

Parallel Multidimensional Scaling Performance on Multicore Systems

Seung-Hee Bae
sebae@cs.indiana.edu

Community Grids Laboratory
Indiana University, Bloomington

Abstract

Multidimensional scaling constructs a configuration points into the target low-dimensional space, while the interpoint distances are approximated to the corresponding known dissimilarity values as much as possible. SMACOF algorithm is an elegant 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. The proposed block decomposition algorithm works very well if the number of block columns is at least a half of the number of threads. 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 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

Since multicore architectures were invented, multicore architectures have been getting important in software development and effecting on client, server and supercomputing systems [2, 8, 10, 22]. As [22] mentioned, the parallelism has become a critical issue to develop softwares for the purpose of getting maximum performance gains of multicore machines. Intel proposed that the Recognition, Mining, and Synthesis (RMS) [10] approach as a killer application for the next data explosion era. Machine learning and data mining algorithms are suggested as important algorithms for the data deluge era by [10]. Those algorithms are, however, usually highly compute-intensive algorithms, so the running time increases in quadratic or even more as the data

size increases. As [10] described, the amount of data is 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 increasing, data visualization would be another interesting area since it is hard to imagine data distribution of the most high-dimensional scientific data. Dimension reduction algorithms are used to reduce dimensionality of high dimensional data into Euclidean low dimensional space, so that dimension reduction algorithms are used as visualization tools. Some dimension reduction approaches, such as generative topographic mapping (GTM) [23] and Self-Organizing Map (SOM) [16], seek to preserve topological properties of given data rather than proximity information, but multidimensional scaling (MDS) [18, 3] works on maintaining dissimilarity information between mapping points as much as possible. The MDS uses several matrices which are full and $n \times n$ matrices for the n given data points. Thus, the matrices could be very large for large problems (n could be as big as million even today). For large problems, we will initially cluster the given data and use the cluster centers to reduce the problem size. In this paper, the author uses parallel matrix multiplication to parallelize an elegant algorithm for computing MDS solution, named SMACOF (Scaling by MAjorizing a COmplicated Function) [6, 7, 14], 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 data size, and a C# language specific issue. Finally, the conclusion and the future works are in Section 6.

2 Multidimensional Scaling (MDS)

Multidimensional scaling (MDS) [18, 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 $n \times n$ dissimilarity matrix $\Delta = (\delta_{ij})$ should be satisfied symmetric ($\delta_{ij} = \delta_{ji}$), nonnegative ($\delta_{ij} \geq 0$), and zero diagonal elements ($\delta_{ii} = 0$) conditions. 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 x_i in Euclidean p -dimensional space. From configuration matrix X , it is easy to compute the Euclidean interpoint distance $d_{ij}(X) = \|x_i - x_j\|$ 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. STRESS [17] and SSTRESS [24] were suggested as objective functions of MDS algorithms. STRESS (σ or $\sigma(X)$) criterion (Eq. (1)) [17] is an weighted squared error between distance of configured points and corresponding dissimilarity, but SSTRESS (σ^2 or $\sigma^2(X)$) criterion (Eq. (2)) [24] is an weighted squared error between squared distance of configured points and corresponding squared dissimilarity.

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

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

where w_{ij} is a weight value, so $w_{ij} \geq 0$.

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

2.1 SMACOF & its Time Complexity

Scaling by MAjorizing a COmplicated Function (SMACOF) [6, 7, 14] is an iterative majorization algorithm in order to minimize STRESS criterion. SMACOF is a variant of gradient 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]).

Algorithm 1 SMACOF algorithm

```

 $Z \leftarrow X^{[0]}$ ;
 $k \leftarrow 0$ ;
 $\varepsilon \leftarrow$  small positive number;
 $MAX \leftarrow$  maximum iteration;
Compute  $\sigma^{[0]} = \sigma(X^{[0]})$ ;
while  $k = 0$  or  $(\Delta\sigma(X^{[k]})) > \varepsilon$  and  $k \leq MAX$  do
     $k \leftarrow k + 1$ ;
     $X^{[k]} = V^\dagger B(X^{[k-1]})X^{[k-1]}$ 
    Compute  $\sigma^{[k]} = \sigma(X^{[k]})$ 
     $Z \leftarrow X^{[k]}$ ;
end while
return  $Z$ ;

```

points	128	256	512	1024	2048
time (sec)	1.4	16.2	176.1	1801.2	17632.5

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

The Algorithm 1 illustrates the SMACOF algorithm for MDS solution.

