Performance of Windows Multicore Systems on Threading and MPI

Judy Qiu[1,2], Seung-Hee Bae[1,2]
[1]Pervasive Technology Institute, [2]School of Informatics and Computing
Indiana University, Bloomington IN, 47408 USA
E-mail: xqiu, sebae @indiana.edu

Abstract

We present performance results on a Windows cluster with up to 768 cores using MPI and two variants of threading – CCR and TPL. CCR (Concurrency and Coordination Runtime) presents a message based interface while TPL (Task Parallel Library) allows for loops to be automatically parallelized. MPI is used between the cluster nodes (up to 32) and either threading or MPI for parallelism on the 24 cores of each node. We look at performance of two significant bioinformatics applications; gene clustering and dimension reduction. We find that the two threading runtimes offer similar performance with MPI outperforming both at low levels of parallelism but threading much better when the grain size (problem size per process/thread) is small. We develop simple models for the performance of the clustering code.

1. Introduction

Multicore technology is still rapidly changing at both the hardware and software levels and so it is challenging to understand how to achieve good performance especially with clusters when one needs to consider both distributed and shared memory issues. In this paper we look at both MPI and threading approaches to parallelism for two significant production datamining codes running on a 768 core Windows cluster. Efficient use of this code requires that one use a hybrid programming paradigm mixing threading and MPI. Here we quantify this and compare the threading model CCR (Concurrency and Coordination Runtime) that we have used for the last 3 years with Microsoft's more recent TPL Task Parallel Library.

Section 2 briefly presents both the clustering and dimension reduction applications used in this paper while section 3 summarizes the three approaches to parallelism – CCR, TPL and MPI – used here. Section 4 looks at the performance of the clustering application with the different software models and as a function of dataset size. We identify the major sources of parallel overhead of which the most important is the usual synchronization and communication overhead. We compare the measured performance with simple one and two factor models which describe most of the performance data well. Both CCR and the newer TPL perform similarly. In section 5, we discuss CCR v TPL on dimension reduction applications. Section VI has conclusions.

In this paper we mainly use a cluster Tempest which has 32 nodes made up of four Intel Xeon E7450 CPUs at 2.40GHz with 6 cores. Each node has 48 GB node memory and is connected by 20Gbps Infiniband. In section 5, we compare with a single AMD machine that is made up of four AMD Opteron 8356 2.3 GHz chips with 6 cores. This machine has 32 GB memory. All machines run Microsoft Window HPC Server 2008 (Service Pack 1) - 64 bit. Note all software was written in C# and runs in .NET3.5 or .NET4.0 (beta 2) environments

2. Applications

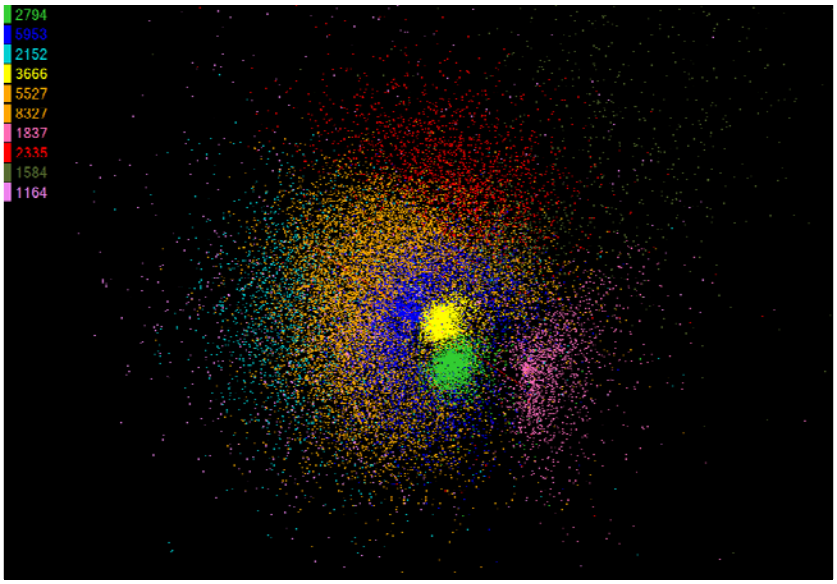

Figure 1 Clustering by Deterministic Annealing for 35339 AluY Sequences

We take our applications from a study of clustering and visualization of gene sequences. We have described in earlier publications, our approach to clustering using deterministic annealing[1-3]. This was introduced by Rose[4, 5] with Hofmann and Buhmann[6] providing a key extension to the "pairwise" case where the points to be clustered do not have known vector representations but rather all that is known is the dissimilarities (distances) between each pair of points. We have substantially improved the published algorithms and implemented efficiently using both MPI and threading. All our current published work[1-3, 7-9] has used Microsoft's CCR threading library[10].

Multidimensional scaling (MDS) [11, 12] aims to construct low dimensional mappings of the given high-dimensional data on the basis of the pairwise proximity information, while the pairwise Euclidean distance within the target dimension between two points is approximated to the corresponding original proximity value. In other words, MDS is a non-linear optimization problem with respect to the mapping in the target dimension and the given pairwise dissimilarity information. The STRESS[13] value is a well-known objective function of MDS, which needs to be minimized. STRESS is defined by:

$$\sigma(X) = \Sigma_{i<j\leq N} \; w_{ij}(d_{ij}(X) - \delta_{ij})^2 \qquad (1)$$

where w_{ij} is an arbitrary weight, $d_{ij}(X)$ is the Euclidean distance between two mapped vectors of x_i and x_j, and δ_{ij} is the corresponding original dissimilarity value.

