Visualization of large multidimensional data sets by using multi-core CPU, GPU and MPI cluster

Piotr Pawliczek[1], Witold Dzwinel[2], David A. Yuen[3]

[1]*University of Texas, Department of Biochemistry and Molecular Biology, Houston, TX 77030, USA*
[2]*AGH University of Science and Technology, Institute of Computer Science, Krakow, Poland*
[3]*University of Minnesota, Minnesota Supercomputing Institute, Minneapolis, MN55455-0219, USA*

Abstract. Multidimensional scaling (MDS) is a very popular and reliable method used in feature extraction and visualization of multidimensional data. The role of MDS is to reconstruct the topology of an original *N*-dimensional feature space consisting of *M* feature vectors in target 2-D (3-D) Euclidean space. It can be achieved by minimization of the error - "stress" function - $F(\|\mathbf{D}-\mathbf{d}\|)$, where **D** and **d** are the *MxM* dissimilarity matrices in the original and in the target spaces, respectively. However, the stress function is in general a multimodal and multidimensional function for which the complexity of finding global minimum increases exponentially with the number of data. We employ here a robust heuristics based on discrete particle method enabling interactive visualization of data for various types of stress functions. Nevertheless, due to at least $O(M^2)$ memory and time complexity, the method becomes computationally demanding when applied for interactive visualization of data sets involving $M\sim10^4$. We present here efficient parallel algorithms developed for various small and pre-medium computer architectures from single and multi-core processors to GPU and multiprocessor MPI clusters. The timings obtained show that the computational efficiency of CUDA implementation of MDS on a PC equipped with a strong GPU board (Tesla M2050 or GeForce 480) is two times greater than its MPI equivalent run on 10 nodes (10x 2xIntel Xeon X5670 = 120 threads) of a professional multiprocessor cluster (HP SL390). We show also that the hybridized two-level MPI/CUDA implementation run on a small cluster of GPU nodes can additionally provide a linear speed-up.

Keywords: data mining, interactive visualization, multidimensional scaling, method of particles, multi-core CPU, GPU-CUDA, MPI cluster

1 Introduction

Multidimensional scaling (MDS) is one of the most popular techniques used for dimensionality reduction and feature extraction (e.g. [1,2]). MDS is also a natural candidate as a computational engine for interactive visualization of multidimensional data and its visual clustering [3,4,5,6].

The general problem which stays behind MDS is to find a bijection F: $\mathbf{\Omega}\rightarrow\mathbf{\Sigma}$ of a "source" space of abstract objects $\mathbf{\Omega}=\{\omega_i;\ i=1,\ldots,M\}$ onto a "target" vector space $\mathbf{\Sigma}=\{\boldsymbol{\xi}_i=(\xi_1, \xi_2, \xi_3, \ldots,\xi_n);\ i=1,\ldots,M\}$ which, within a given accuracy, preserves the topological structure of $\mathbf{\Omega}$ in *n*-dimensional $\mathbf{\Sigma}$ space. The space $\mathbf{\Omega}$ can be an arbitrary space represented by a dissimilarity matrix $\mathbf{D}=\{D_{ij}\}_{MxM}$ where D_{ij} is a dissimilarity measure between objects ω_i and ω_j. It means that the object representation is not important in that case. The dissimilarity matrix **D** is the only information about $\mathbf{\Omega}$.

Correspondingly, $\mathbf{d}=\{d_{ij}\}_{MxM}$ is a distance matrix in the target *n*-dimensional vector space $\mathbf{\Sigma}$, where d_{ij} is a distance (e.g., the Euclidean distance) between vectors $\boldsymbol{\xi}_i$ and $\boldsymbol{\xi}_j$ which are the mappings of corresponding objects ω_i and ω_j in $\mathbf{\Sigma}$. We assume that to preserve the topological structure of $\mathbf{\Omega}$ in $\mathbf{\Sigma}$ an overall error $F(\|\mathbf{D}-\mathbf{d}(\Xi)\|)$ should be minimized, where $F(.)$ is a multidimensional function $\Re^{nxM}\rightarrow\Re^1$ and $\|\ldots\|$ is a discrepancy measure between dissimilarities

D_{ij} from $\mathbf{\Omega}$ and corresponding distances d_{ij} from $\mathbf{\Sigma}$. Then, the matrix $\Xi=(\xi_1, \xi_2, \xi_3, \ldots,\xi_M)$ which minimizes the error function $F(.)$, is the result of multidimensional scaling.

For example, let us assume that the "source" space is the space of shapes. Matrix **D** consists of the elastic distances between shapes [7]. We can map the space into target space of ellipses each defined by two radiuses a_i and b_i (i.e., $\xi_i=(a_i, b_j)$). Distance matrix **d** consists of smallest Hausdorff distances between ellipses. Minimizing the error function $F(\|\mathbf{D}-\mathbf{d}(\Xi)\|)$, the shapes from original space $\mathbf{\Omega}$ will be approximated by ellipses in the target space $\mathbf{\Sigma}$. Simultaneously, the shapes are represented by points $\xi_i=(a_i, b_j)$ in $\mathbf{\Sigma}$ 2-D space. Following this way, one can find vector representation of shapes with an arbitrary accuracy by increasing the dimensionality of $\mathbf{\Sigma}$. Having vector representations of objects one can use classical machine learning algorithms for data mining without using complicated syntactic rules.

The multidimensional scaling methodology has been evolving for many years [8-13]. It is originally defined as a dimensionality reduction procedure which transform $\mathbf{\Omega}=\Re^N$ of feature vectors $\mathbf{Y}=\{\mathbf{y}_i=(y_{i1},..y_{iN})\}_{i=1,\ldots,M}$ into $\mathbf{\Sigma}=\Re^n$ - the Euclidean space of corresponding vectors $\mathbf{X}=\{\mathbf{x}_i=(x_{i1},..x_{in})\}_{i=1,\ldots,M}$. It is assumed that $N>>n$. In this paper, particularly for multidimensional data visualization, we assume that $\dim\mathbf{\Sigma}=n=3$ (or 2). The error function $F(.)$ is called the "stress function" and the problem of finding target configuration $\boldsymbol{X}$ can be formulated as follows:

$$Error = \min_{\mathbf{X}} F(\mathbf{X}) = \min_{\mathbf{X}} \sum_{i<j} w_{ij}\left(D_{ij}^k - d_{ij}^k\right)^m , \tag{1}$$

$$D_{ij} = \left(\mathbf{y}_i - \mathbf{y}_j\right)\cdot\left(\mathbf{y}_i - \mathbf{y}_j\right) \wedge d_{ij} = \left(\mathbf{x}_i - \mathbf{x}_j\right)\cdot\left(\mathbf{x}_i - \mathbf{x}_j\right). \tag{2}$$

In general, the distance matrix **D** can be non-Euclidean. Assuming that $k=1/2$ and $m=2$, $w_{ij}=1/D_{ij}^k$ we obtain the MDS version called Sammon's mapping [13]. The distances array **D** defines unambiguously data topology in the original $\mathbf{\Omega}$ space because all of $L_{Max}=M(M-1)/2$ distances between feature vectors are known. The mapping F() tries to reproduce the topology of $\mathbf{\Omega}$ in $\mathbf{\Sigma}$ by matching corresponding Euclidean distances between feature vectors using mean squared error criterion (1). To find the minimum of criterion (1) the system of $n\times M$ nonlinear equations should be solved. The number of solutions is infinite because the target configuration of feature vectors **X** is invariant with respect to all isometric transformations including rotation, axial and planar symmetries. Moreover, the "stress" function is usually multimodal. There are many methods used for minimization of criterion (1). From simple gradient based techniques [12,13] and Niemann steepest descent algorithm [14] which enable to find a local minimum, to elaborated heuristics (e.g. [4,6,15-19]) searching for a global minimum of (1).

The high computational load is the main obstacle for employing MDS as an interactive tool for visualization of large datasets where $M>10^4$. Both the memory and computational complexity depends linearly on the number of distances $L_{Max}=M(M-1)/2$ between data objects and exponentially on the number of local minima of the "stress function". In general, the second problem is unsolvable and it can be only partially overcome by using very efficient heuristics based on the N-body virtual particle method solver [5,6,20,21]. However, both its time and memory complexity are still bounded from below by the quadratic term $O(M^2)$.

In the following sections we present briefly the virtual particle method employed for MDS mapping. Then we demonstrate the developed parallel algorithms and their implementations in two different parallel environments: multithread multi-core CPU and GPU. We compare the efficiency of mapping on a broad choice of CPU and GPU processors. We present also the results of integration of three heterogeneous parallel environments: OpenMP, CUDA and MPI on a single pre-medium MPI cluster. At the end we summarize and discuss our findings demonstrating that by using on_the_shelf small and pre-medium computer systems the method of

multidimensional scaling based on virtual particle dynamics can be successfully employed for interactive visualization of data sets consisting of $M \sim 10^4$ feature vectors.

2 Virtual particles method as a heuristics

The multidimensional scaling of N-dimensional feature space into n-dimensional target space where $n=3$ (or 2) is used for visualization of multidimensional spaces and visual clustering. In this case to obtain the global minimum of criterion (1) (or a solution located close to the global minimum) we employed the virtual particle method [5,6]. This heuristics consists in the minimization of the potential energy

$$F(\mathbf{X}) = \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} V_{ij}\left(D_{ij}^k - d_{ij}^k\right), \tag{3}$$

representing the sum of potentials $V_{ij}(.)$ of interacting particles i and j (a simple example was shown in Fig. 1). Each particle corresponds to the respective feature vector from the source space $\mathbf{\Omega}$. The interaction potential $V_{ij}(.)$ between particles i and j is a function of difference between corresponding vectors' distance D_{ij} in the source space $\mathbf{\Omega}$ and current particles' distance d_{ij} in $\mathbf{\Sigma}$ (Eqs.2).

Initially, the particles occupy random positions in $\mathbf{\Sigma}$ and their velocities are set to zero. The particles interact with each other via semi-harmonic forces $\mathbf{f}_{ij} = -\Sigma_j \nabla V_{ij}(D_{ij}-d_{ij})$ and they change their positions and velocities according to the Newtonian laws of motion. The kinetic energy is dissipated by the friction force which is proportional to the particle velocity. So, the equations of motion are as follows:

$$\vec{\mathbf{F}}_i = \mathrm{m}_i \frac{d\mathbf{v}_i}{dt} = \gamma \sum_{j=1}^{M} \mathbf{f}_{ij} - \lambda \mathbf{v}_i, \quad \frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i, \tag{4}$$

where m_i, γ and λ are the parameters while $\mathbf{v}_i$ is a velocity of a particle i.

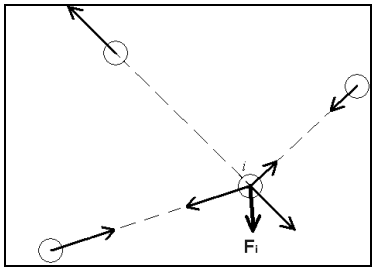

Fig. 1. Forces acting on a particle *i*.

