J Glob Optim (2018) 71:631–649
https://doi.org/10.1007/s10898-018-0669-3

Improving the performance and energy of Non-Dominated Sorting for evolutionary multiobjective optimization on GPU/CPU platforms

J. J. Moreno[1] · G. Ortega[1] · E. Filatovas[2] · J. A. Martínez[1] · E. M. Garzón[1]

Received: 1 February 2017 / Accepted: 27 May 2018 / Published online: 5 June 2018

Abstract Non-Dominated Sorting (NDS) is the most time-consuming procedure used in the majority of evolutionary multiobjective optimization algorithms that are based on Pareto dominance ranking without regard to the computation time of the objective functions. It can be accelerated by the exploitation of its parallelism on High Performance Computing systems, that provide heterogeneous processing units, such as multicore processors and GPUs. The optimization of energy efficiency of such systems is a challenge in scientific computation since it depends on the kind of processing which is performed. Our interest is to solve NDS in an efficient way concerning both runtime and energy consumption. In literature, performance improvement has been extensively studied. Recently, a sequential Best Order Sort (BOS) algorithm for NDS has been introduced as one of the most efficient one in terms of practical performance. This work is focused on the acceleration of the NDS on

This work has been partially supported by the Spanish Ministry of Science throughout Project TIN2015-66680, by J. Andalucía through Projects P12-TIC-301 and P11-TIC7176, and by the European Regional Development Fund (ERDF). This research has been partially funded by a Grant (No. P-MIP-17-60) from the Research Council of Lithuania.

✉ J. J. Moreno
juanjomoreno@ual.es

G. Ortega
gloriaortega@ual.es

E. Filatovas
Ernest.Filatov@gmail.com

J. A. Martínez
jmartine@ual.es

E. M. Garzón
gmartin@ual.es

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

2 Institute of Data Science and Digital Technologies, Vilnius University, Akademijos str. 4, Vilnius, Lithuania

modern architectures. Two efficient parallel NDS algorithms based on Best Order Sort, are introduced, MC-BOS and GPU-BOS. Both algorithms start with the fast sorting of population by objectives. MC-BOS computes in parallel the analysis of the population by objectives on the multicore processors. GPU-BOS is based on the principles of Best Order Sort, with a new scheme designed to harness the massive parallelism provided by GPUs. A wide experimental study of both algorithms on several kinds of CPU and GPU platforms has been carried out. Runtime and energy consumption are analysed to identify the best platform/algorithm of the parallel NDS for every particular population size. The analysis of obtained results defines criteria to help the user when selecting the optimal parallel version/platform for particular dimensions of NDS. The experimental results show that the new parallel NDS algorithms overcome the sequential Best Order Sort in terms of the performance and energy efficiency in relevant factors.

Keywords Evolutionary multiobjective algorithms · Non-Dominated Sorting · Best Order Sort · High Performance Computing · Multicore platforms · GPU

1 Introduction

The increasing computational demand of the next generation applications has driven computer designers to adopt new approaches in designing and constructing large High Performance Computing platforms (HPC), sparking the development and deployment of new technologies. Those technologies include the use of multicore and/or many-core architecture such as GPUs or the modern Xeon Phi platforms and multi-GPU clusters to speed up algorithms with high computational requirements.

The use of evolutionary multiobjective optimization (EMO) algorithms to solve large-scale multiobjective problems has been limited due to its large computational burden. However, thanks to HPC techniques, these kinds of algorithms can be used for solving multiobjective problems with many objectives and/or with large number of variables within a reasonable amount of time with an optimal energy consumption. Examples of such applications are found, e.g., in finance (large scale asset allocation problems) [22] in which thousands or even tens of thousands of variables must be balanced. As consequences, large populations must be used to approximate the Pareto front. Some studies where HPC techniques are developed to solve large-scale EMO problems are [10,14,21,28].

Commonly, parallel implementations of EMO algorithms are focused on the distributions of evaluations of objective functions. The Non-Dominated Sorting (NDS) is the most time-consuming procedure used in the majority of EMO algorithms that are based on Pareto dominance ranking principle, without regard to the computation of the objective functions. Well-known EMO algorithms that use this procedure are: NSGA-II [5], SPEA2 [31], PAES [16], R-NSGA-II [6], Synchronous NSGA-II [8], NSGA-III [4], EPCS [24], etc.

The sequential NDS optimization has been extensively studied. The first NDS version was proposed in [26]. It is based on the brute force method for finding the Pareto sets. The algorithm has $\mathcal{O}(MN^3)$ complexity because of repeated comparisons without auxiliary structures which store dominance information of the individuals. In [5] the popular NSGA-II algorithm for EMO, the Fast Non-Dominated Sorting (FNDS), was introduced with $\mathcal{O}(MN^2)$ complexity. For this sequential method, a specific data structure saves a domination count variable and a set of dominated solutions for every individual. It requires N^2 comparisons because all individuals are compared among them once. However, it is possible to complete the individual classification in fronts avoiding unnecessary comparisons. In this line, several improvements were implemented by developing more efficient sorting strategies [7,11,23, 27,29,30]. It must be noted that the computational burden of these sequential approaches of the NDS procedure has $\mathcal{O}(MN^2)$ complexity in the worst case.

The divide-and-conquer strategy proposed by Jensen reduced the complexity to $\mathcal{O}(N \log^{M-1} N)$ [15], however this algorithm is not applicable to many instances of EMO problems. In [9], the Jensen algorithm was extended removing its limitation that no two solutions can share identical values for any of the problem's objectives and the slight modification of [2] maintains $\mathcal{O}(N \log^{M-1} N)$ complexity at the worst-case running time. However, the divide-and-conquer approach has longer processing times when increasing the number of objectives. Recently, several efficient NDS algorithms have been proposed. In [11], Efficient Non-Dominated Sort with Non-Dominated Tree (ENS-NDT) was introduced. It starts with the population ordered by the first objective and uses a novel Non-Dominated Tree (NDTree) to speed up the NDS to reduce unnecessary comparisons. In [23], an efficient NDS method, referred to as Best Order Sort (BOS) was proposed. It begins ordering the population by all objectives. Then, the fronts are built avoiding unnecessary comparisons. Its performance for sorting large populations according to many objectives overcomes previous proposals. BOS has also been used to define a hybrid NDS in combination with a divide-and-conquer approach [17].

However, the improved sequential NDS versions do not cover the computational needs when solving large-scale EMO problems. Therefore, parallel NDS routines should be developed. In [10], a NSGA-II parallel implementation on a GPU, focusing on the acceleration of NDS, has been analysed. However, this GPU NDS version has the highest complexity $\mathcal{O}(MN^3)$. Every thread computes the dominance of every individual in parallel without considering dominance information about other individuals. Parallel implementations of NDS (with $\mathcal{O}(MN^2)$ complexity) on multicore CPU and GPU have been analysed in [21]. An efficient parallel version of the NDS procedure was formally presented in [25]. The dominance information of individuals is stored in a matrix, but its experimental analysis is very limited. In [19], the same concept is applied to another GPU version of NDS. It is based on a data structure to store the dominance information where the individuals that dominate the population are computed by using fast shuffled reductions of dominance matrices on modern GPUs. Therefore, the GPU versions of NDS analysed in literature are based on the algorithms with the highest computational costs. Moreover, parallel versions of the most efficient sequential NDS algorithms have not been developed since an adaption of the algorithms due to their data dependencies is necessary.

It is remarkable that most NDS schemes, which reduce unnecessary comparisons, became inherently sequential algorithms. In this line, to our knowledge, the fast state-of-the-art algorithm Best Order Sort (BOS), referred to above, makes use of fast implementations of sorting algorithms and removes unnecessary comparisons among individuals. It results in an efficient NDS in terms of runtime, but with a structure which prevents its parallel execution. With the goal to optimize both performance and energy consumption of NDS with respect to the efficient BOS algorithm, two parallel NDS algorithms, MC-BOS and GPU-BOS are introduced in this work. MC-BOS ranks populations in parallel on multicore processors and GPU-BOS solves the same problem on GPUs.