Among many MDS solution, we use SMACOF algorithm which is based on Expectation Maximization (EM)-like iterative majorization method. For details of the SMACOF algorithm, please refer to [14].

The current paper uses samples of Alu repeats[15, 16] coming from the Human and Chimpanzee genomes. Typical result of this analysis is shown in fig. 1 with several identified clusters in the AluY family[1]. The visualization of this data in fig. 1 is an example of the value of dimension reduction. One can clearly see the clusters and one is not solely reliant on statistical measures to quantify them.

Both MDS and clustering algorithms are compute intensive as they are of $O(N^2)$ for N sequences and so we are motivated to seek both improved algorithms[2] and understand the performance of the current code[7-9]. We note the algorithm structure of MDS and deterministic annealing clustering are

rather different. Clustering is similar to classic $O(N^2)$ particle dynamics problems in structure while SMACOF MDS is built around iterative application of non square matrix multiplication.

We note that our conference paper[17] also compares CCR and TPL on a simple matrix multiplication kernel which we do not cover here.

3. Software Models

3.1. CCR (Concurrency and Coordination Runtime)

CCR[10, 18] is a CLR-based common language runtime that supports asynchronous messages for threads. We have discussed its syntax and capabilities in previous papers [3, 7-9]. It offers high performance ports with queues to support messaging between threads and much of its sophistication is not needed in this application. We have used CCR to implement classic MPI like communication patterns including pipeline, shift, rendezvous shift, and rendezvous exchange and it has given good performance [19]. It has been a very reliable tool used in our group for several years. As shown in Figs. 2 and 3, there is a non trivial amount of overhead in implementing a simple parallel loop that is needed 22 times in our clustering application. This does produce relatively ugly code and in fact the MPI version of this is much simpler as it just requires barrier and reduction calls.

MPI and CCR both require the user break up the loops explicitly to express the "data parallelism". The shared memory naturally supported by the threaded model improves both the readability and performance of those parts of the algorithm requiring communication in MPI. These are largely to support linear algebra – especially determination of leading eigenvalue/vector of a cluster correlation matrix.

```
CountdownLatch latch = CountdownLatch(threadCount);
Port<int> port = new Port<int>();

Arbiter.Activate(queue, Arbiter.Receive(true, port, delegate(int
dataBlockIndex)
{
    DataBlock dataBlock = _dataBlocks[MPIRank][dataBlockIndex];
    // do work
    latch.Signal()
}));

for (int dataBlockIndex = 0; dataBlockIndex < dataBlockCount;
dataBlockIndex++)
{
   port.Post(dataBlockIndex);
}

latch.Wait();
```

Fig. 2: Typical Structure of CCR code used in Clustering

```
ParallelOptions parallelOptions = new ParallelOptions();

parallelOptions.MaxDegreeOfParallelism = threadCount;

Parallel.For(0, dataBlockCount,  parallelOptions,
(dataBlockIndex) =>
{
    // do work
});
```

Fig. 3: Typical Structure of CCR code for Clustering

3.2. TPL (Task Parallel Library)

TPL[20] supports a loop parallelism model familiar from OpenMP[21]. Note TPL is a component of the Parallel FX library, the next generation of concurrency support for the Microsoft .NET Framework which supports additional forms of parallelism not needed in our application.

TPL contains sophisticated algorithms for dynamic work distribution and automatically adapts to the workload and particular machine so that the code should run efficiently on any machine whatever its core count. Note TPL involves language changes (unlike CCR which is a runtime library) and so implies that code only runs on Windows.

In Fig. 3 we give the pseudocode for a typical use of TPL in our application. It is clearly simpler than the CCR syntax in fig. 2 but does not help us maintain an OS independent source as it extends language in an idiosyncratic fashion. We note that complete clustering code had 22 separate “Parallel For” invocations while MDS had 7 separate "Parallel.For" subroutines in the parallel implementation; 4 of the MDS "Parallel.For" are called in the iterative part of SMACOF algorithm.

3.3. MPI (Message Passing Interface)

Our codes are implemented to use MPI to support the concurrency across nodes and in addition the threading models described above. The inter-node MPI implementation trivially can support parallelism within the node and that is used in the later studies. In sense, MPI is the “simplest” intra-node paradigm as it re-uses code that must be present anyway. If one only needs intra-node parallelism, then MPI would be more complex to code than the shared memory threading models CCR and TPL.

We have discussed elsewhere how extensions of MapReduce (Twister[22-24]) can be used to replace MPI but that is not the focus here. Twister has a more flexible communication model than MPI and that will lead to poorer performance.

4. Performance of Clustering Code on Tempest Cluster

4.1. Threading vs MPI with CCR and TPL Runtime

In Fig. 4, we show typical execution time measurements with combinations of inter-node parallelism implemented using MPI and intra-node parallelism using either threading (CCR or TPL) or MPI. Four labels are used in the legend to distinguish runtime modes where TPL and CCR represent the type of shared memory parallel runtime implementation, and MPI and Threaded suffixes differentiate whether intra-node parallelism is contributed by concurrent threads or parallel processes. For example, CCR-MPI or TPL-MPI means that the parallelism p is achieved by MPI processes only within a node and TPL-Threaded or CCR-Threaded means that shared memory parallelism is used instead. In the case of p=32 and n=16, TPL-MPI has 1 TPL thread and 2 MPI processes.

