Noname manuscript No.
(will be inserted by the editor)

Improving the Energy-Efficiency of SMACOF for Multidimensional Scaling on Modern Architectures

F. Orts · E. Filatovas · G. Ortega · O. Kurasova · E.M. Garzón

Received: date / Accepted: date

Abstract The reduction of the dimensionality is of great interest in the context of big data processing. Multidimensional scaling methods (MDS) are techniques for dimensionality reduction, where data from a high-dimensional space are mapped into a lower-dimensional space. Such methods consume relevant computational resources therefore intensive research has been developed to accelerate them. In this work, two efficient parallel versions of the well-known and precise SMACOF algorithm to solve MDS problems have been developed and evaluated on multicore and GPU. To help the user of SMACOF, we provide these parallel versions and a complementary Python code based on a heuristic approach to explore the optimal configuration of the parallel SMACOF algorithm on the available platforms in terms of energy efficiency (GFLOPs/watt). Three platforms, 64 and 12 CPU-cores and a GPU device, have been considered for the experimental evaluation.

Keywords Dimensionality Reduction · Multidimensional Scaling · Energy efficiency · SMACOF algorithm

This work has been partially supported by the Spanish Ministry of Science throughout projects TIN2015-66680 and CAPAP-H5 network TIN2014-53522, by J. Andalucía through projects P12-TIC-301 and P11-TIC7176, and by the European Regional Development Fund (ERDF).

F. Orts, G. Ortega and E.M. Garzón
Group of Supercomputation-Algorithms, Dpt. of Informatics, Univ. of Almería, ceiA3, 04120, Almería, Spain
E-mail: francisco.orts@ual.es, gloriaortega@ual.es, gmartin@ual.es

E. Filatovas
Faculty of Fundamental Science, Vilnius Gediminas Technical University, Saulėtekio avn. 11, LT-10223 Vilnius, Lithuania
E-mail: ernest.filatov@gmail.com

O. Kurasova
Inst. of Data Science and Digital Technologies, Vilnius Univ., Akademijos str. 4, LT-08663, Vilnius, Lithuania
E-mail: olga.kurasova@mii.vu.lt

1 Introduction

Real-world data, such as speech signals, images, biomedical, financial, telecommunication and other data usually have a high dimensionality as each data instance (point) is characterized by a set of features. The dimensionality of such data, as well as the amount of data to be processed, is constantly increasing therefore the requirement of processing these data within a reasonable time frame still remains an open problem. Dimensionality reduction methods which aim to map high dimensional data into a lower dimensional space play extremely important role when exploring large datasets. Among such methods Multidimensional Scaling (MDS) remains one of the most popular [5,9].

One of the dimensionality reduction applications is a graphical visualization of the structure of the high-dimensional data in 2D or 3D space for easier data understanding. Some applications in this line can be found in [13,18,22]. Moreover, MDS has proven to be useful as a technique to evaluate criteria of objects classification [15] or discover criteria which initially had not been taken into account [4], serving as a psychological model that allows to discover human patterns [16].

A well-known algorithm for MDS is called SMACOF (Scaling by Majorizing a COmplicated Function) [8]. The experimental investigation has demonstrated that SMACOF is most accurate algorithm comparing to others [17]. It should be noted that the SMACOF algorithm is the most expensive, as its complexity is $O(m^2)$, where m is the number of observations. Several different approaches have been developed to reduce computational complexity of the MDS techniques. In [23], the complexity was reduced to $O(m\sqrt{m})$ by developing iterative MDS spring model. In [32], authors reduced the complexity to $O(m \log m)$ by dividing the original matrix into sub-matrices and then combining the sub-solutions to obtain a final solution. The improved versions of MDS reduce complexity insignificantly, however, optimization accuracy suffers [17]. Consequently, SMACOF version of MDS is usually chosen as it ensures the sufficient accuracy that is essential in many dimensional cases. In short, the MDS techniques remain in high time complexity order therefore parallel strategies should be considered to accelerate the computation of the MDS procedure [24].

During the last decade the High Performance Computing (HPC) has greatly improved and has been widely-applied for MDS techniques. In [29], authors proposed a MDS parallel implementation and explored it under MPI and other libraries. In [12], Fester et al. proposed a CUDA implementation of MDS algorithm based on the high throughput multidimensional scaling (HiT-MDS). In [28], authors suggested a new efficient parallel GPU algorithm for MDS based on virtual particle dynamics [10] and experimentally compared it with multicore CPU version. In [17], the multi-level MDS Glimmer algorithm was developed for GPU by dividing the input data into hierarchical levels and executing the algorithm recursively. It must be noted that currently Glimmer is the most well-known and widely-used GPU tool for MDS. Another CUDA-based technique to get MDS approximation is CFMDS [27] that implements both single-level and multi-level approaches. In [26], authors proposed a corre-

lation clustering framework which uses MDS for layout and GPU-acceleration to speedup visual feedback. In [25], GPU version of MDS was developed to improve content-based image retrieval (CBIR) systems. Summarizing, the research on this HPC field is being carried out actively, it remains relevant as the new GPU architecture and heterogeneous platforms constantly appear, and should be effectively exploited for solving dimensionality reduction problems of different complexities.