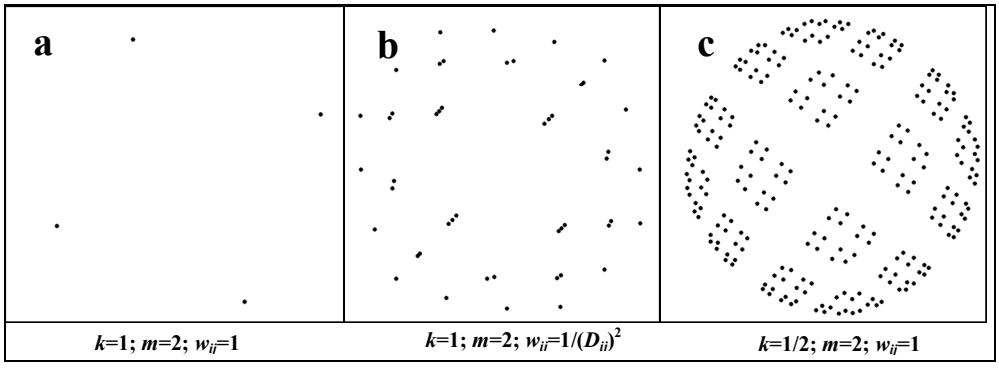

Fig.2 Results of mapping of the nodes of 8-dimensional hypercube into 2-dimensional Euclidean space ($\mathbf{R}^8 \rightarrow \mathbf{R}^2$) by using various criterion functions (see Eq.1).

After a certain number of iterations the equilibrium is reached. Then the sum of potentials represents the total energy of the particle system in the equilibrium (Eqs.3). It is equal to a general form of the "stress function" given by (1). Thus, the positions of particles $\mathbf{x}_i$ represent the solution of the minimization problem (1).

The forces $\mathbf{F}_i$ which act on each particle i are computed on the basis of the current positions of particles $\mathbf{x}_i$. Then, the new positions of particles are calculated by using, e.g., the following *leap-frog* scheme ([22]):

$$\begin{aligned} \mathbf{v}_i^{(n+\frac{1}{2})} &= \frac{1-\beta\cdot\Delta t}{1+\beta\cdot\Delta t}\mathbf{v}_i^{(n-\frac{1}{2})} + \frac{\alpha\cdot\Delta t}{1+\beta\cdot\Delta t}\mathbf{F}_i^{(n)} \\ \mathbf{x}_i^{(n+1)} &= \mathbf{x}_i^{(n)} + \mathbf{v}_i^{(n+\frac{1}{2})}\cdot\Delta t \\ \alpha &= \frac{\gamma}{\mathrm{m}}, \quad \beta = \frac{\lambda}{2\mathrm{m}}, \quad \beta\cdot\Delta t < 1 \end{aligned} \tag{5}$$

where n denotes the consecutive time-step number. The sequential (one thread) version representing a single time step of the method we display in Listing 1.

```
M ← number of vectors;
d ← 0;
Forces[0...(M − 1)] ← 0;
for i ← 0 to (M − 1) do
    for j ← 0 to (i − 1) do
        x_i ← Positions[i];
        x_j ← Positions[j];
        f_ij ← ComputeForce(x_i, x_j, Distances[d]);
        d ← d + 1;
        Forces[i] ← Forces[i] + f_ij;
        Forces[j] ← Forces[j] − f_ij;
for i ← 0 to (M − 1) do
    F_i ← Forces[i];
    v_i ← ComputeVelocity(v_i, F_i);
    x_i ← Positions[i];
    Positions[i] ← ComputePosition(x_i, v_i);
```

Listing 1 Pseudocode of a sequential version of the virtual particles MDS algorithm.

The parameter values α, β and Δt were tuned on the basis of empirical experience. We assumed that in every simulation the number of time-steps is constant and equal to 10^4, albeit it can be also tuned automatically to the chosen mapping error $F(.)$.

As shown in [5,6], the main advantages of the virtual particle MDS over other methods are as follows:

1. The method can be used for an arbitrary error criterion represented by the general formula (3). This allows for better exploration of multidimensional topology of data. For example, in Fig.2 we demonstrate the results of embedding of 8-D hypercube into 2-D target space employing 3 various criteria (see Eq.1). By using Euclidean distance ($k=1/2$) instead of its square (k=1, Fig 2c), we can obtain more detailed view of the feature space topology. Whereas, by using normalized distances or squared distances only the coarse grained structure can be seen (Fig.2a,b).
2. The method efficiently explores multimodal and multidimensional domain of the criterion function, so the probability of finding the global minimum is high. Similarly, as it is in simulated annealing heuristic [23], it can be increased by decreasing dissipation rate and increasing simulation time.

3. Introducing interactively the "energy kicks" we are able to better explore the domain of the stress function (1).
4. The method is entirely interactive, allowing for visual clustering [5,6] and data classification [24]. It means that the particles representing feature vectors can be added removed, grouped and stopped interactively during simulation. The same concerns simulation parameters, such as the time step, friction factor and cost function criteria. The classification and clustering of newly added feature vectors can be estimated visually.

However, despite many advantages of the virtual particle method its computational complexity is not lower than for the other multidimensional scaling algorithms which minimize the stress function (1) and is majored by $O(M^2)$ term. It is too large to enable interactive visualization of data sets which consists of more than 10^4 feature vectors using serial mode of computations. Therefore, the parallelization of MDS algorithm is the first issue to increase its computational efficiency.

As shown in Listing1, two nested loops FOR1 and FOR2 in forces computation module decide about the squared complexity of a single time step. The remaining part, representing particle motion, has only linear complexity (loop FOR3) and its influence on computational time is negligible for large number of feature vectors. Therefore, in the following sections we are dealing with parallelization of the nested loop term representing calculation of forces acting on a single particle.

3 Parallel MDS algorithms

In this section we present the parallel algorithms for multidimensional scaling based on particle dynamics. We discuss and compare three parallel implementations of MDS depending on the multiprocessor type and parallel environment used, namely:

1. Multi-core CPU processor with OpenMP directives;
2. GPU board with CUDA Nvidia computational environment;
3. High Performance Cluster with MPI interface.

3.1 Parallel MDS algorithm for multi-core CPU

There are many well known methods for parallelization of "round robin" molecular dynamics (MD) (i.e. every particle interacts with all the others) developed on shared memory multi-core processors and vector processors (e.g. [25-27]). However, unlike in MD codes, the particle-particle interactions depend also on distances array **D**, which has to be distributed onto computational nodes.

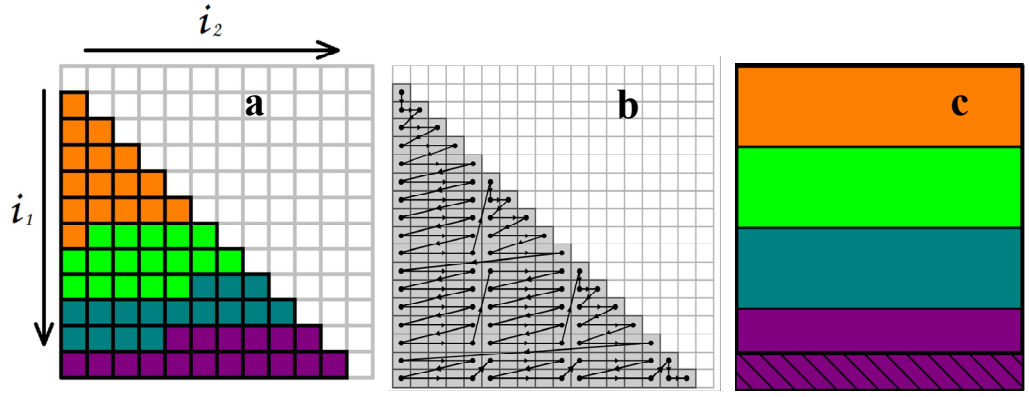

Fig.3 a) Distribution of computations onto 4 threads. b) Division of computations onto blocks. c) Splitting the distances array onto four blocks in GPU.

```
M ← number of vectors;
Forces[0...(M − 1)] ← 0;
create T threads
    t ← index of thread ∈ [0, T − 1];
    o1 ← (M·(M−1))/2 · t/T;
    o2 ← (M·(M−1))/2 · (t+1)/T;
    (i, j) ← ( ⌊(1+√(1+8·o1))/2⌋ , o1 − (i·(i−1))/2 );
    for d ← o1 to o2 − 1 do
        x_i ← Positions[i];
        x_j ← Positions[j];
        f_ij ← ComputeForce(x_i, x_j, Distances_Blocks[d]);
        Forces_t[i] ← Forces_t[i] + f_ij;
        Forces_t[j] ← Forces_t[j] − f_ij;
        if j = i − 1 then
            i ← i + 1;
            j ← 0;
        else
            j ← j + 1;
join threads
for i ← 0 to (M − 1) do
    F_i ← 0;
    for t ← 0 to (T − 1) do
        F_i ← F_i + Forces_t[i];
    v_i ← ComputeVelocity(v_i, F_i);
    x_i ← Positions[i];
    Positions[i] ← ComputePosition(x_i, v_i);
```

Listing 2 Pseudocode representing CPU version of the simulation kernel of MDS algorithm

This seems to be unimportant modification is in fact crucial, because of the large size of distances matrix (owing to O(M^2) memory complexity) what causes dramatic problems connected with efficient use of cache memory. To optimize the use of cache memory both for serial and parallel codes the $\mathbf{F}_i$ array which stores forces (*forces* (.) in Listing 1,2) is merged with the array of particle positions (*positions* (.) in Listing 1,2). In the buffer memory where data are stored the forces are arranged alternately with particle positions. This trick makes data needed for calculations be placed very close in operation memory, decreasing the memory access time and enabling more efficient use of cache (lesser number of cache misses). In Listings 1 and 2 we use two arrays: *forces* (.) and *positions* (.) to make them more explanatory.

To distribute the computations among the threads, the nested loops FOR1-FOR2 from Listing 1 should be modified. To this end all particle pairs were divided onto the sets of equal size. Each set is processed by a single thread. The thread computes partial forces acting on particles from the respective set. In Fig.3a we show the diagram demonstrating how the computations are distributed among the threads. Each array component from Fig.3a stands for the force $\mathbf{f}_{ij}$, distance D_{ij} in source and d_{ij} in target spaces. Because the array is symmetric, only its lower part is processed.

To minimize the access time to the operational memory, the array from Fig.3a should be divided onto square blocks of size *b* and, as shown in Fig.3b, be looked through on block by block basis. The blocks are redistributed among threads in a similar way as particle pairs in Fig.3a. Moreover, to make this algorithm efficient, the size of block *b* should be matched to the type of

processor, the number of threads employed in computations and the size of data processed. If b is too small, more memory hits are expected. Otherwise, too large b causes cache overflow and obstructs the load balancing. The value of b is selected experimentally. Pseudocode of the simulation kernel of MDS method – i.e. the fragment of code representing single time step, is shown in Listing 2.

