Parallel Multidimensional Scaling Performance on Multicore Systems

Seung-Hee Bae
sebae@cs.indiana.edu

Community Grids Laboratory
Indiana University, Bloomington

Abstract

SMACOF algorithm is a gradient descent approach to solve Multidimensional scaling problem. We design parallel SMACOF program using parallel matrix multiplication to run on a multicore machine. Also, we propose a block decomposition algorithm based on the number of threads for the purpose of keeping good load balance. In this paper, we investigate performance results of the implemented parallel SMACOF in terms of the block size, data size, and the number of threads. The speedup (or efficiency) factor is almost 7.7 *with 2048 points data over 8 running threads. In addition, performance comparison between jagged array and two-dimensional array in C# language is carried out. The jagged array data structure performs at least* 40% *better than the two-dimensional array structure.*

1. Introduction

As CPU clock speed-up is limited by several physical limitations, such as heat, energy loss, and limit of light speed, most CPU architecture companies introduced multicore architectures, which is the most popular CPU architecture now. Since multicore architectures were invented, multicore architectures are getting important in software development and effecting on client, server and supercomputing systems [1, 7, 8, 16]. In addition, the CPU developers have a vision to develop many-core chips which will have hundreds or thousands cores on a single chip. As [16] mentioned, the parallelism became a critical issue to develop softwares for the purpose of getting maximum performance gains of multicore machines.

Also, Intel proposed that the Recoginition, Mining, and Synthesis (RMS) approach as a killer application for the next data explosion era, and machine learning and data mining algorithms are suggested as important algorithms for the data deluge era by [8]. Those algorithms are, however, usually highly compute-intensive algorithms, so the running time will be increasing in quadratic or even more as the data size increases. As [8] already described, the amount of data will be huge in every domain, due to digitization of not only scientific data but personal documents, photos, and videos. From the above statements, the necessary computation will be enormous for data mining algorithms in the future, so that implementing in scalable parallelism of those algorithms will be one of the most important procedures for the coming many-core and data explosion era.

As the amount of the scientific data is also increasing, data visualization could be another interesting area since it is hard to imagine data distribution of the most scientific data or to depict relative position among data points due to high dimensionality. Dimension reduction algorithms is used to reduce dimensionality of high dimensional data into viewable low dimensional space, so that dimension reduction algorithms are used as visualization tools. Some dimension reduction approaches, such as generative topographic mapping (GTM) [2, 17] and Self-Organizing Map (SOM) [11], seek to preserve topological properties of given data rather than proximity information, and other methods, i.e. multidimensional scaling (MDS) [13, 3], work on maintaining proximity information, similarity or dissimilarity information, between points as much as possible. In this paper, the author uses parallel matrix multiplication to parallelize an elegant algorithm, named SMACOF (Scaling by MAjorizing a COmplicated Function) [5, 6, 10], for computing MDS solution in C# language and presents performance analysis of parallel implementation of SMACOF on multicore machines.

Section 2 describes MDS and the SMACOF algorithm which is used in this paper, briefly. How to parallelize the described SMACOF algorithm is shown in Section 3. We explain the experimental setup in Section 4. In Section 5, the author discusses the performance results with respect to several aspects, such as block size, the number of threads, and a C# language specific issue. Finally, the conclusion and the future works are in Section 6.

2. Multidimensional Scaling (MDS)

Multidimensional scaling (MDS) [13, 3] is a general term for a collection of techniques to configure data points with proximity information, typically dissimilarity (interpoint distance), into a target space which is normally Euclidean low-dimensional space. Formally, the dissimilarity matrix should be symmetric, nonnegative, and zero diagonal elements. Thus, the conditions of nonnegative elements ($\delta_{ij} \geq 0$), zero diagonal elements ($\delta_{ii} = 0$), and symmetricity ($\delta_{ij} = \delta_{ji}$) should be met for an $n \times n$ dissimilarity matrix $\Delta = (\delta_{ij})$, where n is the number of points. From the given dissimilarity matrix Δ, a configuration of points is constructed by the MDS algorithm in a Euclidean target space with dimension p. The output of MDS algorithm can be an $n \times p$ configuration matrix X, whose rows represent each data points in Euclidean p-dimensional space. From configuration matrix X, it is easy to compute the Euclidean interpoint distance $d_{ij}(X)$ among n configured points in the target space and to build the $n \times n$ Euclidean interpoint distance matrix $D(X) = (d_{ij}(X))$. The purpose of MDS algorithm is to construct a configuration points into the target p-dimensional space, while the interpoint distance $d_{ij}(X)$ is approximated to δ_{ij} as much as possible. Two different measurements were suggested as an objective value of MDS algorithms. First, Kruskal proposed STRESS (σ or $\sigma(X)$) criterion (Eq. 1) which is based on squared error between distance of configured points and corresponding dissimilarity [12]. The SSTRESS (σ^2 or $\sigma^2(X)$) criterion (Eq. 2) is based on squared error between squared distance of mapped points and squared dissimilarity [18].