Currently, the target of HPC includes the optimization of energy consumption. The ratio of the computational speed to the electrical power (*GFLOPs/watt*) is usually defined as a parameter that is a suitable indicator of the energy efficiency [19]. The increase of this parameter means that the system achieves better performance (GFLOPs) with less electrical power (watts) and, as consequence, less energy is consumed. Therefore, for the optimal parallel executions of SMACOF, the ratio should be maximized.

In this paper, parallel versions of the SMACOF algorithm on multicore and GPU are developed and evaluated on prototypes of modern architectures. As the parallel SMACOF algorithm can be executed on different alternative platforms, the kind of platform and its resources that optimize the runtime and/or energy efficiency need to be to determined. Bearing in mind that the parallel performance depends on the problem sizes, the users of parallel SMACOF need support to configure it. For this purpose, a benchmarking process to find the optimal solutions has been developed. It is based on a heuristic approach which combines two concepts: the analysis of the first iterations of SMACOF representative computation and functional models of performance and power consumption of homogeneous parallel platforms. The benchmarking process has been evaluated using different platforms (multicore and GPU) and various sizes of the problem. Moreover, the energy efficiency of SMACOF has been experimentally evaluated on two different multicore platforms and a GPU device.

The paper is organized as follows. In Section 2, the descriptions of the Multidimensional Scaling and the SMACOF algorithm are provided. Section 3 describes the proposed multicore and GPU parallel implementations of the SMACOF algorithm. In Section 4, the algorithm for tuning the energy efficiency of SMACOF is presented. Experimental evaluations of the parallel implementations on three platforms are discussed in Section 5. Finally, conclusions are drawn in Section 6.

2 SMACOF algorithm for MDS

Multidimensional Scaling is a technique for the analysis of similarity or dissimilarity data on a set of objects (items). It aims at finding points $Y_1, Y_2, \ldots, Y_m \equiv Y$ in the low-dimensional space $\mathbb{R}^s$, $s < n$, such that the distances between them are as close as possible to the distances between the original points $X_1, X_2, \ldots, X_m \equiv X$ in the space $\mathbb{R}^n$. This is achieved by minimizing the

Algorithm 1 SMACOF(m, s, Δ, $kmax$, ϵ, Y)

Require:
m: number of items;
s: dimension of low-dimensional space;
Δ: $m \times m$ matrix of dissimilarities of observed data on the high-dimensional space (n);
$kmax$: maximum number of iterations;
ϵ: threshold for the stress variance
Ensure:
Y: set of finding points in the low-dimensional space stored in a $m \times s$ matrix
1: Initial Solution randomly generated, Y^0
2: **Compute Euclidean distances**, $D^0 = [d(Y_i^0, Y_j^0)]$ ▷ $O(m^2 s)$
3: $k = 0$, $error = 1$
4: **if** $(k < kmax)$ and $(error > \epsilon)$ **then**
5: **Compute Guttman transform matrix**, $B^k \equiv B^k(\Delta, D^{k-1})$ (Alg. 2) ▷ $O(m^2)$
6: **Compute Guttman transform**, $Y^k = 1/m \cdot B^k \cdot Y^{k-1}$ ▷ $O(m^2 s)$
7: **Update distances** $D^k = [d(Y_i^k, Y_j^k)]$ ▷ $O(m^2 s)$
8: **Compute** E_{MDS}^k (Eq. 1)
9: $error = |E_{MDS}^k - E_{MDS}^{k-1}|$
10: $k = k + 1$
11: **return** Y

stress function:

$$E_{MDS} = \sum_{i<j} \Big(\delta_{ij} - d(Y_i, Y_j)\Big)^2 \tag{1}$$

Here $d(\cdot,\cdot)$ (δ) is the distance between two points in the low-dimensional space (multi-dimensional space).

There are many strategies to solve MDS problems [9]. We focus our attention on the well-known SMACOF algorithm which is based on a particular minimization process of the stress function [8]. The theoretical background of SMACOF is simpler and more powerful than other approaches from convex analysis, because it guarantees monotone convergence of stress [5]. SMACOF has demonstrated better results when optimizing stress function comparing to other proposals in the literature [17]. The main idea is based on the majorizing concept which consists in approximating a complex function by another one simpler. This method iteratively finds a new function, which is located above the original function and touches at the supporting point. At every iteration of the algorithm, the minimum of the new function is closer of the minimum of the complex function, in our case the stress function [5]. SMACOF can be expressed by Alg. 1 in which the complexity order of the most relevant tasks appeared between parenthesis.