The optimal block size b is chosen performing tens of iterations (time steps) for sizes of blocks being the powers of two from 64 to 2048. The tests presented in the following sections use the optimal block size. The algorithm was implemented by using OpenMP directives.

3.2. Parallel MDS algorithm for GPU

Using GPU instead of CPU is one of the ways for accelerating the computations. In this section we present the MDS algorithm developed for CUDA Nvidia environment.

The GPU is SIMD type of processor architecture. The basic components of GPU responsible for computations are multiprocessors (MP). Each multiprocessor executes simultaneously 16 up to 32 similar operations on various variables depending on the type of operation and processor. From the point of view of a programmer the number of registers and the size of MP cache – the memory which speed is comparable to the speed of registers - are the most significant parameters of GPU multiprocessor. The most important properties of full GPU board are the number of threads and the size of global memory i.e., relatively slow memory which can be used as a buffer for data transfer from operational memory of the whole computer system to the MPs caches. The compute capabilities of GPU boards – usually represented by two digits separated by a dot - are presented in Table 1.

Table 1 The compute capability of modern GPU boards

<table>
<tr><th></th><th colspan="7">Compute Capability</th></tr>
<tr><th></th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.0</th><th>2.1</th><th>3.0</th></tr>
<tr><td>Maximum number of threads (warps) per block</td><td colspan="4">512 (16)</td><td colspan="3">1024 (32)</td></tr>
<tr><td>Maximum number of resident blocks per multiprocessor</td><td colspan="6">8</td><td>16</td></tr>
<tr><td>Maximum number of resident threads (warps) per multiprocessor</td><td colspan="2">768 (24)</td><td colspan="2">1024 (32)</td><td colspan="2">1536 (48)</td><td>2048 (64)</td></tr>
<tr><td>Number of 32-bit registers per multiprocessor</td><td colspan="2">8192</td><td colspan="2">16384</td><td colspan="2">32768</td><td>65536</td></tr>
<tr><td>Maximum amount of shared memory per multiprocessor</td><td colspan="4">16 KB</td><td colspan="3">48 KB</td></tr>
</table>

The codes which run on GPU boards should minimize the number of operations involved global memory due to its relatively long access time. Therefore, the efficient algorithms employing GPU use the following steps:

1. Copying the fragment of global memory to the shared memory.
2. Executing the calculations on data located in GPU registers and shared memory.
3. Write-back the results to the global memory.

The read-write operations to the global memory do not block the MPs. In the moment when a warp is blocked by read-write operations, the multiprocessor can execute other warps. It allows for overlapping by calculations the delay caused by usage of global memory. The maximum number of warps assigned to a single processor depends on both compute capability of GPU board (see Table 1) and organization of the running code.

The program consists of two parts: PC module executed on CPU and GPU module representing simulation kernel. The PC module reads data, such as particle positions and velocities to its operational memory and is responsible for calculating dissimilarity array **D**. It also takes part in transferring these data to the global memory of GPU board. The whole simulation is executed on GPU board.

Our algorithm bases on the following assumptions:

1. The whole dissimilarity array **D** is transferred from operational memory to the GPU global memory and resides there to the end of computations. This is because the transfer from operational to the global GPU memory is very slow.
2. Unlike in the CPU case the whole **D** is used for computations (M^2 components instead of $M(M-1)/2$). It allows for considerable simplification of the algorithm by removing conditional instructions which are very inefficient on GPU.
3. Buffering arrays which keep positions and velocities of the particles are divided onto blocks containing 32 particles each. If a block is not full, additional artificial particles are generated. Consequently, the number of rows in **D** is increased while the number of columns is kept unchanged. This allows the algorithm to be described on the level of warps operation. Each warp executes in parallel the instructions on the all 32 particles.
4. If two particles are too close to each other, their interactions are ignored. It allows for elimination of extreme cases and processing diagonal components of **D** array without the need to process additional *if* instructions.

The code developed is represented by a single kernel. It is executed once per each time step.

At the beginning of computations the following data are transferred from operational (CPU) to the global (GPU) memory.

1. Dissimilarity array `gdistances` (**D**).
2. Buffer `gpositionsA` ($\mathbf{x}^n$) which contains current particle positions.
3. Buffer `gpositionsB` ($\mathbf{x}^{n+1}$) which contains computed (new) particle positions.
4. Buffer `gVelocities` ($\mathbf{v}^n$) which contains current particle velocities, replaced subsequently by the newly computed.

The compute capability of the simulation kernel located on GPU board is determined by two parameters: w_B – the number of warps in a single block (warp consists of 32 threads) and w_C – the number of particles transferred to a buffer in the shared memory during the computations (must be a multiple of 32). From these two parameters the number of threads T on a single block and the number of blocks B are computed. Pseudocode representing the MDS algorithm version intended for calculations on GPU board is presented in Listing 3. It demonstrates a series of instructions executed in the scope of one block. The variables which begin with the letter „s" and „g" are stored in the shared and global memories, respectively.

Each thread calculates the force $\boldsymbol{F}_i$ acting on i. To this end the positions of all the particles and respective fragment of dissimilarity array should be copied to the shared memory. Data are copied and processed in a controlled way to use the shared memory efficiently. Then the velocities $\mathbf{v}_i$ (stored in `gVelocities` buffer) of all particles are computed and their positions (stored in $\texttt{gPositions}_\texttt{B}$ buffer) updated. At the end of the iteration (a time step) the arrays

$\texttt{gPositions}_\texttt{B}$ and $\texttt{gPositions}_\texttt{A}$ are swapped to avoid the memory conflicts caused by the threads from various blocks accessing the arrays holding particle positions.

The distribution of computations over blocks is shown in Fig.3c. The computations are split onto four blocks each of height equal to $32w_B$ rows. Because the number of particles (feature vectors) is not divisible by $32w_B$, additional rows - virtual particles - were added (the hatched region in Fig.3c). So, the last block executes some operations on artificial particles. However, they do not influence other particles because the number of columns remains unchanged. The size of global memory - about 3GB on standard GPU boards - imposes an upper limit on the size of data. Because the dissimilarity array $M \times M$ which should be hold in global memory is responsible mainly for the size of data, only around of 10^4 feature vectors (particles) can be processed on a single GPU board.

$W \leftarrow$ ilość wątków w pojedynczej ścieżce ($warp\ size$) $= 32$;
$T \leftarrow$ liczba wątków w bloku $= w_B \cdot W$;
$B \leftarrow$ liczba bloków $= \lceil \frac{n}{T} \rceil$;
compute in B blocks with T threads
  $b \leftarrow$ numer bloku $\in [0, B-1]$;
  $t \leftarrow$ numer wątku w bloku $\in [0, T-1]$;
  $i_1 \leftarrow b \cdot T + t$;
  $\overline{f} \leftarrow \overline{0}$;
  $sPos_1[t] \leftarrow gPositions_A[i_1]$;
  for $k \leftarrow 0$ **to** $\lfloor \frac{n}{w_c \cdot W} \rfloor$ **do**
    for $j \leftarrow 0$ **to** $j < \lfloor \frac{w_c}{w_B} \rfloor$ **do**
      $i_2 \leftarrow k \cdot w_c \cdot W + j \cdot T + t$;
      $sPos_2[j \cdot T + t] \leftarrow gPositions_A[i_2]$;
    if $t < (w_c \bmod w_B) \cdot W$ **then**
      $i_2 \leftarrow k \cdot w_c \cdot W + \lfloor \frac{w_c}{w_B} \rfloor \cdot T + t$;
      $sPos_2[\lfloor \frac{w_c}{w_B} \rfloor \cdot T + t] \leftarrow gPositions_A[i_2]$;
    synchronizeThreadsWithinBlock;
    for $l \leftarrow 0$ **to** $w_c \cdot W$ **do**
      $i_2 \leftarrow k \cdot w_c \cdot W + l$;
      $sD[t] \leftarrow gD[i_1, i_2]$;
      $\overline{f} \leftarrow \overline{f}$+ComputeForce$(sPos_1[t], sPos_2[l], sD[t])$;
    synchronizeThreadsWithinBlock;
  $r \leftarrow n \bmod (w_c \cdot W)$;
  if $r > 0$ **then**
    for $j \leftarrow 0$ **to** $j < \lfloor \frac{r}{T} \rfloor$ **do**
      $i_2 \leftarrow n - r + j \cdot T + t$;
      $sPos_2[j \cdot T + t] \leftarrow gPositions_A[i_2]$;
    if $t < (r \bmod T) \cdot W$ **then**
      $i_2 \leftarrow n - r + \lfloor \frac{r}{T} \rfloor \cdot T + t$;
      $sPos_2[\lfloor \frac{r}{T} \rfloor \cdot T + t] \leftarrow gPositions_A[i_2]$;
    synchronizeThreadsWithinBlock;
    for $l \leftarrow 0$ **to** r **do**
      $i_2 \leftarrow n - r + l$;
      $sD[t] \leftarrow gD[i_1, i_2]$;
      $\overline{f} \leftarrow \overline{f}$+ComputeForce$(sPos_1[t], sPos_2[l], sD[t])$;
  $sV[t] \leftarrow gVelocities[i_1]$;
  $sV[t] \leftarrow$ ComputeVelocity$(sV[t], \overline{f})$;
  $gPositions_B[i_1] \leftarrow$ ComputePosition$(sV[t], sPos_1[t])$;
  $gVelocities[i_1] \leftarrow sV[t]$;
join threads
$gPositions_A \leftarrow gPositions_B$;

Listing 3 Pseudocode representing GPU version of the kernel of MDS algorithm

The total memory complexity of the code is:

$$Mem = 32 \cdot w_B \cdot \left\lfloor \frac{M}{32 \cdot w_B} \right\rfloor \cdot (M + 9) \cdot sizeof(\text{float/double}) \tag{6}$$

The number of registers r_T used by a single thread is another important parameter which influence computational efficiency. This parameter is matched during compilation. Both w_B and r_T determine the number of registers used by a single block. The optimal values of r_T, $w_{B,C}$ for tests presented in this paper are matched on the basis of approximate assessments given in [28] and trial runs executing several time steps of simulation.

3.3. Parallel MDS algorithm for MPI cluster

The MDS algorithm implemented on cluster uses two-level parallelism. The first one is connected with internal architecture of cluster nodes. The cluster node can be just a shared memory multiprocessor (one or more multi-core CPUs) or it can be additionally empowered with GPU boards. In the former, calculations within MPI (Message Passing Interface) process are parallelized with the use of OpenMP or POSIX threads (both versions are implemented) while in the latter, CUDA environment is used.

The higher – coarse-grained - parallelization level corresponds to the topology of cluster nodes. The dissimilarity matrix and calculations are distributed among nodes with the use of adapted UTFBD algorithm ([29]) which is an optimized version of Taylor algorithm [30]. These algorithms were originally developed for parallel molecular dynamics simulations and were adopted by us to the requirements of our MDS method. Each MPI process corresponds to one system process executed on a dedicated cluster node. Following, we present the details of this parallel algorithm.