As shown in Algorithm 1, SMACOF algorithm consists of iterative matrix multiplication, where V^\dagger and $B(X^{[k-1]})$ are $n \times n$ matrices, and $X^{[k-1]}$ is $n \times p$ matrix. Since the inside of the while loop is the major computing 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 target dimension p is typically two or three, the author may have the assumption that $p \ll n$, without loss of generality. Thus, the time complexity of SMACOF method might be considered as $O(k \cdot n^3)$. Since the running time of SMACOF method is proportional to n^3 , the running time is highly dependent on the number of points (n).

Table 1 shows the average of the 5 running times of a naive (non-parallel, non-block based) SMACOF program in C# on **Intel8b** machine (refer to Table 2 for the detail information of the machine) for mapping four-dimensional Gaussian distribution data set into 3D space. It takes more than 4 hour 50 minutes (17632.5 seconds) for just 2048 points Gaussian distribution data. You should note that even though **Intel8b** machine has two 4-core processors which has actually 8 cores, the naive SMACOF implementation uses only one of the 8 cores. It took, however, only 22 minutes 4 seconds (1324.0 seconds) for running the proposed parallel SMACOF with 8 threads on the same machine with the same data. It is 13.3 times faster than the naive SMACOF for the 2048 points data. The speedup comes not only

from parallelism of the computation but also from effective cache memory utilization. As in multicore era, parallel approach is essential to software implementation 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 [4, 1, 9, 25, 12] 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 multiplications, which would be more fitted into cache line than the whole matrix multiplication. Fitting into cache line reduces unnecessary cache I/O overhead. The cache memory issue is significant in performance, as you see in Section 5.1.

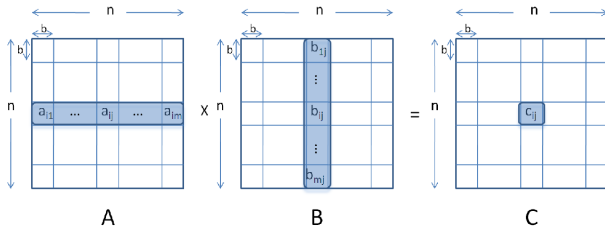


Figure 1. Parallel Matrix multiplication.

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 i th block-row of matrix A with j th block-column of matrix B , correspondingly, as in Eq. (3):

$$c_{ij} = \sum_{k=1}^m a_{ik} \cdot b_{kj} \quad (3)$$

Algorithm 2 First mapping block index of each thread

```

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

```

Algorithm 3 Mapping blocks to a thread

```

while col ≥ the number of the block columns do
  row ← row + 1;
  if row ≥ the number of the block rows then
    return false
  end if
  end if
  col ← (row + id + row · (TH - 1)/2)%TH;
end while

return true

```

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 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 ($1 \leq i, j \leq m$, $c_{ij} \subset C$) is 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 (c_{ij} 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 to each thread should be as equal as possible, which means either $\lceil m^2/TH \rceil$ or $\lceil m^2/TH \rceil - 1$ blocks assigned to all threads. Algorithm 3 is used for block assignment to each thread in consideration of load balance in our implementation, and it works very well if the number of block columns is at least a half of the number of running threads. 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 block multiplication is done, the thread will execute Algorithm 3 again with assigning col value to $col + TH$. Each thread keeps those iteration, to do matrix multiplication and find the next working block by Algorithm 3, until the thread gets *false* from calling Algorithm 3. Finally, every assigned blocks to each thread is finished for the full matrix multiplication.

After finishing assigned computation, each thread sends a signal to the main thread which is waiting on the Rendezvous (actually 'waitingAll' method in CCR library is

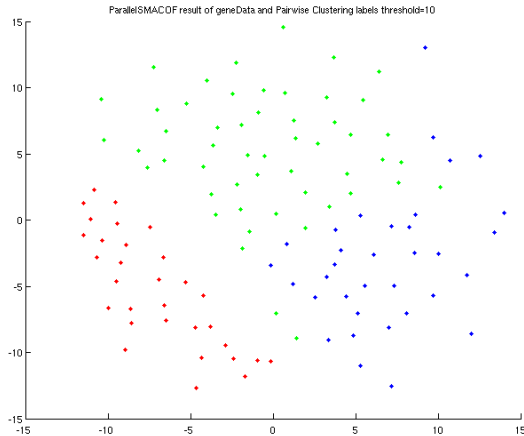


Figure 2. MDS result by SMACOF algorithm of 117 ALU sequence data with labeled by DA Pairwise Clustering.

called). Finally, the main thread will wrap up 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) [5, 19] is used for this application.