Algorithm 1 has a high computational cost and high memory requirements due to the large data structures involved: input matrix Δ ($m \times m$), output and auxiliary matrices ($m \times s$) and three auxiliary matrices ($m \times m$) to store the similarities among the objects of the low-dimensional space. The symmetry has not been exploited in the storage of the data structures, however, it has been considered for the above-mentioned matrices update. Bearing in mind this fact, the number of floating point operations of Alg. 1 is: $3s/2m^2 + 3s/2m$

Algorithm 2 Two pseudocodes to compute B^k (line 5 of Alg 1): On the left, the approach from Eq. 8.24 of [5]; on the right, the two nested loops are collapsed in only one to later obtain a balanced parallel execution of Guttman transform matrix.

Require:
m: number of *items*;
Δ: $[\delta_{ij}]$, $m \times m$ matrix of dissimilarities;
D: $[d_{ij}]$, Euclidean distances matrix
Ensure:
B: $[b_{ij}]$, Guttman transform matrix
1: **for** $i = 0; i < m; i++$ **do**
2: **for** $j = i + 1; j < m; j++$ **do**
3: **if** $d_{ij} \neq 0$ **then**
4: $b_{ij} = -\delta_{ij}/d_{ij}$
5: **else**
6: $b_{ij} = 0$
7: **for** $i = 0; i < m; i++$ **do**
8: $b_{ii} = -\sum_{j=1, j\neq i}^{m} b_{ij}$
9: **return** B

Require:
m: number of *items*;
Δ: $[\delta_{ij}]$, $m \times m$ matrix of dissimilarities;
D: $[d_{ij}]$, Euclidean distances matrix
Ensure:
B: $[b_{ij}]$, Guttman transform matrix
1: **for** $l = 0; l < (m \cdot (m+1)/2); l++$ **do**
2: $i = \lfloor l/(m+1) \rfloor, j = l\%(m+1)$
3: **if** $j > i$ **then**
4: $i = m - i - 1, j = m - j$
5: **if** $d_{ij} \neq 0$ **then**
6: $b_{ij} = -\delta_{ij}/d_{ij}$
7: **else**
8: $b_{ij} = 0$
9: $b_{ji} = b_{ij}$
10: **for** $i = 0; i < m; i++$ **do**
11: $b_{ii} = -\sum_{j=1, j\neq i}^{m} b_{ij}$
12: **return** B

for the initialization (line 2 of Alg. 1) and $(7/2s + 3/2)m^2 + 1/2(3s + 1)m$ for the iterative process.

3 Parallel implementations of the SMACOF algorithm

The SMACOF computational cost is $O(s \cdot m^2)$ and memory requirements are $O(m^2)$. This feature has limited for years the applicability of SMACOF to solve large MDS problems. The use of HPC techniques helps to overcome this drawback. In this work, we propose two parallel versions based on the exploitation of large-scale modern multicore and GPU architectures. This section is devoted to describing these parallel implementations.

Both implementations are focused on the parallel execution of the computation of the Euclidean distances matrices (lines 2 and 7 of Alg. 1) and the Guttman transform (lines 5 and 6 of Alg. 1). Parallel procedures are highlighted in bold in Alg. 1. To calculate the outputs of these procedures, we have taken into account that we are working with symmetric matrices (B^k, D^k and Δ). For example, to compute the symmetric matrix B^k (which defines Guttman transform) is only necessary to calculate a triangular sub-matrix of $L = (m \cdot (m + 1)/2)$ elements. Thus, B^k can be managed as a unidimensional vector of L elements which can be updated in parallel. To distribute this computation among the processing elements, the left part of Alg. 2 has been transformed into the right one. This way, two nested loops are collapsed into a regular loop to compute the triangular matrix of L elements. It can be

easily parallelized with maintaining the load balance. This idea has also been applied to the parallel computation of D^k.

The multicore version has been implemented using C, OpenMP [6] and MKL library [3]. The parallel computations of B^k and D^k consider the symmetry of these matrices. Therefore, Alg. 2 on the right is taken as reference for the parallel computation of B^k. The l-loop of such algorithm is distributed among the cores and when it has finished a synchronization point is included to ensure that the non-diagonal elements of B^k are computed before starting the parallel i-loop. Moreover, the MKL library (concretely the *cblas_dgemm* routine) is in charge of computing in parallel the matrix-matrix product linked to the Guttman transform (line 6 of the Alg. 1).

