Parallel Data Mining from Multicore to Cloudy Grids

Geoffrey Fox[a,b,1] Seung-Hee Bae[b], Jaliya Ekanayake[b], Xiaohong Qiu[c], and Huapeng Yuan[b]

[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 [4] or workflow technologies [5] 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 usually the "almost embarrassingly parallel" style illustrated by the independent events in

[1] Corresponding Author.

particle physics or the independent documents of information retrieval – these lead to independent "maps" (processing) which 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 [6] 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 N^2 (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 (4 core Head Node)	1	1 AMD Quad Core Opteron 2356 2.3GHz	2x1MB 8 GB	Windows Server HPC Edition (Service Pack 1)
B	Barcelona (8 core Compute Node)	4	2 AMD Quad Core Opteron 2356 2.3 GHz	4×512K 16GB	Windows Server 2003 Enterprise x64 bit Edition
C	Barcelona (16 core Compute Node)	2	4 AMD Quad Core Opteron 8356 2.3GHz	4×512K 16 GB	Windows Server HPC Edition (Service Pack 1)
D	Barcelona (24 core Compute Node)	1	4 Intel Six Core Xeon E7450 2.4GHz	12 M 48GB	Windows Server HPC Edition (Service Pack 1)
E	Madrid (4 core Head Node)	1	1 AMD Quad Core Opteron 2356 2.3GHz	2x1MB 8 GB	Windows Server HPC Edition (Service Pack 1)
F	Madrid (16 core Compute Node)	8	4 AMD Quad Core Opteron 8356 2.3GHz	4x512K 16 GB	Windows Server HPC Edition (Service Pack 1)
G	Gridfarm 8 core	8	2 Quad core Intel Xeon E5345 2.3GHz	4x1MB 8GB	Red Hat Enterprise Linux 4
H	IU Quarry 8 core	112	2 Quad-core Intel Xeon 5335 2.00GHz	4x4MB, 8 GB	Red Hat Enterprise Linux 4

Our algorithms are parallel MDS (Multi dimensional scaling) [7] and clustering. The latter has been discussed earlier by us [8-12] 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 20 years ago [13] and has a good reputation [14] and there is no clearly better clustering approach. The pairwise extension PWDA was developed by Hofmann and Buhmann [15] 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.

1. 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, 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;

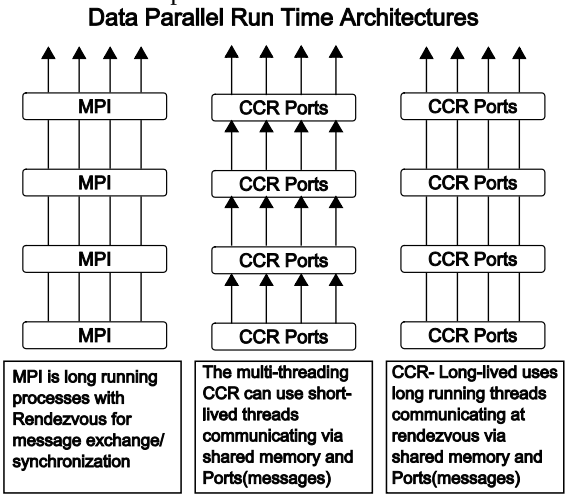

Figure 1(a). First three of seven different combinations of processes/threads and intercommunication mechanisms discussed in the text

the new generation of data intensive programming systems typified by Hadoop [16] (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.

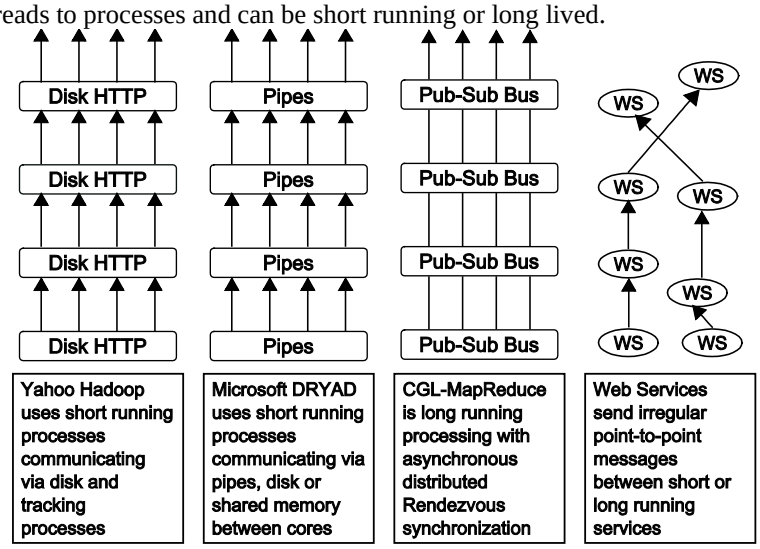

Figure 1(b). Last four of seven different combinations of processes/threads and intercommunication mechanisms discussed in the text

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 due to the distributed memory 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 processing 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