After k^{th} iteration of the matrix multiplication is completed, the parallel SMACOF application calculates STRESS value ($\sigma^{[k]}$) of the current solution ($X^{[k]}$), and measures $\Delta\sigma^{[k]} (= \sigma^{[k-1]} - \sigma^{[k]})$. If $\Delta\sigma^{[k]} < \varepsilon$, 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 above procedure again to find $X^{[k+1]}$ using $X^{[k]}$.

Figure 2 is an example of MDS results by SMACOF algorithm of 117 ALU sequence data. The sequences are clustered in three clusters by deterministic annealing pairwise clustering method [15]. Three clusters are represented in red, blue, and green in the Figure 2. Though we use only 117 ALU sequence data for the gene visualization problem now, we need to apply MDS to very large system (e.g. one million ALU sequences) so we should consider parallelism for the large systems.

4 Experimental Settings

4.1 Experimental Platforms

For the performance experiments on multicore machines of the parallel SMACOF, two multicore machines depicted in Table 2 are used. Both **intel8a** and **intel8b** have two

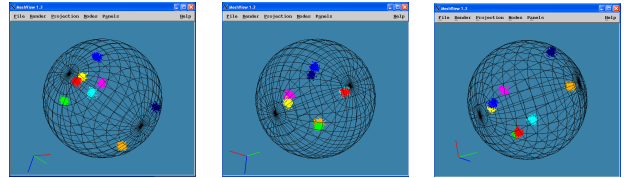


Figure 3. The example of SMACOF results with 8-centered Gaussian distribution data with 2000 points shown with Meshview in 3D.

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 [5, 19].

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

Table 2. Test machine specification

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.

In fact, the real scientific data would be much more complex than those simple Gaussian distribution data in this paper. However, nobody can verify that the mapping results of those high dimensional data is correct or not, since it is hard to imagine original high dimensional space. Figure 3 shows three different screen captures of the parallel SMACOF results of the given 2000 points data in a 3D image viewer, called Meshview [13]. Combining three images in Figure 3 shows that the expected mapping is achieved by the parallel SMACOF program.

4.3 Experimental Designs

Several different aspects of performance issues 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 affects on the cache memory performance, since computers fetch data into cache by a cache line amount of data, whenever it needs to fetch data, without regard to the actual necessary 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 and data points:** 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 than 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. This experiment setup will investigate the thread performance overhead and how many threads would be used to run the parallel SMACOF on the test machines. The bigger data uses more computation time. The author examines how speedup and overhead change as the data size differs. The tested number of data points are from 128 to 2048 as increased by factor of 2.
- Two-dimensional array vs. jagged array:** It is known as jagged array (array of arrays) shows better performance than two-dimensional array in C# [21][26]. 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 to run several times to increase the probability to get better solution. If the initial mapping is different, however, the computation amount can be varied whenever the application runs, so that we could not measure any performance comparison between two experimental setups, since it could 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 for each data.

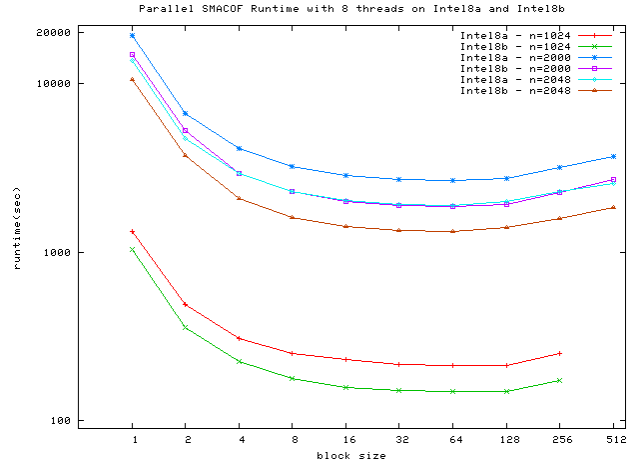


Figure 4. Parallel SMACOF running time with respect to block sizes with 8 threads.

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 by the factor of 2 from one to 256 for 1024 points data, in order to keep the number of blocks (m^2) more than the number of threads (TH), or to 512 for 2000 and 2048 points data, as shown in Figure 4. 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, block size 64 could be more fitted than other block sizes for the **Intel8a** and **Intel8b** in Table 2. 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 performance criteria but the block size. Table 3 describes the running time with only one thread with 512, 1024, and 2048 points data. Based on the 8-thread results, we chose block sizes $b = 32, 64$ to test cache effect and measure the speedup of selected block sizes based on the result of using

one big whole $n \times n$ matrix. 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 block decomposition, finding correct starting positions of the current block, and the iteration of submatrix multiplication.

# points	blockSize	avgTime(sec)	speedup
Intel8a			
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	
Intel8b			
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 3. Running results with only one thread with different block sizes for 512, 1024, and 2048 points data on Intel8a and Intel8b.