The proposed algorithms start by ordering the population according to the different objectives with a fast parallel routine and then the fronts are built from two ideas: (1) if an individual is not dominated by any solution with a particular rank, then it belongs to that particular rank; and (2) if an individual is dominated by at least one individual of all ranks then the individual is defining a new front of an upper rank. If these principles are considered, the number of comparisons can be reduced since the population is ordered by the objectives.

The contribution of this work is twofold. First, new parallel implementations of NDS procedure on multicore and GPU are analysed, referred to as MC-BOS and GPU-BOS respectively. Second, an experimental evaluation of MC-BOS and GPU-BOS is carried out using modern multicore processors and GPUs to rank populations of different sizes and number of objectives. The runtime and energy consumption are analysed in relation to the sequential BOS. For all tests, both parallel algorithms accelerate the NDS and reduce its energy consumption in relation to the sequential BOS in relevant percentages. The analysis of the obtained results allows us to identify the best platform/algorithm of the parallel NDS according to every problem size.

The paper is organized as follows: Sect. 2 is devoted to describing some relevant concepts related to the EMO problems and BOS algorithm, a state-of-the-art NDS algorithm. Section 3 explains in detail the parallel implementations of BOS introduced in this work, MC-BOS and GPU-BOS. An experimental study of performance and energy consumption of both parallel implementations and the sequential BOS is carried out in Sect. 4. The analysis of the experimental results allows us to define criteria to choose the optimal version/platform in terms of performance and energy consumption. Finally, Sect. 5 shows the conclusions of this work.

2 Background

2.1 Evolutionary multiobjective optimization

We can formulate a multiobjective minimization problem as follows [18]:

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

where $\mathbf{z} = \mathbf{f}(\mathbf{x})$ is an *objective vector*, defining the values for all objective functions $f_1(\mathbf{x})$, $f_2(\mathbf{x}), \ldots, f_M(\mathbf{x})$, $f_i : \mathbb{R}^V \to \mathbb{R}$, $i \in \{1, 2, \ldots, M\}$, $M \geq 2$ is the number of objective functions; $\mathbf{x} = (x_1, x_2, \ldots, x_V)$ is a vector of variables (*decision vector*) and V is the number of variables $\mathbf{S} \subset \mathbb{R}^V$ is *search space*, which defines all feasible decision vectors.

A decision vector $\mathbf{x}' \in \mathbf{S}$ is a *Pareto-optimal solution* if $f_i(\mathbf{x}') \leq f_i(\mathbf{x})$ for all $\mathbf{x} \in \mathbf{S}$, $i \in \{1, 2, \ldots, M\}$ and $f_j(\mathbf{x}') < f_j(\mathbf{x})$ for at least one $j \in \{1, 2, \ldots, M\}$. The set of all the *Pareto-optimal solutions* is called the *Pareto set*. An objective vector $\mathbf{f}(\mathbf{x}')$ is a *Pareto-optimal vector* if $\mathbf{x}'$ is a *Pareto-optimal solution*. The region defined by all the *Pareto-optimal vectors* is called the *Pareto front*.

For two objective vectors $\mathbf{z}$ and $\mathbf{z}'$, $\mathbf{z}'$ *dominates* $\mathbf{z}$ (or $z' \succ z$) if $z'_i \leq z_i$ for all $i \in \{1, 2, \ldots, M\}$ and there exists at least one $j \in \{1, 2, \ldots, M\}$ 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. Many EMO algorithms have been designed and various operator techniques, fitness functions and chromosomal representations can be found in literature, however the outline remains similar. As a rule, the solution process of an EMO algorithm is iterative. An EMO algorithm starts with an initial population consisting of decision vectors randomly generated in the search space. Each iteration of an EMO consists of the following operations:

– Evaluating each individual.
– Assigning fitness to each individual.
– Checking if termination condition is satisfied.
– Modifying population using selection, mutation and crossover operators.
– Creating a new population.

The solution process is continued till a stopping criterion is not satisfied, which is usually based on the maximum number of iterations or number of function evaluations. Without regard to the operation of evaluation, every individual which computes the objective functions, the most computationally expensive part of such algorithms is the dominance ranking operators. They are performed in the fitness assignment operation in each iteration of an EMO algorithm. In algorithms based on Pareto dominance, this is implemented by the NDS procedure.

2.2 Non-Dominated Sorting

Non-Dominated Sorting aims to assign different ranks to the individuals, and dividing the population into several non-dominated levels (fronts). According to Pareto dominance, the individuals with the same rank are non-dominated among themselves and they can only be dominated by a solution in a lower rank. It should be noted that dominance comparisons between the individuals are repeated in every iteration on an EMO algorithm and takes the most computational burden.

Best Order Sort[1] algorithm for NDS has been introduced and analysed in [23]. It is considered one of the most efficient NDS algorithms in terms of practical performance [17]. In this work, the principles of BOS are used for developing parallel NDS procedures with a reduced number of comparisons.

Algorithm 1 describes the BOS procedure to compute fronts of the population P with N individuals for a problem with M objectives. It consists of two stages. In the first one the global data structures are initialized and the population is sorted in the ordered sets Q_j by objectives $j = 1, \dots M$, using lexicographical ordering in case of a tie. This way, Q_j can be considered the columns of a matrix, referred to as Q, which is computed in the first stage. So, Q_{ij} represents the i-th individual in the list of objective j. SC and RC represent the numbers of ranked individuals and fronts computed, respectively. The sets L_j^r for $1 \le j \le M$ and $1 \le r \le N$ define the subset of the front r by the analysis of objective j. When BOS finishes the front r is the union of L_j^r with $1 \le j \le M$. Moreover, two N vectors, F and $isRanked$, are defined to store the rank of every individual and to mark the ranked individuals, respectively.

In the second stage, the individuals are ranked by the comparison of their dominance. It starts with the analysis of individuals with better objective values in the sorted sets Q_j. Every individual is checked in the corresponding objective. Then, the sets L_j^r are filled to compute the fronts and to save the ranked individuals by comparisons to objective j. If an individual, s, is checking for one objective j and previously it had been ranked by the analysis of another objective, then it is added to the set $L_j^{F_s}$. If the individual s had not been previously ranked, then the routine FINDRANK(s,j) ranks the individual by the dominance analysis in the objective j as described in Algorithm 2. This way, s is compared to the individuals t in L_j^k for all computed ranks $1 \le k \le RC$. If there is any individual, t, which cannot dominate s, then s is classified in the front of t, $F_s = F_t$ and added to $L_j^{F_s}$. If the comparisons finish and s is dominated by the checked individuals, then s defines a new front of higher rank.

1 https://github.com/Proteek/Best-Order-Sort.

