`Parallel Data Mining from Multicore to Cloudy Grids

Geoffrey Foxa,b,[[1]](#footnote-2) Seung-Hee Baeb, Jaliya Ekanayakeb, Xiaohong Qiuc, and Huapeng Yuanb

a Informatics Department, Indiana University 919 E. 10th Street Bloomington, IN 47408 USA

b Computer Science Department and Community Grids Laboratory, Indiana University 501 N. Morton St., Suite 224, Bloomington IN 47404 USA

c UITS Research Technologies, Indiana University, 501 N. Morton St., Suite 211, Bloomington, IN 47404

**Abstract.** We describe a suite of data mining tools that cover clustering, information retrieval and the mapping of high dimensional data to low dimensions for visualization. Preliminary applications are given to particle physics, bioinformatics and medical informatics. The data vary in dimension from low (2-20), high (thousands) to undefined (sequences with dissimilarities but not vectors defined). We use deterministic annealing to provide more robust algorithms that are relatively insensitive to local minima. We discuss the algorithm structure and their mapping to parallel architectures of different types and look at the performance of the algorithms on three classes of system; multicore, cluster and Grid using a MapReduce style algorithm. Each approach is suitable in different application scenarios. We stress that data analysis/mining of large datasets can be a supercomputer application

**Keywords.** MPI, MapReduce, CCR, Performance, Clustering, Multidimensional Scaling

Introduction

Computation and data intensive scientific data analyses are increasingly prevalent. In the near future, data volumes processed by many applications will routinely cross the peta-scale threshold, which would in turn increase the computational requirements. Efficient parallel/concurrent algorithms and implementation techniques are the key to meeting the scalability and performance requirements entailed in such scientific data analyses Most of these analyses can be thought of as a Single Program Multiple Data (SPMD)[1] algorithms or a collection thereof. These SPMDs can be implemented using different parallelization techniques such as threads, MPI [2], MapReduce [3], and mash-up or workflow technologies yielding different performance and usability characteristics. In some fields like particle physics, parallel data analysis is already commonplace and indeed essential. In others such as biology, data volumes are still such that much of the work can be performed on sequential machines linked together by workflow systems such as Taverna. The parallelism currently exploited is the “almost embarrassingly parallel” style given by the independent events in particle physics or the independent documents of information retrieval – these independent maps are followed by a reduction to give histograms in particle physics or aggregated queries in web searches. The excellent quality of service (QoS) and ease of programming provided by the MapReduce programming model has gained itself a lot of traction for this type of problem. However, the architectural and performance limitations of the current MapReduce architectures make their use questionable for many applications. These include many of the machine learning algorithms such as those discussed in this paper which need iterative closely coupled computations. In section 2 we compare various versions of this data intensive programming model with other implementations for both closely and loosely coupled problems. However the more general workflow or dataflow paradigm (which is seen in Dryad and MapReduce extensions) is always valuable. In sections 3 and 4 we turn to some data mining algorithms that surely need parallel implementations for large data sets; interesting both sections see algorithms that scale like N2 (N is dataset size) and use full matrix algebra.

**Table 1.** Hardware and software configurations of the clusters used for testing.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ref** | **Cluster Name** | **# Nodes** | **CPU** | **L2 Cache**  **Memory** | **Operating System** |
| A | Barcelona | 1 | 1 AMD Quad Core | 2x1MB | Windows Server |
|  | (4 core |  | Opteron 2356 | 8 GB | HPC Edition |
|  | Head Node) |  | 2.3GHz |  | (Service Pack 1) |
| B | Barcelona | 4 | 2 AMD Quad Core | 4×512K | Windows Server 2003 |
|  | (8 core |  | Opteron 2356 | 16GB | Enterprise x64 bit |
|  | Compute Node) |  | 2.3 GHz |  | Edition |
| C | Barcelona | 2 | 4 AMD Quad Core | 4×512K | Windows Server |
|  | (16 core |  | Opteron 8356 | 16 GB | HPC Edition |
|  | Compute Node) |  | 2.3GHz |  | (Service Pack 1) |
| D | Barcelona | 1 | 4 Intel Six Core | 12 M | Windows Server |
|  | (24 core |  | Xeon E7450 | 48GB | HPC Edition |
|  | Compute Node) |  | 2.4GHz |  | (Service Pack 1) |
| E | Madrid | 1 | 1 AMD Quad Core | 2x1MB | Windows Server |
|  | (4 core  Head Node) |  | Opteron 2356 | 8 GB | HPC Edition |
|  |  |  | 2.3GHz |  | (Service Pack 1) |
| F | Madrid | 1 | 4 AMD Quad Core | 4x512K | Windows Server |
|  | (16 core |  | Opteron 8356 | 16 GB | HPC Edition |
|  | Compute Node) |  | 2.3GHz |  | (Service Pack 1) |
| G | Gridfarm | 8 | 2 Quad core Intel | 4x1MB | Red Hat Enterprise |
|  | 8 core |  | Xeon E5345 | 8GB | Linux 4 |
|  |  |  | 2.3GHz |  |  |
| H | IU Quarry | 112 | 2 Quad-core Intel | 4x4MB, | Red Hat Enterprise |
|  | 8 core |  | Xeon 5335 | 8 GB | Linux 4 |
|  |  |  | 2.00GHz |  |  |