5.2 Performance Analysis of the Number of Threads and Data Sizes

We also investigated the relation between the performance gains and the data sizes. Based on the result of the Section 5.1, 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, ..., and 2048 points data. For the 128 points data set we measure only $b = 32$ case, since the number of blocks is only 4 if $b = 64$ so only 4 threads does actual submatrix multiplication. Figure 5 illustrates the speedup of the Parallel SMACOF with respect

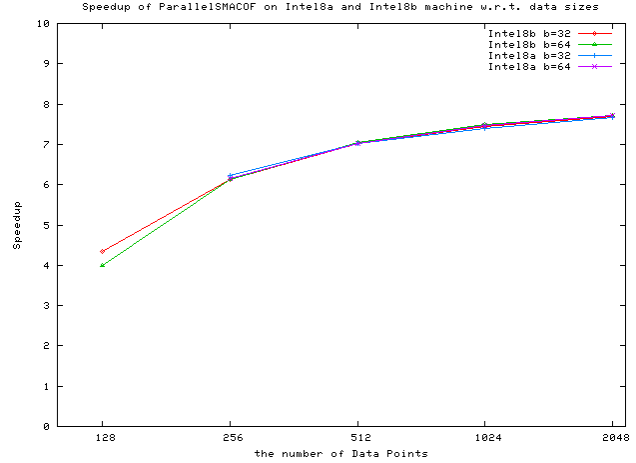


Figure 5. Speedup by 8 threads on 8 core machines with respect to different Data sizes

to different data sizes. As Figure 5 depicted, the speedup ratio increases as the data size increases. Following equations are the definition of overhead (f), efficiency (e), and speedup($S(N)$):

$$f = \frac{NT(N) - T(1)}{T(1)} \quad (4)$$

$$e = \frac{1}{1 + f} = \frac{S(N)}{N} \quad (5)$$

where N is the number of threads or processes, $T(N)$ and $T(1)$ are the running time with N threads and one thread, and $S(N)$ is the speedup of N threads. For the two small data set, i.e. 128 and 256 points data, the overhead ratio would be relatively high due to the short running time. 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 overhead is around 0.03 for both block sizes on both **Intel8a** and **Intel8b** machines. Those values represent that our parallel SMACOF implementation works very well, since it shows significant speedup but negligible overhead.

In addition to the experiments on data size, the performance gain with respect to the number of threads (TH) is also experimented with 1024 points data. Figure 6 illustrates the speedup of the Parallel SMACOF with respect to TH on **Intel8a** and **Intel8b** machines. As we expected, the speedup increases almost linearly during $TH \leq 8$, which is the number of cores. Then, when $TH = 9$, 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

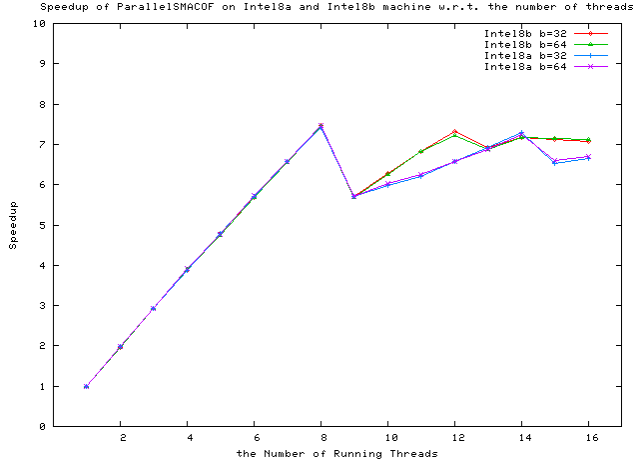


Figure 6. Speedup on 8 core machines with respect to the number of threads with 1024 points data

is above 9, the performance gain increases again, but it is always less than when the number of threads is equal to the number of cores on the machine. Note that the speedup curves in Figure 6 are different for more than 9 threads on the machines **Intel8a** and **Intel8b**. That must be affected by thread management algorithm of the operating systems on those machines (refer to Table 2).

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 [21, 26]. The author also wants to measure performance efficiency of jagged array over two-dimensional array on the proposed parallel SMACOF algorithm. The Figure 7 demonstrates the efficiency of jagged array over two-dimensional array on both **Intel8a** and **Intel8b**.