Algorithm 1 Best Order Sort algorithm for NDS

```
Input: $P$: population; $M$: number of objective functions
    Initialization of the global data structures
 1: $N \leftarrow |P|$; $SC \leftarrow 0$; $RC \leftarrow 1$;
 2: $L^r_j \leftarrow \emptyset$ for $1 \leq j \leq M$ and $1 \leq r \leq N$
 3: $isRanked_i \leftarrow$ false; $F_i \leftarrow 0$ for $1 \leq i \leq N$
 4: $Q \leftarrow [Q_j] \leftarrow$ sorted $P$ by the objective $j$ for $1 \leq j \leq M$ in every column of $Q$
    Building fronts from ordered population $Q_j$
 5: for $i \leftarrow 1$ to $N$ do
 6:     for $j \leftarrow 1$ to $M$ do
 7:         $s \leftarrow Q_{i,j}$                    ▶ $i$-th individual in the sorted population by objective $j$
 8:         if ISRANKED($s$) then
 9:             $L^{F_s}_j = L^{F_s}_j \cup \{s\}$
10:         else
11:             $F_s \leftarrow$ FINDRANK($s$, $j$)                              ▶ Algorithm 2
12:             $isRanked_s \leftarrow$ true; $SC \leftarrow SC + 1$
13:         if $SC = N$ then break
14: return $F$
```

Algorithm 2 FINDRANK Procedure of Best Order Sort

```
Input: $s$: individual; $j$: number of list
 1: $done \leftarrow$ false
 2: for $k \leftarrow 1$ to $RC$ do
 3:     for $t \in L^k_j$ do
 4:         $check \leftarrow$ true                               ▶ It is assumed $t$ dominates $s$
 5:         for $l \leftarrow 1$ to $M$ do
 6:             if $s$ is better than $t$ in objective $l$ then
 7:                 $check \leftarrow$ false                           ▶ $t$ cannot dominate $s$
 8:                 break
 9:         if $check$ then break
10:     if not $check$ then                                  ▶ $s$ is non-dominated by $L^k_j$
11:         $F_s \leftarrow k$; $L^{F_s}_j = L^{F_s}_j \cup \{s\}$
12:         $done \leftarrow$ true
13:         break
14: if not $done$ then                    ▶ $s$ is dominated by all fronts previously defined
15:     $RC \leftarrow RC + 1$
16:     $Fs \leftarrow RC$
17:     $L^{F_s}_j = L^{F_s}_j \cup \{s\}$
18: return $Fs$
```

Thus, Algorithm 1 optimizes the comparisons to compute NDS as it is based on: (1) the previous sorting of the population by objectives, this way the comparisons to identify the ‘not-worse’ individuals in the corresponding objective can be optimized; (2) the sorted checking which ranks firstly individuals with ‘better’ objectives.

The computational complexity of BOS in the worst case is $\mathcal{O}(MN^2)$, however it is relevant to underline that BOS optimizes the number of comparisons at the expense of incrementing its data dependencies. Therefore, it is inherently a sequential algorithm. However, in next sections, several modifications of the BOS algorithm are studied with the goal of defining parallel versions of BOS which can exploit modern parallel architectures and be competitive in relation to the efficient sequential BOS.

3 Parallel implementations of the Non-Dominated Sorting based on Best Order Sort algorithm

Most of the HPC platforms and also modern computers are composed of multicore and GPU devices. CUDA (Compute Unified Device Architecture) is the parallel interface introduced by NVIDIA to help develop GPU codes using C or C++ language. CUDA provides some abstraction to the GPU hardware, and it provides the SIMT (Single Instruction, Multiple Threads) programming model to exploit the GPU. However, the programmer has to take into account several features of the architecture, 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 parallel routine (kernels) invocations to the device. The input/output data to/from the GPU kernels are communicated between the CPU and the 'global' GPU memories. GPUs have hundreds of cores which can collectively run thousands of computing threads. Each core, called Scalar Processor (SP), belongs to a set of multiprocessor units called Streaming Multiprocessors (SM). The SMs are composed of 192 (or 128) SPs on Kepler (or Maxwell) GPU architectures [12,20]. This way, the GPU device consists of a set of SMs and each kernel is executed as a batch of threads organized as a grid of thread blocks [1].

3.1 Multicore version of the Best Order Sort algorithm (MC-BOS)

The multicore version is implemented on Pthreads[2] and C to exploit the parallelism available on modern processors. While the original BOS algorithm loops through the Q matrix rowwise, trying to reduce the number of comparisons needed to rank each individual, each Q_j set has its own L_j structure. Therefore, they can be processed in parallel without synchronization.

Algorithm 3 describes the operation of each of the M compute threads. The initial sorting of the population by each objective is efficiently computed by each thread using QSORT_R[3] with a custom comparison function to consider multiple objectives in case of a tie. Every thread j ranks the population in the order defined by Q_j and writes the rank of every individual in the shared data structure F (line 9, Algorithm 3). Thus, every thread will rank only the individuals which have not been studied by another thread.

Although not using any synchronization points increases the performance of the multicore algorithm, it may cause a 'write after write' data hazard on the shared array F (that contains the ranks of the population) when several threads try to rank the same individual at the same time. Nevertheless, the definition of domination given in Sect. 2 guarantees that the individuals that dominate a given individual are in lower positions of every sorted set Q_j. As the rank of an individual is the maximum rank of the individuals that dominate it plus one, the ranks computed by the threads are the same and this hazard does not cause wrong results.

3.2 GPU implementation of NDS based on Best Order Sort (GPU-BOS)

The parallel scheme of MC-BOS is not appropriate to exploit the massive parallelism provided by GPU. As a consequence it is necessary to design a new scheme with specific data structures which allow us to increase the parallelism level of the algorithm. GPU-BOS has been designed by adapting the key ideas of BOS to a massively parallel architecture.

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

[3] https://linux.die.net/man/3/qsort_r.

Algorithm 3 Pseudocode of the MC-BOS function computed by every thread

```
Input:
    $N$: Number of individuals
    $M$: Number of objective functions
    $j$: Index of the objective function this thread will study
    $P$: Matrix of dimensions $N \times M$ containing the definition of the population
    $F$: Array of size $N$ to store the rank of each individual (initialized to $-1$)

 1: $RC \leftarrow 1$
 2: $L_j^r \leftarrow \emptyset$ for $1 \leq r \leq N$
 3: $Q_{i,j} \leftarrow i$ for $0 \leq i < N$
 4: Key-value ordering of $Q_j$ by $P_j$, using other objectives in case of a tie.

 5: for $i \leftarrow 0$ to $N-1$ do
 6:     $s \leftarrow Q_{i,j}$
 7:     if $F_s < 0$ then                          ▶ This individual is unranked
 8:         $F_s \leftarrow$ FINDRANK($s, j$)         ▶ Defined on Algorithm 2
 9:     else                       ▶ The individual was ranked by another thread
10:         $L_j^{F_s} \leftarrow L_j^{F_s} \cup s$             ▶ Add individual $s$ to front $F_s$
11: return $F$
```

Algorithm 4 GPU-BOS host pseudocode to compute NDS on GPU

```
Input:
    $N$: Number of individuals
    $M$: Number of objective functions
    $P$: Matrix of dimensions $N \times M$ containing the definition of the population

    Phase 1: Initialization and sorting of the population.
 1: Communicate P matrix to GPU global memory.
 2: $Q_{i,j} \leftarrow i$ for $0 \leq i < N$ and $0 \leq j < M$    ▶ Initialize matrix Q of dimensions $N \times M$
 3: for $j \leftarrow 0$ to $M-1$ do
 4:     $Q_j \leftarrow$ DEVICERADIXSORT($P_j$)               ▶ Key-value ordering of $Q_j$ by $P_j$

    Phase 2: Compute best objective and position for each individual
 5: $B_{0,i} \leftarrow N$ for $0 \leq i < N$              ▶ Initialize first row of matrix $B$ (best positions)
 6: $B_{1,i} \leftarrow 0$ for $0 \leq i < N$            ▶ Initialize second row of matrix $B$ (best objective)
 7: $P_{aux} \leftarrow$ FINDPOSITIONS($Q, P$)          ▶ Best position for each individual in each objective
 8: $B \leftarrow$ BESTOBJECTIVE($P_{aux}$)          ▶ Objective with the lowest position for each individual

    Phase 3: Compute fronts from the information of Phase 2
 9: $batch_i \leftarrow 0$ for $0 \leq i < N$
10: $F_i \leftarrow -1$ for $0 \leq i < N$
11: $\Delta_{i,j} \leftarrow -1$ for $0 \leq i < N$ and $0 \leq j < blockDim$
12: $Rank \leftarrow 0$
13: $SC \leftarrow 0$
14: while $SC < N$ do
15:     $F \leftarrow$ CUNEWFRONT($N, M, Rank, P, Q, \Delta, B, batch, F$)        ▶ Algorithm 5
16:     Communicate $F$ from GPU memory to CPU memory
17:     $SC \leftarrow SC + |\{k : F_k = Rank\}|$
18:     $Rank \leftarrow Rank + 1$
19: return $F$
```

Algorithm 4 shows the host pseudocode of GPU-BOS to compute the NDS on the GPU. It includes three phases. In Phase 1, global parameters are defined, the population data structure is sent from the CPU to the GPU memory and then, it is efficiently sorted by objectives on the GPU. The sorting is computed on the GPU by M executions of the DEVICERADIXSORT kernel defined by the CUB[4] library in streaming mode.