$$\sigma(X) = \sum_{i<j} w_{ij}(d_{ij} - \delta_{ij})^2 \tag{1}$$

$$\sigma^2(X) = \sum_{i<j} w_{ij}[(d_{ij})^2 - (\delta_{ij})^2]^2 \tag{2}$$

where $d_{ij} = \|x_i - x_j\|$ and $w_{ij} \geq 0$.

Therefore, the MDS can be thought of as an optimization problem, which is minimization of the STRESS or SSTRESS criteria during construction of a configuration points in the p-dimension target space.

2.1. Scaling by MAjorizing a COmplicated Function (SMACOF)

Scaling by MAjorizing a COmplicated Function (SMACOF) [5, 6, 10] is an iterative majorization algorithm in order to minimize STRESS criterion. SMACOF is a variant of steepest descent approach so likely to find local minima. Though it is trapped in local minima, it is powerful since it guarantees monotone decreasing the STRESS (σ) criterion. We will not explain the mathematical details of SMACOF in this paper (for detail, refer to [3]), but introduce the essential equation, called Guttman transform

$$X = V^{\dagger} B(Z) Z \tag{3}$$

where $V^{\dagger}$ is the Moore-Penrose inverse (or Pseudo inverse).

The Guttman transform is generated from the equation $\nabla\sigma(X) = 0$ which can be written as $VX = B(Z)Z$.

$$V = (v_{ij}) \tag{4}$$

$$v_{ij} = \begin{cases} -w_{ij} & \text{if } i \neq j \\ \sum_{i \neq j} w_{ij} & \text{if } i = j \end{cases} \tag{5}$$

$$B(Z) = (b_{ij}) \tag{6}$$

$$b_{ij} = \begin{cases} -w_{ij}\delta_{ij}/d_{ij}(Z) & \text{if } i \neq j \\ 0 & \text{if } d_{ij}(Z) = 0, i \neq j \\ -\sum_{i \neq j} b_{ij} & \text{if } i = j \end{cases} \tag{7}$$

If the weights are equal for all distances ($w_{ij} = 1$), then

$$V = n\left(I - \frac{ee^t}{n}\right) \tag{8}$$

$$V^{\dagger} = \frac{1}{n}\left(I - \frac{ee^t}{n}\right) \tag{9}$$

where $e = (1, \ldots, 1)^t$ is unit vector whose length is p. In this paper, we use equal weights.

The iteration is processed by substitution $X^{[k-1]}$ into Z like following Eq. (10)

$$X^{[k]} = V^{\dagger} B(X^{[k-1]}) X^{[k-1]} \tag{10}$$

where $X^{[k]}$ is the configuration matrix of k iteration, and $X^{[0]}$ is random initial configuration or a prior initial configuration matrix. Finally, SMACOF algorithm stops when $\Delta\sigma(X^{[k]}) = \sigma^{[k-1]} - \sigma^{[k]} < \varepsilon$, where ε is a relatively small threshold constant. The Algorithm 1 illustrates the SMACOF algorithm for MDS solution.

Algorithm 1 SMACOF algorithm

$Z \Leftarrow X^{[0]}$;
$k \Leftarrow 0$;
$\varepsilon \Leftarrow$ small positive number;
Compute $\sigma^{[0]} = \sigma(X^{[0]})$;
while $k = 0$ or ($\Delta\sigma(X^{[k]}) > \varepsilon$ and $k \leq$ max iterations) **do**
  $k \Leftarrow k + 1$;
  Compute the Guttman transform $X^{[k]}$ by Eq. (10)
  Compute $\sigma^{[k]} = \sigma(X^{[k]})$
  $Z \Leftarrow X^{[k]}$;
end while

points	128	256	512	1024	2048
iteration	47	40	52	46	53
time (sec)	1.43	16.22	176.12	1801.21	17632.51

Table 1. The running times of SMACOF program in naive way for different data on Intel8b machine.