In the GPU version, three kernels have been coded using C and CUDA to compute in parallel D^k (lines 2 and 7 of Alg. 1) and B^k (line 5 of Alg. 1). The Euclidean distances require one kernel, and the Guttman transform requires two, as it is explained below. To compute the distances matrix, every thread updates two symmetric elements of D^k matrix. Moreover, shuffle instructions have been used for the reductions involved in the computation of D^k elements. These instructions, available from Kepler NVIDIA architecture, essentially allow threads in the same warp to share information. They can improve the reduction processes [2]. In our experiments, shuffle instructions have demonstrated to improve the performance compared to the reductions based on shared memory. We have observed that the advantage of shuffle instructions versus the shared memory version increases with s. Specifically, we have evaluated the performance for sizes of problem from $m = 10000$ to $m = 40000$ with $s = 64$ and the shuffle version has obtained the same or better performance (up 30%) than shared memory version in the computation of D^k matrix (lines 2 and 7 of Alg. 1).

The CUDA version of Alg. 2 to compute B^k on GPU consists of two kernels. In the first kernel, each thread starts by calculating a non-diagonal element of B^k. Next, its symmetric element is copied without requiring any synchronization. When this kernel finishes, the second one computes the diagonal elements from the non-diagonal ones. For Y^k, *cublasDgemm* routine from the cuBLAS library [1] has been used to accelerate matrix-matrix product on GPU (line 6 of Alg. 1).

4 Tuning the Energy Efficiency of the SMACOF algorithm

In this work, two parallel implementations have been developed to accelerate the SMACOF algorithm. When solving real-world problems, it is reasonable to run the most energy efficient parallel SMACOF version on a particular subset of resources of available computational platforms.

The idea consists in an initial benchmarking that identifies, for every available platform, the optimal selection of resources for a size of problem of interest. Then, the user can choose the optimal platform for the subsequent execution of the SMACOF algorithm. According to the developed parallel

versions, multicore processors and GPUs are considered as target platforms in this work.

The Energy-Efficiency (EE) is usually defined as the ratio of the computational speed to the electrical power, that is *GFLOPs/watt* [19]. Therefore, for the optimal parallel executions of SMACOF, the ratio should be maximized.

The optimization of the EE of parallel applications on modern platforms can be viewed as a problem of scheduling parallel machines with costs [30]. The parallel SMACOF versions can be executed on one of the alternative platforms, for example the different multicore or GPUs architectures. Every platform is denoted by $\mathcal{F}^k \in \mathcal{F}$, $k = 1, \ldots, f$ where $\mathcal{F}$ is the set of f available parallel platforms. Every platform $\mathcal{F}^k$ consists in a set of parallel machines $\mathcal{M}^k$, $\mathcal{F}^k = \{\mathcal{M}_i^k\}_{i=1}^{c_k}$ where c_k is the number of available machines of the platform k. The corresponding energy efficiency depends on the number of machines involved in the computation and the particular input size.

Then, the solution of the scheduling problem corresponds to the subset of platforms $\mathcal{F}^{k_o} \subseteq \mathcal{F}$ with their optimal configurations defined by the machines number $r_{k_o}^o$ that optimizes EE ($r_{k_o}^o \leq c_{k_o}$). We propose a heuristic approach for solving this problem. It is based on a functional model of EE for modern platforms (multicore and GPUs) and the definition of the significant computation in the SMACOF algorithm.

The functional performance models were introduced by Lastovetsky [7,33]. The processor performance depends on the problem size and can be empirically estimated by a benchmarking process. In this way, the modeling performance depends on the combination of the architecture and the application. In similar lines other authors have been focussed on the benchmarking and they have proposed the concepts of application signature and Small-Scale Executions [11, 31]. If the parallel application is iterative, then a subset of iterations can define the significant portion of the application and can be used in the benchmarking [20,21].

The models to estimate EE have to combine performance and power. Previous works have proposed functional models for the EE estimation on iterative applications [14]. If it is focused on a particular execution of the application with F floating point operations on one homogeneous platform k, and it is assumed a perfect load balance among r_k actives machines, then the following model of EE as function of r_k is reasonable:

$$EE(r_k) = \frac{F}{\mathcal{T}^k(r_k)\mathcal{P}^k(r_k)} = \frac{F}{(\frac{\mathcal{T}^k(1)}{r_k} + \mathcal{TC}^k(r_k))(\mathcal{P}_{idle}^k + r_k p^k(r_k))} \quad (2)$$

where $\mathcal{T}^k(r_k)$ and $\mathcal{P}^k(r_k)$ are the runtime and power consumption on r_k machines respectively, $\mathcal{TC}^k(r_k)$ represents the runtime penalties due to the contention among the actives machines on the k platform, $\mathcal{P}_{idle}^k$ represents the idle power consumption when no process is actively using any machine and $p^k(r_k)$ is the contribution to the power of every machine.