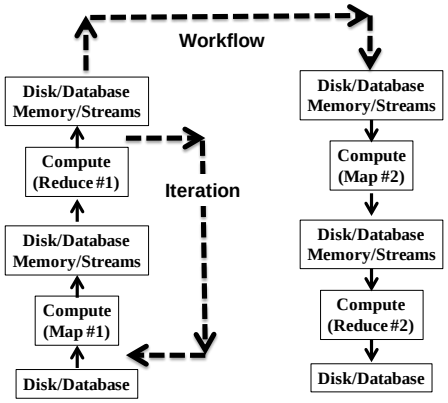

Figure 2: Data Intensive Iteration and Workflow

a) **Iteration** where results of one stage are iterated many times. 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 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 here messaging can be attractive as it avoids many of the error scenarios seen in shared memory thread synchronization. The CCR threading [8-11] used in this paper is coordinated by reading and writing messages to ports. As a further example of runtimes crossing different application caharacteristics, MPI has often been used in Grid (distributed) applications with MPICH-G popular here. Again the paper of Chu [17] 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 files) the data on it's file. Future scientific programming models must clearly capture this concept.

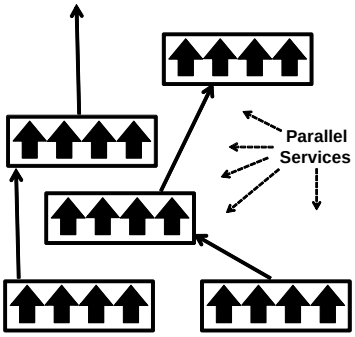

Figure 3. Workflow of Parallel Services

2. 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 and Dryad.

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.

2.1. Current MapReduce Implementations

Google's MapReduce implementation is coupled with a distributed file system named Google File System (GFS) [18] 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 [19] – another open source MapReduce runtime developed using a functional programming language named Erlang [20]. 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.

2.2. 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 4 shows the main components of the CGL-MapReduce.

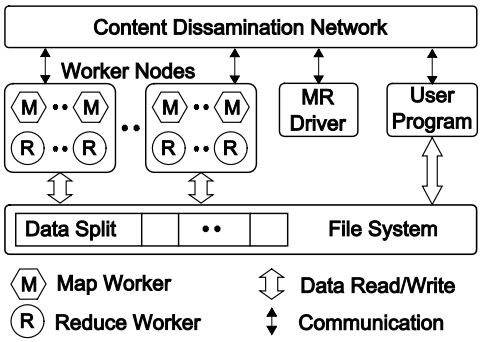

Figure 4. Components of the CGL-MapReduce

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.

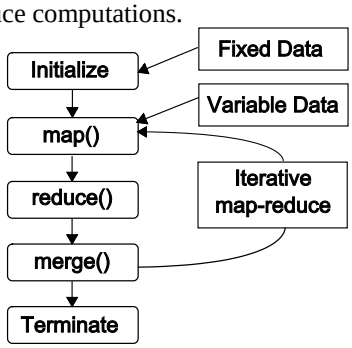

Figure 5. Computation phases of CGL-MapReduce

A MapReduce computation under CGL-MapReduce passes through several phases of computations as shown in figure 5. 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 transfers the intermediate result from *map* tasks 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[21], 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.

2.3. 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 [22] algorithm and implemented it using Hadoop and CGL-MapReduce. Details of these applications and the challenges we faced in implementing them can be found in [23]. 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. Test configurations.

Feature	**HEP Data Analysis**	**Kmeans clustering**	**Matrix Multiplication**	**Histogramming & Grep**
Cluster Ref	H	G	G	B
Number of Nodes	12	5	5	4
Number of Cores	96	40	40	32
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 6 depicts our results.

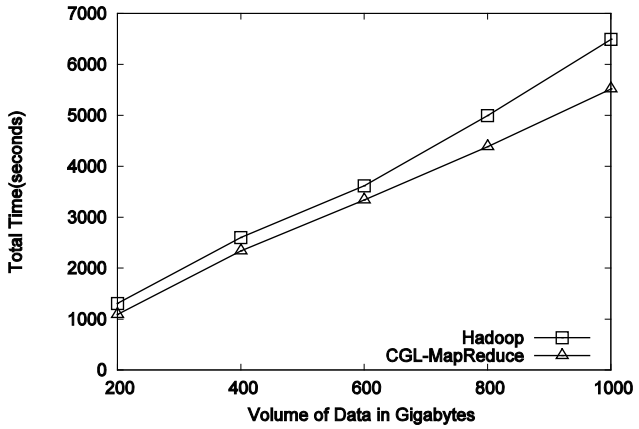

Figure 6. HEP data analysis, execution time vs. the volume of data (fixed compute resources)

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 parallel 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 7 depicts our results.

$$\phi(P) = [PT(P) - T(1)] / T(1) \quad (1)$$

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 8.