2.2. Time Complexity of SMACOF

As discussed in Section 2.1, SMACOF algorithm consists of iterative matrix multiplication as in Eq. (10), where V and $B(X^{[k-1]})$ are $n \times n$ matrices, and $X^{[k-1]}$ is $n \times p$ matrix. Since the inside of the iteration steps are the dominant steps of the SMACOF approach, the order of SMACOF algorithm must be O($k \cdot (n^3 + p \cdot n^2)$), where k is the iteration number and p is the target dimension. Since the purpose of the MDS approach is the mapping high dimensional data into conceivable low-dimensional space, in most cases two- or three-dimensional space, the author may have the assumption that $p \ll n$, without loss of generality. Thus, the order of SMACOF method might be considered as O($k \cdot n^3$). The iteration number is totally dependent on the stop criterion, threshold value (ε). It can be $k > n$ for small n, but $k < n$ could be typical for large n. We may think of k as a constant integer, which is not too small constant for some large enough data set. Since the order of SMACOF method is proportional to n^3, the running time is highly dependent on the number of points (n).

Table 1 shows the iteration numbers and the average of the 5 running times of SMACOF program in C# on **Intel8b** machine (refer to Table 2 for the detail information of the machine) for each data, which are four-dimensional Gaussian distribution data having different number of points, by a naive (non-parallel, non-block based) SMACOF implementation. In Table 1, the average running time is increased about 10 ($> 2^3$) times as the data size is increased in twice. This result supports the above time complexity analysis of SMACOF. It takes more than 30 minutes for the 1024 points Gaussian distribution data and more than 4 hour 50 minutes for the 2048 points Gaussian distribution data. The number of points, however, could be more than millions in real scientific data which we are interested in, so it would take more than 0.5×10^{10} hours to run the ordinary SMACOF algorithm with those large scientific data. Even though **Intel8b** machine has two 4-core processors which actually 8 cores, the normal SMACOF program uses only one of the 8 cores. (The other 7 cores do nothing, even the program takes almost 5 hours for only 2048-point data.) As in multicore era, implementation in a parallel approach is critical not only for high performance but for hardware utilization. These are the reasons why parallel implementation of SMACOF (or any application which takes huge amount of time) is interesting now.

3. Parallel Implementation of SMACOF

Since the dominant time consuming part of SMACOF program is the iterative matrix multiplication, which is O($k \cdot n^3$), Building parallel matrix multiplication is the most natural thought to implement parallel SMACOF in efficient way. Parallel matrix multiplication makes two benefits in terms of performance issue. First, High hardware utilization and computation speed-up is achieved as implied in parallelism on multicore machines. In addition, the performance gain of cache memory usage is also achieved, since parallel matrix multiplication is composed of a number of small block matrix multiplication, which would be fit into cache line. Fitting into cache line reduces unnecessary cache interferences, such as false sharing and cache I/O overhead. The cache memory issue is significant in performance, as you see in Section 5.1.

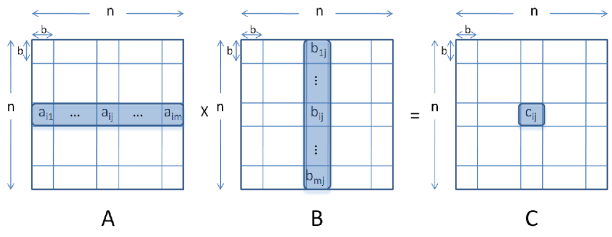

Figure 1. Parallel Matrix multiplication.

Parallel matrix multiplication is composed of a block decomposition and block multiplications of the decomposed blocks. Figure 1 illustrates how to operate matrix multiplication ($A \cdot B = C$) using block decomposition. In Figure 1, $n \times n$ square matrices are used as an example, but square property of matrix is not necessary for the block version of matrix multiplication. Also, decomposed blocks can be rectangular if row of block in matrix A is equal to column of block in matrix B, though the Figure 1 use $b \times b$ square block. In order to compute block c_{ij} in Figure 1, we should multiply ith block-row of matrix A with jth block-column of matrix B, correspondingly, as in Eq. (11).