We define $\mathbf{U}_{n,n}=[u_{rc}]_{n1,n1}$ matrix of MPI processes. Only its lower triangle and diagonal are used. As it was shown in Fig.4a, the processes are ordered row-wise. For every element u_{rc} of matrix $\mathbf{U}_{n,n}$ for which $c \leq r$, exactly one process is assigned with MPI rank equal to:

$$u_{rc} = \frac{(r-1) \cdot r}{2} + c - 1 . \tag{7}$$

This data structure defines the constrained number of MPI processes that may be used during computations to $n1 \cdot (n1+1)/2$ where $n1$ is the matrix size.

The particles positions and velocities of particles represented by the sequence of vectors are divided onto $n1$ subsequences indexed by integers starting from one. To describe relationship between vectors' indices and numbers of subsequences, two function named *size(g)* and *first(g)* are defined:

$$\begin{aligned} size(g) &= \begin{cases} \lceil N/n1 \rceil & for\ g \leq N \bmod n1 \\ \lfloor N/n1 \rfloor & for\ g > N \bmod n1 \end{cases} \qquad g = 1,2,...,n1 \\ first(g) &= \begin{cases} 1 & for\ g = 1 \\ \sum_{i=1}^{g-1} size(i) & for\ g > 1 \end{cases} \end{aligned} \tag{8}$$

Then, the subsequence of vectors representing particles positions with index g $(0 < g < n1)$ is defined as follows:

$$S_x(g) = \left(\mathbf{x}_{first(g)}, \mathbf{x}_{first(g)+1}, \mathbf{x}_{first(g)+2}, ..., \mathbf{x}_{first(g)+size(g)-1} \right) \tag{9}$$

Analogically, for velocities of corresponding particles we obtain:

$$S_v(g) = \left(\mathbf{v}_{first(g)}, \mathbf{v}_{first(g)+1}, \mathbf{v}_{first(g)+2}, \ldots, \mathbf{v}_{first(g)+size(g)-1}\right). \tag{10}$$

The sequences S_x and S_v are assigned to respective MPI processes i.e., $S_x(r)$ and $S_v(r)$ sequences to processes u_{rc} lying on main diagonal ($r=c$), while two sequences of particle positions: $S_x(r)$ and $S_x(c)$, are assigned to the rest of processes ($r > c$).

As shown in Fig.4b, during simulation every process u_{rc} from diagonal ($r=c$) uses and updates all distances (both in "target" and "source" spaces) from the set:

$$\left\{ \left|\mathbf{x}_i \mathbf{x}_j\right| : \mathbf{x}_i, \mathbf{x}_j \in S_x(r) \right\}, \tag{11}$$

while the rest of processes, u_{rc} ($r>c$), keep track of all distances from the sets:

$$\left\{ \left|\mathbf{x}_i \mathbf{x}_j\right| : \mathbf{x}_i \in S_x(r), \mathbf{x}_j \in S_x(c) \right\}. \tag{12}$$

In each time step, current distances between particles from **d** array are compared with the values from the input dissimilarity matrix **D**. Consequently, total forces acting on each particle are computed and particles are moved according to the Newtonian dynamics.

Because particles' positions are scattered between processes, each of them may compute only partial forces acting on their particles. The sequences of partial forces are denoted as S_F. They are defined analogically as sequences S_x in formula (9):

$$S_F(g,h) = \left(f_{h,\,first(g)}, f_{h,\,first(g)+1}, f_{h,\,first(g)+2}, \ldots, f_{h,\,first(g)+size(g)-1}\right) \tag{13}$$

where $\boldsymbol{f}_{h,i}$ is a part of the total force acting on particle i, exerted by particles from $S_x(h)$ sequence.. Every sequence $S_F(g, h)$ may be computed by the process u_{gh} for $g<h$ or by the process u_{hg} when $g \geq h$. At this stage processes do not communicate with each other. To compute sequence of total forces $S_F(g)$ acting on particles from $S_x(g)$, corresponding elements from $S_F(g,h)$ sequences must be added. They are gathered by the processes u_{gg} from $\mathbf{U}_{n,n}$ matrix (see Figs.4c,d). This procedure can be expressed as follows:

$$S_F(g) = \left(\sum_{h=1}^{n} f_{h,\,first(g)}, \sum_{h=1}^{n} f_{h,\,first(g)+1}, \ldots, \sum_{h=1}^{n} f_{h,\,first(g)+size(g)-1} \right) \tag{14}$$

The computation stages of a single time step are presented in the Table 2.

To exchange data between processes only two MPI functions are needed: `MPI_Bcast` and `MPI_Reduce`. Unfortunately, both of them are blocking procedures. This generates a serious problem with rapid drop in computational efficiency. During every time step, the processes lying out of the diagonal of matrix **U** have to execute each function twice, i.e.:

- the first time - for synchronization of the first dataset along row;
- the second time - for synchronization of the second dataset along column.

However, these two operations may be executed in parallel, because they work on separate buffers. To bypass the lack of non-blocking versions of these functions in MPI standard, POSIX threads were employed. Every process lying out of diagonal uses two threads during data synchronization. One thread synchronizes data along a row while another along a column.

Table 2. The functions of processes u from matrix $\mathbf{U}_{n,n}$ during a single time step

processes from diagonal (r=c)	**processes lying out of diagonal (r>c)**
Update velocities vectors from $S_v(r)$ and positions vectors from $S_x(r)$ according to formulas (9,10).	do nothing
Broadcast particles positions vectors (sequence $S_x(r)$) along row and column (Fig.4a).	Receive two sequences with positions vectors: $S_x(r)$ from process u_{rr} and $S_x(c)$ from process u_{cc} (Fig.4a)
Compute $S_F(r, r)$ on the ground of difference between current particles distances and original dissimilarities (Fig.4b)	Compute $S_F(r, c)$ and $S_F(c, r)$ on the ground of difference between current particles distances and original dissimilarities (Fig.4b)
Compute $S_F(r)$ by gather and addition proper s_F sequences from processes lying in the same row and column (Fig.4c,d)	Send $S_F(r, c)$ to process u_{rr} and $S_F(c, r)$ to process u_{cc} (Fig.4c,d)

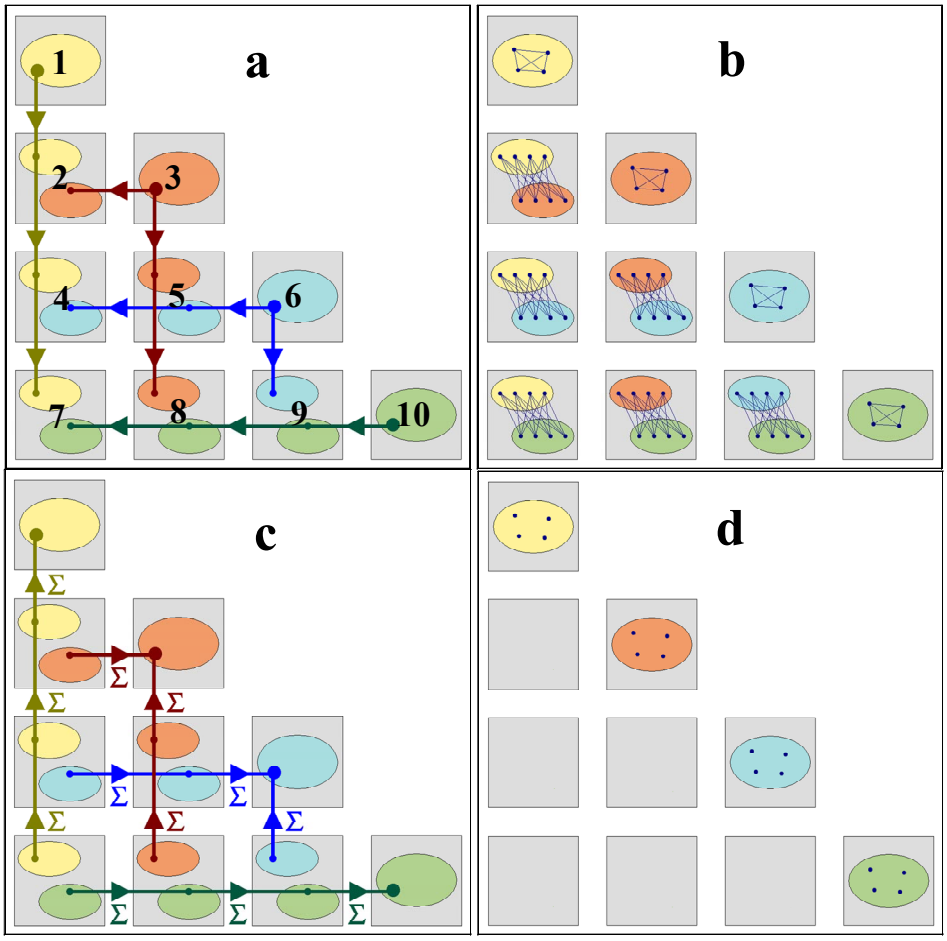

Fig.4 Diagrams demonstrating the distribution of computations onto cluster nodes.

4 Tests and results

In all the tests reported below, we compare the average execution times of one time step of particle based MDS parallel algorithms which peudocodes are shown in Listings 1-3. The averages were calculated for runs executing at list 200 time steps. The tests were carried out employing optimized C++ codes with MPI functions, OpenMP directives or CUDA Nvidia procedures in respect to computational environment used. All the operations performed on real numbers were executed using the same variable type, i.e., *float* and/or *double*, which can be set up at the beginning of program execution. The efficiency of the sequential code was tested on several computer architectures described in Table 3. All the tests were performed using 64-bits Linux system. The codes were compiled by using g++ compiler ver.4 with O3 option (optimized code). Only on the older "Baribal" cluster (see Table 3) the code was compiled using *icpc* Intel compiler.

4.1 Tested computer systems

In Table 3 we provide a brief description of the computer systems used in our tests. The list contains heterogeneous computer systems ranging from older cluster systems (SGI Altix 3700) to strong computational clusters (HP SL390) empowered with GPU boards through PCs and workstations with single multi-core CPU and GPU boards.