Our algorithms are parallel MDS (Multi dimensional scaling) [6] and clustering. The latter has been discussed earlier by us [7-11] but here we extend our results to larger systems – single workstations with 16 and 24 cores and a 128 core (8 nodes with 16 cores each) cluster described in table 1. Further we study a significantly different clustering approach that only uses pairwise distances (dissimilarities between points) and so can be applied to cases where vectors are not easily available. This is common in biology where sequences can have mutual distances determined by BLAST like algorithms but will often not have a vector representation. Our MDS algorithm also only uses pairwise distances and so it and the new clustering method can be applied broadly. Both our original vector-based (VECDA) and the new pairwise distance (PWDA) clustering algorithms use deterministic annealing to obtain robust results. VECDA was introduced by Rose and Fox almost 30 years ago [12] and has a good reputation [13] and there is no clearly better clustering approach. The pairwise extension PWDA was developed by Hofmann and Buhmann [14] around 10 years ago but does not seem to have used in spite of its attractive features – robustness and applicability to data without vector representation.

As seen in table 1, we use both Linux and Windows platforms in our multicore and our work uses a mix of C#, C++ and Java. Our results study three variants of MapReduce, threads and MPI. The algorithms are applied across a mix of paradigms to indicate the performance characteristics.

# Choices in Messaging Runtime

Although high level languages – especially for parallel programming – have been a holy grail for computer science research, there has been more progress in the area of runtime environments and this is our focus in this paper.

To be more precise there are successful workflow languages but in parallel programming where the precise constraints of correct parallel semantics have thwarted research so far. This observation however underlies our approach which is to use workflow technologies – defined as orchestration languages for distributed computing for the coarse grain functional components of parallel computing with dedicated low level direct parallelism of kernels. At the run time level, there is much similarity between parallel and distributed run times with both supporting messaging with different properties. Some of the choices are shown in figure 1 and differ by both hardware and software models. The hardware support of parallelism/concurrency varies from shared memory multicore, closely coupled (e.g. Infiniband connected) clusters, and the higher latency and possibly lower bandwidth distributed systems. The coordination (communication/ synchronization) of the different execution units vary from threads (with shared memory on cores); MPI between cores or nodes of a cluster; workflow or mash-ups linking services together; the new generation of data intensive programming systems typified by Hadoop (implementing MapReduce) and Dryad. These can be considered as the workflow systems of the information retrieval industry but are of general interest as they support parallel analysis of large datasets. As illustrated in the figure the execution units vary from threads to processes and can be short running or long lived.

**Figure 1(a).** First three of seven different combinations of processes/threads and intercommunication mechanisms discussed in the text

The former gives greater flexibility and fault tolerance but has significantly higher overheads. Short running threads can be spawned up in the context of persistent data in memory and so have modest overhead seen in section 4. Short running processes in the spirit of stateless services are seen in Dryad and Hadoop and can have substantially higher overhead than long running processes which are coordinated by rendezvous messaging as later do not need to communicate large amounts of data – just the smaller change information needed. The importance of this is emphasized in figure 2 showing data intensive passing through multiple “map” (each map is for example a particular data analysis or filtering operation) and “reduce” operations that gather together the results of different map instances corresponding typically to a data parallel break up of an algorithm. The figure notes two important patterns

**Figure 2:** Data Intensive Iteration and Workflow



**Figure 1(b).** Seven different combinations of processes/threads and intercommunication mechanisms discussed in the text

a) Iteration where results of one stage are iterated. This is seen in the “Expectation Maximization” EM steps in the later sections where for clustering and MDS thousands of iterations are needed. This is typical of most MPI style algorithms.

b) Pipelining where results of one stage are forwarded to another; this is functional parallelism typical of workflow applications. In applications of this paper we implement a three stage pipeline:

Data (from disk) 🡪 Clustering 🡪 Dimension Reduction (MDS) 🡪 Visualization

Each of the first two stages is parallel and one can break up the compute and reduce modules of figure 2 into parallel components as shown in figure 3. There is an important ambiguity in parallel/distributed programming models/runtimes that both the parallel MPI style parallelism and the distributed Hadoop/ Dryad/ Web Service/Workflow models are coordinated are implemented by messaging. Thus the same software can in fact be used for all the decompositions seen in figures 1-3. Thread coordination can avoid messaging but even this is attractive as it avoids many of the error scenarios seen in shared memory communication. The CCR threading used in this paper is coordinated by reading and writing messages to ports. As an example, MPI has often been used in Grid (distributed) applications with MPICH-G popular here. Again the paper of Chu [] noted that the MapReduce approach can be used in many machine learning algorithms and one of our data mining algorithms VECDA only uses map and reduce operations (it does not need SEND or RECEIVE MPI operations). We will note in this paper that MPI gives excellent performance and ease of programming for MapReduce as it has elegant support for general reductions. It does not have the fault tolerance and flexibility of Hadoop or Dryad. Further MPI is designed for the “owner-computes” rule of SPMD – if a given datum is stored in a compute node’s memory, that node’s CPU computes (evolves or analyzes) it. Hadoop and Dryad combine this idea with the notion of “taking the computing to the data”. This leads to the generalized “owner stores and computes” rule or crudely that a file (disk or database) is assigned a compute node that analyzes (in parallel with nodes assigned different disks) the data on each file. Future scientific programming models must clearly capture this concept.