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj} \tag{11}$$

where c_{ij}, a_{ik}, and b_{kj} are $b \times b$ blocks, and m is the number of blocks in block-row of matrix A and in block-column of matrix B. Thus, if we assume that the matrices A, B, and C are decomposed $m \times m$ blocks by $b \times b$ block, without loss

Algorithm 2 First mapping block index of each thread

/* starting block index of each thread is (0, id) */
$row \Leftarrow 0$;
$col \Leftarrow id$; /* $0 \leq id \leq TH - 1$ */
/* where id is the thread ID, and TH is the number of threads */

Algorithm 3 Mapping blocks to a thread

while $col \geq$ the number of the column blocks **do**
  $row \Leftarrow row + 1$;
  if $row \geq$ the number of the row blocks **then**
    return $false$
  end if
  $col \Leftarrow (row + id + row \cdot (TH - 1)/2)\%TH$;
end while

return $true$

of generality, the matrix multiplication ($A \cdot B = C$) can be finished with m^2 block matrix multiplications. Note that computation of c_{ij} blocks ($1 \leq i, j \leq m$, $c_{ij} \subset C$) are independent each other.

After decomposing matrices, we should assign a number of c_{ij} blocks to each threads. The load balance is essential for assigning jobs (blocks in this paper) to threads (or processes) for the maximum performance gain (or the minimum parallel overhead). In the best case, the number of decomposed blocks assigned each thread should be as equal as possible, i.e. $\lceil m^2/TH \rceil$ or $\lceil m^2/TH \rceil - 1$. Algorithm 3 is used for block assignment to each thread in consideration of load balance in our implementation. As in Algorithm 2, (0, id) block is the starting position of each thread. If (0, id) block exists, then thread id starts computing (0, id). Otherwise, the Algorithm 3 will find an appropriate block to compute for the thread, and then the thread computes the found block. After computation a block multiplication is done, the thread will call again Algorithm 3 with increase only col value to $col + TH$. Iteration these steps, until the thread gets $false$ from calling Algorithm 3. Then the assigned blocks to the thread is done for the matrix multiplication.

After finishing assigned computation, each thread sends a signal to the main thread which is waiting on the Rendezvous (actually 'waitingAll' method is called). Finally, the main thread will complete the matrix multiplication after getting all signals from the participated threads. For the purpose of synchronization and other thread issues, a novel messaging runtime library CCR (Concurrency and Coordination Runtime) [4, 14] developed by Microsoft Research is used for this application.

After needed matrix multiplications are completed, the parallel SMACOF application calculates STRESS value (σ) of the current solution, and measures difference between previous STRESS value and current STRESS value ($\Delta\sigma$). If $\Delta\sigma < \varepsilon$, where ε is a threshold value for the stop condition, the application returns the current solution as the final answer. Otherwise, the application will iterate the procedure again using the current solution as previous solution.

4. Experimental Settings

4.1 Experimental Platforms

For the experiments of the parallel SMACOF, two multicore machines depicted in Table 2 are used. Both **intel8a** and **intel8b** have two quad-core CPU chips, and a total of 8 cores. Both of them use Microsoft Windows OS system, since the program is written in C# programming language and CCR for this test.

ID	Intel8a	Intel8b
CPU	Intel Xeon E5320	Intel Xeon x5355
CPU Clock	1.86 GHz	2.66 GHz
Core	4-core × 2	4-core × 2
L2 Cache	2 × 4 MB	2 × 4 MB
Memory	8GB	4GB
OS	XP pro 64 bit	Vista Ultimate 64 bit

Table 2. Multicore PC's used in this experiments

4.2 Experimental Data

For the purpose of checking quality of the implemented parallel SMACOF results, the author generated simple 4-dimensional 8-centered Gaussian distribution data with different number of points, i.e. 128, 256, 512, 1024, 2000, and 2048. The 8 center positions in 4-dimension are following: (0,0,0,0), (2,0,0,0), (0,2,0,1), (2,2,0,1), (0,0,1,0), (2,2,1,0), (2,0,4,1), and (0,2,4,1). Note that the fourth dimension values are only 0 and 1. Thus, for those data, it would be natural to be mapping into three-dimensional space near by center positions of the first three dimensions.