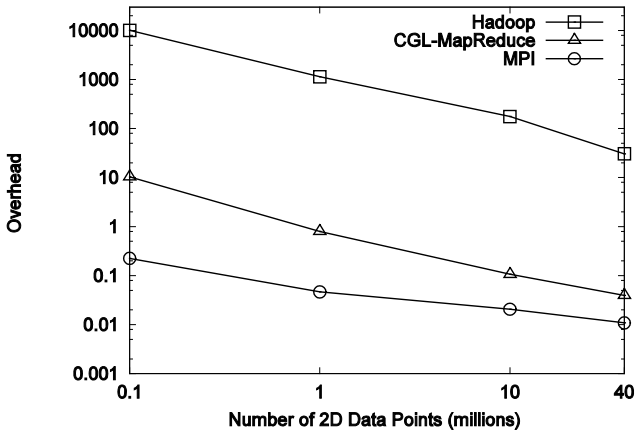

Figure 7. Overheads associated with Hadoop, CGL-MapReduce and MPI for Kmeans clustering – iterative MapReduce - (Both axes are in log scale)

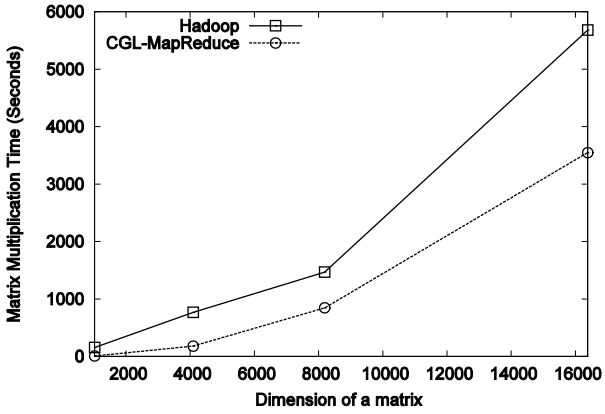

Figure 8. Performance of the Hadoop and CGL-MapReduce for matrix multiplication

The results in figure 7 and figure 8 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[24] programming environment. The results of these two evaluations are shown in figure 9 and figure 10.

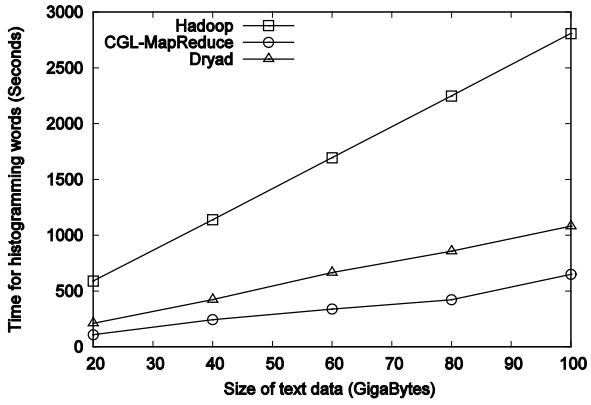

Figure 9. Performance of Dryad, Hadoop, and CGL-MapReduce for "histogramming of words" operation.

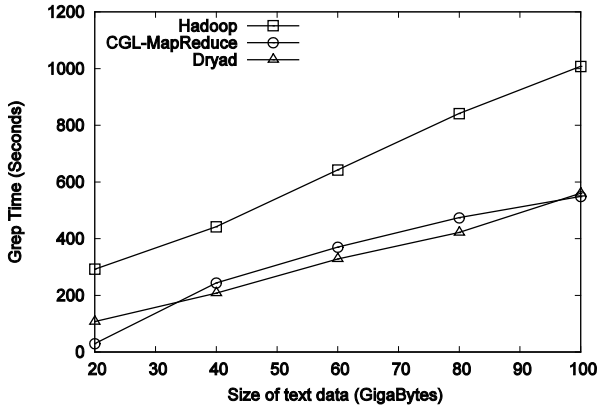

Figure 10. Performance of Dryad, Hadoop, and CGL-MapReduce for "distributed grep" operation

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 transfers 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 show close performance results.

3. Multidimensional Scaling

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) [25] and Self-Organizing Map (SOM) [26], seek to preserve topological properties of given data rather than proximity information. On the other hand, multidimensional scaling (MDS) [27, 28] tries to maintain dissimilarity information between mapping points as much as possible. The MDS uses several full matrices which are $N \times N$ matrices for the N given data points. Thus, the matrices could be very large for large problems (N could be as big as millions even today). For large problems, we will initially cluster the given data and use the cluster centers to reduce the problem size. Here we parallelize an elegant algorithm for computing MDS solution, named SMACOF (Scaling by MAjorizing a COmplicated Function) [29, 30], using MPI.NET [31, 32] which is an implementation of message passing interface (MPI) for C# language and presents performance analysis of the parallel implementation of SMACOF on multicore cluster systems. We show some examples of the use of MDS to visualize the results of the clustering algorithms of section 4 in figure 11. These are datasets in high dimension (from 20 in figure 11(right) to over a thousand in figure 11(left)) which are projected to 3D using proximity (distance/dissimilarity) information. The figure shows 2D projections determined by us from rotating 3D MDS results.