In Phase 2, for every individual s, the sub-population which could dominate it is obtained from the matrix Q whose M columns define the sorted population by the objectives. Every individual, s, is classified at each column Q_j. Then, the lowest row of Q which stores s is obtained, $\mathbf{i}$, and the corresponding column is defined as $\mathbf{j}$. Thus, $\mathbf{j}$ represents the objective with best order for the individual s and $\mathbf{i}$ is the index of s in the sorted population by such objective. As it is justified in [23], the sub-population which could dominate the individual s is defined by the array $S_s = [Q_{0,\mathbf{j}}, \ldots, Q_{\mathbf{i}-1,\mathbf{j}}]$.

So, the B matrix is $2 \times N$ and it saves the mentioned pairs of indexes for every individual and it is computed in parallel on the GPU by the kernels FINDPOSITIONS and BESTOBJECTIVE (lines 7 and 8, Algorithm 4). The kernel FINDPOSITIONS also contains a sub-routine to include in the set S_s of each individual s, the individuals with higher positions but the same value in the objective function studied, in such a way allowing us to use a faster sorting algorithm that does not need lexicographical sorting in case of ties.

Then, Phase 3 iteratively computes a new front till the entire population is ranked. The kernel CUNEWFRONT (Algorithm 5) computes in parallel a new front at every iteration. N blocks of threads are defined and every block analyses if the corresponding individual s is classified in the new front. This analysis is based on the following property [23]: Let s, D_s, and $r - 1$ be an individual, the set of dominators of such individual and the maximal rank of the dominators, respectively, then the rank of s is r.

In kernel CUNEWFRONT every thread block analyses if its individual s belongs to the new front. When s has been previously ranked then the block stops. Otherwise, if the first front is being analysed, every block checks the dominance of s by comparisons to the lists S_s. So, iteratively every batch of $blockDim$ individuals is analysed in parallel. When one dominator is identified the thread block stops checking and the threads that have identified dominators save that information in the column s of the matrix Δ. Also, the first thread of each block stores in $batch_s$ the index of the next batch of individuals to be checked. When one front of $Rank > 0$ is studied, every thread in the s block reads the dominators previously identified from the s column of Δ. If at least one of them has not been ranked, then its rank is $q \geq Rank$, the rank of s is higher than $Rank$ and therefore s is not classified in the new front, and the thread block stops.

Otherwise, if all dominators have been classified in a front $q < Rank$ then every thread block checks the dominance of s with an initial sorted set of candidates as dominators of s, $S_s^* \equiv [Q_{k,\mathbf{j}}, \ldots, Q_{\mathbf{i}-1,\mathbf{j}}]$, where $k = B_{0,\mathbf{j}}$. When the dominance is checked by a thread block this initial set can include individuals previously ranked which are erased from S_s^*. When at least one dominator of s is identified then, it is saved in Δ and also k_{stop} in $batch$ and the computation stops. Otherwise, there are no new dominators of s, therefore it is classified in the new front. Figure 1 illustrates the scheme to analyse the dominance of the individual s by a thread block with $blockDim = 4$ for a very reduced population when 1st and 2nd fronts are computed by GPU-BOS.

The performance of GPU-BOS is bounded by the CUNEWFRONT kernel. On this kernel there are two types of comparisons. The first type (lines 8–14, Algorithm 5) is the comparison of an individual with its saved dominators. Although each such comparison runs in $\mathcal{O}(1)$, one individual can be compared with the same dominator multiple times if the dominator is not yet ranked. However, the maximum number of ranks is N and the rank is incremented after each call to the CUNEWFRONT kernel, so there cannot be more than $\mathcal{O}(N)$ of such comparisons

[4] https://nvlabs.github.io/cub/structcub_1_1_device_radix_sort.html.

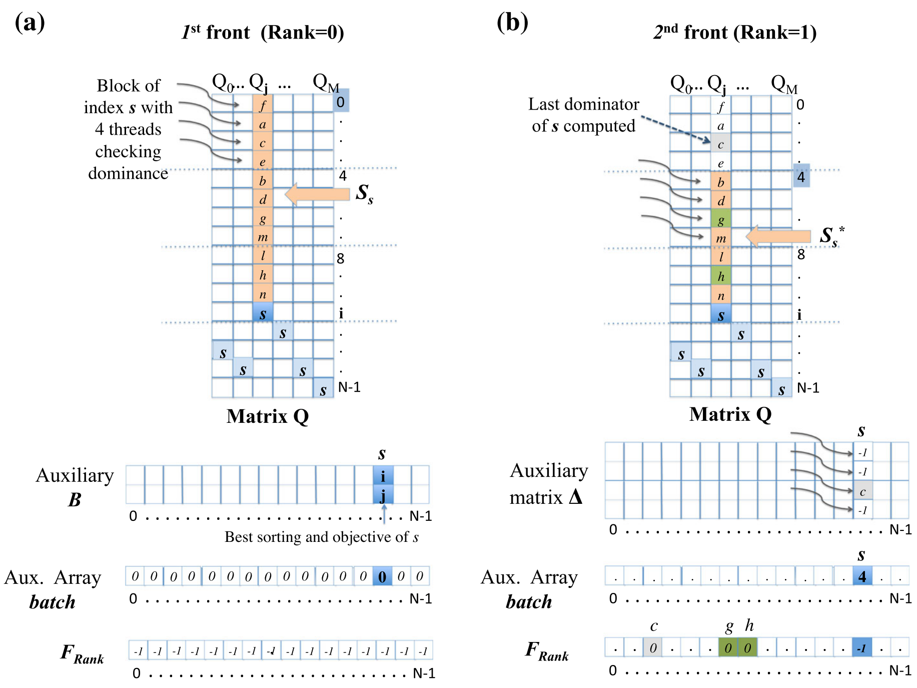

Fig. 1 Dominance checking to analyse if the individual s belongs to the 1st or 2nd front and data structures involved in CuNewFront. **a** On the left, the dominance of the sorted list $S_s = \{f, a, c, e, b, d, g, m, l, h, n\}$ over s is computed to check if the individual s belongs to the first front. The checking is computed in a parallel loop by a thread block. At first iteration, c is identified as a dominator and the s-block stops. Then, the threads write in Δ_s the last dominators computed, c, and also they write 4 in the array $batch$, the next index to continue the dominance checking for a new call of CuNewFront. These written values are shown on the right of the figure. **b** On the right, to check if the individual s belongs to the 2nd front, when the last dominators of s computed, saved in Δ_s, have no rank then s-block stops. But in the example the last dominator computed, c, has been previously classified in the first front. Then, the dominance over the sorted list $S_s^* = \{b, d, m, l, n\}$ is computed. Notice that, the individuals g and h have been previously classified in the first front, then they are not included in the checking list. If new dominators of s are not identified, then s is classified in the 2nd front. Otherwise, the individual s is not ranked and it will be involved in the successive calls of CuNewFront to compute new fronts

for each individual, $\mathcal{O}(N^2)$ in total. The second type (lines 15–31, Algorithm 5) is the comparison of an individual with its potential dominators. There are $\mathcal{O}(N^2)$ combinations and each comparison runs in $\mathcal{O}(M)$.