It is sure that the real scientific data are much more complex than those simple Gaussian distribution data in this paper. However, nobody can verify the mapping results of those high dimensional data visually, since it is hard to imagine high dimensional space. That is the reason those proposed simple data used in this paper. Figure 2 shows three different screen captures of the parallel SMACOF results of the given 2000 points data in a 3D image viewer, called Meshview [9]. Combining three images in Figure 2 shows that the expected mapping is achieved by the SMACOF program.

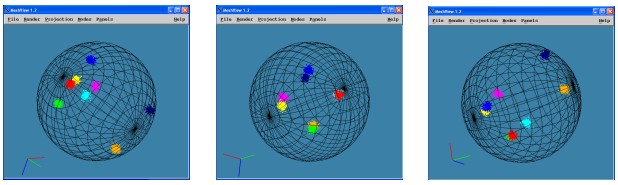

Figure 2. The example of SMACOF results with 8-centered Gaussian distribution data with 2000 points shown with Meshview in 3D.

4.3 Experimental Designs

Several different aspects of performance issue are investigated by the following experimental designs:

- **Different number of block sizes:** This experiment demonstrates the performance of parallel SMACOF program with respect to block size. Three different data set of 4-dimensional Gaussian distribution are used for this test. As Section 5.1 describes, different block size will affect on the cache memory performance, since computers will fetch (or flush) data into cache by a cache line amount of data, whenever it needs to fetch (or flush) data, without regart to the actual needed data size. Note that whenever the author mentions the block size, says b, it means a $b \times b$ block, so that the actual block size is not b but b^2.

- **Different number of threads:** As the number of threads increases, the number of used core will be increased unless the number of threads is bigger than the number of cores in the system. However, when we use more threads then the number of cores, thread performance overhead, like context switching and thread scheduling overhead, will increase. We experiment the number of thread from one to sixteen, double of the number of cores of the used systems in Table 2. This experiment setup will investigate the thread performance overhead and what is the appropriate number of thread for the parallel SMACOF program.

- **Different number of data points:** The bigger data will use more computation time as in Table 1. Using this test setup, The author also examines how efficiency and overhead will change as the data size differs. The tested number of data points are from 128 to 2048 as increased by factor of 2, i.e. 128, 256, ..., 2048.

- **Two-dimensional array vs. jagged array:** It is known as jagged array (array of arrays) shows better performance than two-dimensional array in C# [15][19]. This knowledge is investigated by comparing performance results of the two different versions of parallel SMACOF program, which are based on jagged array and two-dimensional array.

Due to gradient descent attribute of SMACOF algorithm, the final solution highly depends on the initial mapping. Thus, it is appropriate to use random initial mapping for the SMACOF algorithm unless specific prior initial mapping exists, and run several times to increase the probability to get optimal solution. If the initial mapping is different, however, the computation amount can be varied whenever the application runs, so that we could not verify the overhead, efficiency, and speed up of the parallel SMACOF application. Also, any other performance comparison between two experimental setups would be inconsistent. Therefore, though the application is originally implemented with random initial mapping for real solutions, the random seed is fixed for the performance measures of this paper to generate the same answer and the same necessary computation for the same problem. The stop condition threshold value (ε) is also fixed at 1000 for 2000 and 2048 points data, and at 100 for the other data.

5. Experimental Results and Performance Analysis

5.1. Performance Analysis of Different Block Sizes (Cache Effects)

As mentioned the previous section, the author tested performance of parallel SMACOF program with respect to block sizes. We used three different data set of 1024, 2000, and 2048 points data and experimented with 8 threads on **Intel8a** and **Intel8b** machines. The tested block size is increased in factor of 2 from one to 256 for 1024 points data (in order to keep the number of blocks more than the number of threads or cores) or 512 for 2000 and 2048 points data, as shown in Figure 3. Note that the curves for every test case show the same shape. When the block size is 64, the application performs best with 2000 and 2048 points data. For the data with 1024 points, performance of the block size 64 is comparative with block size 128. From those results, we could make conclusion that there are fitter block sizes for each problem and it might different for each machines. For the two machines in Table 2, block size 64 could be fitter than other block sizes. Note that the running time of 2000 points data is longer than that of 2048 points data on both test platforms, even though the number of points are less. The reason is that the iteration number of 2000 data is 80, but that of 2048 data is only 53 for the performance tests.