**Figure 3.** Workflow of Parallel Services



# Data Intensive Workflow Paradigms

In this section, we will present an architecture and a prototype implementation of a new programming model that can be applied to most composable class of applications with various program/data flow models, by combining the MapReduce and data streaming techniques and compare its performance with other parallel programming runtimes such as MPI, Hadoop[4] and Dryad[5].

MapReduce is a parallel programming technique derived from the functional programming concepts and proposed by Google for large-scale data processing in a distributed computing environment. The *map* and *reduce* programming constructs offered by MapReduce model is a limited subset of programming constructs provided by the classical distributed parallel programming models such as MPI. However, our current experimental results highlight that many problems can be implemented using MapReduce style by adopting slightly different parallel algorithms compared to the algorithms used in MPI, yet achieve similar performance for appropriately large problems. The main advantage of the MapReduce programming model is that the easiness in providing various quality of services. Google and Hadoop both provide MapReduce runtimes with fault tolerance and dynamic flexibility support.

Dryad is a distributed execution engine for coarse grain data parallel applications. It combines the MapReduce programming style with dataflow graphs to solve the computation tasks. Dryad considers computation tasks as directed acyclic graph (DAG)s where the vertices represent computation tasks –typically, sequential programs with no thread creation or locking, and the edges as communication channels over which the data flow from one vertex to another.

Moving computation to data is another advantage of the MapReduce and Dryad have over the other parallel programming runtimes. With the ever-increasing requirement of processing large volumes of data, we believe that this approach has a greater impact on the usability of the parallel programming runtimes in the future.

## Current MapReduce Implementations

Google's MapReduce implementation is coupled with a distributed file system named Google File System (GFS) [6] where it reads the data for MapReduce computations and stores the results. According to J. Dean et al., in their MapReduce implementation, the intermediate data are first written to the local files and then accessed by the reduce tasks. The same architecture is adopted by the Apache's MapReduce implementation – Hadoop.

Hadoop stores the intermediate results of the computations in local disks, where the computation tasks are executed, and informs the appropriate workers to retrieve (pull) them for further processing. The same approach is adopted by Disco [7] – another open source MapReduce runtime developed using a functional programming language named Erlang [8]. Although this strategy of writing intermediate result to the file system makes the above runtimes robust, it introduces an additional step and a considerable communication overhead, which could be a limiting factor for some MapReduce computations. Apart from the above, all these runtimes focus mainly on computations that utilize a single *map/reduce* computational unit. Iterative MapReduce computations are not well supported.

## CGL-MapReduce

CGL-MapReduce is a novel MapReduce runtime that uses streaming for all the communications, which eliminates the overheads associated with communicating via a file system. The use of streaming enables the CGL-MapReduce to send the intermediate results directly from its producers to its consumers. Currently, we have not integrated a distributed file system such as HDFS with CGL-MapReduce, and hence the data should be available in all computing nodes or in a typical distributed file system such as NFS. The fault tolerance support for the CGL-MapReduce will harness the reliable delivery mechanisms of the content dissemination network that we use. Figure 1 shows the main components of the CGL-MapReduce.

# D:\Academic\Ph.D\eScience2008\images\eps\architecture.eps

**Figure 1.** Components of the CGL-MapReduce.

CGL MapReduce runtime comprises a set of workers, which perform *map* and *reduce* tasks and a content dissemination network that handles all the underlying communications. As in other MapReduce runtimes, a master worker (*MRDriver*) controls the other workers according to instructions given by the user program. However, unlike typical MapReduce runtimes, CGL-MapReduce supports both single-step and iterative MapReduce computations.

D:\Academic\Ph.D\eScience2008\images\eps\program_flow.eps

**Figure 2.** Computation phases of CGL-MapReduce.

A MapReduce computation under CGL-MapReduce passes through several phases of computations as shown in figure 2. In CGL-MapReduce the initialization phase is used to configure both the *map*/*reduce* tasks and can be used to load any fixed data necessary for the *map/reduce*  tasks. The *map* and *reduce* stages perform the necessary data processing while the framework directly transfer the intermediate result from *map* task to the *reduce* tasks. The *merge* phase is another form of reduction which is used to collect the results of the *reduce* stage to a single value. The User Program has access to the results of the *merge* operation. In the case of iterative MapReduce computations, the user program can call for another iteration of MapReduce by looking at the result of the merge operation and the framework performs anther iteration of MapReduce using the already configured *map*/*reduce* tasks eliminating the necessity of configuring *map*/*reduce* tasks again and again as it is done in Hadoop.

CGL-MapReduce is implemented in Java and utilizes NaradaBrokering[9], a streaming-based content dissemination network. The CGL-MapReduce research prototype provides the runtime capabilities of executing MapReduce computations

written in the Java language. MapReduce tasks written in other programming languages require wrapper *map* and *reduce* tasks in order for them to be executed using CGL-MapReduce.

## Evaluations

To evaluate the different runtimes we have selected several data analysis applications. First, we applied the MapReduce technique to parallelize a High Energy Physics (HEP) data analysis application and implemented it using Hadoop and CGL-MapReduce. The HEP data analysis application process large volumes of data and perform a histogramming operation on a collection of event files produced by HEP experiments. Next, we applied the MapReduce technique to parallelize a Kmeans clustering [10] algorithm and implemented it using Hadoop and CGL-MapReduce. In addition, we implemented the same algorithm using MPI (C++) as well. We have also implemented a matrix multiplication algorithm using Hadoop and CGL-MapReduce. To compare the performance of Dryad with other parallel runtimes, we use two text-processing applications, which perform a “word histogramming” operation, and a “distributed grep” operation implemented using Dryad, Hadoop, and CGL-MapReduce. Table 1 and Table 2 highlight the details of the hardware and software configurations and the various test configurations that we used for our evaluations.