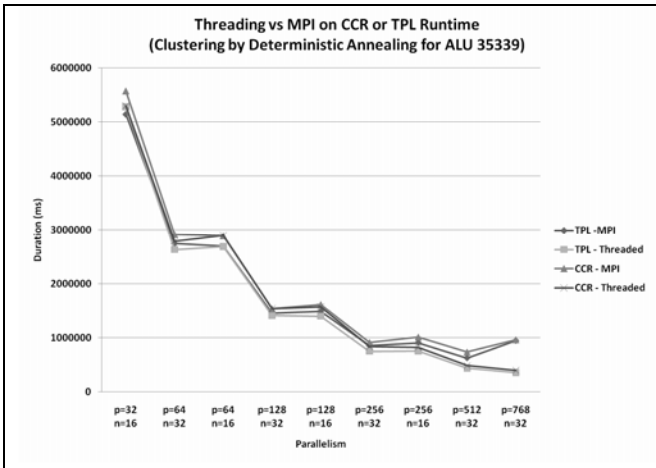

Fig. 4. Execution Time of Clustering Code for selected parallelism *p* and node counts *n*. Shown are CCR and TPL internal threading as well as intra-node MPI implemented in these two frameworks

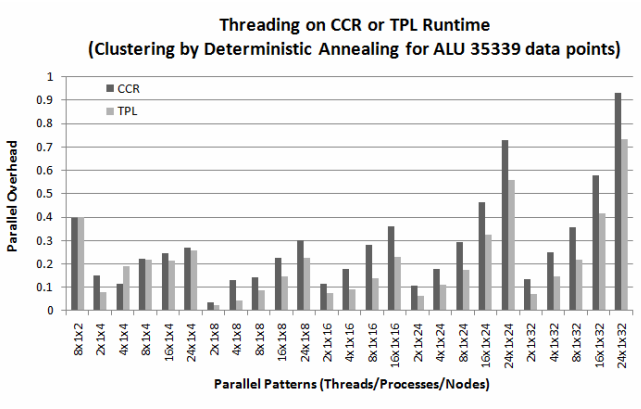

Fig. 5. Parallel Overhead for 35399 AluY sequence clustering for cases of pure threading in the node

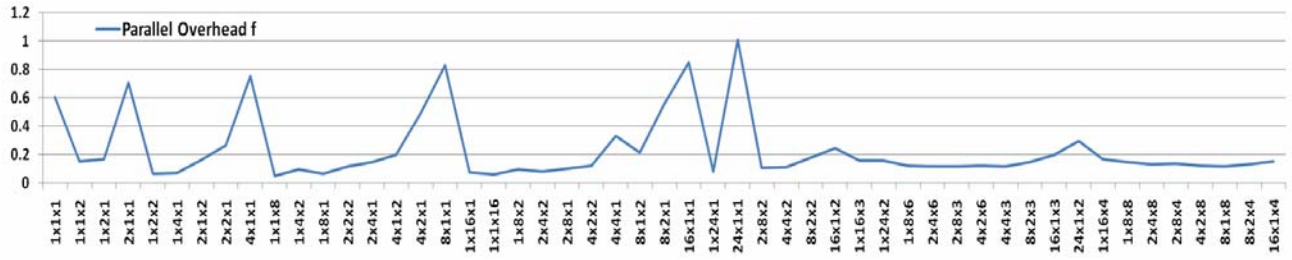

Fig. 6. Parallel Overhead *f* as a function of pattern *t*X*m*X*n* for a sample of 30,000 Metagenomics sequences. [1] (Selected on small parallel counts $p \leq 64$)

Comparable performance at low levels of parallelism is observed but that threading is much faster on the extreme case on left – 24 way internal parallelism on 32 nodes. We will explore this effect in more detail below. Also note here that TPL is a little faster than CCR even in case of internal MPI when there is only one thread per process. We convert execution time into an efficiency ε or an overhead *f* where

$$\varepsilon = S(p)/p \text{ or } (p_{ref}T(p_{ref}))/(pT(p)) \quad (2)$$

$$f = 1/\varepsilon - 1 = pT(p)/ (p_{ref}T(p_{ref})) - 1 \quad (3)$$

where T(p) is execution time on p processors and S(p) is speedup. Normally the reference process count is $p_{ref} = 1$ but we will sometimes use a larger value where execution with p=1 is inconvenient. Efficiency is usually between 0 and 1 but the overhead is unbounded and so when large is not as easy to plot. However *f* is linear in execution time of parallel code whereas efficiency is inversely proportional to T(p). Typically deviations from "perfect speedup" correspond to extra terms added into T(p) and often these overheads from load imbalance or communication are just additive quantities to T(p). Thus such overheads are easier to study in the overhead f than efficiency ε. Deviations from zero of *f* directly measure (scaled) overhead. We exploit this in section 4 where we

show a simple and expected model for f describes our measurements in "linear" region where overheads are small (and $f <\sim 1$).

Note that we label parallelism as tXmXn where

$$p = t\ m\ n \qquad (4)$$

Here each node has t threads or m MPI internal processes and the run involves n nodes. In most of data in this section either t or m is one i.e. we use pure MPI or pure threading in a node although we have performed a rather complete set of tests with both m and t not equal to 1 reported in [3].