Running results with only one thread will be more helpful to investigate the cache effect, since there is no other

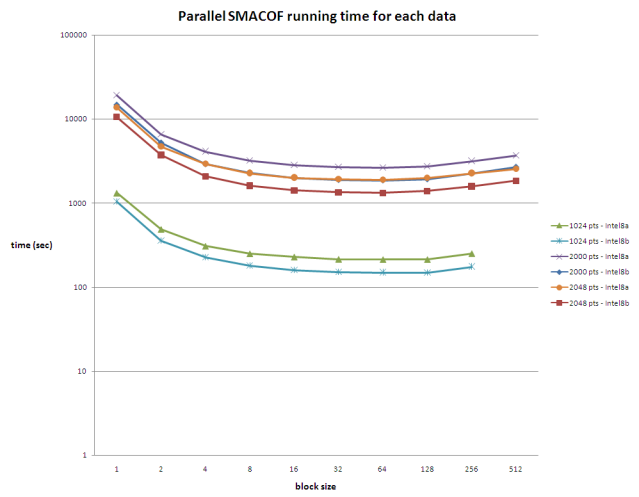

Figure 3. Parallel SMACOF running time with respect to block sizes with 8 threads.

performance criteria except the block size. Table 3 and Table 4 describe the running time with only one thread with 512, 1024, and 2048 points data. Based on the 8-thread results, we chose more fitted block sizes which are $b = 32, 64$ to test cache effect and measure the speedup of selected block sizes based on the result of using one big whole matrix ($n \times n$). As we expected, the result shows that there are more than 1.6 speedup for the 1024 and 2048 points data, and around 1.1 speedup for the even small 512 points data on **Intel8b** and a little smaller speedup on **Intel8a**. Also, performance of $b = 64$ is better than $b = 32$ in all cases. Note that there is some additional tasks (a kind of overheads) for the block matrix multiplication, such as dividing blocks, finding correct block positions, and the iteration of submatrix multiplication.

# points	blockSize	avgTime(sec)	speedup
512	32	228.39	1.10
512	64	226.70	1.11
512	512	250.52	
1024	32	1597.93	1.50
1024	64	1592.96	1.50
1024	1024	2390.87	
2048	32	14657.47	1.61
2048	64	14601.83	1.61
2048	2048	23542.70	

Table 3. Running results with only one thread with different block sizes for 512, 1024, and 2048 points data on *Intel8a* machine.

# points	blockSize	avgTime(sec)	speedup
512	32	160.17	1.10
512	64	159.02	1.11
512	512	176.12	
1024	32	1121.96	1.61
1024	64	1111.27	1.62
1024	1024	1801.21	
2048	32	10300.82	1.71
2048	64	10249.28	1.72
2048	2048	17632.51	

Table 4. Running results with only one thread with different block sizes for 512, 1024, and 2048 points data on *Intel8b* machine.

5.2. Performance Analysis with respect to the Number of Threads and Data Sizes

We also investigated the relation between performance gains and the data sizes. Based on the result of the previous section, block size $b = 32, 64$ cases are only examined and we compute the speedup of 8 threads running on the two 8-core test machines over 1 thread running with the same block size with respect to five different Gaussian distribution data, such as 128, 256, 512, 1024, and 2048 points data. For the 128 points data set we measure only $b = 32$ case, since the number of blocks will be only 4 if $b = 64$ so only 4 threads will do actual submatrix multiplication. Figure 4 illustrates the speedup of the Parallel SMACOF with respect to different data sizes. As Figure 4 depicted, the speedup ratio increases as the data size increases. For the two small data set, i.e. 128 and 256 points data, the overhead ratio would be relatively high due to the running time is quite short. However, for the 512 points and bigger data set, the speedup ratio is more than 7.0. Note that the speedup ratio for the 2048 points data is around 7.7 and the parallel efficiency is around 0.96 for both block sizes on both **Intel8a** and **Intel8b** machines. Those values represent that our parallel SMACOF implementation works very well in parallel, since the it shows significant performance gains but negligible overhead.