According to this model $\mathcal{T}^k(r_k)$, one minimum for a number of active machines is obtained since $\mathcal{TC}^k(r_k)$ is an increasing function and $\mathcal{P}^k(r_k)$ is

also an increasing function for r_k. Therefore, $EE(r_k)$ achieves a maximum for $r^o_{k_o}$ machines.

Then, from the point of view of the SMACOF usage, to optimize EE, it should be identified r^o_k on the set of available platforms for the sizes problem and choose the platform k_o which optimizes EE, i.e. achieves $EE(r^o_{k_o})$. Modern computers provide two different platforms, multicore processors and GPUs and the number of kinds of platforms can increase if clusters of heterogeneous nodes with several kinds of multicore and GPUs platforms are available. We have defined a heuristic to decide what is the best platform to run their particular instances of the parallel SMACOF. Our proposal is organized in two stages, first the identification of the optimal configuration of every platform and second the selection of optimal platforms and configurations. Previous considerations about the EE model help us to define an efficient benchmarking exploration to find the optimal configurations on every platform. Therefore, selective search described in Algorithm 3 can be used to find the optimal platforms and their configurations in the benchmarking process.

As above mentioned, the benchmarking is usually based on the execution of a significant core of the application. SMACOF consists in an iterative procedure to compute the Guttman transforms. The computational cost of every iteration is the same, therefore a subset of iterations can be considered as the SMACOF significant core to compute the profiling in a efficient way. SMACOF can be configured using the information provided by this preprocess based on exploration of several resources selection on particular combinations of platforms and data sizes.

5 Experimental Evaluation

In this section, the SMACOF algorithm to solve MDS problems is evaluated in terms of runtime and energy efficiency on three computational architectures:

$\mathcal{F}_1$: Bullion S8: 4 Intel Xeon E7 8860v3 (16 × 4 CPU-cores);
$\mathcal{F}_2$: Bullx R421-E4 Intel Xeon E5 2620v2 (12 CPU-cores and 64 GB RAM);
$\mathcal{F}_3$: NVIDIA K80 (composed by two Kepler GK210 GPUs) connected to the host Bullx R421-E4 Intel Xeon E5 2620v2.

$\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$ run Ubuntu 16.04 LTS and $\mathcal{F}_3$ runs CUDA Toolkit 8. The programs have been compiled using gcc 5.4.0 and nvcc 8.0.44 with optimization flags O3 for GPU architecture 3.5. For the acquisition of energy measurement data, we have collected this information from various hardware counters. For Intel, we have used the Running Average Power Limit (RAPL) interface and, for NVIDIA, the NVIDIA Management Library (NVML).

For the evaluation of SMACOF, test problems of different sizes defined by values of m, n and s have been considered (see Table 1). For this experimental investigation, randomly generated input data were used. The number of evaluated iterations of the SMACOF algorithm has been 100.

Algorithm 3 Heuristic for computing the set of optimal platforms $\{k_o\}$, with their configurations $\{r^o_{k_o}\}$, which optimize the EE of the SMACOF

Require:
$\mathcal{F} = \{\mathcal{F}^k\}_{k=1}^{f}$ with $\mathcal{F}^k = \{\mathcal{M}_i^k\}_{i=1}^{c_k}$; ▷ Set of platforms
Parallel versions of SMACOF(m, s, Δ, $kmax$, ϵ, Y) to execute on the f available platforms;
m (items), s (output dimensions); ▷ Particular data size *sampling*.
Ensure:
$\{k_o, r^o_{k_o}\}$ optimize the EE on the f available platforms
1: Evaluate the number of FLOAT operations of SMACOF(m, s, Δ, $kmax$, ϵ, Y)
2: **for** $k \leftarrow 1$ **to** f **do**
3: Execute Parallel SMACOF(m, s, Δ, $kmax$, ϵ, Y) on $r_k = c_k$ machines and evaluate its EE denoted by $\mathcal{EE}^k$
4: **for** $i \leftarrow c_k - sampling$ **to** $sampling$ **do**
5: Execute Parallel SMACOF(m, s, Δ, $kmax$, ϵ, Y) on $r_k = i$ machines and evaluate $\mathcal{EE}^{Aux}$
6: **if** $\mathcal{EE}^{Aux} \leq \mathcal{EE}^k$ **then**
7: $r^o_k = i + sampling$
8: Break i-loop
9: **else**
10: $\mathcal{EE}^k = \mathcal{EE}^{Aux}$
11: Select the platforms $\{k_o\}$ with their optimal configurations $\{r^o_{k_o}\}$ which maximize EE
12: **return** $\{k_o, r^o_{k_o}\}$

Table 1 Test problems using several number of items (m), dimensions of multi-dimensional space (n), and dimensions of low-dimensional space (s).