4.2. Threading Internal to Node

In Fig. 5, we show a set of runs with pure threading in each node with different choices for thread count t and node count n. The overhead clearly increases as expected as one increases parallelism reaching (for TPL) 0.72 for a 768 core run. This corresponds to an efficiency of 58%. However the figure also shows a surprising increase at low parallelism values $n < 8$. This is a reproducible effect over several applications and corresponds to poor Windows performance where processes have large memory. The effect is shown in more detail in a sample from an earlier paper with Fig. 6 showing the overhead for many cases of low parallelism counts such as patterns 2x1x1, 4x1x1, 8x1x1, 16x1x1 and 24x1x1. This figure shows that here MPI internal (or external) to the node outperforms threading at low parallelism as it reduces per process memory size. The phenomenon is illustrated in patterns 1x2x1, 2x2x1, 4x2x1, 8x2x1, and 1x24x1.

Often the most important overhead in parallel computing is the time spent synchronizing or communicating in the parallel code. In the parallel clustering code with p parallel units and N points total, one stores N/p points in each parallel unit. Then for an $O(N^2)$ problem at each iteration, one communicates approximately N/p points between nodes and does a total of (N/p) times (N/p) computations involving the communicated points and those stored in each node. Letting n = N/p be the grain size, the overhead takes the form

$$f \propto n\ t_{comm}/[\ (n*n)\ t_{calc}\ ] \qquad (5)$$

$$\text{or} \quad f \propto 1/n \text{ or } f \propto \text{p at fixed data set size N}$$

The overall constant in (5) can be derived in terms of core hardware performance for calculation and communication. However in practice the exact value of f is hard to make quantitative Here we adopt a phenomenological view and take the functional dependence on p or n from (5) but fit the constant. In fact we take a general one or two-factor model

$$f = a_1\ x_1 \quad \text{or} \quad f = a_1\ x_1 + a_2\ x_2 \qquad (6)$$

where we take various choices for x_1 and x_2 and perform a simple one or two parameter least squares fit to find a_1 and a_2. We show the results of this analysis in Fig. 7 for the choices $x_1 = p$ and x_2 = node count N_{node} as the factors. This is performed separately for the CCR and TPL cases. Note the model describes the data quite well except for the case of low parallelism $N_{node} < 8$ where we had already suggested that the overhead was coming from a totally different effect (large process memory) than the usual communication and synchronization overheads that equs. (5) and (6) are designed to model. We note that a one factor model that only keeps the dependence on total parallelism gives similar quality fits to that with two factors – this is to be expected if one analyzes the natural forms of

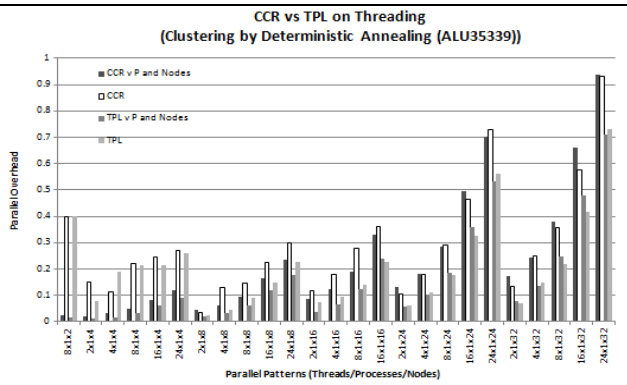

Fig. 7. The data of Fig. 5 compared with a simple model described in text for MPI and threading. For each pattern, we show in order the model CCR prediction, the measured CCR, the model TPL prediction and finally the measured TPL

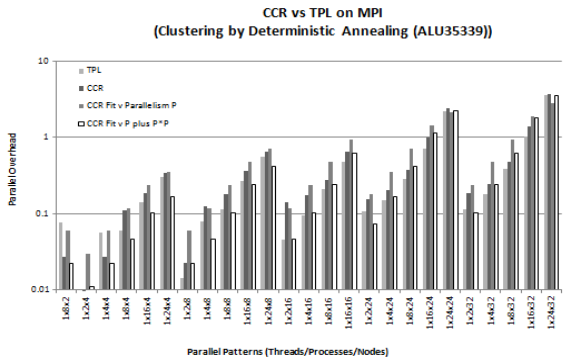

Fig. 10. Parallel Overhead for 35399 AluY sequence clustering for cases of pure MPI internal to the node. For each pattern, we show in order the measured TPL, the measured CCR, the single factor model CCR prediction and finally the two factor model CCR prediction.

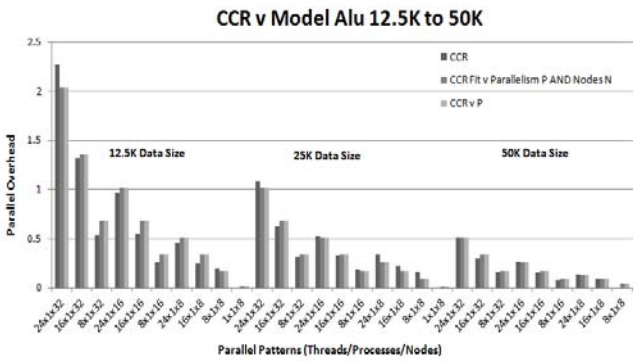

Fig. 8. Parallel Overhead *f* as a function of pattern *tXmXn* for three samples of respectively 12,500 25.000 and 50,000 AluY sequences in the case *m*=1 of threading internal to node. We show for each pattern, the CCR measurement followed by the two factor and single factor model.