**Table 2.** Tests configurations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature** | **HEP Data Analysis** | **Kmeans clustering** | **Matrix Multiplication** | **Histogramming & Grep** |
| Cluster Ref | H | G | G | B |
| Number of Nodes | 12 | 5 | 5 | 4 |
| Amount of Data | Up to 1TB of HEP data | Up to 40 million data points | Up to 16000 rows and columns | 100GB of text data |
| Data Location | IU Data Capacitor: a high-speed and high-bandwidth storage system running the Lustre File System | Hadoop : HDFS  CGL-  MapReduce : NFS | Hadoop : HDFS  CGL-MapReduce : NFS | Hadoop : HDFS  CGL-MapReduce: Local Disc  Dryad :  Local Disc |
| Language | Java, C++ (ROOT) | Java, C++ | Java | Java, C# |

For the HEP data analysis, we measured the total execution time it takes to process the data under different implementations by increasing the amount of data. Figure 3 depicts our results.

D:\programs\gp422win32\gnuplot\bin\es_hep.eps

**Figure 3.** HEP data analysis, execution time vs. the volume of data (fixed compute resources).

Hadoop and CGL-MapReduce both show similar performance. The amount of data accessed in each analysis is extremely large and hence the performance is limited by the I/O bandwidth of a given node rather than the total processor cores. The overhead induced by the MapReduce implementations has negligible effect on the overall computation.

We evaluate the performance of different implementations for the Kmeans clustering application and calculated the overhead (φ) induced by the different parallel programming runtime using the formula given below. In this formula P denotes the number of hardware processing units used and T(P) denotes the total execution time of the program when P processing units are used. T(1) denotes the total execution time for a single threaded program. Figure 4 depicts our results.

*φ(P) = [PT(P) –T(1)] /T(1)* (1)

For the matrix multiplication program, we measured the total execution time by increasing the size of the matrices used for the multiplication, using both Hadoop and CGL-MapReduce implementations. The result of this evaluation is shown in figure 5.

D:\programs\gp422win32\gnuplot\bin\es_oh.eps

**Figure 4.** Overheads associated with Hadoop, CGL-MapReduce and MPI for iterative MapReduce (Both axes are in log scale).

D:\programs\gp422win32\gnuplot\bin\mat_mult.eps

**Figure 5.** Performance of the Hadoop and CGL-MapReduce for matrix multiplication.

The results in figure 4 and figure 5 show how the approach of configuring once and re-using of *map/reduce* tasks for multiple iterations and the use of streaming have improved the performance of CGL-MapReduce for iterative MapReduce tasks. The communication overhead and the loading of data multiple times have caused the Hadoop to induce large overhead to the computation making the results comparably larger than that of CGL-MapReduce.

We compare the above two MapReduce runtimes with Microsoft Dryad implementation using two text processing applications. We develop the Dryad applications using the DryadLINQ[11] programming environment. The results of these two evaluations are shown in figure 6 and figure 7.

D:\programs\gp422win32\gnuplot\bin\wordcount_4.eps

**Figure 6.** Performance of Dryad, Hadoop, and CGL-MapReduce for “histogramming of words” operation.

D:\programs\gp422win32\gnuplot\bin\grep_4.eps

**Figure 7.** Performance of Dryad, Hadoop, and CGL-MapReduce for “distributed grep” operation

In both these tests, Hadoop shows higher overall processing time compared to Dryad and CGL-MapReduce. This could be mainly due to its distributed file system and the file based communication mechanism. Dryad uses in memory data transfer for intra-node data transfers and a file based communication mechanism for inter-node data transfers where as in CGL-MapReduce all data transfer occur via streaming. The “word histogramming” operation had data transfer requirements compared to the “distributed grep” operation and hence the streaming data transfer approach adopted by the CGL-MapReduce shows lowest execution times for the “word histogramming” operation. In “distributed grep” operation both Dryad and CGL-MapReduce shows close performance results.

# Multidimensional Scaling

Multidimensional scaling (MDS) [3, 4] 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 × N* dissimilarity matrix *Δ = (δij)* should be satisfied symmetric (*δij* = *δji*), nonnegative (*δij* ≥ 0), and zero diagonal elements (*δii* = 0) conditions. From 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 × p* configuration matrix *X*, whose rows represent each data points *xi* in Euclidean *p*-dimensional space. From configuration matrix *X*, it is easy to compute the Euclidean interpoint distance *dij(X) = ||xi – xj||* among *N* configured points in the target space and to build the *N × N* Euclidean interpoint distance matrix *D(X) = (dij(X))*. The purpose of MDS algorithm is to construct a configuration points into the target *p*-dimensional space, while the interpoint distance *dij(X)* is approximated to *δij* as much as possible. STRESS [7] and SSTRESS [8] were suggested as objective functions of MDS algorithms. STRESS (σ or σ(X)) criterion (Eq. (1)) [7] is an weighted squared error between distance of configured points and corresponding dissimilarity, but SSTRESS (σ2 or σ2(X)) criterion (Eq. (2)) [8] is an weighted squared error between squared distance of configured points and corresponding squared dissimilarity.