In addition to experiments on data sizes, the performance gain with respect to the number of threads is also experimented. Again, we test the cases, where block size $b = 32, 64$ with 1024 points data. Figure 5 illustrates the speedup of the Parallel SMACOF with respect to the number of threads on **Intel8a** and **Intel8b** machines. As we expected, the speedup increase almost linearly until the number of threads becomes 8, which is the number of cores.

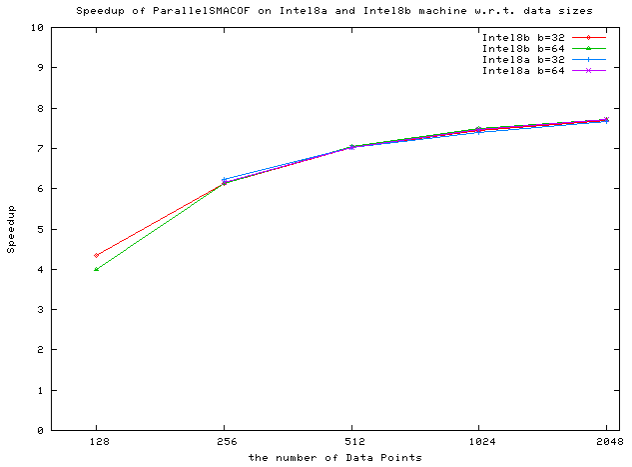

Figure 4. Speedup by 8 threads on 8 core machines with respect to different Data sizes

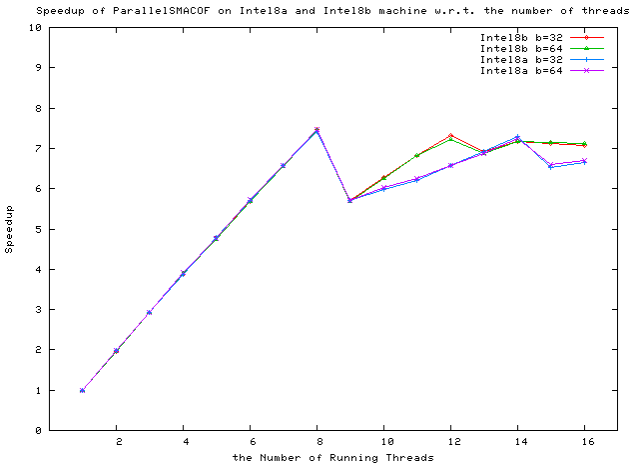

Figure 5. Speedup on 8 core machines with respect to the number of threads with 1024 points data

Then, when the number of threads is 9, the speedup factor suddenly decreases on both machines. That might be dependent on context switch and thread scheduling algorithms of the operating system. When the number of threads is above 9, the performance gain increases again, but it is less than when the number of threads equals to the number of cores on the machine. Note that the the speedup curves in the Figure 5 are different for more than 9 threads on the machines **Intel8a** and **Intel8b**. That must be effected by the operating systems on those machines (refer to Table 2). Based on Figure 5, Windows Vista system shows better thread management than Windows XP for those tests.

5.3. Performance of two-dimensional Array vs. Jagged Array (C# Language Specific Issue)

It is known that jagged array (array of arrays) performs better than two-dimensional (multidimensional) array in C# language [15, 19]. The author also wants to measure performance efficiency of jagged array over two-dimensional array on the proposed parallel SMACOF algorithm. The Figure 6 describes the efficiency value with respect to different size of data set. The left figure of Figure 6 is the runtime plottings of both jagged and two-dimensional array on both test machines with block size $b = 64$ and the right figure demonstrates the efficiency of jagged array over two-dimensional array on both **Intel8a** and **Intel8b**. Note that the left figure of Figure 6 use log scale on x- and y-axes. As shown in the right figure of Figure 6, the efficiency is about 1.5 on **Intel8b** and 1.4 on **Intel8a** for all the test data set, 128, 256, 512, ..., 2048 points data. In other words, jagged array data structure is more than 40% faster than

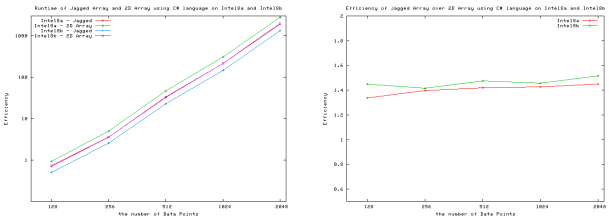

Figure 6. Performance comparison of Jagged and 2-Dimensional array in C# language on *Intel8a* and *Intel8b*.

two-dimensional array structure in C# language.