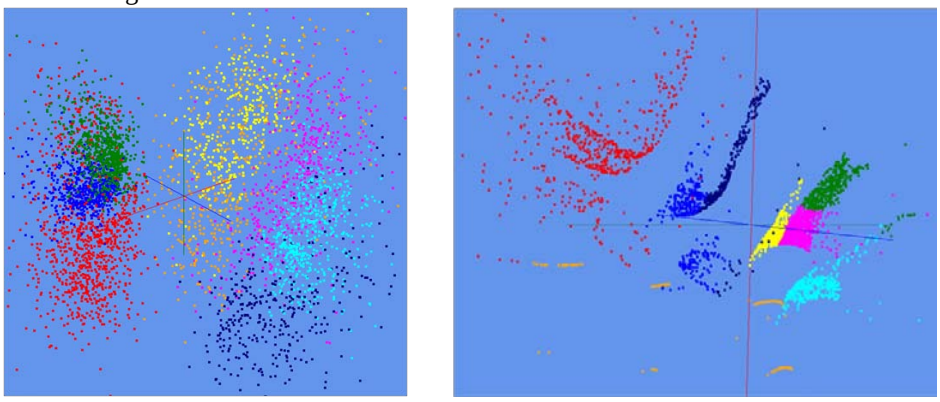

Figure 11. Visualization of MDS projections using parallel SMACOF described in section 3. Each color represents a cluster determined by the PWDA algorithm of section 4. Figure 11(left) corresponds to 4500 ALU pairwise aligned Gene Sequences with 8 clusters [33] and 11(right) to 4000 Patient Records with 8 clusters from [34]

Multidimensional scaling (MDS) 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 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 point 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 [35] and SSTRESS [36] were suggested as objective functions of MDS algorithms. STRESS (σ or $\sigma(X)$) criterion (Eq. (2)) is a weighted squared error between distance of configured points and corresponding dissimilarity, but SSTRESS (σ^2 or $\sigma^2(X)$) criterion (Eq. (3)) is a weighted squared error between squared distance of configured points and corresponding squared dissimilarity.

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

$$\sigma^2(X) = \Sigma_{i<j\leq n} w_{ij} [(d_{ij}(X))^2 - (\delta_{ij})^2]^2 \tag{3}$$

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.

3.1. Scaling by MAjorizing a COmplicated Function (SMACOF)

Scaling by MAjorizing a COmplicated Function (SMACOF) [29, 30] 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 [28].

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;

3.2. Distributed-Memory Parallel SMACOF

In order to implement distributed-memory parallel SMACOF, one must address two issues: one is the data decomposition which is actually block matrix decomposition for the SMACOF implementation since SMACOF is composed of an iterative matrix

multiplication, and the other is the required communication between decomposed processes. For the data decomposition, our implementation allows users to choose the number of row-blocks and column-blocks with a constraint that the product of the number of row-blocks and column-blocks 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 $N \times N$ 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 is used for the communication.

3.2.1. Advantages of Distributed-memory Parallel SMACOF

The running time of SMACOF algorithm is O (N^2). Though matrix multiplication of $V^{\dagger} \cdot B(X)$ takes O (N^3), you can reduce the computation order by using association property of matrix multiplication, since $V^{\dagger} \cdot (B(X) \cdot X)$ is O ($2N^2p$), 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 $N \times N$ double matrices, i.e. Δ, D, $V^{\dagger}$, and $B(X)$, which means at least $32 \times N^2$ 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 $32 \times N^2$ 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, a multicore workstation, which has a 8GB of memory will be able to run SMACOF algorithm with 10000 data points. However, this workstation cannot be used to run the same algorithm with 20000 data points. Shared memory parallelism increases performance but does not increase size of problem that can be addressed. 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 the era of a data deluge.

3.3. Experimental Results and Analysis

For the performance experiments of the distributed-memory parallel SMACOF, we use two nodes of Ref C and one node of Ref D 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.

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. We will investigate the dependence on starting point more thoroughly using other approaches discussed in section 3.4.

3.3.1. Performance Analysis

For the purpose of performance comparison, we implemented the sequential version of SMACOF algorithm. The sequential SMACOF is executed on each test node, and the test results are in Table 3. Note that the running time of D 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. L2 cache of two Ref C nodes (**C1** and **C2**) is much smaller than that of **D** node.

Table 3. Sequential Running time on each test node

Data size	C1	C2	D
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

Initially we measured the performance of the distributed-memory parallel SMACOF (MPI_SMACOF) on each test node only. Figure 12 shows the speedup of each test node with different number of processes. Both axes of the Figure 12 are in logarithmic scale. As the Figure 12 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 or more data points on **C1** and **C2** nodes are bigger than the actual processes number using the MPI_SMACOF application, which corresponds to super-linear speedup. However, on the **D** 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 12 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.