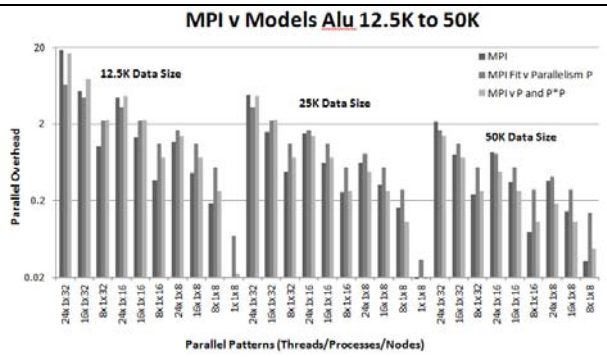

Fig. 11. Parallel Overhead *f* as a function of pattern *tXmXn* for three samples of respectively 12,500 25.000 and 50,000 AluY sequences in the case *t*=1 of MPI internal to node. We show for each pattern, the CCR measurement followed by the single factor and two factor model.

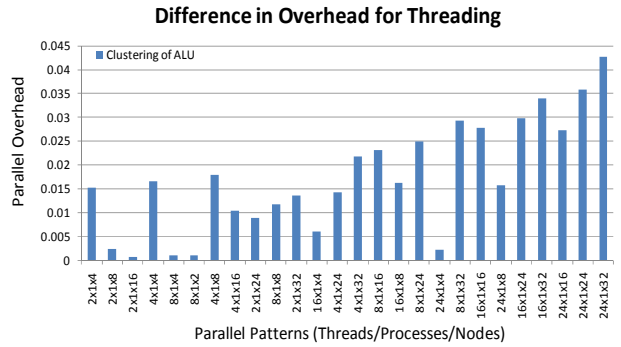

Fig. 9. Parallel Overhead difference CCR minus TPL for threading internal to node with Clustering by Deterministic Annealing for 35339 AluY Sequences

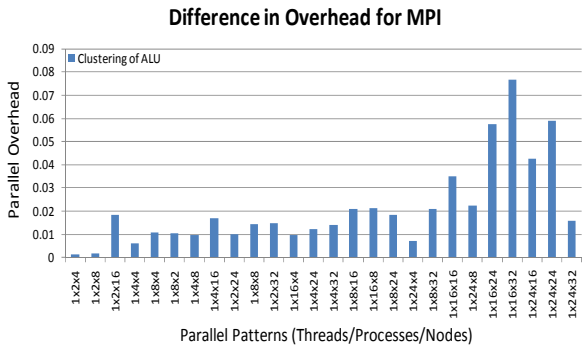

Fig. 12. Parallel Overhead difference CCR minus TPL for MPI internal to node with Clustering by Deterministic Annealing for 35339 AluY Sequences

overhead. We illustrate this in fig. 8 which compares one and two factor fits for another AluY sample chosen as it was homogeneous and could therefore be used to test data set size dependence of the performance. The one factor fit just uses $x_1 = p$ while the two factor fit uses $x_1 = p$ and $x_2 = N_{node}$. The two fits are indistinguishable and also simultaneously describe three dataset sizes with 12.5K, 25K and 50K points. Precisely we used a factor x_1 that was $1/n$ = Parallelism p/(Data set size) that is precisely the inverse of grain size. We note that TPL is usually faster than CCR although the difference is often small as seen in Fig. 9.

4.3. MPI Internal to Node

We now look at the analogous runs to the previous section but with pure MPI and not pure threading in each node. We still get results for both CCR and TPL as our code bases are implemented in the threading frameworks and can get some overheads even though the thread count is one in all cases. Fig. 10 plots the basic overhead measurements plus two models that we only apply to CCR case. One model has a single factor x_1 as the parallelism p and the second model has x_1 as p and x_2 as p^2. Again the models are approximately correct but now for all patterns as we have internal MPI parallelism, we do not have the large process memory effect at low parallelism values.

The simple linear fits are less good than for threading case. This is particularly clear in Fig. 11 which analyzes the dataset size dependence for MPI intra-node parallelism for the three AluY samples. Now the fits are significantly poorer than in Fig. 8. This is not surprising as the large size of the overhead makes it hard to justify a linear (or even quadratic) model. In Fig. 12, we show the small overhead increases for CCR compared to TPL in the case when MPI is used internal to a node.

5. Parallelization of Multidimensional Scaling (MDS)

5.1. Parallel Implementation of MDS

In order to analyze performance of two different shared memory parallel libraries CCR[10, 18] and TPL[20], we implement a well-known MDS application, called SMACOF[14] corresponding to weight function of 1 for all i, j in Eq. (1). We have parallelized this using MPI and threading but here only discuss the CCR and TPL – discussed in section 3 – parallelism of MDS on 24 core multicore systems with a standard shared memory model.

SMACOF[14] algorithm consists of iteration of the following three components: (1) non-square matrix multiplication, (2) calculating the objective function known as STRESS value, and (3) updating necessary matrices. the computational complexity of those three components is $\mathbf{O}(N^2)$. For parallelization of the non-square matrix multiplication, block decomposition is used for load balancing and efficient cache memory utilization. Fig. 13. illustrates block decomposition for the matrix multiplication.

For the load balancing, the N x D output matrix C is decomposed by b x D sub-blocks where N is the number of points, and b is the specified block size, and D is the target dimension. Then, each block C_i is assigned to one of the threads. Then each threads calculates the assigned sub-blocks C_i based on the block decomposition of Fig. 13. The following block matrix multiplication equation represents how compute C_i exactly:

$$C_i = \Sigma_j M_{ij} * X_j \quad (7)$$

The reason we use block matrix multiplication as in Eq. (7) instead of $C_i = M_i * X$ is to achieve better cache hit ratio, and we use $b = 64$ based on our previous experience in [25].

