.0) sec. for the optimized sequential version (FNDS-BDC) coded on C. Parallel versions of the FNDS-UDC algorithm are interesting alternatives to reduce the computational wall-clock time. The selected parallel version will depend on the available hardware and the number of objective functions. On A_1, the multicore version is better than the GPU version only when the number of objective functions is the smallest (i.e., a_05, b_05 and c_05 instances of the problem). On A_2, the GPU version is faster than the multicore version. If a GPU with CUDA support is available and the population and objectives are large enough, the GPU or Hybrid versions are good options to reduce the wall-clock time of FNDS algorithms. Results on A_1 (A_2) show that the runtime of the fastest version ranges from 0.18 – 6.81 s. (0.07 – 1.96 s). The best acceleration factor on A_1 (A_2) has been obtained for the most computationally expensive instance of the problem (d_15), where the runtime of SeqC, 127.07 s (62.04 s), has been reduced by a factor of 18.6× (31.6×) with respect to the fastest version, 6.8 s (1.96 s).

7 Conclusions

In this paper, several implementations of the non-dominated sorting procedure, one of the most time consuming procedures in several state-of-the-art EMO algorithms, have been proposed and experimentally evaluated: a multicore implementation based on Pthreads, a GPU implementation based on CUDA and a hybrid implementation as the combination of both. Results show that the number of objectives, the size of the population and the characteristics of the available platforms are the key parameters to establish the best implementation in terms of performance. To ease the selection of the most suitable FNDS version a supplementary training procedure has also been developed. This way, scientists can use the versions of FNDS algorithms described and analyzed in this work to extend their EMO algorithms for solving multi-objective optimization problems.

The optimized FNDS implementations are available on-line[8] to be called from Matlab or C++ routines, in combination to the different considered parallel architectures.

However, further work is needed towards the evaluation of these new routines on multi-objective optimization algorithms. Another future research line is related to the development of the multi-GPU version and the measurement of the energy efficiency of these proposals to select the most suitable parallel version based on performance and energy efficiency criteria.

References

1. Aguirre, H., Oyama, A., Tanaka, K.: Adaptive ε-sampling and ε-hood for evolutionary many-objective optimization. In: Evolutionary Multi-Criterion Optimization, *LNCS*, vol. 7811, pp. 322–336. Springer Berlin (2013)
2. Arrondo, A., Redondo, J., Fernández, J., Ortigosa, P.: Parallelization of a non-linear multi-objective optimization algorithm: application to a location problem. Appl. Math. Comput. **255**, 114–124 (2015)
3. Butenhof, D.: Programming with POSIX Threads. Professional Computing Series. Addison-Wesley, Boston (1997)
4. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2013)
5. Corne, D.W., Jerram, N.R., Knowles, J.D., Oates, M.J.: PESA-II: Region-based selection in evolutionary multiobjective optimization. In: GECCO, pp. 283–290 (2001)
6. Deb, K.: An efficient constraint handling method for genetic algorithms. Comput. Method Appl. M. **186**(2), 311–338 (2000)
7. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE T. Evolut. Comput. **6**(2), 182–197 (2002)
8. Deb, K., Sundar, J., Udaya Bhaskara Rao, N., Chaudhuri, S.: Reference point based multi-objective optimization using evolutionary algorithms. Int. J. Comput. Intell. Res. **2**(3), 273–286 (2006)
9. Deb, K., Thiele, L., Laumanns, M., Zitzler, E.: Scalable multi-objective optimization test problems. In: CEC, vol. 1, pp. 825–830 (2002)
10. Deb, K., Tiwari, S.: Omni-optimizer: a procedure for single and multi-objective optimization. In: Evolutionary Multi-Criterion Optimization, pp. 47–61. Springer Berlin (2005)
11. Du, J., Cai, Z.: A sorting based algorithm for finding a non-dominated set in multi-objective optimization. In: ICNC, vol. 4, pp. 436–440. IEEE (2007)
12. Fang, H., Wang, Q., Tu, Y.C., Horstemeyer, M.F.: An efficient non-dominated sorting method for evolutionary algorithms. Evol. Comput. **16**(3), 355–384 (2008)
13. Filatovas, E., Kurasova, O., Sindhya, K.: Reference point based multi-objective optimization using evolutionary algorithms. Informatica **26**(1), 33–50 (2015)
14. Fonseca, C.M., Fleming, P.J., et al.: Genetic algorithms for multiobjective optimization: Formulationdiscussion and generalization. In: ICGA, vol. 93, pp. 416–423. Morgan Kaufmann Publishers Inc., San Francisco (1993)
15. Gupta, S., Tan, G.: A scalable parallel implementation of Evolutionary Algorithms for Multi-Objective optimization on GPUs. In: CEC, pp. 1567–1574. IEEE (2015)
16. Hennessy, J., Patterson, D.: Computer Architecture - A Quantitative Approach, 5th edn. Morgan Kaufmann, Burlington (2012)
17. Huband, S., Hingston, P., Barone, L., While, L.: A review of multiobjective test problems and a scalable test problem toolkit. IEEE T. Evolut. Comput. **10**(5), 477–506 (2006)

[8] https://sites.google.com/site/hpcoptimizationproblems/FNDS.