Table 3. The description of computer systems used for performance analysis a) CPU systems and b) GPU boards.

a)

Processor name	**Computer**	**Number of processors**	**Number of cores per processor**	**Clock speed [GHz]**	**Cache size [MB]**	**Memory size [GB]**
Intel Xeon X5670	one node of HP SL390 cluster, (in total)	**2 (48)**	**6 (288)**	**2.93**	**12**	**70 (1,700)**
Intel Core2 Duo E8400	PC	**1**	**2**	**3**	**6**	**8**
Intel Core i5 M 430	laptop Samsung R780	**1**	**2**	**2.27**	**3**	**4**
Dual Core AMD Opteron 270	workstation	**2**	**2**	**2**	**1**	**16**
Intel Itanium 2 Madison (IA-64)	SGI Altix 3700, supercomputer "Baribal", ACK Cyfronet AGH	**256**	**1**	**1.5**	**6 or 4**	**512**

b)

Name of device	Compute capability	Number of multi-processors	Number of CUDA cores per multiprocessor	Multi-processors clock rate [GHz]	Global memory size [GB]	Global memory clock rate [GHz]
GeForce 8500 GT	**1.1**	**2**	**8**	**0.92**	**0.25**	**0.40**

GeForce 8800 Ultra	**1.0**	**16**	**8**	**1.51**	**0.75**	**1.15**
GeForce 9500 GT	**1.1**	**4**	**8**	**1.40**	**0.50**	**0.40**
GeForce 9800 GT	**1.1**	**14**	**8**	**1.38**	**1.00**	**0.90**
GeForce GT 330M	**1.2**	**6**	**8**	**1.27**	**1.00**	**0.79**
GeForce GTX 260	**1.3**	**27**	**8**	**1.26**	**1.75**	**1.00**
GeForce GTX 460	**2.1**	**7**	**48**	**1.40**	**2.00**	**1.80**
GeForce GTX 480	**2.0**	**15**	**32**	**1.40**	**1.50**	**1.85**
Tesla M2050	**2.0**	**14**	**32**	**1.15**	**2.62**	**1.57**

3.2 Data test beds

In the tests we used a few data sets of very different character [28,31]. However, we have noticed that data topology does not noticeably influence the timings of a single iteration albeit it has considerable effect on the quality of final mapping and the number of iterations necessary to obtain optimal value of the "stress" function. We are focused here on the efficiency of a single iteration (a time step) and on parallel implementation issues rather than on the quality of mapping. The former depends on the hardware and software issues while the latter on a proper choice of heuristics and its parameters. Therefore, to make the results consistent, we used in the tests only one artificially generated dataset consisting of 40-dimensional vectors (i.e., N=dim(**Ω**)=40) representing 2 classes of the same size: class 1 and class 2. First 1-20 vector coordinates were generated randomly from [-3/2, 3/2] interval. For vectors belonging to the class 1 their 21-40 coordinates are random numbers from [-3/2, ½] interval while those from class 2 were generated in [1/2, 2/3] interval. We use several datasets of various sizes: H1, H2, H3 etc. consisting from 1024 up to 256k (~$2.6\cdot10^5$) vectors. The final results of H4 data visualization (**Ω→Σ** where dim(**Σ**)=n=3**)** using our MDS algorithm is demonstrated in Fig.5.

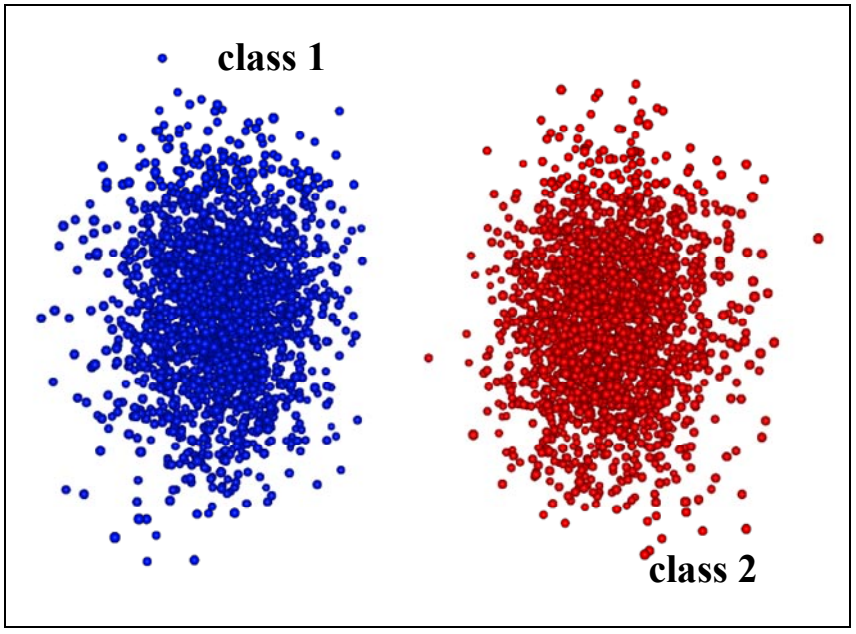

Fig.5 The H4 dataset visualized by MDS employing virtual particle method for minimization of stress function (1) for $k=1/2$, $m=2$, $w_{ij}=1$.

We assume additionally that two dissimilarity matrices, **D** and **d,** are Euclidean. The minimized stress function is represented byu the formula (1) with $k=1/2$, $m=2$, $w_{ij}=1$. The computational efficiency measured for average execution time of one time step is very similar for other choices of stress function type.

4.3 Results of tests – single thread case

In Figs.6a,b we display the average execution times of one time step of MDS algorithm for tested CPU boards and H1k-H30k testing datasets. The plots show that for the fastest three processors the computational times obtained for *float* arithmetic (Fig.6a) are reverse proportional to the CPU frequency (Table 3). In this case the Intel Xeon X5670 dedicated for number crunching is slightly slower then two years older one i.e., the popular Intel Core2Duo. Its advantage over other CPUs is evident for *double* arithmetic (Fig.6b) concerning both the computational speed and memory access time.

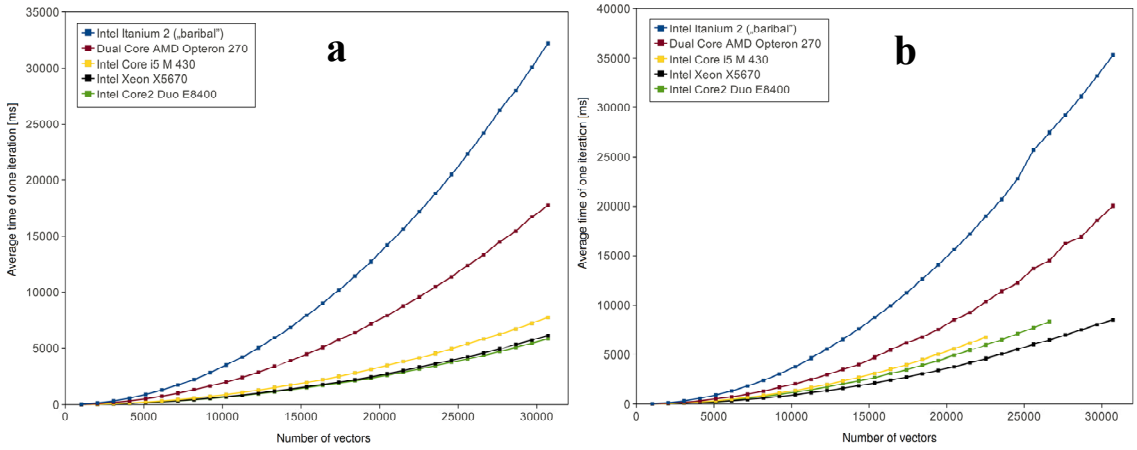

Fig.6 The averaged timings of the sequential version of MDS algorithm for a single time step for a) *float* and b) *double* arithmetic.

4.4 Results of tests – multi-core CPU

The pseudocode of the algorithm developed for multithread implementation of MDS is presented in Listing 2. The C++ code was tested on all the platforms from Table 3a. The compiler was the same as for the sequential code. For distribution of computations onto threads we used the OpenMP technology. The maximal number of threads does not exceed the total accessible number of cores for tested configurations. The only exception was Intel Core i5, which due to Hyper-Threading technology allows for concurrent execution of two treads on a single core. The calculations on Itanium cluster ("Baribal") was performed by using 20 processors while the tests on Xeon 5670 processor was conveyed on one node of HP SL390 cluster, i.e., 12 threads. As shown in Fig.7, the concept of reading the large distances array using block_by_block method is more efficient than previously used row_by_row one. It allows for better optimization of the cache memory for larger data files. Therefore, just block_by_block method was used in our tests presented below.

In Figs.8a,b we display the average execution times of one time step of the multithread MDS code for tested platforms and H1k-H30k testing datasets. The plots from Fig.8 demonstrate that one node of HP SL390 cluster consisting of two Intel Xeon X5670 CPUs (12 cores) remains

unbeatable. It is 2-3 times faster than 20 processors of SGI Altix 3700 cluster. As before, Xeon X5670 shows its advantage over the other platforms in *double* arithmetic.

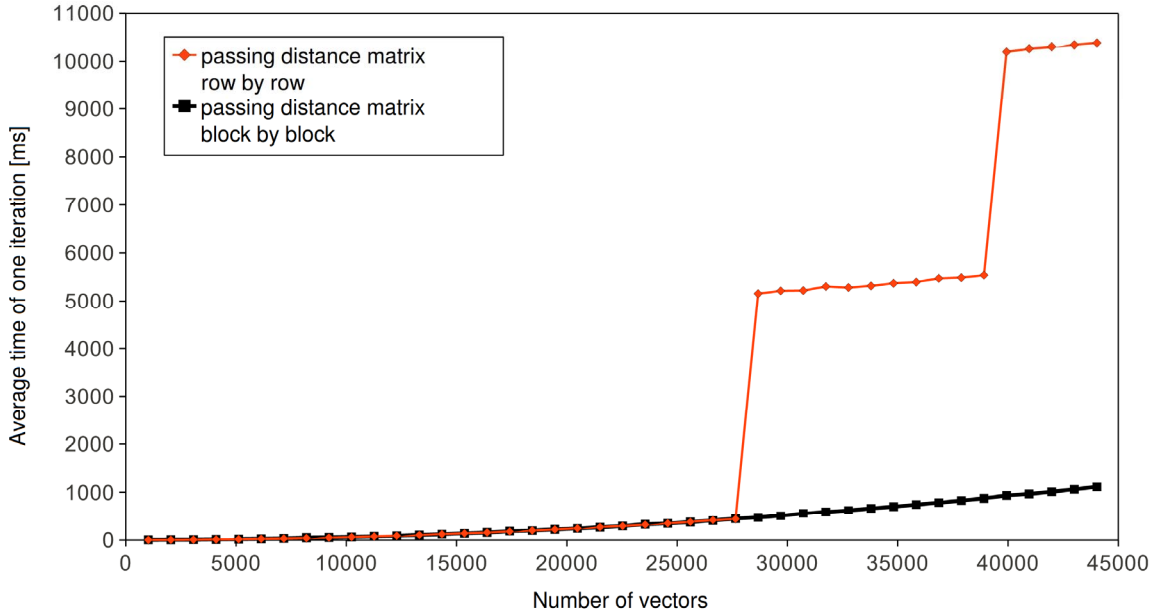

Fig.7 The average times of one MDS timestep for two methods of dissimilarity array reading. The red plot corresponds to the row_by_row method while the black one to block_by_block algorithm (see Fig.3b).