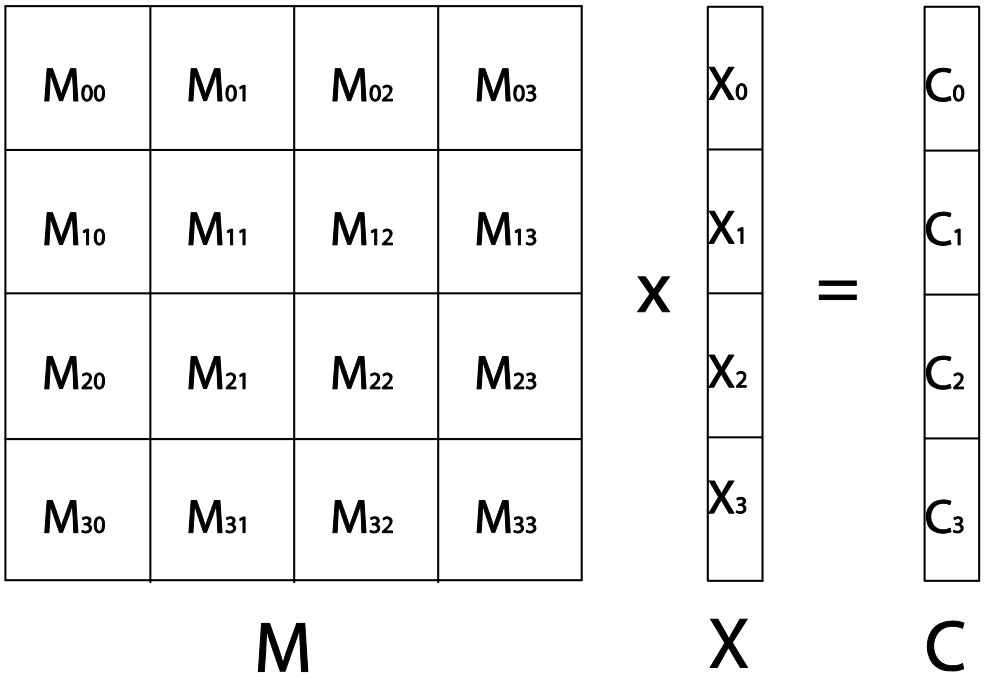

Fig. 13. Block decomposition for the non-square matrix multiplication.

```
inputs: pNum, length, myRank;
outputs: startOffset, endOffset, nRows;

        int nRows;

        if (length % pNum == 0)
        {
           nRows = length / pNum;
           startOffset = nRows * myRank;
           endOffset = startOffset + nRows;
        }
        else
        {
           if (myRank < pNum - (length % pNum))
           {
              nRows = length / pNum;
              startOffset = nRows * myRank;
              endOffset = startOffset + nRows;
           }
           else
           {
              nRows = length / pNum + 1;
              startOffset = nRows * myRank - pNum + (length % pNum);
              endOffset = startOffset + nRows;
           }
        }
```

Fig. 14. row-block assigning algorithm for load balancing of updating matrices and calculating STRESS.

In addition to parallelizing the non-square matrix multiplication, we need to parallelize the updating of matrices and calculation of the STRESS value for each iteration, since the time complexity of them is also $\mathbf{O}(N^2)$. Although we consider cache performance for the matrix multiplication part, cache effect is not a big issue for updating matrices and calculating STRESS since each element is accessed only once for those subroutines. Therefore, simple row-block assigning algorithm is used for the purpose of load balancing as in Fig. 14. *pNum* is the number of threads, *length* is equal to *N*, *myRank* is the ID of the current thread, and *nRows* is the number of rows assigned to the current threads. The difference of the number of assigned rows is at most one through the algorithm shown in Fig. 14.

5.2. Parallel Performance of MDS

In this paper, we compare the performance of the two different parallel runtimes for Microsoft .NET frameworks, CCR [10, 18] and TPL [20] by applying them to MDS application SMACOF [14] algorithm with the parallel approach given in Section 5.1. We explore the performances on two different 24-core systems. One node is made up of four Intel Xeon E7450 CPUs at 2.40GHz with 6 cores and 48 GB of main memory, and the other node is an AMD machine that is made up of four AMD Opteron 8356 2.3 GHz chips with 6 cores and 32 GB of main memory.

For the experiment, we run the SMACOF implemented with CCR and TPL (labeled as CCR-MDS and TPL-MDS, correspondingly) with three different scientific data sets that we have studied. Two of them are ALU sequence data with 3000 and 4499 points[3] (sequences), and the other is 10,000 health records[1] related to 8 clinical features, such as Body Mass Index (BMI) and blood pressures etc., as well as 97 environmental factors which is geographically related to the dataset, like greenness of neighborhood and the number of nearby fast food restaurants, and so on. Since MDS applications work based on pairwise dissimilarity information, we constructed pairwise dissimilarity matrices for ALU3000, ALU4499, and Patient10000 data set through different methods. For instance, in ALU3000 data case, a pairwise dissimilarity matrix was constructed by Manhattan distance (a.k.a. hamming distance) of each pair of sequences which contain about 950 DNA characters. For the Patient10000 data, we employed the Euclidean distance of normalized vector representation of 8 clinical features and 97 environmental factors to build the pairwise dissimilarity matrix.