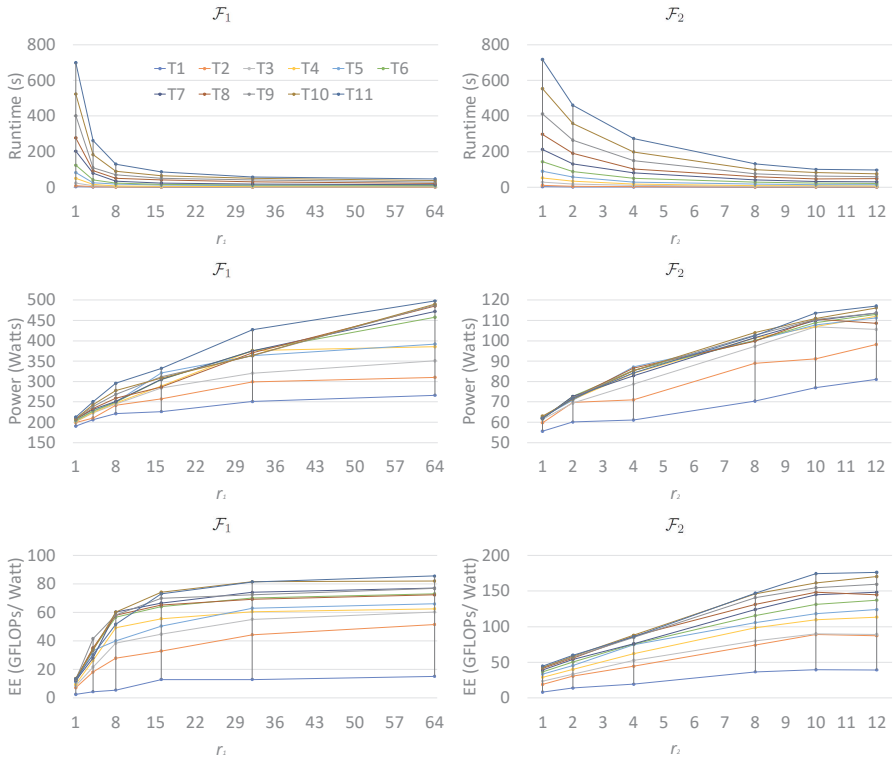
	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$	$T7$	$T8$	$T9$	$T10$	$T11$
m	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000	22000
n	100	200	300	400	500	600	700	800	900	1000	1100
s	2	3	4	5	6	7	8	9	10	11	12

Table 2 Runtime, power and energy efficiency of the set of test problems (Table 1) on $\mathcal{F}_3$ (GPU) platform.

$\mathcal{F}_3$	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$	$T7$	$T8$	$T9$	$T10$	$T11$
Time (s)	2.8	4.9	5.1	5.9	6.5	11.0	18.8	28.7	42.3	64.9	91.5
Power (Watts)	38.7	98.6	105.2	108.0	112.6	113.6	112.8	110.1	112.3	111.5	110.3
EE (GFLOPs/ Watt)	13.8	24.7	75.1	150.0	255.1	257.2	240.7	241.7	228.7	205.9	196.6

Figure 1 shows, the runtime, power and energy efficiency of the set of test problems on $\mathcal{F}_1$ and $\mathcal{F}_2$ (multicore) platforms and Table 2 shows similar parameters on $\mathcal{F}_3$ (GPU) platform with the same test cases. Execution times of the multicore versions (plotted on the top of Fig. 1) are according with runtime models described in Section 4. The runtime decreases with the values of r_1 and r_2, therefore the best performance is achieved for the maximum number of cores. The experimental power measurements are plotted in the middle of Fig. 1. It is remarkable that the temporal evolution of the power partially depends on unpredictable factors for programmers. To overcome this drawback it has been necessary to collect the measurements after an activity period on

Fig. 1 Runtime (top), power (middle) and energy efficiency (bottom) of the set of test problems (Table 1) on $\mathcal{F}_1$ and $\mathcal{F}_2$ platforms.



the processor to minimize their variance due to changes in the temperature. This instability can be observed in the power plot for both platforms, but we can conclude that power consumption trend increases as the number of cores and the size of the problem.

Focusing our attention on the energy efficiency (plotted on the bottom of Fig. 1) it increases as the number of cores. The highest values of r_1 and r_2 optimize the energy efficiency for the plateau in the plot. Therefore, the optimal value of r_k in both platforms is in a wide interval, for instance 32-64 (10-12) for $\mathcal{F}_1$ ($\mathcal{F}_2$).