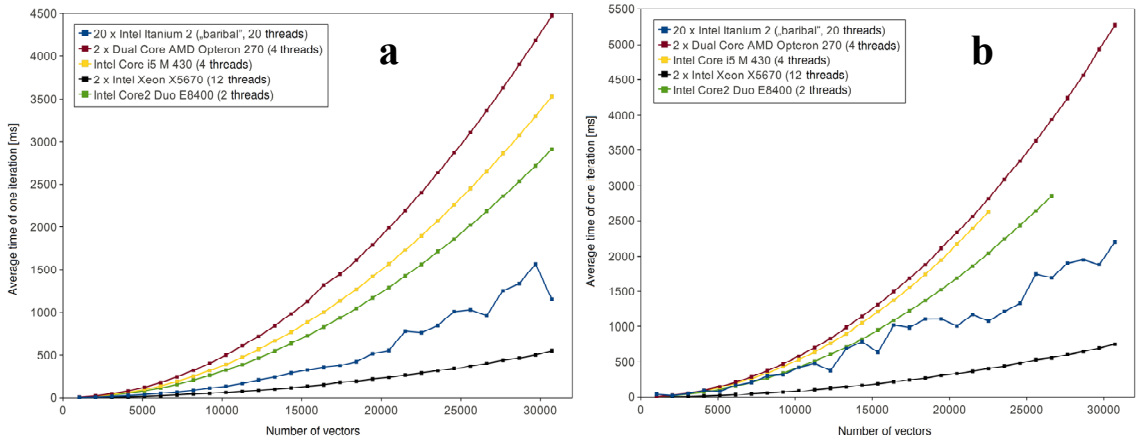

Fig.8 The averaged timings of the multithread version of MDS algorithm for a single timestep for a) *float* and b) *double* arithmetic.

The high efficiency of parallel implementation of our algorithm from Listing 2 is confirmed by nearly liner speedups collected in Table 4. They were obtained for H30k data file and *float* arithmetic. It means that the efficiency achieved is higher than 90%. The worst efficiency was obtained for four threads executed on two cores of Intel Core i5. The Hyper-Threading technology was not able to substitute two additional cores, so the speedup is relatively low.

Table 4. Speedups obtained for various computer architectures and number of threads

CPU type	Threads count			
	2	**4**	**12**	**20**
Intel Itanium 2 („Baribal")	1.95	3.99	11.74	18.56
Dual Core AMD Opteron 270	2.00	3.97	----	----
Intel Core i5 M 430	2.00	2.19	----	----
Intel Xeon X5670	2.00	3.92	11.18	----
Intel Core2 Duo E8400	2.00	----	----	----

4.5 Results of tests – GPU

All of GPU boards from Table 3b were tested assuming simplified floating-point arithmetic. Additionally, the GPU boards with compute capability equal to 2.0 or greater were tested for IEEE-754 standard of arithmetic (*ieee* version). In Fig.9 we present the averaged timings of a single iteration of our MDS algorithm from Listing 3 for a set of GPU boards. The timings obtained on two Intel Xeon X5670 CPUs by multithread version of MDS algorithm described in the previous section are shown for comparison.

In Table 5 we collected the speedups for tested GPU boards both for *fast* and *ieee* arithmetic. They were calculated versus both single thread code version and full 12-threads node of HP SL390 cluster (2 x Intel Xeon 5670). The measurements were performed for the largest testing datasets which fit to the GPU global memory, i.e., H13k for older GPUs (Fig.9a) and H18k for newer ones (Fig.9b).

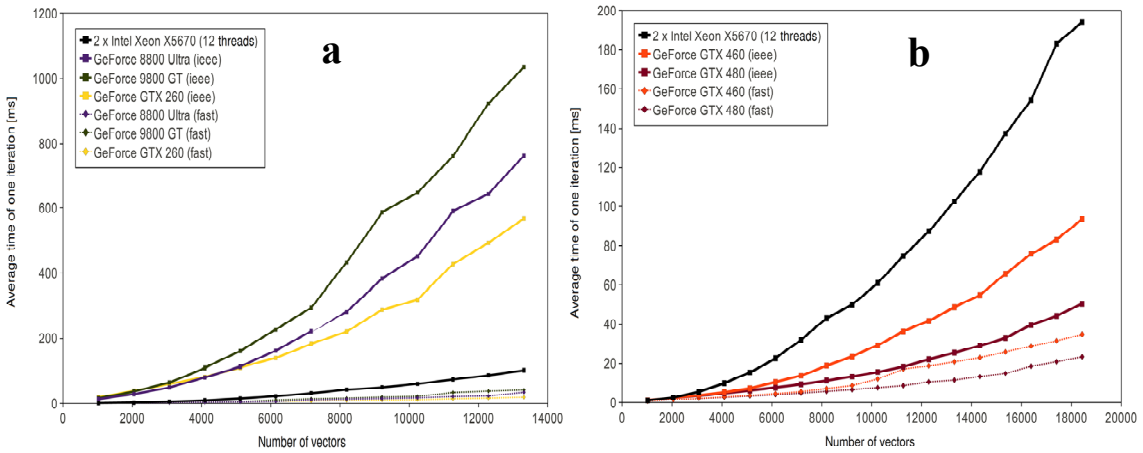

Fig.9 Timings obtained for a) older GeForce boards and b) GTX 460 and Fermi boards versus two Intel Xeon X5670 CPUs for two types of arithmetic (*fast* and *ieee*).

Apart from the oldest GPU boards the rest ones were faster than the cluster node. The timings obtained by the optimal OpenMP code on the cluster node were comparable to the CUDA code executed on the weak GeForce GT 330M GPU board from a medium class laptop. The PCs with stronger graphic boards such as GeForce GTX 260 or GeForce GTX 460 are five times faster while Tesla M2050 and GeForce GTX 480 beats 7-8 times the cluster node performance for fast mode arithmetic (Table 5).

Table 5 Speedups obtained for various NVIDIA GPU boards measured against single node with two Intel Xeon X5670 processors for *ieee* arithmetic.

GPU type	*FAST floating points operations*		*IEEE floating points operations*	
	vs. 2 processors (12 threads)	***vs. single core (1 thread)***	***vs. 2 processors (12 threads)***	***vs. single core (1 thread)***
GeForce 8500 GT	0.3	2.8	----	----
GeForce 8800 Ultra	2.9	32.8	----	----
GeForce 9500 GT	0.8	8.4	----	----
GeForce 9800 GT	2.5	28.3	----	----
GeForce GT 330M	1.2	13.4	----	----
GeForce GTX 260	5.0	57.1	----	----
GeForce GTX 460	5.1	58.0	2.1	24.3
GeForce GTX 480	8.3	94.6	3.9	43.7
Tesla M2050	7.1	80.1	3.6	40.6

The advantage of GPU processors shrinks if the full floating-point arithmetic according to IEEE-754 standard is necessary. As shown in Fig.9 and Table5, only the GPUs with compute capability greater than 2 meet this requirement. The performance drops 2 times in that case. However, as shown in Table 6, increasing requirements concerning computational accuracy additionally diminishes this difference. For *double* arithmetic CUDA codes when run on Fermi and TeslaM 2050 are almost 2 and, respectively, 3 times faster than corresponding OpenMP code executed on two Intel Xeon 5076 processors. Albeit the advantage over cluster node is still evident, three additional factors which additionally shrink GPU advantage over CPU should be taken into account:

1. The coding time using CUDA is a few times slower and much more sophisticated than exploiting OpenMP standard.
2. The Fermi and Tesla boards are expensive and are not scalable.
3. The global memory of GPU boards is at least two times smaller than operation memory of tested CPU board.

Table 6 Speedup measured against single node with two Intel Xeon X5670 processors for *double* arithmetic.

GPU type	**vs. 2 processors (12 threads)**	**vs. single core (1 thread)**
GeForce GTX 260	0.5	5.8
GeForce GTX 460	0.8	9.4
GeForce GTX 480	1.8	20.1
Tesla M2050	2.8	31.3

In respect to MDS and interactive visualization of large datasets, the third aspect is especially painful. The idea of keeping only a part of distances array in GPU global memory is extremely

inefficient because of small throughput between operational and global memories (16 GB/s for PCI-Express x16.2x).

4.6 Results of tests – MPI cluster

As was mention in the introduction, processing a very large dataset is not only computationally but also memory bounded by O(M^2) factor. For large M the memory consumption is beyond the ability of a single processor board. The memory shortage problem becomes more obvious if the running OS is 32-bit which can handle at most 4GB virtual memory per process. Therefore, we can utilize distributed resources of a MPI computer cluster to handle very large data. For large data processing we have developed a parallel MPI version of the MDS which is described in Section 3.3.

For testing purposes we used two MDS code versions exploiting two-level parallelism on a small MPI cluster (HP SL390). As the first level we employed the MDS algorithm written for CPU nodes with OpenMP directives and the GPU version of MDS with CUDA programming interface. The second level parallelism exploits cluster node topology and is realized by the MPI based algorithm described in Section 3.3. In the tests we used MPICH2 [32] environment which is consistent with MPI2 standard. It allows the code to be executed both on the computer cluster and a single multiprocessor.

The CPU and GPU versions were tested using H40k and H24 datasets, respectively. Different size of testing datasets is due to the limitations imposed by the size of GPU global memory. The computations were performed using *float* arithmetic. The speedups for these two parallel versions of MDS were compared to the timings obtained for multithread MDS developed for multi-core CPU on one cluster node (Listing 2) and for one Tesla M2050, respectively. The MPI code was developed in accordance with the algorithm described in section 3.3. As shown in Fig.10, the efficiency of a single node is around 40-50% for both versions. In comparison to CPU version the GPU one gives lower speedup. This is due to smaller size of data processed and faster execution time on a single GPU board. In that case the serial component in the Amdahl law, i.e., the communication between the GPU boards, is greater than for slower CPU version.

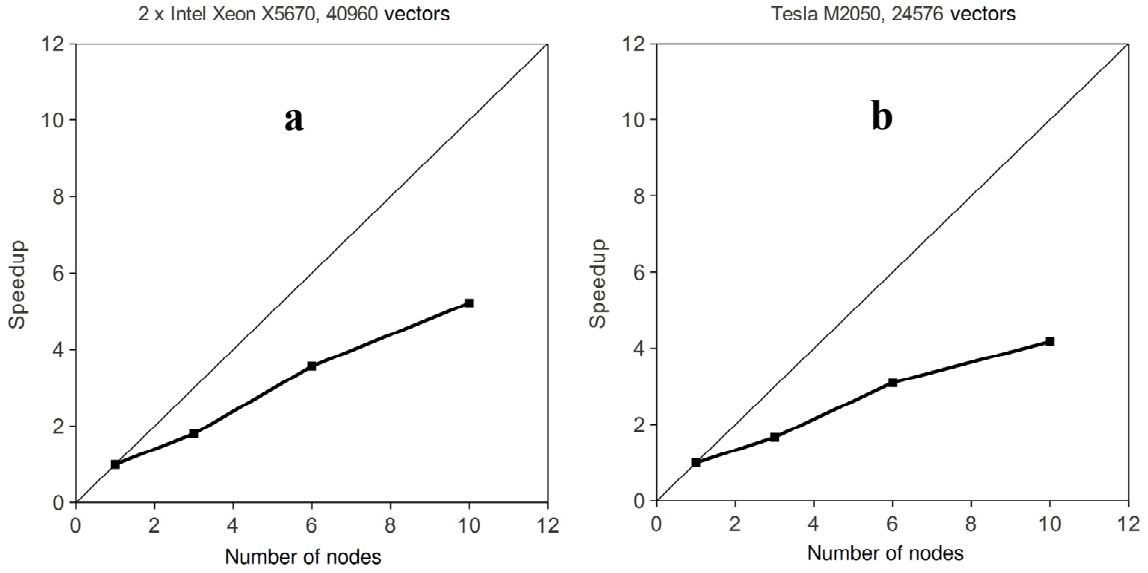

Fig.10 The speedups for HP SL390 MPI cluster employing two level parallelism: MPI interface and a) OpenMP on CPU nodes (2x XenonX5670), b) CUDA on GPU nodes (Tesla M2050).