*σ(X) = Σi<j≤n wij(dij(X) − δij)2* (1)

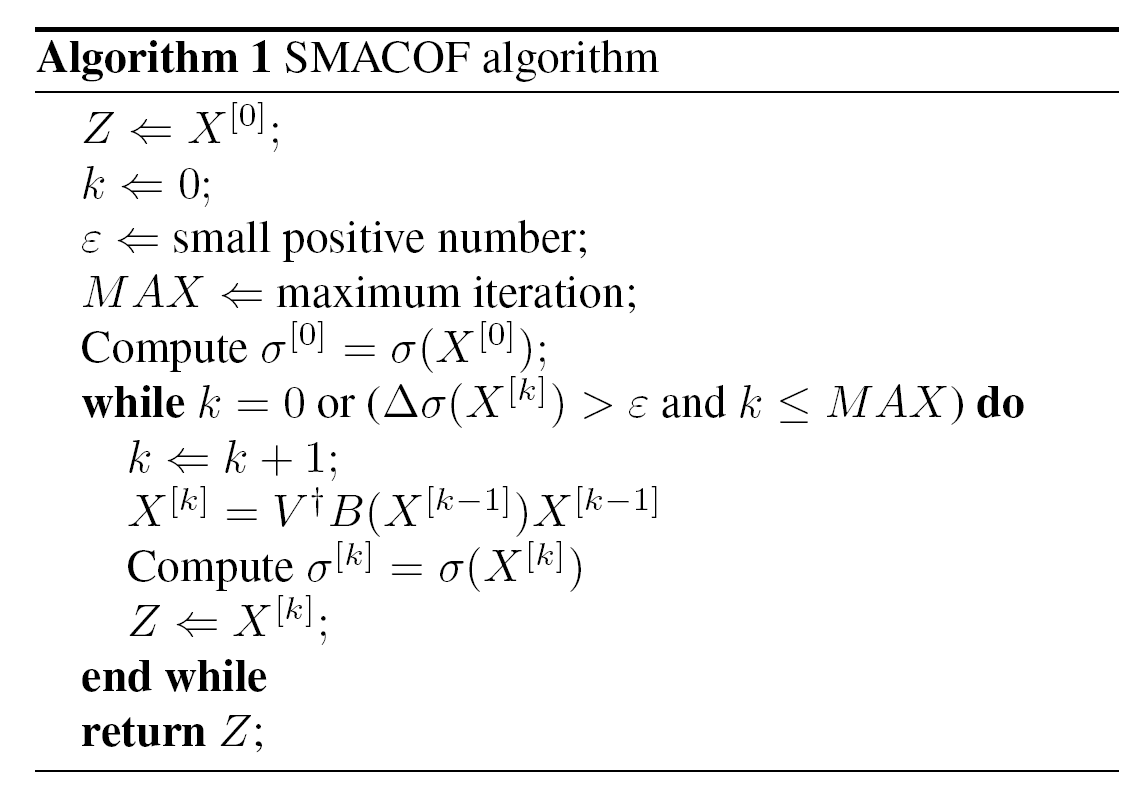
*σ2(X) = Σi<j≤n wij [(dij(X))2 − (δij)2]2* (2)

where *wij* is a weight value, so *wij* ≥ 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.

## Scaling by MAjorizing a COmplicated Function (SMACOF)

Scaling by MAjorizing a COmplicated Function (SMACOF) [5, 6] is an iterative majorization algorithm in order to minimize objective function of MDS. SMACOF is likely to find local minima due to gradient descent property. Nevertheless, it is powerful since it guarantees monotonic decreasing the objective function. The procedure of SMACOF is described in Algorithm 1. For the mathematical details of SMACOF, please refer to [4].



## Distributed-Memory Parallel SMACOF

In order to implement distributed-memory parallel SMACOF, there must be done two things: one is the program decomposition which is actually block matrix decomposition for the SMACOF implementation since SMACOF is composed of an iterative matrix multiplication, and the other is communication between decomposed processes if necessary. For the program decomposition, our implementation allows users to choose the number of block-row and block-column with a constraint that the multiplication of the number of block-row and block-column should be equal to the number of processes, so that each process will be assigned corresponding decomposed sub-matrix. For instance, if we run this program with 16 processes, then users can decompose the *N×N* full matrices into not only *4×4* block matrices but also 16×1, 8×2, 2×8, and 1×16 block matrices. In addition, message passing interface (MPI) is used to communicate between processes, and MPI.NET [9], which is an implementation of message passing interface (MPI) for C# language, is used for the communication.

### Advantages of Distributed-memory Parallel SMACOF

The running time of SMACOF algorithm is O (*N2*). Though matrix multiplication of *V†*∙*B(X)* takes O (*N3*), you can reduce the computation order by using association property of matrix multiplication, since *V†*∙(*B(X)*∙*X*) is O (2*N2p*), where *N* is the number of points and *p* is the target dimension that we would like to find a configuration for given data. Also, SMACOF algorithm uses at least four full *N*×*N* double matrices, i.e. *Δ, D, V†,* and *B(X)*, which means at least 32× *N*2 bytes of memory should be allocated to run SMACOF program.