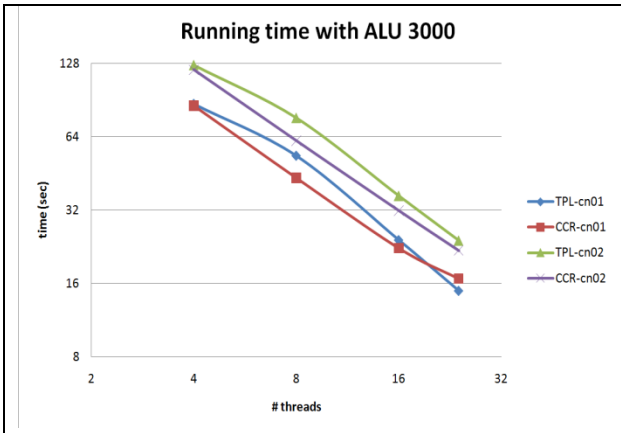

Fig. 15. Execution Time of CCR vs. TPL for Multi-Dimensional Scaling using 3000 Alu sequences on 24 core Intel vs. AMD machines

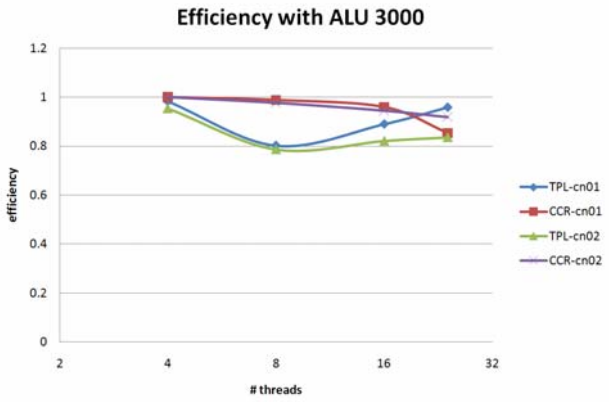

Fig. 16. Parallel Efficiency comparing CCR and TPL for Multi-Dimensional Scaling using 3000 Alu sequences on 24 core Intel and AMD machines

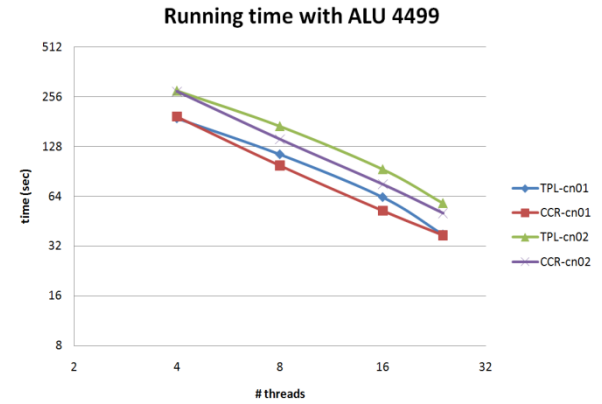

Fig. 17. Execution Time of CCR vs. TPL for Multi-Dimensional Scaling using 4499 Alu sequences on 24 core Intel vs. AMD machines

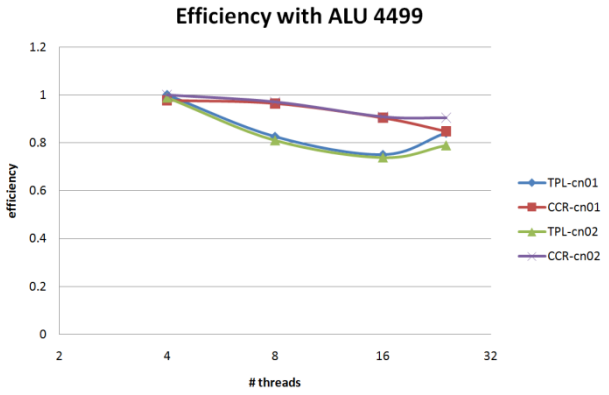

Fig. 18. Parallel Efficiency comparing CCR and TPL for Multi-Dimensional Scaling using 4499 Alu sequences on 24 core Intel and AMD machines

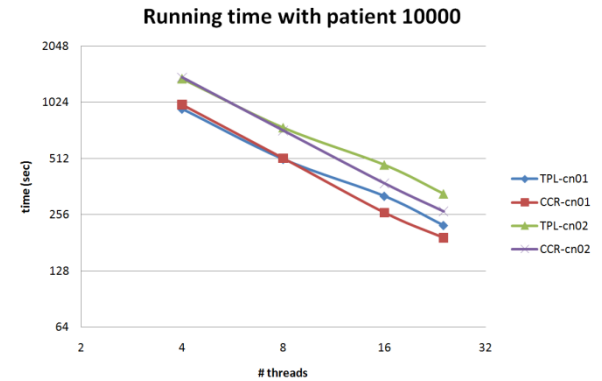

Fig. 19. Execution Time of CCR vs. TPL for Multi-Dimensional Scaling using 10,000 point patient data on 24 core Intel vs. AMD machines