5 Related work

Because data mining of large data sets became one of the hottest topics in computer science, multidimensional scaling has recently attracted much attention as a robust visualization tool for data exploration [1,2[. Knowing MDS limitations such as quadratic memory and computational complexity, plenty of approaches were developed concerning both methodological and implementation issues. The most recent reviews can be found in [1,2]. Among many approaches to multidimensional scaling the classical concept dominates [9,12,13,14]. It employs dissimilarity matrix between objects as a most reliable representation of the multidimensional data topology. The major factors which differentiate the MDS methods based on dissimilarity matrix are as follows:

1. Definition of dissimilarity and the metrics in the context of non-metric and metric spaces, respectively [8-13].
2. Usage of the partial dissimilarity matrix or its approximation (such as in [19, 20, 33, 34, 35]).
3. The type of minimized cost function ("stress function") [9,11,1213,14].
4. The choice of minimization procedure (e.g in [1,2,4,5,17,36]).
5. The implementation issues (e.g., [1,2,37]).

Probably, one of the first papers which address the problem of implementation of MDS method on GPU was written by Reina and Ertl, 2005 [38]. The paper presents GPU version of FastMap [39] - a quite simple and very fast visualization method. Its GPU version was implemented with the use of OpenGL library. Execution times obtained on GeForce 6800 GT were about 40 times lower than those obtained by CPU implementation run on a single Pentium 4 2.4 GHz processor.

Another GPU implementation of multidimensional scaling method can be found in [40]. This paper describes an implementation of HitMDS algorithm in CUDA environment. The HitMDS algorithm bases on maximization of Pearson correlation between original and target distance matrices and was presented in earlier papers [41] and [42]. The Authors reported that on NVidia TESLA S870 GPU rack they gained speedup within interval 50-60 measured against Matlab implementation run in multi-thread mode on a 16 core server equipped with 3 GHz AMD Opteron CPUs. The NVidia TESLA S870 GPU rack consists of four Tesla C870 processors. Single Tesla C870 processor is comparable to GeForce 8800 Ultra graphic card.

According to the review paper [2] the most efficient implementation of MDS algorithm allowing for visualization of $2x10^5$+ feature vectors is the GPU implementation of GLIMMER algorithm described in [35]. It integrates two other approaches: Chalmers's algorithm ([34]) and Multigrid MDS ([40]). GLIMMER was designed in a way allowing for its direct and efficient implementation in GPU environment. However, as shown in [28], the final results of GLIMMER mapping are far from the global minimum of the cost function (1). It was shown in [28] that the MDS method employing particle dynamics and incomplete distances matrix can achieve similar GPU efficiency as GLIMMER with considerably smaller error (1).

Anyway, in this paper we concentrate on the implementation of MDS algorithm which uses full dissimilarity matrix. Unlike approximate algorithms, such as GLIMMER, it ensures that the mapped structure is unambiguous. By using full dissimilarity matrix we avoid systematic errors caused by overridden number of degrees of freedom in approximate algorithms.

In [5,6,31,33] we have presented a few parallel implementations of MDS based on particle dynamics. Our algorithms were inspired by well known molecular dynamics parallel codes. In [31] we reported satisfactory linear speedup with 40%-80% efficiency using 36 nodes (144 treads) and $6x10^4$ feature vectors. The most recent parallel MPI implementation of MDS with full dissimilarity matrix – SMACOF - is presented in [37]. The SMACOF (Scaling by MAjorizing a COmplicated Function) algorithm bases on function majorizing concept [36]. In general, the

minimum obtained by using this type of algorithm is local and is not as good as those obtained by heuristics (e.g. [5,6,28]). However, unlike for heuristics, it can be achieved much faster. In [37] the Authors report a very good performance of their parallel MDS algorithm on the clusters of AMD Opteron 8356 (2.3GHz) and Intel Xeon E7450 (2.4 GHz) consisting of 256 and 768 nodes, respectively. The largest data set visualized consists of 10^5 feature vectors.

Though the MPI implementations allow for visualization of the datasets of the largest sizes due to the scalable memory, the simulation times are still unsatisfactory to enable interactive visualization of 10^4 feature vectors. Meanwhile, as shown in Fig.9b, one iteration of simulation of dynamics of 19,000 particles (feature vectors) using our MDS virtual particle algorithm running on Fermi GPU board and using *fast* arithmetic requires about 20 milliseconds. For a typical number of time steps needed to obtain a stable minimum i.e., n=1000-5000, we obtain the total computational time equal to 2-10 seconds. This result is more than satisfactory for interactive visualization and control having in mind that the system can be additionally interactively controlled during particle system evolution.

6 Discussion and conclusions

Mapping of original matrix **D** representing dissimilarities between data objects onto feature vectors from 3D-2D Euclidean space allows for interactive visualization of non-metric, non-Euclidean feature spaces. The visualization of the process of minimization of the error function, possibility of deleting outliers, changing on the fly the parameters and error function are the basic procedures which can be used for interactive exploration of the feature space and search for data dependences. Such the interactivity is possible by using robust heuristics which is based on dynamics of virtual particles corresponding to respective data objects [5,6,20,28]. Because the particle system dynamics consisting of dissipative particles is a metaphor of the stress function minimization, the process of particle evolution is meaningful on its own. Tracing particles allows for better matching the parameters of simulation, helps to find the best minimum of error function which corresponds to the minimum of potential energy of the whole particle system.

However, due to quadratic complexity of the MDS problem, to enable visualization of datasets consisting of 10^4+ feature vectors two approaches are possible. The first one consists in developing efficient parallel algorithms exploiting modern parallel programming paradigms, interfaces and processor architectures. This is just what we do in this paper. The second way is to develop approximate algorithms of lower computational complexity such as those using sparser dissimilarity matrices [19, 20, 33, 34, 35] (i.e., disabling or approximating most of distances in **D**). Although such the approach allows for visualization of larger datasets [35], it can be done at the expense of more difficult control of the total error and longer time of coding. The sparser distances matrix needs to be fit in more sophisticated data structures and the conditions of efficient parallelization impose additional constrains on these structures (e.g. requirement of data locality).

To compare the efficiency of the developed parallel MDS algorithms in different programming environments, we have collected in Fig.11 and Fig.12 the most important timings obtained for the same datasets. The timings for *float* arithmetic shown in from Fig.11 were obtained for H18k dataset. The identical tests for *double* arithmetic presented in Fig.12 were performed for H12k dataset. The size of datasets was limited by the size of global memory of GeForce GTX 480 (Fermi) GPU board. In the Figs.11-12, we present the two best timings obtained by multithread OpenMP version of the algorithm and three best timings obtained on the GPU boards. The timings for GPU boards were made for *ieee* arithmetic. We have added also the timings obtained on MPI cluster with both CPU and GPU nodes. Unfortunately, due to the small size of datasets and lesser number of nodes the speedups obtained were considerably worse than those described in section 4.6.

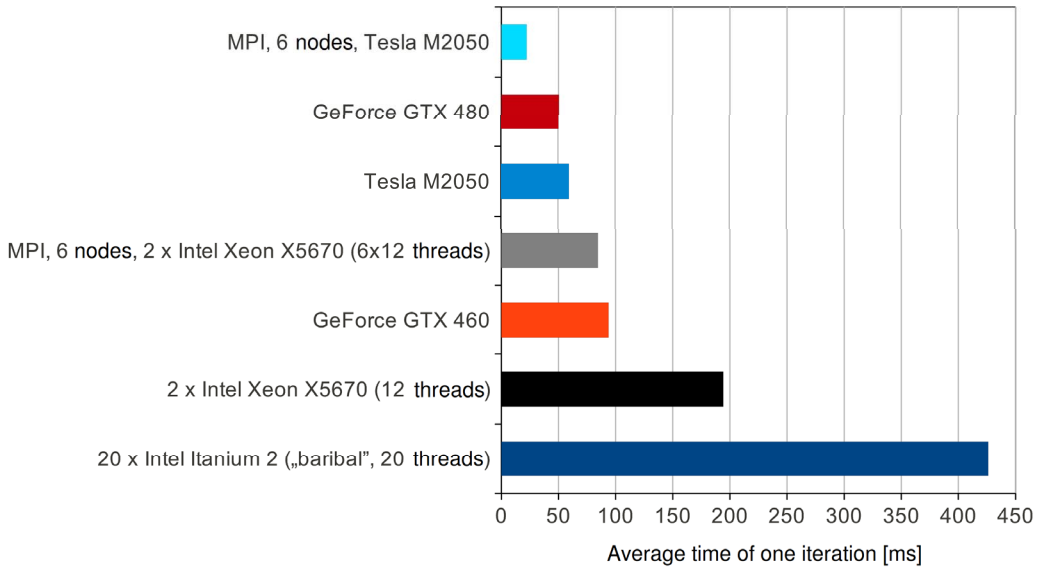

Fig.11 Comparison of average times of a single MDS iteration on chosen CPUs, GPUs, nodes and MPI clusters made for *float* arithmetic.

As shown in Fig.11, the advantage of GPU over CPU systems is evident for *float* arithmetic. Just by employing small 6 nodes cluster (12 Intel Xeon X5670 processors) allows for achieving timings a little bit better than for a single GeForce 460 GTX GPU board. From results obtained in section 4.6 we expect that for larger datasets the MPI cluster will be more efficient. So, the computational power of 6 two-processor server should be comparable rather to Fermi or Tesla boards. It means that a workstation with 4 strong GPU boards can have similar computational power as a professional sever equipped with several two-processor nodes.

As demonstrated in Fig.12, if a *double* precision arithmetic is required the advantage of GPU boards becomes questionable. Only the strongest (and the most expensive) GPU boards, such as Fermi and Tesla, are about 2 times faster than two-processor cluster node. As shown in Figs.11-12, the limited size of global memory of GPU boards can be partially compensated by using more GPU nodes. However, comparing the tendency of speedups from Fig.10 we expect that the gap between CPU and GPU performance may shrink additionally by employing tens of CPU nodes of MPI-cluster.