As in general, there are temporal and spatial advantages when we use distributed-memory parallelism. First, computational advantage should be achieved by both shared-memory and distributed-memory parallel implementation of SMACOF. While shared-memory parallelism is limited by the number of processors (or cores) in a single machine, distributed-memory parallelism can be extended the available number of processors (or cores) as much as machines are available, theoretically. SMACOF algorithm uses at least 32× N2 bytes of memory as we mentioned above. For example, 32MB, 3.2GB, 12.8GB, and 320GB are necessary for N = 1000, 10000, 20000, 100000, correspondingly. Therefore, if you have a multicore workstation which has 8GB memory, then it is capable to run SMACOF algorithm with 10000 data points on your workstation, but not with 20000 data points. If you develop shared-memory parallel SMACOF, then you may get speedup for computation, but still you cannot run SMACOF with 20000 data points. Thus, the distributed-memory parallelism provides us to be able to run SMACOF algorithm with much more data, and this benefit is quite important in such a data deluge era.

## Experimental Results and Analysis

For the performance experiments, the authors use three nodes in our Windows cluster. The hardware specification of each node is in Table 1. For the performance test, we generate artificial random data set which is in 8-centered Gaussian distribution in 4-dimension with different number of data points, such as 128, 256, 512, 1024, 2048, and 4096.

**Table 1.** Hardware specification of test nodes

|  |  |  |  |
| --- | --- | --- | --- |
| **Node ID** | **AMD16a** | **AMD16b** | **Intel24** |
| CPU | AMD Opteron 8356 | AMD Opteron 8356 | Intel Xeon E7450 |
| Core Clock Speed | 2.30 GHz / core | 2.30 GHz / core | 2.40 GHz / core |
| # of Cores | 4-core × 4 | 4-core × 4 | 6-core × 4 |
| L2 cache | 4 × 512KB / processor | 4 × 512KB / processor | 3 × 3MB / processor |
| L3 cache | 2MB / processor | 2MB / processor | 12MB / processor |
| Main Memory | 16GB | 16GB | 48GB |
| OS | Windows Server 2008 HPC Edition, SP1 | Windows Server 2008 HPC Edition, SP1 | Windows Server 2008 HPC Edition, SP1 |

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, 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.

### Performance Analysis

For the purpose of performance comparison, the authors implement naïve sequential version of SMACOF algorithm. The sequential SMACOF is executed on each test node, and the test results are in Table 2. Note that the running time of Intel24 is almost twice faster than the other two nodes, though the core’s clock speed of each node is similar. The reason would be the cache memory size. Both L2 and L3 cache of two AMD nodes are much smaller than those of **Intel24** node.

**Table 2.** Sequential Running time on each test node

|  |  |  |  |
| --- | --- | --- | --- |
|  | **AMD16a** | **AMD16b** | **Intel24** |
| 128 | 0.3437 | 0.3344 | 0.1685 |
| 256 | 1.9031 | 1.9156 | 0.9204 |
| 512 | 9.128 | 9.2312 | 4.8456 |
| 1024 | 32.2871 | 32.356 | 18.1281 |
| 2048 | 150.5793 | 150.949 | 83.4924 |
| 4096 | 722.3845 | 722.9172 | 384.7344 |

At first, the authors experimented performance of the implemented distributed-memory parallel SMACOF (MPI\_SMACOF) on each test node only. Figure 1 shows the speedup of each test node with different number of processes. Both axes of the Figure 1 are in logarithmic scale. As the Figure 1 depicted, the MPI\_SMACOF is not good for small data, such as 128 and 256 data points. However, for larger data, i.e. 512 and more data points, the MPI\_SMACOF shows great performance on the test data. You should notice those speedup values of larger data, such as 1024 and more data points on **AMD16a** and **AMD16b** nodes are bigger than the actual processes number using the MPI\_SMACOF application, which means super-linear speedup. However, on the **Intel24** node, it represented good speedup but not super-linear speedup at all. The reason of super-linear speedup is related to cache-hit ratio, as we discussed about sequential running results. MPI\_SMACOF implemented in the way of block decomposition, so that those sub-matrix would be better matched in the cache line size and the portion of sub-matrix which is in cache memory at a moment would be bigger than the portion of whole matrix in it. The Figure 1 also describes that the speedup ratio (or efficiency) becomes worse when you run MPI\_SMACOF with more processes on single node. It seems natural that as the number of computing units increases, the assigned computing job will be decreased but the communication overhead will be increased.

**Figure 1.** Speedup of MPI\_SMACOF performance on each test node

**Figure 2.** Speedup of MPI\_SMACOF on combine nodes

Also, the authors experimented performance of the proposed MPI\_SMACOF algorithm on all three test nodes with different number of processes. Figure 2 illustrates the speedup of those experiments with respect to the average of the sequential SMACOF running time on each node. The comparison with average might be reasonable since, for every test case, the processes are equally spread as much as possible on those three test nodes except the case of 56 processes running. The Figure 2 represents that the speedup values are increasing as the data size is getting bigger. This result shows that the communication overhead on different nodes is larger than communication overhead on single node, so that the speedup is still increasing, even with large test data such as 2048 and 4096 points, instead of being converged as in Figure 1.