18. Ishibuchi, H., Sakane, Y., Tsukamoto, N., Nojima, Y.: Evolutionary many-objective optimization by NSGA-II and MOEA/D with large populations. In: IEE SMC, pp. 1758–1763. IEEE (2009)
19. Jensen, M.T.: Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms. IEEE T. Evolut. Comput. **7**(5), 503–515 (2003)
20. Knowles, J.D., Corne, D.W.: Approximating the non-dominated front using the Pareto archived evolution strategy. Evol. Comput. **8**(2), 149–172 (2000)
21. Lančinskas, A., Žilinskas, J.: Solution of multi-objective competitive facility location problems using parallel NSGA-II on large scale computing systems. In: PARA, pp. 422–433. Springer Berlin (2013)
22. Laumanns, M., Thiele, L., Deb, K., Zitzler, E.: Combining convergence and diversity in evolutionary multiobjective optimization. Evol. Comput. **10**(3), 263–282 (2002)
23. McClymont, K., Keedwell, E.: Deductive sort and climbing sort: New methods for non-dominated sorting. Evol. Comput. **20**(1), 1–26 (2012)
24. Miettinen, K.: Nonlinear Multiobjective Optimization. Springer, Berlin (1999)
25. Munshi, A., Gaster, B., Mattson, T.G., Fung, J., Ginsburg, D.: OpenCL Programming Guide, 1st edn. Addison-Wesley Professional, Boston (2011)
26. NVIDIA Corporation: CUDA C PROGRAMMING GUIDE PG-02829-001_v7.5 (2015). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide
27. Ortega, G., Lobera, J., García, I., Arroyo, M.P., Garzón, E.M.: Parallel resolution of the 3D Helmholtz equation based on multi-graphics processing unit clusters. Concurrency-Pract. Ex. **27**(13), 3205–3219 (2015)
28. Sharma, D., Collet, P.: GPGPU-compatible archive based stochastic ranking evolutionary algorithm (g-asrea) for multi-objective optimization. In: Parallel Problem Solving from Nature, PPSN XI, pp. 111–120. Springer Berlin (2010)
29. Sharma, D., Collet, P.: Implementation techniques for massively parallel multi-objective optimization. In: Massively Parallel Evolutionary Computation on GPGPUs, pp. 267–286. Springer Berlin (2013)
30. Shi, C., Chen, M., Shi, Z.: A fast nondominated sorting algorithm. In: ICNN, vol. 3, pp. 1605–1610. IEEE (2005)
31. Smutnicki, C., Rudy, J., Żelazny, D.: Very fast non-dominated sorting. Decis. Mak. in Manuf. and Serv. **8**(1–2), 13–23 (2014)
32. Srinivas, N., Deb, K.: Multiobjective optimization using non-dominated sorting in genetic algorithms. Evol. Comput. **2**(3), 221–248 (1994)
33. Suh, J.W., Kim, Y.: Accelerating MATLAB with GPU Computing: A Primer with Examples, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2013)
34. Tang, S., Cai, Z., Zheng, J.: A fast method of constructing the non-dominated set: Arena's principle. In: ICNC, vol. 1, pp. 391–395 (2008)
35. Wong, M.L.: Parallel multi-objective evolutionary algorithms on graphics processing units. In: GECCO, pp. 2515–2522. ACM (2009)
36. Zhang, Q., Li, H.: Moea/d: A multiobjective evolutionary algorithm based on decomposition. IEEE T. Evolut. Comput. **11**(6), 712–731 (2007)
37. Zhang, X., Ye, T., Cheng, R., Jin, Y.: An efficient approach to non-dominated sorting for evolutionary multi-objective optimization. IEEE T. Evolut. Comput. **19**(2) (2012)
38. Zheng, J., Ling, C.X., Shi, Z., Xie, Y.: Some discussions about mogas: Individual relations, non-dominated set, and application on automatic negotiation. In: CEC, vol. 1, pp. 706–712. IEEE (2004)
39. Zitzler, E., Künzli, S.: Indicator-based selection in multiobjective search. In: Parallel Problem Solving from Nature-PPSN VIII, pp. 832–842. Springer (2004)
40. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Tech. Rep. 103, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Zurich (2001)