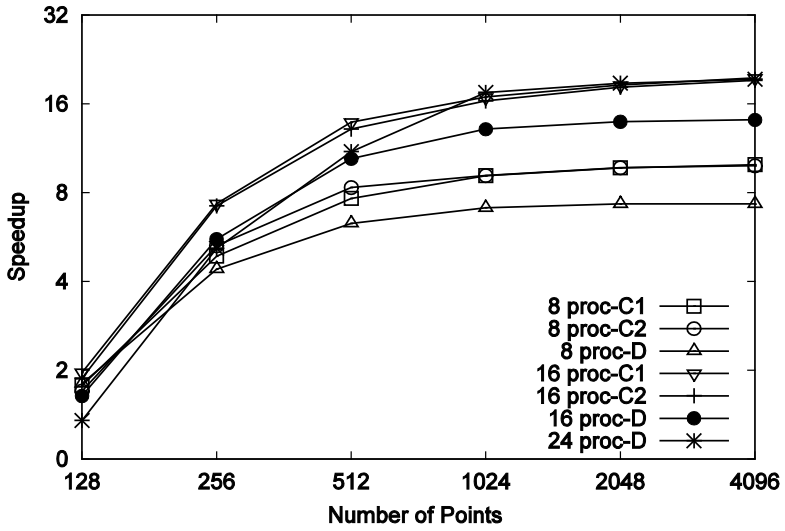

Figure 12. Speedup of MPI_SMACOF performance on each test node

In addition, we have measured the performance of the proposed MPI_SMACOF algorithm on all the three test nodes with different number of processes. Figure 13 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 13 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 12.

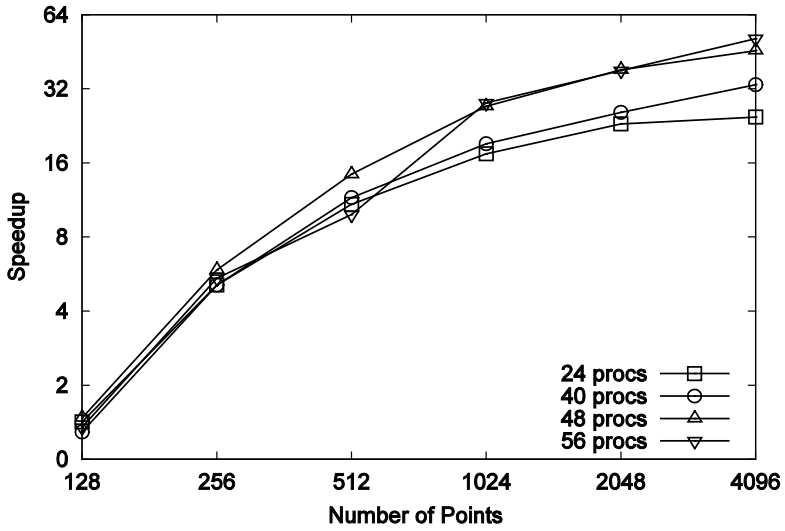

Figure 13. Speedup of MPI_SMACOF on combine nodes

3.4. Conclusions

We have developed a dimension mapping tool that is broadly applicable as it only uses dissimilarity values and does not require the points to be in a vector space. We have good parallel performance and are starting to use it for science as illustrated in figure 11. We will compare the method described with alternatives that can also be parallelized and avoid the steepest descent approach of SMACOF which can lead to local minima. One first described in [37] and [38] uses deterministic annealing based on ideas sketched in section 4. This still uses Expectation Maximization (EM) (steepest descent) but only for the small steps needed as temperature is decreased. We will also implement the straightforward but possibly best method from ref [39] that solves equations (2) and (3) as χ^2 problems and uses optimal solution methods for this.

4. Multicore Clustering

Clustering can be viewed as an optimization problem that determines a set of K clusters by minimizing

$H_{VEC} = \sum_{i=1}^{N} \sum_{k=1}^{K} M_i(k)\, D_{VEC}(i,k)$ (2)

where $D_{VEC}(i,k)$ is the distance between point *i* and cluster center *k*. N is the number of points and $M_i(k)$ is the probability that point *i* belongs to cluster *k*. This is the vector version and one obtains the pairwise distance model with:

$H_{PW} = 0.5 \sum_{i=1}^{N} \sum_{j=1}^{N} D(i, j) \sum_{k=1}^{K} M_i(k)\, M_j(k) / C(k)$ (3)

and $C(k) = \sum_{i=1}^{N} M_i(k)$ is the expected number of points in the k'th cluster. Equation (2) requires one be able to calculate the distance between a point *i* and the cluster center *k* and this is only possible when one knows the vectors corresponding to the points *i*. (3) reduces to (2) when one inserts vector formulae and drops terms $\sum_{i=1}^{N} \sum_{j=1}^{N} D_{VEC}(i,k)\, D_{VEC}(j,k) \sum_{k=1}^{K} M_i(k)\, M_j(k)$ that average to zero.

One must minimize (2) or (3) as a function of cluster centers (1) and cluster assignments $M_i(k)$. One can derive deterministic annealing from an informatics theoretic [14] or physics formalism [15]. In latter case one smoothes out the cost function (2) or (3) by averaging with the Gibbs distribution exp(-H/T). This implies in a physics language that one is minimizing not H but the free energy F at temperature T and entropy S

$$F = H - TS \qquad (4)$$

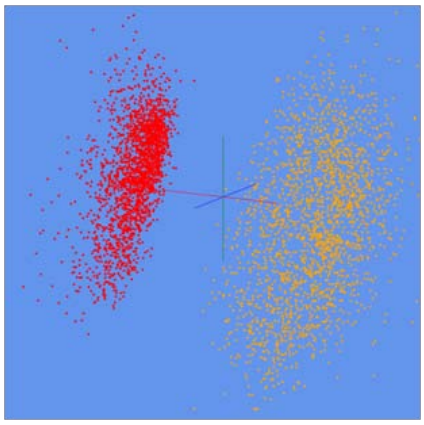

Figure 14. Preliminary stage of clustering shown in figure 11(left) corresponding to 4500 ALU pairwise aligned Gene Sequences with 2 clusters [33]