To choose the optimal platform, we could compare the three platforms in terms of performance. This way, the best option for $T11$ is $\mathcal{F}_1$, since the execution times are 46.6s, 96.2s and 91.5s on $\mathcal{F}_1$ with $r_1^o = 64$, $\mathcal{F}_2$ with $r_2^o = 12$ and $\mathcal{F}_3$ respectively. This selection is the same for all test cases. If we focus on the energy efficiency, the best option is the GPU when the problem size is enough high since it consumes less power than $\mathcal{F}_1$ and achieves a reasonable performance. Then, to optimize the energy efficiency, the best option is the use of the GPU platform for the test cases $T4 - T11$. For instance for $T11$, the energy efficiencies on the different platforms are 85.5, 176.1 and 196.6

Table 3 Sampling of EE for $T11$ test according to the benchmarking proposed in Alg. 3 for multicore platforms $\mathcal{F}_1$ and $\mathcal{F}_2$.

$\mathcal{F}_1$		
r_1	64	61
EE (GFLOPs/Watt)	85.5	85.0

$\mathcal{F}_2$		
r_2	12	9
EE (GFLOPs/Watt)	176.1	155.8

GFLOPs/watt for $\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$, respectively. The best platform for $T1-T3$ is the multicore $\mathcal{F}_2$ which consumes less power than $\mathcal{F}_1$.

These results support the benchmarking process explained in Section 4 to explore in an automatic way the selection of the optimal parallel platform and its best resource selection. This procedure has been developed in Python. We have chosen $sampling = 3$ to obtain relevant differences between successive experimental evaluations on both platforms. The results support the idea of starting the benchmarking process by the highest numbers of CPU-cores available on every platform to find the optimal r_k is efficient. To illustrate the behaviour of the benchmarking (Alg. 3) for multicore platforms, we focus on the $T11$ test. Table 3 shows the EE obtained when a set of ten iterations of SMACOF are executed on platforms $\mathcal{F}_1$ and $\mathcal{F}_2$. Only two samples for the benchmarking exploration are required for $T11$ since $r_1^o = 64$ and $r_2^o = 12$ are identified by the preprocess. So, we can conclude that the proposed benchmarking can execute an efficient exploration to optimize the energy efficiency of parallel SMACOF.

6 Conclusions

This work has analyzed an approach to optimize the energy efficiency (GFLOPs/watt) of the SMACOF algorithm, a well-known and precise method to solve MDS problems. Two parallel versions of SMACOF, multicore and GPU, have been developed and evaluated. To help the user of SMACOF parallel codes, we provide these versions and a complementary Python code based on a heuristic approach to explore the optimum configuration of the available platforms.

An experimental evaluation has been carried out on three platforms based on architectures with 64CPU-cores, 12CPU-cores and a GPU device. The results show 64-cores processor is the best platform to optimize the runtime of SMACOF; the 12-cores processor is the best option to improve the energy efficiency for the smallest test problems and, for the largest test problems, the optimal energy efficiency is achieved on the GPU.

In currently known parallel versions of SMACOF only the runtime is considered, and neither the energy consumption nor adaptive capability to the platform and problem-size are optimized. Therefore, our SMACOF implementation is of great interest for developing energy efficiency aware applications based on MDS problems. Our implemented versions of the SMACOF algorithm are freely available through the following website:

https://github.com/2forts/SMACOF. As future work, we consider to implement a distributed parallel version of SMACOF and to analyze and develop other methods for solving MDS problems.