Therefore, GPU-BOS retains the worst-case complexity of the original BOS algorithm: $\mathcal{O}(MN^2)$. Although the computational schemes of GPU-BOS and BOS are different, they are based on the same rules to organize the individuals in order to reduce the number of dominance comparisons.

4 Evaluation

This section opens with a technical description of the experimental hardware setup, followed by an analysis of the performance and energy consumption of the BOS, MC-BOS and GPU-BOS algorithms when applied to compute the fronts of populations with different sizes and

Algorithm 5 CuNewFront GPU kernel computes a new front in parallel

```
Input:
    $N$: Number of individuals
    $M$: Number of objective functions
    $Rank$: Current rank
    $P$: Matrix of dimensions $N \times M$ containing the definition of the population
    $Q$: Matrix of dimensions $N \times M$ containing the indexes of the population sorted by each objective function
    $\Delta$: Matrix of dimensions $N \times blockDim$ where we store partial domination information
    $B$: Matrix of size $N \times 2$ containing, for each individual, the objective where it is best sorted and its position
    in the sorted list
    $batch$: Array of size $N$ to store the starting index of the next batch to be processed
    $F$: Array of size $N$ to store front information for each individual

    Initialization.
 1: __shared__ $dominated$                          ▶ Block scope shared variable
 2: $s \leftarrow blockIdx.x$                       ▶ The individual this block will study
 3: $x \leftarrow threadIdx.x$                      ▶ Block scope thread ID
 4: if $F_s \geq 0$ then return                     ▶ This individual is already ranked
 5: if $threadIdx.x = 0$ then
 6:     $dominated \leftarrow$ false                ▶ The first thread initializes the shared variable
 7: __syncthreads                                   ▶ Synchronization point for all threads in each block

    Checking if the dominators of $s$ (saved in $\Delta$) have been ranked
 8: if $\Delta_{s,x} \geq 0$ then
 9:     if $F_{\Delta_{s,x}} \geq 0$ and $F_{\Delta_{s,x}} < Rank$ then
10:         $\Delta_{s,x} \leftarrow -1$            ▶ The individual that dominated $s$ is already ranked
11:     else
12:         $dominated \leftarrow$ true             ▶ $s$ is dominated by an unranked individual
13: __syncthreads
14: if $dominated$ then return

    If all computed dominators of $s$ are ranked, analyse remaining candidates
15: $j \leftarrow B_{1,s}$                          ▶ Objective where $s$ is best sorted
16: for $k \leftarrow batch_s$ to $B_{0,s} - 1$ with $k$ += $blockDim$ do
17:     __syncthreads
18:     if $k + threadIdx.x < B_{0,s}$ then
19:         $c \leftarrow Q_{k,j}$                  ▶ Index of a candidate to a new dominator
20:         $is\_candidate \leftarrow$ true
21:         if $Rank > 0$ then
22:             if $F_c \geq 0$ and $F_c < Rank$ then
23:                 $is\_candidate \leftarrow$ false ▶ Already ranked, not a candidate
24:         if $is\_candidate$ and IS_DOMINATED($P, s, c$) then
25:             $\Delta_{s,x} \leftarrow c$         ▶ $s$ is dominated by an unranked individual
26:             $dominated \leftarrow$ true
27:     __syncthreads
28:     if $dominated$ then
29:         if $threadIdx.x = 0$ then
30:             $batch_s \leftarrow k + blockDim$   ▶ Store the batch where $s$ left
31:         return

    All the candidates for dominance have been checked
32: if $threadIdx.x = 0$ then
33:     $F_s \leftarrow Rank$                       ▶ If this instruction is reached, $s$ belongs to this rank
```

number of objectives on the target architectures. We assume that users of these routines are interested in specific EMO problems with particular number of objectives and population sizes. Our goal with this experimental analysis is to provide general criteria to help users

Table 1 Characteristics of the test GPUs

	NVIDIA K80	Tesla M2075
Peak performance (double prec.) (TFLOPS)	2.91	0.5
Peak performance (single prec.) (TFLOPS)	8.74	1.03
Device memory (GB)	2 × 12	5
Memory bandwidth (GB/s)	2 × 240	144
Multiprocessors	2 × 13	14
CUDA cores	2 × 2496	448
Compute Capability	3.7	2.0

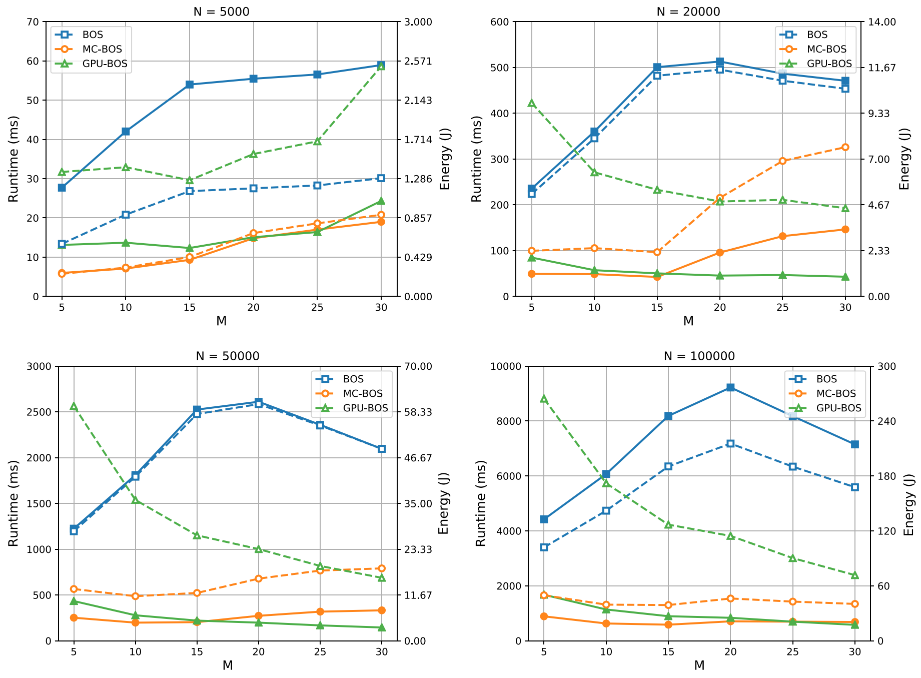

Fig. 2 Experimental Runtime (solid lines) and Energy Consumption (dashed lines) of BOS, MC-BOS and GPU-BOS on the platform $\mathcal{F}_1$ for four random populations of four sizes when the number of objectives changes from $M = 5$ to $M = 30$

to choose the best platform/version to obtain the best performance and/or lowest energy consumption for solving specific NDS processes.

Three computational architectures have been considered in the experiments:

- $\mathcal{F}_1$: Bullx R424-E3: 2 Intel Xeon E5 2650 processors with 8 cores each and 64 GB of RAM. It is connected to a NVIDIA Tesla M2070 GPU. Table 1 provides technical details about this GPU platform.
- $\mathcal{F}_2$: Bullx R421-E4: 2 Intel Xeon E5 2620v2 processor with 6 cores each and 64 GB of RAM. It is connected to 2 NVIDIA K80 (each NVIDIA K80 is composed by two Kepler GK210 GPUs). The characteristics of each NVIDIA K80 are given in Table 1.
- $\mathcal{F}_3$: Bullion S8: 8 Intel Xeon E7 8860v3 processors with 16 cores each and 2.3 TB of RAM.