# Multicore Datamining

**Table 1. Machines used**

|  |  |  |
| --- | --- | --- |
| **Barcelona Cluster** | **Head**  **Node (1)** | Dell PowerEdge T605 workstation,  1 AMD Quad Core Opteron 2356 at 2.3GHz,  L2 Cache 2x1MB, Memory 8 GB,  Windows Server HPC Edition (Service Pack 1) |
| **Compute node (4)** | Dell PowerEdge 2970,  2 AMD Quad Core Opteron 2356 at 2.3GHz,  4×512K L2 Cache, Memory 16 GB,  Windows Server 2003 Enterprise x64 bit Edition |
| **Compute node (2)** | Dell PowerEdge R905,  4 AMD Quad Core Opteron 8356 at 2.3GHz,  4×512K L2 Cache, Memory 16 GB,  Windows Server HPC Edition (Service Pack 1) |
| **Compute node (1)** | Dell PowerEdge R900,  4 Intel Six Core Xeon E7450 at 2.4GHz,  12 M L2 Cache, Memory 48GB,  Windows Server HPC Edition (Service Pack 1) |
| **Madrid Cluster** | **Head**  **Node (1)** | Dell PowerEdge T605 workstation,  1 AMD Quad Core Opteron 2356 at 2.3GHz,  L2 Cache 2x1MB, Memory 8 GB,  Windows Server HPC Edition (Service Pack 1) |
| **Compute node (8)** | Dell PowerEdge R905,  4 AMD Quad Core Opteron 8356 at 2.3GHz,  4x512K Cache, Memory 16 GB,  Windows Server HPC Edition (Service Pack 1) |

# Acknowledgements

We thank other members of the SALSA parallel computing group for conversations. Our work on patient data would not have been possible with Gilbert Liu from IU Medical School and the gene sequence data was produced by Haixu Tang and Mina Rho from the IU Bioinformatics group who also answered our endless naïve questions. The IU MPI.NET group led by Andrew Lumsdaine was very helpful as we learnt how to use this. This research was partially supported by Microsoft. We have obtained continued help on CCR from George Chrysanthakopoulos and Henrik Frystyk Nielsen. Scott Beason provided key support on visualization and set up of Clusters.