As shown in Figure 7, 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 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 an elegant algorithm to find a solution for MDS problem. Since the SMACOF algorithm highly depends on matrix multiplication operation, the parallel matrix multiplication approach

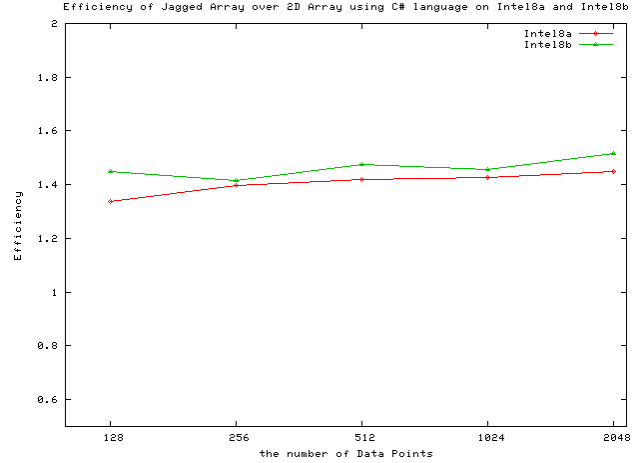


Figure 7. Performance comparison of Jagged and 2-Dimensional array in C# language on Intel8a and Intel8b with $b = 64, TH = 8$.

is used to implement parallel SMACOF. For the load balance issue of the parallel SMACOF, we suggested a quite nice block decomposition algorithm. The algorithm works well if the number of blocks in a row is more than a half of the number of running threads.

Furthermore, several different performance analyses have been done. The experimental results show that quite high efficiency and speed up is achieved by the proposed parallel SMACOF, about 0.95% efficiency and 7.5 speedup 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 the 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 at least 1.4 times faster than two-dimensional array in our experiments.

Our group have already tested deterministic annealing clustering [20] algorithm using MPI and CCR under 8 node dual-core Windows cluster and it shows scaled speedup [11], and we will improve the proposed parallel SMACOF to run on cluster of multicore. Based on high efficiency of the proposed parallel SMACOF on a single multicore machine, it would be highly interesting to develop reasonably high efficient multicore cluster level parallel SMACOF.

Acknowledgements

The author give thanks to Prof. Geoffrey C. Fox and Dr. Xiaohong Qiu at Community Grids Lab for helpful discus-

sion and advice for this paper.

References

- [1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.
- [2] 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, California, Dec 2006. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.
- [3] I. Borg and P. J. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, New York, NY, U.S.A., 2005.
- [4] J. Choi, D. W. Walker, and J. J. Dongarra. Pumma:parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency:Practice and Experience*, 6(7):543–570, Oct. 1994.
- [5] 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.
- [6] J. de Leeuw. Applications of convex analysis to multidimensional scaling. *Recent Developments in Statistics*, pages 133–145, 1977.
- [7] J. de Leeuw. Convergence of the majorization method for multidimensional scaling. *Journal of Classification*, 5(2):163–180, 1988.
- [8] 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>.
- [9] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, Jun. 1995.
- [10] P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology@Intel Magazine*, 2005.
- [11] G. C. Fox. Parallel data mining from multicore to cloudy grids. In *International Advanced Research Workshop on High Performance Computing and Grids, HPC2008*, Cetraro, Italy, Jul. 2008.
- [12] J. Gunnels, C. Lin, G. Morrow, and R. Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.
- [13] 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>.
- [14] W. J. Heiser and J. de Leeuw. Smacof-1. Technical Report UG-86-02, Department of Data Theory, University of Leiden, Leiden, The Netherlands, 1986.
- [15] T. Hofmann and J. M. Buhmann. Pairwise data clustering by deterministic annealing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(1):1–14, 1997.
- [16] T. Kohonen. *Self-Organizing Maps*. Springer-Verlag, Berlin, Germany, 2001.
- [17] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.
- [18] J. B. Kruskal and M. Wish. *Multidimensional Scaling*. Sage Publications Inc., Beverly Hills, CA, U.S.A., 1978.
- [19] J. Richter. Concurrent affairs: Concurrency and coordination runtime. *MSDN Magazine*, Sep. 2006. <http://msdn.microsoft.com/en-us/magazine/cc163556.aspx>.
- [20] K. Rose. Deterministic annealing for clustering, compression, classification, regression, and related optimization problems. *Proceedings of the IEEE*, 86(11):2210–2239, 1998.
- [21] D. Solis. *Illustrated C# 2005*. Apress, Berkely, CA, 2006.
- [22] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3), 2005.
- [23] M. Svensén. *GTM: The Generative Topographic Mapping*. PhD thesis, Neural Computing Research Group, Aston University, Birmingham, U.K., 1998.
- [24] 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.
- [25] R. A. van de Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency:Practice and Experience*, 9(4):255–274, Apr. 1997.
- [26] 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.