References

1. cuBLAS library (2017). URL `http://docs.nvidia.com/cuda/cublas/index.html`
2. CUDA Pro Tip: Do The Kepler Shuffle (2017). URL `https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle/`
3. Intel Math Kernel Library (Documentation) (2017). URL `https://software.intel.com/en-us/mkl/documentation`
4. Bilsky, W., Borg, I., Wetzels, P.: Assessing conflict tactics in close relationships: A reanlysis of a research instrument. Facet theory: Analysis and design p. 3946 (1994)
5. Borg, I., Groenen, P.J.: Modern multidimensional scaling: Theory and applications. Springer Science & Business Media (2005)
6. Chapman, B., Jost, G., Pas, R.v.d.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press (2007)
7. Clarke, D., Ilic, A., Lastovetsky, A., Rychkov, V., Sousa, L., Zhong, Z.: Design and Optimization of Scientific Applications for Highly Heterogeneous and Hierarchical HPC Platforms Using Functional Computation Performance Models, pp. 235–260. John Wiley & Sons, Inc. (2014)
8. De Leeuw, J.: Applications of convex analysis to multidimensional scaling. Recent Developments in Statistics pp. 133–145 (1977)
9. Dzemyda, G., Kurasova, O., Žilinskas, J.: Multidimensional Data Visualization: Methods and Applications, vol. 75. Springer Science & Business Media (2013)
10. Dzwinel, W., Blasiak, J.: Method of particles in visual clustering of multi-dimensional and large data sets. Future Generation Computer Systems **15**(3), 365–379 (1999)
11. Escobar, R., Boppana, R.V.: Performance prediction of parallel applications based on small-scale executions. In: 2016 IEEE 23rd HiPC, pp. 362–371 (2016)
12. Fester, T., Schreiber, F., Strickert, M.: CUDA-based Multi-core Implementation of MDS-based Bioinformatics Algorithms". In: I. Grosse, S. Neumann, S. Posch, F. Schreiber, P.F. Stadler (eds.) GCB, *LNI*, vol. 157, pp. 67–79. GI (2009)
13. Filatovas, E., Podkopaev, D., Kurasova, O.: A visualization technique for accessing solution pool in interactive methods of multiobjective optimization. International Journal of Computers Communications and Control **10**, 508–519 (2015)
14. Garzón, E.M., Moreno, J.J., Martínez, J.A.: An approach to optimise the energy efficiency of iterative computation on integrated GPU–CPU systems. The Journal of Supercomputing **73**(1), 114–125 (2017)
15. Goldberger, J., Gordon, S., Greenspan, H.: An efficient image similarity measure based on approximations of kl-divergence between two gaussian mixtures. In: ICCV, pp. 487–493. IEEE Computer Society (2003)
16. Hout, M.C., Goldinger, S.D., Brady, K.J.: MM-MDS: A Multidimensional scaling database with similarity ratings for 240 object categories from the massive memory picture database. PLOS ONE **9**(11), 1–11 (2014)
17. Ingram, S., Munzner, T., Olano, M.: Glimmer: Multilevel MDS on the GPU. IEEE Trans. Vis. Comput. Graph. **15**(2), 249–261 (2009)
18. Kurasova, O., Petkus, T., Filatovas, E.: Visualization of pareto front points when solving multi-objective optimization problems. Information Technology And Control **42**(4), 353–361 (2013)
19. Leng, J., et al.: GPUWattch: Enabling Energy Optimizations in GPGPUs. SIGARCH Comput. Archit. News **41**(3), 487–498 (2013)
20. Martínez, J.A., Almeida, F., Garzón, E.M., Acosta, A., Blanco, V.: Adaptive load balancing of iterative computation on heterogeneous nondedicated systems. The Journal of Supercomputing **58**(3), 385–393 (2011). DOI 10.1007/s11227-011-0595-3
21. Martínez, J.A., Garzón, E.M., Plaza, A., García, I.: Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE. The Journal of Supercomputing **58**(2), 151–159 (2011)

22. Medvedev, V., Kurasova, O., Bernatavičienė, J., Treigys, P., Marcinkevičius, V., Dzemyda, G.: A new web-based solution for modelling data mining processes. Simulation Modelling Practice and Theory **76**, 34–46 (2017)
23. Morrison, A., Ross, G., Chalmers, M.: Fast multidimensional scaling through sampling, springs and interpolation. Information Visualization **2**(1), 68–77 (2003)
24. Orts, F., Filatovas, E., Ortega, G., Kurasova, O., Garzón, E.M.: HPC Tool for Multidimensional Scaling. In: Proceedings of the 17th International Conference on Computational and Mathematical Methods in Science and Engineering, vol. 5, pp. 1611–1614. J. Vigo-Aguiar (2017)
25. Osipyan, H., Morton, A., Marchand-Maillet, S.: Fast interactive information retrieval with sampling-based MDS on GPU architectures. In: Information Retrieval Facility Conference, pp. 96–107. Springer (2014)
26. Papenhausen, E., Wang, B., Ha, S., Zelenyuk, A., Imre, D., Mueller, K.: GPU-accelerated incremental correlation clustering of large data with visual feedback. In: Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA, pp. 63–70 (2013)
27. Park, S., Shin, S.Y., Hwang, K.B.: CFMDS: CUDA-based fast multidimensional scaling for genome-scale data. BMC Bioinformatics **13**(17), S23 (2012)
28. Pawliczek, P., Dzwinel, W., Yuen, D.A.: Visual exploration of data by using multidimensional scaling on multicore CPU, GPU, and MPI cluster. Concurr Comput **26**(3), 662–682 (2014)
29. Qiu, J., Bae, S.H.: Performance of windows multicore systems on threading and mpi. Concurrency and Computation: Practice and Experience **24**(1), 14–28 (2012)
30. Shmoys, D.B., Tardos, E.: An approximation algorithm for the generalized assignment problem. Math. Program. **62**(3), 461–474 (1993)
31. Wong, A., Rexachs, D., Luque, E.: Parallel application signature for performance analysis and prediction. IEEE Transactions on Parallel and Distributed Systems **26**(7), 2009–2019 (2015)
32. Yang, T., Liu, J., McMillan, L., Wang, W.: A fast approximation to multidimensional scaling. In: IEEE workshop on Computation Intensive Methods for Computer Vision (2006)
33. Zhong, Z., Rychkov, V., Lastovetsky, A.: Data partitioning on multicore and Multi-GPU platforms using functional performance models. IEEE Transactions on Computers **64**(9), 2506–2518 (2014)