References

1. F. Darema, *SPMD model: past, present and future, Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 8th European PVM/MPI Users' Group Meeting, Santorini/Thera, Greece, 2001.
2. MPI (Message Passing Interface), http://www-unix.mcs.anl.gov/mpi/
3. J. Dean and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters*, ACM Commun., vol. 51, Jan. 2008, pp. 107-113.
4. Apache Hadoop, http://hadoop.apache.org/core/
5. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, *Dryad: Distributed data-parallel programs from sequential building blocks*, European Conference on Computer Systems , March 2007.
6. S. Ghemawat, H. Gobioff, and S. Leung, *The Google file system*, Symposium on Operating Systems Principles 2003, pp 29–43, 2003.
7. Disco project, http://discoproject.org/
8. Erlang programming language, http://www.erlang.org/
9. S. Pallickara and G. Fox, “NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids,” Middleware 2003, pp. 41-61.
10. J. B. MacQueen , *Some Methods for classification and Analysis of Multivariate Observations*, Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, University of California Press, vol. 1, pp. 281-297.
11. Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Kumar and J. Currey, *DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language* Symposium on Operating System Design and Implementation (OSDI), San Diego, CA, December 8-10, 2008.
12. M. Svens´en. GTM: *The Generative Topographic Mapping*. PhD thesis, Neural Computing Research Group, Aston University, Birmingham, U.K., 1998.
13. T. Kohonen. *Self-Organizing Maps*. Springer-Verlag, Berlin, Germany, 2001.
14. J. B. Kruskal and M.Wish. *Multidimensional Scaling*. Sage Publications Inc., Beverly Hills, CA, U.S.A., 1978.
15. I. Borg and P. J. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, New York, NY, U.S.A., 2005.
16. J. de Leeuw. *Applications of convex analysis to multidimensional scaling*. Recent Developments in Statistics, pages 133–145, 1977.
17. J. de Leeuw. *Convergence of the majorization method for multidimensional scaling*. Journal of Classification, 5(2):163–180, 1988.
18. J. B. Kruskal. *Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis*. Psychometrika, 29(1):1–27, 1964.
19. 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.
20. Douglas Gregor and Andrew Lumsdaine. *Design and Implementation of a High-Performance MPI for C# and the Common Language Infrastructure*. Principles and Practice of Parallel Programming (PPoPP), pages 133-142, February 2008. ACM.
21. Dennis Gannon and Geoffrey Fox, *Workflow in Grid Systems* Concurrency and Computation: Practice & Experience 18 (10), 1009-19 (Aug 2006), Editorial of special issue prepared from GGF10 Berlin.
22. Geoffrey Fox and Marlon Pierce *Grids Challenged by a Web 2.0 and Multicore Sandwich* Special Issue of Concurrency&Compuitation:Practice&Experience on Seventh IEEE International Symposium on Cluster Computing and the Grid — CCGrid 2007, Keynote Talk Rio de Janeiro Brazil May 15 2007 <http://grids.ucs.indiana.edu/ptliupages/publications/CCGridDec07-Final.pdf>
23. Seung-Hee Bae Parallel *Multidimensional Scaling Performance on Multicore Systems* at workshop on Advances in High-Performance E-Science Middleware and Applications in Proceedings of eScience 2008 Indianapolis IN December 7-12 2008 <http://grids.ucs.indiana.edu/ptliupages/publications/eScience2008_bae3.pdf>
24. Xiaohong Qiu, Geoffrey Fox, H. Yuan, Seung-Hee Bae, George Chrysanthakopoulos, Henrik Frystyk Nielsen *High Performance Multi-Paradigm Messaging Runtime Integrating Grids and Multicore Systems* September 23 2007 published in proceedings of eScience 2007 Conference Bangalore India December 10-13 2007 <http://grids.ucs.indiana.edu/ptliupages/publications/CCRSept23-07eScience07.pdf>
25. Xiaohong Qiu, Geoffrey C. Fox, Huapeng Yuan, Seung-Hee Bae, George Chrysanthakopoulos, Henrik Frystyk Nielsen *PARALLEL CLUSTERING AND DIMENSIONAL SCALING ON MULTICORE SYSTEMS* Invited talk at he 2008 High Performance Computing & Simulation Conference (HPCS 2008) In Conjunction With The 22nd EUROPEAN CONFERENCE ON MODELLING AND SIMULATION (ECMS 2008) Nicosia, Cyprus June 3 - 6, 2008; Springer Berlin / Heidelberg Lecture Notes in Computer Science Volume 5101/2008 "Computational Science: ICCS 2008" ISBN 978-3-540-69383-3 Pages 407-416 DOI: <http://dx.doi.org/10.1007/978-3-540-69384-0_46>
26. Xiaohong Qiu, Geoffrey C. Fox, Huapeng Yuan, Seung-Hee Bae, George Chrysanthakopoulos, Henrik Frystyk Nielsen *Performance of Multicore Systems on Parallel Data Clustering with Deterministic Annealing* ICCS 2008: "Advancing Science through Computation" Conference; ACC CYFRONET and Institute of Computer Science AGH University of Science and Technology Kraków, POLAND; June 23-25, 2008. Springer Lecture Notes in Computer Science Volume 5101, pages 407-416, 2008. DOI: <http://dx.doi.org/10.1007/978-3-540-69384-0_46>
27. Xiaohong Qiu , Geoffrey C. Fox, Huapeng Yuan, Seung-Hee Bae, George Chrysanthakopoulos, Henrik Frystyk Nielsen Parallel *Data Mining on Multicore Clusters* 7th International Conference on Grid and Cooperative Computing GCC2008 Shenzhen China October 24-26 2008 <http://grids.ucs.indiana.edu/ptliupages/publications/qiu-ParallelDataMiningMulticoreClusters.pdf>
28. Home Page for SALSA Project at Indiana University <http://www.infomall.org/salsa>.
29. Kenneth Rose, Eitan Gurewitz, and Geoffrey C. Fox *Statistical mechanics and phase transitions in clustering* Phys. Rev. Lett. 65, 945 - 948 (1990)
30. Rose, K. *Deterministic annealing for clustering, compression, classification, regression, and related optimization problems*, Proceedings of the IEEE Vol. 86, pages 2210-2239, Nov 1998
31. T Hofmann, JM Buhmann *Pairwise data clustering by deterministic annealing*, IEEE Transactions on Pattern Analysis and Machine Intelligence 19, pp1-13 1997
32. MPI.NET Home Page <http://www.osl.iu.edu/research/mpi.net>
33. Microsoft Robotics Studio is a Windows-based environment that includes end-to-end Robotics Development Platform, lightweight service-oriented runtime, and a scalable and extensible platform. For details, see <http://msdn.microsoft.com/robotics/>
34. Georgio Chrysanthakopoulos and Satnam Singh *An Asynchronous Messaging Library for C#*, Synchronization and Concurrency in Object-Oriented Languages (SCOOL) at OOPSLA October 2005 Workshop, San Diego, CA.
35. Hansjorg Klock, Joachim M. Buhmann *Multidimensional scaling by deterministic annealing* in Energy Minimization Methods in Computer Vision and Pattern Recognition, Eds Pelillo M. and Hancock E.R., Proc. Intl. Workshop EMMCVPR Venice Italy, Springer Lecture Notes in Computer Science 1223 ppg. 246-260 May 1997
36. Anthony J. Kearsley, Richard A. Tapia, Michael W. Trosset *The Solution of the Metric STRESS and SSTRESS Problems in Multidimensional Scaling Using Newton’s Method*, technical report 1995.
37. C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun*. Map-reduce for machine learning on multicore.* In B. Scholkopf, J. Platt, and T. Hoffman, editors, Advances in Neural Information Processing Systems 19, pages 281–288. MIT Press, Cambridge, MA, 2007.
38. Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox *Map-Reduce for Data Intensive Scientific Analyses* Proceedings of the IEEE International Conference on e-Science. Indianapolis. 2008. December 7-12 2008 <http://grids.ucs.indiana.edu/ptliupages/publications/ekanayake-MapReduce.pdf>
39. Hansjorg Klock, Joachim M. Buhmann, *Data visualization by multidimensional scaling: a deterministic annealing approach*, Pattern Recognition 33 (2000) 651}669
40. Alkes L. Price, Eleazar Eskin and Pavel A. Pevzner, *Whole-genome analysis of Alu repeat elements reveals complex evolutionary history*. Genome Res. 2004 14: 2245-2252 DOI: http://dx.doi.org/10.1101/gr.2693004
41. Bell JF, Wilson JS, Liu GC. *Neighborhood greenness and 2-year changes in body mass index of children and youth*. Am J Prev Med. Dec 2008;35(6):547-553.

1. Corresponding Author. [↑](#footnote-ref-2)