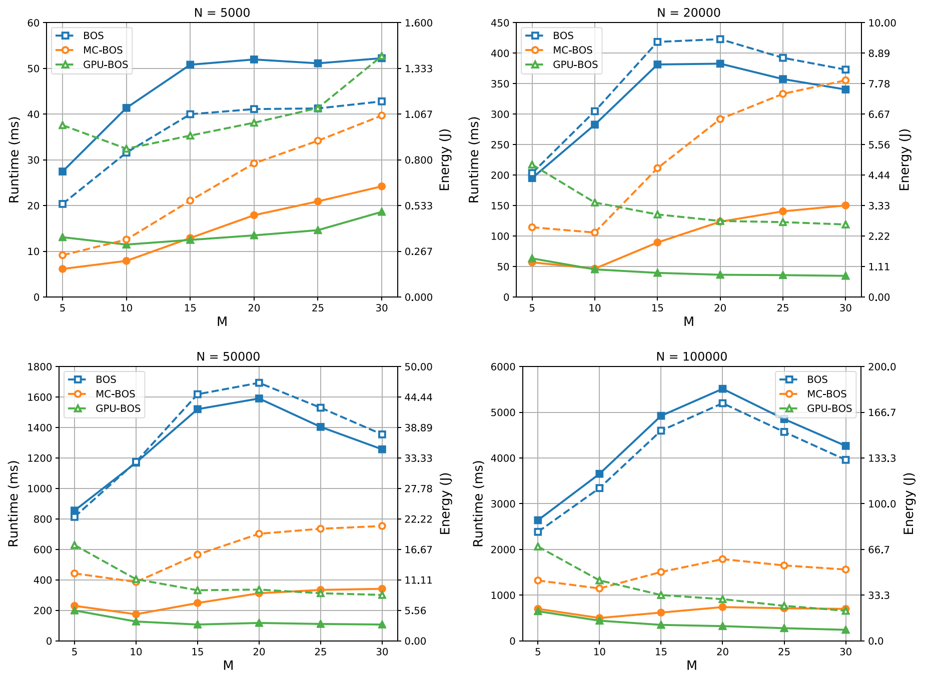

Fig. 3 Experimental Runtime (solid lines) and Energy Consumption (dashed lines) of BOS, MC-BOS and GPU-BOS on the platform $\mathcal{F}_2$ for four random populations of four sizes when the number of objectives changes from $M = 5$ to $M = 30$

The test platforms do not include the most recent multicore processors and GPUs. However, this hardware is readily available on many currently accessible clusters for scientific computation. Therefore, they can be considered representative examples of available platforms for users of the routines that will be tested. Three kinds of multicore processors are considered.

$\mathcal{F}_1$ contains 2 Intel Sandy Bridge EP processors for a total of 16 CPU cores with 64 GB of RAM and 2 NVIDIA Tesla M2070 GPUs of the Fermi microarchitecture. $\mathcal{F}_2$ is a newer platform, containing 2 Intel Ivy Bridge EP processors with 64 GB of RAM for a total of 12 CPU cores and 2 NVIDIA Tesla K80 GPUs of the Kepler microarchitecture. Although both of these platforms have multiple GPUs, our GPU-BOS implementation uses only one. $\mathcal{F}_3$ contains 8 Intel Haswell processors for a total of 128 cores with 2.3 TB of RAM. This platform is composed of four nodes, with two sockets and 576 GB of RAM per node, interconnected by a proprietary bus that converts them into a single NUMA node.

To provide a fair comparison and avoid the overhead of the JVM, the original sequential BOS Java code[5] has been reimplemented in C. All the programs have been compiled using gcc 5.4.0 and nvcc 8.0.44 with optimization flags O3. All platforms run Ubuntu 16.04 LTS with CUDA SDK 8.

For the acquisition of the energy consumption data, we have developed a software tool that collects metrics from various hardware counters integrated on each platform. It uses the Running Average Power Limit (RAPL) interface on Intel processors, introduced on the

[5] https://github.com/Proteek/Best-Order-Sort.

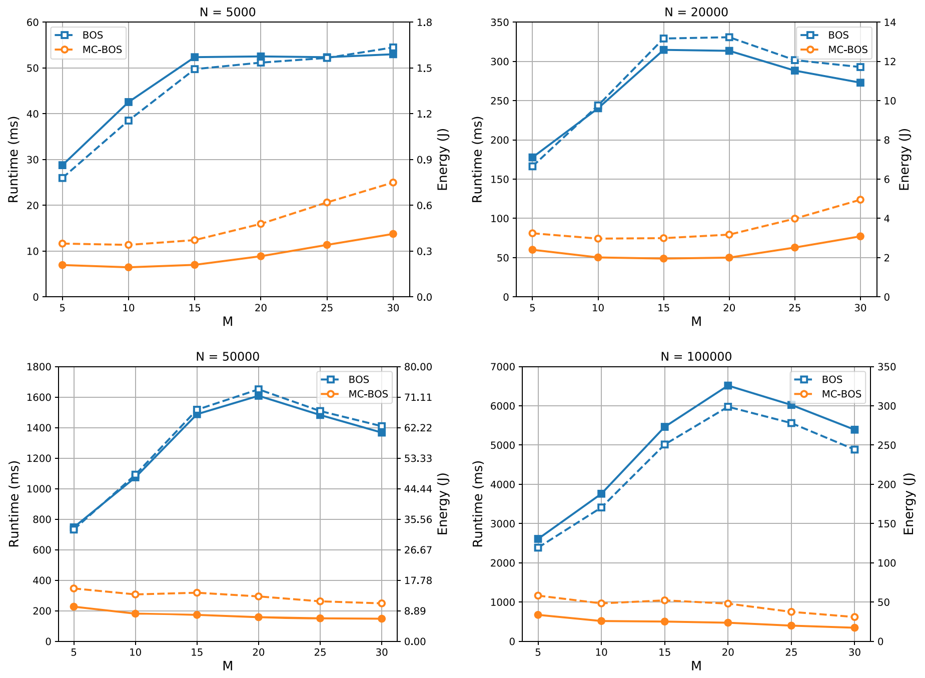

Fig. 4 Experimental Runtime (solid lines) and Energy Consumption (dashed lines) of BOS and MC-BOS on the platform $\mathcal{F}_3$ for four random populations of different sizes when the number of objectives changes from $M = 5$ to $M = 30$

Sandy Bridge microarchitecture [3] via the Linux Power Cap sysfs. It uses the NVIDIA Management Library[6] (NVML) API on NVIDIA GPUs.

The runtime and energy of sequential BOS, MC-BOS and GPU-BOS has been evaluated when the number of objectives, M, and population size, N, vary on the platforms $\mathcal{F}_1$ and $\mathcal{F}_2$. On $\mathcal{F}_3$ only results for BOS and MC-BOS are shown since there is no GPU on this platform. Figures 2, 3 and 4 represent the experimental results with four graphics for random populations with different numbers of individuals $N = 5000, 20{,}000, 50{,}000, 100{,}000$. To obtain these results, 100 test populations have been randomly generated by the testing scripts, following a uniform distribution in the range $[0, 1)$. For each population and implementation, we have repeated the experiment 10 times, for each case discarding the best and worst runtime, and averaging the rest to obtain the experimental results shown in this section.

Every plot represents, on the left, the runtimes in milliseconds (with solid lines) and, on the right, energy consumption in joules (with dashed lines) for the different NDS versions. The energy measurement results are generated taking into consideration only the domains used by each implementation. This means that, for the sequential algorithm, we only consider the processor where it is running. For the multicore algorithm we consider as many processors as needed to allocate the number of threads spawned. For the GPU algorithm, we consider the processor where the host code is running and the GPU where the kernels are launched.

The general trends of BOS, MC-BOS and GPU-BOS when the objectives increase are similar on every plot, although the random populations are different on each platform. The runtimes of BOS increases almost linearly for the smallest populations with $M = 5, 10, 15$

6 https://developer.nvidia.com/nvidia-management-library-nvml.