6. Conclusions & Future Works

In this paper, a machine-wide multicore parallelism is designed for the SMACOF algorithm, which is used to find a solution for MDS problem, and a several different performance analyses have been done. Since the SMACOF algorithm highly depends on matrix multiplication operation, the parallel matrix multiplication approach is used to implement parallel SMACOF. For load balance issue of the parallel SMACOF, we suggested a quite nice block decomposition algorithm. The algorithm works well if the the number of blocks in a row is more than a half of the number of running threads. The experimental results show that quite high efficiency and speed up is achieved by the proposed parallel SMACOF, about 0.95% and 7.5 over 8 cores, for larger test data set with 8 threads running, and the efficiency is increased as the data size increased. Also, we

tested the cache effect of the performance with the different block sizes, and the block size $b = 64$ is most fitted on both tested 8-core machines for the proposed parallel SMACOF application. In addition, the performance comparison between jagged array and two-dimensional array in C# is carried out. Jagged array shows about 1.5 times efficiency than two-dimensional array in our experiments.

Investigation over other programming languages and OS, for instance C/C++ under Linux and Windows systems, and over the multicore clusters or clouding computing systems would be interesting research issues. Based on high efficiency of the proposed parallel SMACOF on a single multicore machine, it would be highly intertesting to develop reasonably high efficient multicore cluster level (or even cloud computing system level) parallel SMACOF implementation.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Berkeley, California, Dec 2006. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[2] C. M. Bishop, M. Svensén, and C. K. I. Williams. GTM: The generative topographic mapping. *Neural Computation*, 10(1):215–234, 1998. `http://citeseer.ist.psu.edu/bishop98gtm.html`.

[3] I. Borg and P. J. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, New York, NY, U.S.A., 2005.

[4] G. Chrysanthakopoulos and S. Singh. An asynchronous messaging library for c#. In *Proceedings of Workshop on Synchronization and Concurrency in Object-Oriented Language (SCOOL), OOPSLA*, San Diego, CA, U.S.A., 2005.

[5] J. de Leeuw. Applications of convex analysis to multidimensional scaling. *Recent Developments in Statistics*, pages 133–145, 1977.

[6] J. de Leeuw. Convergence of the majorization method for multidimensional scaling. *Journal of Classification*, 5(2):163–180, 1988.

[7] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy. The impact of multicore on computational science software. *CTWatch Quarterly*, 3(1), Feb 2007. `http://www.ctwatch.org/quarterly/archives/february-2007`.

[8] P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology@Intel Magazine*, Feb 2005.

[9] A. J. Hanson, K. I. Ishkov, and J. H. Ma. Meshview: Visualizing the fourth dimension, 1999. `http://www.cs.indiana.edu/~hanson/papers/meshview.pdf`.

[10] W. J. Heiser and J. de Leeuw. Smacof-1. Technical Report UG-86-02, Department of Data Theory, University of Leiden, Leiden, The Nethelands, 1986.

[11] T. Kohonen. *Self-Organizing Maps*. Springer-Verlag, Berlin, Germany, 2001.

[12] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[13] J. B. Kruskal and M. Wish. *Multidimensional Scaling*. Sage Publications Inc., Beverly Hills, CA, U.S.A., 1978.

[14] J. Richter. Concurrent affairs: Concurrency and coordination runtime. *MSDN Magazine*, Sep. 2006. `http://msdn.microsoft.com/en-us/magazine/cc163556.aspx`.

[15] D. Solis. *Illustrated C# 2005*. Apress, Berkely, CA, U.S.A., 2006.

[16] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), Mar 2005.

[17] M. Svensén. *GTM: The Generative Topographic Mapping*. PhD thesis, Neural Computing Research Group, Aston University, Birmingham, U.K., 1998.

[18] Y. Takane, F. W. Young, and J. de Leeuw. Nonmetric individual differences multidimensional scaling: an alternating least squares method with optimal scaling features. *Psychometrika*, 42(1):7–67, 1977.

[19] W. Vogels. Hpc.net - are cli-based virtual machines suitable for high performance computing? In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 36, Washington, DC, U.S.A., 2003. IEEE Computer Society.