Such the bottlenecks like double precision arithmetic, relatively small global memory of GPU boards, algorithmic constrains, the difficulty in CUDA or OpenCL programming and limited portability of CUDA codes, are still serious disadvantages of more broad exploitation of computational capabilities of GPU boards. On the other hand for the problems like multidimensional scaling and interactive visualization of large datasets, where *fast* arithmetic mode is sufficient for obtaining satisfactory results, the advantage of GPU boards and clusters over CPU equivalents is overwhelming (see Fig.9b). The Fermi GPU board is then about 10 times faster than two-processor, 12-thread Intel Xeon X5670 board.

To sum up, the implementation of multidimensional scaling employing particle dynamics in GPU computational environment allows to visualize interactively datasets consisting of more than 10^4 objects (feature vectors) on PCs equipped with GPU boards with compute capability around 2 (Tables 1,3b).

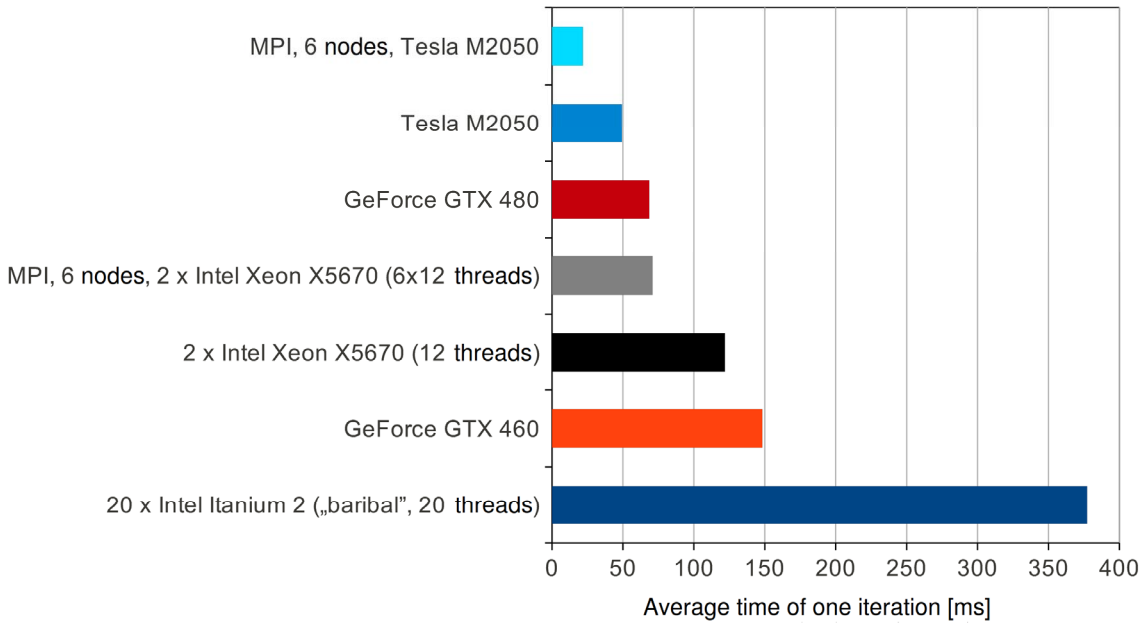

Fig.12 Comparison of average times of a single MDS iteration on chosen CPUs, GPUs, nodes and MPI clusters made for *double* arithmetic.

Acknowledgements This research is financed by the Polish Ministry of Higher Education and Science, project NN519 443039 and partially by AGH grant No.11.11.120.777. We would like to thank Nvidia Company for support and donating the Authors with Fermi GPU board.

References

1. Borg I., Groenen P.J.F. *Metric and Nonmetric MDS. Modern multidimensional scaling: theory and applications*. Springer Verlag, second edition, 2005.
2. France S.L., Carroll J.D. Two-Way Multidimensional Scaling: A Review. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 41, 644-661, 2011.
3. Li Y. Distance-preserving mapping of patterns to 3-space. *Pattern Recognition Letters*, 25, 119-128, 2004.
4. Klock H., Buhmann J.M. Data visualization by multidimensional scaling: a deterministic annealing approach. *Pattern Recognition*, 33, 651-669, 2000.
5. Dzwinel W., Błasiak J., Method of particles in visual clustering of multi-dimensional and large data sets, *Future Generation Computer Systems*. 15, 365-379, 1999
6. Dzwinel W., Virtual particles and search for global minimum, *Future Generation Computer Systems*. 12, 371-389, 1997
7. Younes L., Computable elastic distances between shapes. *SIAM J. Appl.Math*. 58(2): 565-586, 1998
8. Young G., Householder A. S., Discussion of a set of points in terms of their mutual distances, *Psychometrika*. 3(1):19-22, March 1938.
9. Torgerson W. S., Multidimensional scaling. 1. Theory and method, *Psychometrika*, 17:401-419, 1952.
10. Torgerson W. S., *Theory and methods of scaling*, John Wiley & Sons, New York, 1958.
11. Coombs C. H., *A theory of data*, John Wiley & Sons, New York 1964, Chapters 5, 6, 7, pp. 80-180.

12. Kruskal J., Multidimensional scaling by optimizing goodness-of-fit to a nonmetric hypothesis, *Psychometrika* 29, 1-27, 1964.
13. Sammon J.W., A nonlinear mapping for data structure analysis, *IEEE Trans. Comput*. C-18 (5), 401-409, 1969.
14. Niemann, H. Linear and nonlinear mapping of patterns, *Pattern Recogn*. 12(2), 83-87, 1980
15. Dzwinel, W., How to Make Sammon's Mapping Useful for Multidimensional Data Structures Analysis? *Pattern Recognition*. 27(7), 949-959, 1994
16. Mathar R and Zilinskas A., On global optimization in two dimensional scaling, *Acta Applicandae Mathematicae*. 33/1, 109–118, 1993.
17. Varoneckas, A. Zilinskas, and J. Zilinskas, Multidimensional scaling using parallel genetic algorithm. *Computer Aided Methods in Optimal Design and Operations*, 129–138, 2006.
18. De Leeuw J., Mair P. Multidimensional Scaling Using Majorization: SMACOF in R. Journal Of Statistical Software, 31, 1_30, 2009.
19. Seung-Hee Bae, Judy Qiu, Geoffrey Fox. Adaptive Interpolation of Multidimensional Scaling, *Procedia Computer Science*, 9, ICCS 2012, 393–402, 2012.
20. Kurdziel, M., Boryczko, K., Dzwinel W., Procrustes analysis of truncated least squares multidimensional scaling, *Computing and Informatics,* 2012 in press
21. Nguyen, D., Dzwinel, W., Cios, K.J., Visualization of Highly-Dimensional Data in 3-D Space, *Proceedings of the 2011 11th International Conference on Intelligent Systems Design and Applications*, 22-24 November 2011, Cordoba, Spain, 225-230.
22. Rapaport D.C., *The Art of Molecular Dynamics Simulation*, Cambridge University Press, New York, 1996
23. Kirkpatrick, S., Gelatt Jr. CD, Vecchi, MP. Optimization by simulated annealing. *Science.* 220/4598, 671-680, 1983.
24. Dzwinel, W., Yuen, D.A., Boryczko, K., Ben-Zion, Y., Yoshioka, S., Ito, T., Nonlinear multidimensional scaling and visualization of earthquake clusters over space, time and feature space, *Nonlinear Processes in Geophysics,* 12, 117-128, 2005
25. Ahlrichs R., Brode S., An optimized MD program for the vector computer CYBER-205, *Comput. Phys. Commun*. 42/1, 51-55, 1986.
26. Ahlrichs R., Brode S., A new rigid motion algorithm for MD simulations, *Comput. Phys. Commun*. 42, 59-64, 1986.
27. Smith, W., Forester, TR., Parallel macromolecular simulations and the replicated data strategy: I. The computation of atomic forces. *Comput. Phys. Commun*. 79/1, 52-62, 1994.
28. Pawliczek, P., Improvement of multidimensional scaling efficiency in the context of interactive visualization of large data sets. PhD thesis AGH University of Science and Technology, Department of Computer Science, Krakow, Poland, 2012
29. Shu J, Wang B, Chen M, Wang J, Zheng W., Optimization techniques for parallel force-decomposition algorithm in molecular dynamic simulations, *Computer Physics Communications*, 154, 121-130, 2003
30. Taylor VE., Stevens RL, Arnold KE, Parallel molecular dynamics: Communication requirements for massively parallel machines, *Proceedings of Frontiers'95, Fifth Symposium on the Frontiers of Massively Parallel Computation*, p. 156, IEEE Comput. Society Press, 1994
31. Pawliczek, P., Dzwinel, W., Parallel Implementation of Multidimensional Scaling Algorithm Based on Particle Dynamics, *Lecture Notes in Computer Science*, PPAM, Wrocław, 13-16 September 2009, LNCS 6067, 312-321, 2010
32. Gropp W. MPICH2: A New Start for MPI Implementations. D. Kranzlm üller, J. Volkert, P. Kacsuk, J. Dongarra (editors), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, tom 2474, 7_7. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
33. Pawliczek, P., Dzwinel, W., Visual Analysis of Multidimensional Data Using Fast MDS

algorithm, Proceedings of 20th IEEE-SPIE Symposium Photonics, Electronics and Web Engineering, Signal Processing Syposium, 24-26 May 2007, Jachranka, Poland.
34. Chalmers M. A linear iteration time layout algorithm for visualizing high-dimensional data. *Visualization '96. Proceedings*, 127-131. IEEE, 1996.
35. Ingram S., Munzner T., Olano M. Glimmer: Multilevel MDS on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 15, 249-261, 2009.
36. De Leeuw J., Mair P. Multidimensional Scaling Using Majorization: SMACOF in R. Journal Of Statistical Software, 31, 1_30, 2009.
37. Seung-Hee Bae, Judy Qiu and Geoffrey Fox, High Performance Multidimensional Scaling for Large High-Dimensional Data Visualization, *IEEE Transaction of Parallel and Distributed System*, January 2012, (in press)
38. Reina G., Ertl T. Implementing FastMap on the GPU: Considerations on General-Purpose Computation on Graphics Hardware. Theory and Practice of Computer Graphics 2005, 51-58, 2005.
39. Faloutsos C, Lin K.-I FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (1995), 163-17
40. Bronstein M.M., Bronstein A.M., Kimmel R., Yavneh I. Multigrid multidimensional scaling. *Numerical Linear Algebra with Applications*. 13, 149-171, 2006. Fester T, Schreiber F, Strickert M CUDA-based multi-core implementation of MDS-based bioinformatics algorithms. Lecture Notes in Informatics Series of the German Informatics Society (GI). German Conference on Bioinformatics, 67-79, 2009
41. Strickert M, Teichmann S, Sreenivasulu N, Seiffert U: High-Throughput Multi-Dimensional Scaling (HiT-MDS) for cDNA-Array Expression Data. *Lecture Notes in Computer Science*, 3696, 625-634, 2005.
42. Strickert M, Sreenivasulu N, Usadel B, Seiffert U. Correlation-maximizing surrogate gene space for visual mining of gene expression patterns in developing barley endosperm tissue. *BMC Bioinformatics*, 8:165, 2007