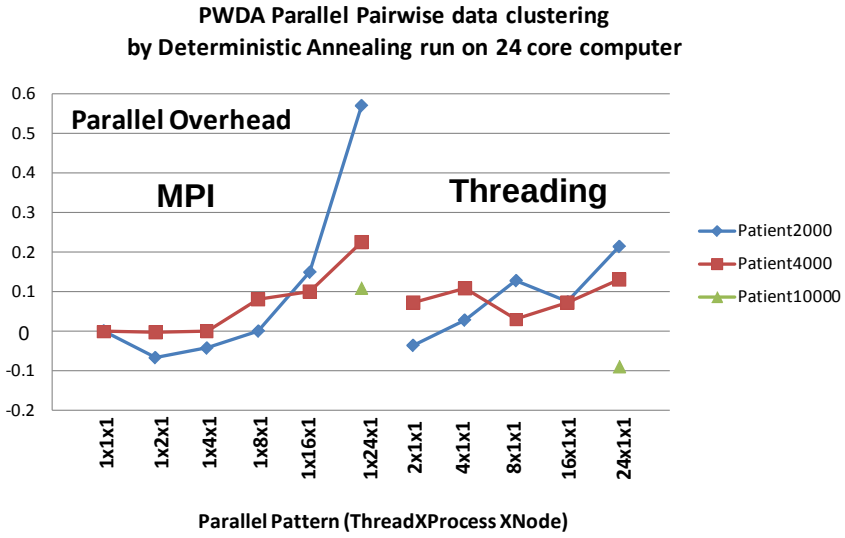

Figure 15. Parallel Overhead for pure threading or pure MPI on 24 core Ref D of Table 1 for three different patient datasets with 2000, 4000 and 10,000 elements. The leftmost results are MPI.NET runs labeled 1XNX1 for N MPI processes. The rightmost results are CCR threading labeled NX1X1 for N threads

In [9] and [11], we explain how a single formalism describes multiple different problems, VECDA (Clustering of points defined by vectors with deterministic annealing) [13, 14], Gaussian Mixture Models (GMM) [40]; Gaussian Mixture Models

with deterministic annealing (GMMDA) [41]; and Generative Topographic Maps (GTM) [25]. One can also add deterministic annealing to GTM and derive asimilar formalism although there is no study of this yet. Annealing in most problems corresponds to a multi-scale approach with temperature corresponding to a distance scale that starts very large and decreases. For example the eight clusters in figure 11(left) were found systematically with clusters being added as one reduced temperature so that at a higher temperature one first split from one to two clusters to find results of figure 14. The splits are determined from the structure of second derivative matrix and continuing figure 14 leads to figure 11(left). The vector clustering model is suitable for low dimensional spaces such as our earlier work on census data [9] but the results of figure 11 and 14 correspond to our implementation of PWDA – the pairwise distance clustering approach of [15] which starts from equation (3). As described in [42] this has similarities to familiar $O(N^2)$ problems such astrophysical particle dynamics. Whereas VECDA is pure MapReduce just using broadcast, allreduce and barrier in MPI, PWDA has significant use of send-receive in MPI as information must passed around a ring of processors. As N is potentially of order a million we see that both MDS and pairwise clustering are potential supercomputing data analysis applications. We have performed extensive performance measurements [8-11, 42] showing the effect of cache and for Windows runtime fluctuations can be quite significant. Here we give some typical results with figure 15 showing the performance of PWDA on the single 24 core workstation (ref D of table 1). The results are expressed as an overhead using the definitions of equation (1) introduced in section 2. We compare both MPI and thread based parallelism using Microsoft's CCR package [43, 44]. As these codes are written in C#, we use MPI.NET[31, 32] finding this to allow an elegant object-based extension of traditional MPI and good performance. MPI.NET is a wrapper for the production Microsoft MPI.

Figure 15 shows that although threading and MPI both get good performance, their systematics are different. For the extreme case of 24-way parallelism, the thread implementation shows an overhead that varies between 10 and 20% depending on the data set size. MPI shows a large overhead for small datasets that decreases with increasing dataset size so in fact 24-way MPI parallelism is 20% faster than the thread version on the largest 10,000 element dataset. This is due to the different sources of the overhead. For MPI the overhead is due to the communication calls which are due to reduce (20%) and send-receive (80%) and this as expected decreases (inversely proportional to dataset size) as the dataset size increases. For threads there is no memory movement overhead but rather the overhead is due to the Windows thread scheduling that leads to large fluctuations that can have severe effects on tightly synchronized parallel codes such as those in this paper as discussed in refs. [8-11, 42]. We see some cases where the overhead is negative (super-linear speedup) which is due to better use of cache in the higher parallelism cases compared to sequential runs. This effect is seen in all our runs but differs between the AMD and Intel architectures reflecting their different cache size and architecture.