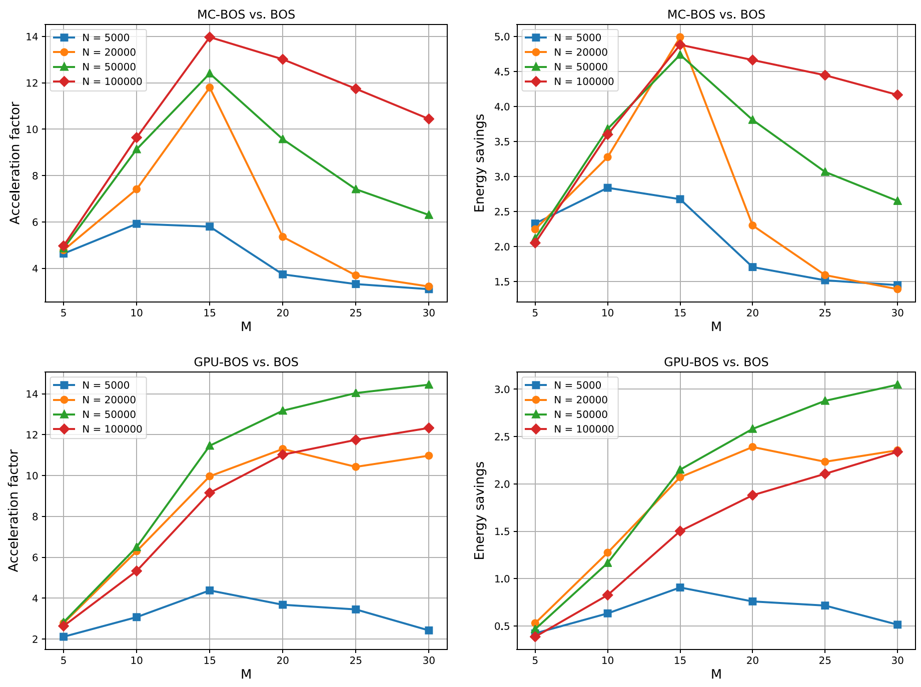

Fig. 5 Experimental acceleration factors of runtime and energy savings for multicore version (on the top) and GPU version (at the botton) in relation to the sequential BOS on the platform $\mathcal{F}_1$ for four random populations of four sizes when the number of objectives changes from $M = 5$ to $M = 30$

and for $M = 20, 25, 30$ the runtime lightly increases for small populations. BOS runtime even decreases, since for the larger populations the percentage of non-dominated individuals increases as N and M, decreasing in such cases the comparisons needed to compute fronts. MC-BOS achieves a high acceleration in relation to BOS in terms of runtime and energy. MC-BOS runtimes increase as M increases, and the slope of this increment is less relevant as the individual count increases. This trend is clearly appreciated on the three platforms in general terms. On $\mathcal{F}_1$ ($\mathcal{F}_2$) there are relevant runtime increments between $M = 15$ and 20 ($M = 10$ and 15) because the number of cores available on the platform $\mathcal{F}_1$ ($\mathcal{F}_2$) becomes less than M, that is $M > 16$ ($M > 12$). Therefore, since the number of threads is defined by M, as M increases several threads are concurrently executed on the same core. The runtimes of GPU-BOS for the smaller populations decrease as M increases with a more relevant slope when the population is large.

On platform $\mathcal{F}_1$, GPU-BOS is slower than MC-BOS when $M \leq 15$ and $N < 10{,}000$. For $10{,}000 \leq N < 100{,}000$ and $M > 15$, GPU-BOS is faster than MC-BOS and, in some cases, more energy efficient. For populations larger than that, the multicore implementation scales better than the GPU implementation. The high energy consumption of these old NVIDIA Fermi cards is relevant, making GPU-BOS less energy efficient than even the sequential implementation for $M = 5$.

On platform $\mathcal{F}_2$, the newer NVIDIA Fermi cards show better performance than their Fermi counterparts, being faster than the multicore implementation when $M \geq 15$ for all the population sizes shown. They are also energy efficient, making the GPU implementation the best one in both metrics for most of the test cases.

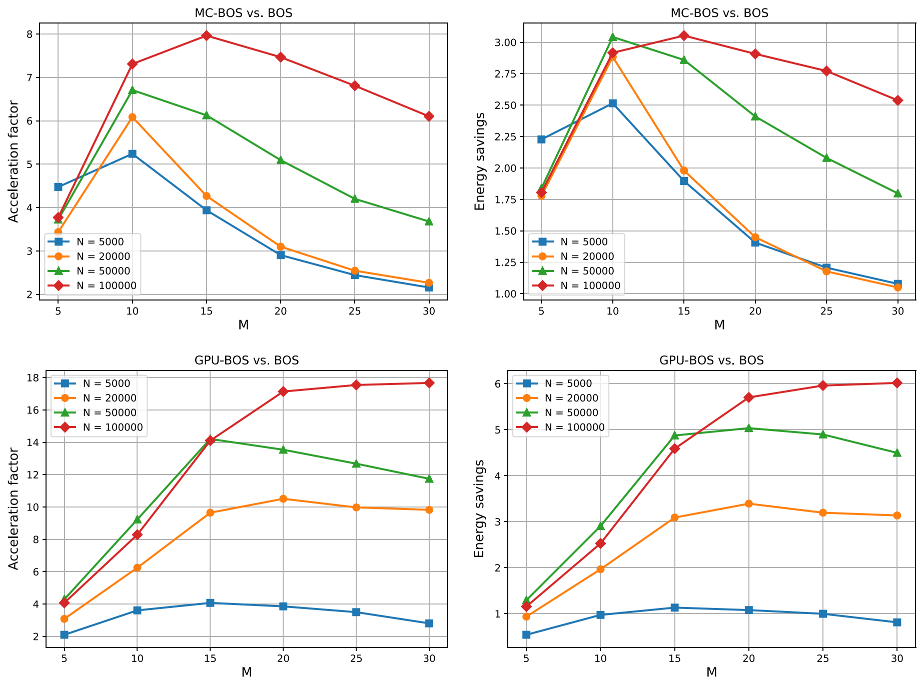

Fig. 6 Experimental acceleration factors of runtime and energy savings for multicore version (on the top) and GPU version (at the botton) in relation to the sequential BOS on the platform $\mathcal{F}_2$ for four random populations of four sizes when the number of objectives changes from $M = 5$ to $M = 30$

On platform $\mathcal{F}_3$, there are enough cores for the number of threads spawned, so the situation where multiple threads execute concurrently on the same core does not appear. So, the slope of the performance as M increases is much more gradual.

Additionally, experiments on NDS were performed on the basis of the NSGA-II algorithm. Populations after several NSGA-II generations with DTLZ2 test functions [13] with $M = 5, \ldots, 30$ have also been analysed. This experimental study has not been included in this section because the acceleration factors achieved by MC-BOS and GPU-BOS, in terms of performance and energy, are similar to the study carried out for random populations. Therefore, the general conclusions of this study were the same.

The advantages in performance and energy consumption of both parallel versions in relation to the sequential BOS are analysed in Figs. 5, 6 and 7 for the three test platforms respectively. These plots have been obtained from the runtime and energy consumption above shown in Figs. 2, 3 and 4. The acceleration factors of MC-BOS vs BOS (GPU-BOS vs BOS) range from 3× to 14× (2× to 14.5×) on platform $\mathcal{F}_1$, 2× to 8× (2× to 17.5×) on platform $\mathcal{F}_2$ and 3× to 15.6× on platform $\mathcal{F}_3$. The energy savings factors are slightly lower, from 1.5× to 5× (0.5× to 2.8×) on platform $\mathcal{F}_1$, 1× to 3× (0.5× to 6×) on platform $\mathcal{F}_2$ and 2× to 8× on platform $\mathcal{F}_3$.

Summarizing, parallel versions are faster and consume less energy than the sequential BOS in relevant percentages, in spite of the irregularity of both parallel algorithms. If the number of objectives is less than the number of cores in the processor, then the best option is the multicore version to optimize performance and energy. If there are many objectives, then, the GPU version is the optimal selection.