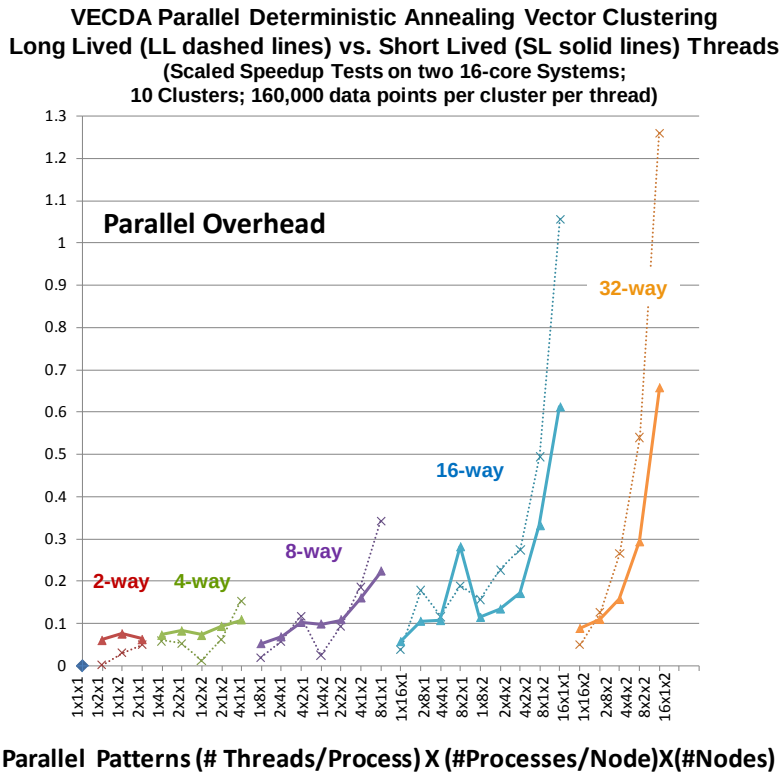

Figure 16. Comparison of use of short lived (solid lines) and long lived (dashed lines) threads for the Vector-based deterministic annealing VECDA. The results achieve a given parallelism by choosing number of nodes, MPI processes per node and threads per MPI process. The number of threads increases as you move from left to right for given level of parallelism.

Figure 16 looks at the vector clustering VECDA comparing MPI versus two versions of threading. MPI is very efficient – the 32 way parallel code with 16 MPI processes on each of two 16 core nodes has overheads (given by equation (1) and roughly 1 – efficiency) of 0.05 to 0.10. For the case of 16 threads on each of two nodes the overhead is 0.65 (short lived) to 1.25 (long lived) threads. This example is particularly favorable to MPI as only reduction is an important operation; there are no send or receive calls and a negligible amount of time on broadcast and barrier. The short lived threads are the natural implementation with threads spawned for parallel for loops. In the long lived case, the paradigm is similar to MPI with long running threads synchronizing with rendezvous semantics.

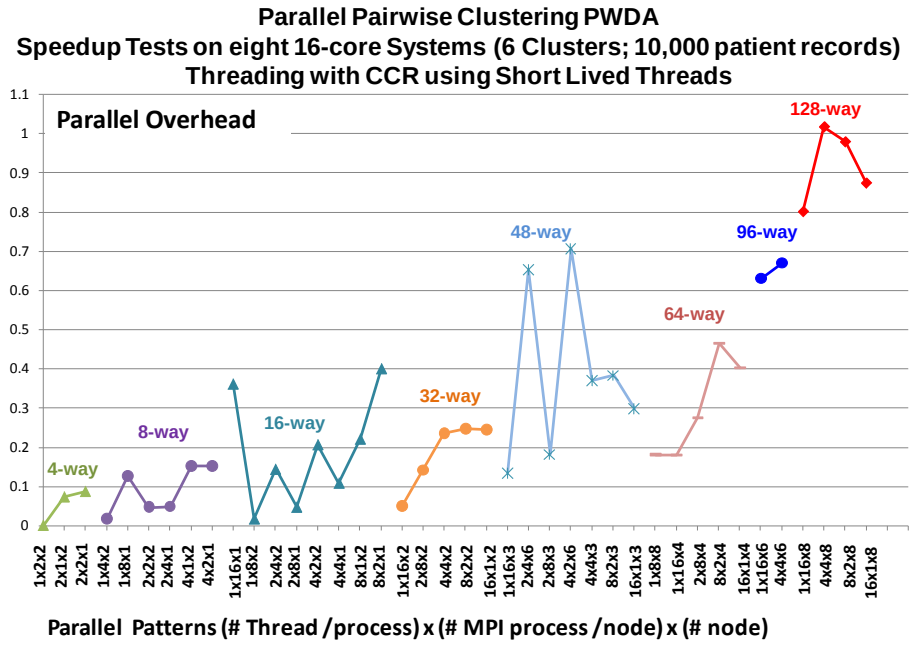

Figure 17. Parallel Overhead for PWDA runs on 128 core Cluster Ref. F in table 1 with patterns defined in figure 16.

Figure 17 shows preliminary results of PWDA for a 10,000 element dataset on the 128 core cluster. The results show less difference between MPI and threading than figure 16. That is partly due to MPI being dominated in this case by the use of send-receive as discussed above for the results of figure 15. The results also show effects of the cache which are still being investigated. This slows down some of low parallelism results – in particular the purely sequential 1x1x1 case. The patterns are always labeled as (threads per process)x(MPI processes per node)x(nodes). Note figure 17 studies the overhead for a fixed problem whereas figure 16 looks at scaled speedup with problem size increasing proportional to number of parallel units. We see that the 10,000 dataset can run well up to 32- or 64-way parallelism.

5. Conclusions

This paper has addressed several issues. It has studied the performance of a variety of different programming models on data intensive problems. It has presented novel clustering and MDS algorithms which are shown to parallelize well and could become supercomputer applications for large million point problems. It has compared MPI and threading on multicore systems showing both to be effective but with different overheads. We see these complemented by the data intensive programming models including Dryad and Hadoop as well as an in house version of MapReduce. These support an “owner stores and computes” programming paradigm that will be of increasing importance.

6. Acknowledgements

We thank other members of the SALSA parallel computing group including Jong Choi and Yang Ruan 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. Our research was partially supported by Microsoft. We have obtained continued help on Dryad from Roger Barga and 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] 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 ://grids.ucs.indiana.edu/ptliupages/publications/CCGridDec07-Final.pdf

[5] 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.

[6] 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.

[7] 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 ://grids.ucs.indiana.edu/ptliupages/publications/eScience2008_bae3.pdf

[8] 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 ://grids.ucs.indiana.edu/ptliupages/publications/CCRSept23-07eScience07.pdf

[9] 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 the 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: ://dx.doi.org/10.1007/978-3-540-69384-0_46

[10] 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: ://dx.doi.org/10.1007/978-3-540-69384-0_46

[11] 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 ://grids.ucs.indiana.edu/ptliupages/publications/qiu-ParallelDataMiningMulticoreClusters.pdf

[12] Home Page for SALSA Project at Indiana University ://www.infomall.org/salsa.

[13] Kenneth Rose, Eitan Gurewitz, and Geoffrey C. Fox *Statistical mechanics and phase transitions in clustering* Phys. Rev. Lett. 65, 945 - 948 (1990)

[14] Rose, K. *Deterministic annealing for clustering, compression, classification, regression, and related optimization problems*, Proceedings of the IEEE Vol. 86, pages 2210-2239, Nov 1998
[15] T Hofmann, JM Buhmann *Pairwise data clustering by deterministic annealing*, IEEE Transactions on Pattern Analysis and Machine Intelligence 19, pp1-13 1997
[16] Apache Hadoop, ://hadoop.apache.org/core/
[17] 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.
[18] S. Ghemawat, H. Gobioff, and S. Leung, *The Google file system*, Symposium on Operating Systems Principles 2003, pp 29–43, 2003.
[19] Disco project, http://discoproject.org/
[20] Erlang programming language, http://www.erlang.org/
[21] S. Pallickara and G. Fox, "NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids," Middleware 2003, pp. 41-61.
[22] 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.
[23] 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 ://grids.ucs.indiana.edu/ptliupages/publications/ekanayake-MapReduce.pdf
[24] 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.
[25] M. Svens´en. GTM: *The Generative Topographic Mapping*. PhD thesis, Neural Computing Research Group, Aston University, Birmingham, U.K., 1998.
[26] T. Kohonen. *Self-Organizing Maps*. Springer-Verlag, Berlin, Germany, 2001.
[27] J. B. Kruskal and M.Wish. *Multidimensional Scaling*. Sage Publications Inc., Beverly Hills, CA, U.S.A., 1978.
[28] I. Borg and P. J. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, New York, NY, U.S.A., 2005.
[29] J. de Leeuw. *Applications of convex analysis to multidimensional scaling*. Recent Developments in Statistics, pages 133–145, 1977.
[30] J. de Leeuw. *Convergence of the majorization method for multidimensional scaling*. Journal of Classification, 5(2):163–180, 1988.
[31] 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.
[32] MPI.NET Home Page ://www.osl.iu.edu/research/mpi.net
[33] 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
[34] 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.
[35] J. B. Kruskal. *Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis*. Psychometrika, 29(1):1–27, 1964.
[36] 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.
[37] 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
[38] Hansjorg Klock, Joachim M. Buhmann, *Data visualization by multidimensional scaling: a deterministic annealing approach*, Pattern Recognition 33 (2000) 651}669
[39] 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.
[40] Dempster, A.P., Laird, N.M., & Rubin, D.B. (1977). *Maximum-likelihood from incomplete data via the EM algorithm*. J. R. Statist. Soc. Ser. B (methodological), 39, 1–38.
[41] Naonori Ueda and Ryohei Nakano *Deterministic annealing EM algorithm* Neural Networks Volume 11, Issue 2, 31 March 1998, Pages 271-282 ://dx.doi.org/10.1016/S0893-6080(97)00133-0
[42] Xiaohong Qiu and Geoffrey Fox, *Parallel Data Mining for Medical Informatics*, Technical Report January 13 2009 ://grids.ucs.indiana.edu/ptliupages/publications/DataminingMedicalInformatics.pdf

[43] 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 ://msdn.microsoft.com/robotics/

[44] 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.