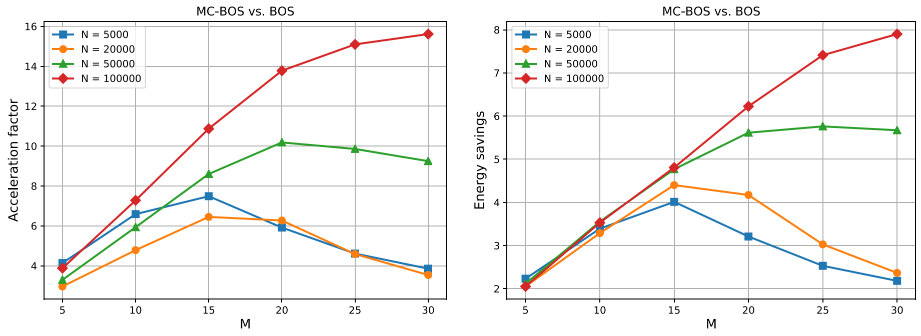

Fig. 7 Experimental acceleration factors of runtime and energy savings for the multicore version in relation to the sequential BOS on the platform $\mathcal{F}_3$ for four random populations of four sizes when the number of objectives changes from $M = 5$ to $M = 30$

5 Conclusions

This work has proposed two parallel procedures, MC-BOS and GPU-BOS, to improve the performance and the energy consumption to compute Non-Dominated Sorting on multicore and GPU architectures. Both procedures are based on the principles of the Best Order Sort algorithm and their source code is publicly available at GitHub[7]. It is a state-of-the-art procedure to efficiently rank populations avoiding unnecessary comparisons to a sequential structure. To improve the performance of BOS by the exploitation of modern multicore processors and GPUs, two schemes have been developed to increase the parallelism level of this algorithm. MC-BOS defines the same number of threads as objectives and efficiently exploits multicore processors. GPU-BOS defines a scheme that tries to reduce the number of comparisons needed for sorting without the need for large dominance data structures. It reduces both the comparisons and the memory requirements in relation to other GPU versions of NDS.

From the evaluation results, it can be concluded that both parallel algorithms improve the performance of the BOS algorithm in factors that reach 17.5× and reduce the energy consumption in factors that reach 8×. The multicore version is the best option to optimize performance and energy consumption when the number of objectives is not excessive, otherwise the best option is the GPU version. These results are a milestone due to the irregularity of NDS procedures that optimize the sequential performance, such as BOS. They are inherently sequential and a re-definition of schemes of the parallel algorithms to exploit the moderate and massive parallelism of multicore processors and GPUs has been necessary.

Our future work is focused on the evaluation of MC-BOS and GPU-BOS on novel architectures, such as Skylake processors of Intel and Volta GPUs of NVIDIA.

References

1. Brodtkorb, A.R., Hagen, T.R., Sætra, M.L.: Graphics processing unit (GPU) programming strategies and trends in GPU computing. J. Parallel Distrib. Comput. **73**(1), 4–13 (2013)
2. Buzdalov, M., Shalyto, A.: A provably asymptotically fast version of the generalized Jensen algorithm for non-dominated sorting. In: International Conference on Parallel Problem Solving from Nature, pp. 528–537. Springer, Berlin (2014)

[7] https://github.com/juanjonrg/nds.

3. David, H., Gorbatov, E., Hanebutte, U.R., Khanna, R., Le, C.: Rapl: memory power estimation and capping. In: 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED), pp. 189–194 (2010). https://doi.org/10.1145/1840845.1840883
4. Deb, K., Jain, H.: An improved NSGA-II procedure for many-objective optimization, Part I: Solving problems with box constraints. KanGAL Report (2012009) (2012)
5. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
6. 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)
7. 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)
8. Filatovas, E., Kurasova, O., Sindhya, K.: Reference point based multi-objective optimization using evolutionary algorithms. Informatica **26**(1), 33–50 (2015)
9. Fortin, F.A., Grenier, S., Parizeau, M.: Generalizing the improved run-time complexity algorithm for non-dominated sorting. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, GECCO ’13, pp. 615–622. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2463372.2463454
10. Gupta, S., Tan, G.: A scalable parallel implementation of Evolutionary Algorithms for Multi-Objective optimization on GPUs. In: CEC, pp. 1567–1574. IEEE (2015)
11. Gustavsson, P., Syberfeldt, A.: A new algorithm using the non-dominated tree to improve non-dominated sorting. Comput. Evol. (2017). https://doi.org/10.1162/EVCO_a_00204
12. Harris, M.: Maxwell: the most advanced CUDA GPU ever made (2014). https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/
13. Huband, S., Hingston, P., Barone, L., While, L.: A review of multiobjective test problems and a scalable test problem toolkit. IEEE Trans. Evol. Comput. **10**(5), 477–506 (2006)
14. 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)
15. Jensen, M.T.: Reducing the run-time complexity of multiobjective EAs: the NSGA-II and other algorithms. IEEE Trans. Evol. Comput. **7**(5), 503–515 (2003)
16. Knowles, J.D., Corne, D.W.: Approximating the non-dominated front using the Pareto archived evolution strategy. Evol. Comput. **8**(2), 149–172 (2000)
17. Markina, M., Buzdalov, M.: Hybridizing non-dominated sorting algorithms: divide-and-conquer meets best order sort 2017). CoRR arxiv:1704.04205
18. Miettinen, K.: Nonlinear Multiobjective Optimization. Springer KK, Tokyo (1999)
19. Moreno, J.J., Ortega, G., Filatovas, E., Martínez, J.A., Garzón, E.M.: Using low-power platforms for evolutionary multi-objective optimization algorithms. J. Supercomput. **73**(1), 302–315 (2017)
20. NVIDIA: NVIDIA’s next generation CUDA compute architecture: Kepler GK110 (2012). https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf
21. Ortega, G., Filatovas, E., Garzón, E.M., Casado, L.G.: Non-dominated sorting procedure for Pareto dominance ranking on multicore CPU and/or GPU. J. Glob. Optim. **69**(3), 607–627 (2017)
22. Ponsich, A., Jaimes, A.L., Coello, C.A.C.: A survey on multiobjective evolutionary algorithms for the solution of the portfolio optimization problem and other finance and economics applications. IEEE Trans. Evol. Comput. **17**(3), 321–344 (2013)
23. Roy, P.C., Islam, M.M., Deb, K.: Best order sort: A new algorithm to non-dominated sorting for evolutionary multi-objective optimization. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, GECCO ’16 Companion, pp. 1113–1120. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2908961.2931684
24. Roy, P.C., Islam, M.M., Murase, K., Yao, X.: Evolutionary path control strategy for solving many-objective optimization problem. IEEE Trans. Cybern. **45**(4), 702–715 (2015)
25. Smutnicki, C., Rudy, J., Żelazny, D.: Very fast non-dominated sorting. Decis. Mak. Manuf. Serv. **8**(1–2), 13–23 (2014)
26. Srinivas, N., Deb, K.: Muiltiobjective optimization using nondominated sorting in genetic algorithms. Evol. Comput. **2**(3), 221–248 (1994). https://doi.org/10.1162/evco.1994.2.3.221
27. Tang, S., Cai, Z., Zheng, J.: A fast method of constructing the non-dominated set: Arena’s principle. ICNC **1**, 391–395 (2008)
28. Wong, M.L.: Parallel multi-objective evolutionary algorithms on graphics processing units. In: GECCO, pp. 2515–2522. ACM (2009)
29. Zhang, X., Ye, T., Cheng, R., Jin, Y.: An efficient approach to non-dominated sorting for evolutionary multi-objective optimization. IEEE Trans. Evol. Comput. **19**(2), 201–213 (2012)

30. 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)
31. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength Pareto evolutionary algorithm. Technical Report 103, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Zurich, Switzerland (2001)