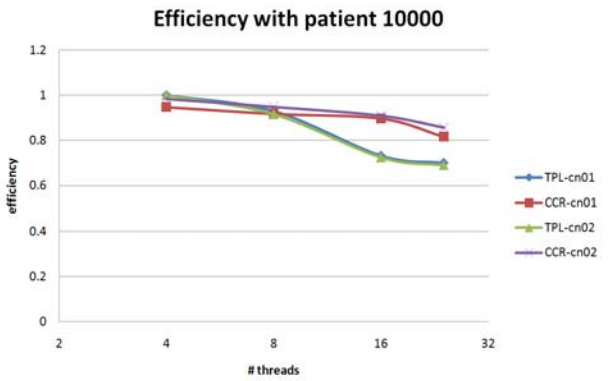

Fig. 20. Parallel Efficiency comparing CCR and TPL for Multi-Dimensional Scaling using 10,000 pt patient data on 24 core Intel and AMD machines

Fig. 15 and Fig. 17 describe the execution time of CCR-MDS and TPL-MDS on two test-beds, Intel Xeon 24-core node (hereafter **cn01**) and AMD Opteron 24-core node (hereafter **cn02**) for ALU3000 and ALU4499 datasets with respect to the number of threads, and Fig. 16 and Fig. 18 show the efficiency of execution time in Fig.15 and Fig.17, correspondingly. In Fig.16, CCR-MDS shows high efficiency in most cases except 24-thread cases, and it is expected. In Fig. 18, however, TPL-MDS shows somewhat more overhead with 8-/16-thread cases with both ALU3000 and ALU4499 datasets, although it shows almost comparable efficiency to CCR-MDS with 24-thread cases. Execution time and its efficiency of CCR-MDS and TPL-MDS with 10,000 point patient data are illustrated by Fig. 19 and Fig. 20, respectively. Again, CCR-MDS shows almost linear running time and high efficiency in Fig. 20. In contrast, efficiency of TPL-MDS drops to around 70% with 16-/24-thread cases on both Intel and AMD machines. Another interesting feature is that both CCR-MDS and TPL-MDS achieve better performance on the Intel Xeon CPU node than the AMD Opteron CPU node.

In summary, we implemented SMACOF algorithm in parallel with CCR and TPL parallel runtime and find that CCR is comparable to and better than TPL for MDS implementation with single node thread only parallelism. CCR is particularly better than TPL on the larger thread counts of 16 or 24 cores. Also, both parallel runtimes perform better on Intel CPU node than on AMD CPU node.

6. Conclusions

We have examined parallel programming tools supporting Microsoft Windows environment for both distributed and shared memory. We show that the new TPL Task Parallel Library produces simpler code than the older CCR runtime. Good performance on the cluster of 24 core nodes requires use of a hybrid programming paradigm using MPI between nodes and threading internal to the node. We are able to describe both MPI and threading overheads with a simple single factor model with a linear dependence on the inverse grain size (number of data points in each thread). This breaks down when the overhead gets very large and also at small levels of parallelism when Windows performs poorly with large memory processes. TPL is slightly faster than CCR on the clustering example but the opposite is true for MDS.

Acknowledgements

We would like to thank Microsoft for their collaboration and support. Tony Hey, George Chrysanthakopoulos and Henrik Frystyk Nielsen played key roles in providing technical support. We appreciate our collaborators from IU School of Informatics and Computing. Haixu Tang and Mina Rho gave us important feedback on Alu and Metagenomics data and we would like to thank co-authors on our conference paper[17] for their earlier work on which we build.

References

1. Geoffrey Fox, Xiaohong Qiu, Scott Beason, Jong Youl Choi, Mina Rho, Haixu Tang, Neil Devadasan, and Gilbert Liu, "Biomedical Case Studies in Data Intensive Computing," in *Proceedings of The 1st International Conference on Cloud Computing (CloudCom 2009)*, Martin Jaatun, Gansen Zhao, and Chunming Rong, Editors. December 1-4, 2009, Springer Verlag LNCS "Cloud Computing": Vol. 5931. Beijing Jiaotong University, China.

2. Judy Qiu, Jaliya Ekanayake, Thilina Gunarathne, Jong Youl Choi, Seung-Hee Bae, Yang Ruan, Saliya Ekanayake, Stephen Wu, Scott Beason, Geoffrey Fox, Mina Rho, and Haixu Tang, "Data Intensive Computing for Bioinformatics," in *Data Intensive Distributed Computing*. 2010, IGI Publishers.
3. Geoffrey Fox, Seung-Hee Bae, Jaliya Ekanayake, Xiaohong Qiu, and H. Yuan, "Parallel Data Mining from Multicore to Cloudy Grids," book chapter of *High Speed and Large Scale Scientific Computing*. 2009: IOS Press, Amsterdam. ISBN:978-1-60750-073-5
4. Kenneth Rose, Eitan Gurewitz, and Geoffrey C Fox, "Statistical mechanics and phase transitions in clustering," *Phys. Rev. Lett.*, Aug, 1990. **65**: p. 945--948.
5. Ken Rose, "Deterministic Annealing for Clustering, Compression, Classification, Regression, and Related Optimization Problems," *Proceedings of the IEEE*, 1998. ***86***: p. 2210--2239.
6. Hofmann T. and Buhmann J. M., "Pairwise data clustering by deterministic annealing," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 1997. ***19***: p. 1--14.
7. Xiaohong Qiu , Geoffrey C. Fox, Huapeng Yuan, Seung-Hee Bae, George Chrysanthakopoulos, and Henrik Frystyk Nielsen, "Parallel Data Mining on Multicore Clusters," in *Proceedings of 7th International Conference on Grid and Cooperative Computing GCC2008*. October 24-26, 2008. Shenzhen China.
8. Xiaohong Qiu, Geoffrey C. Fox, Huapeng Yuan, Seung-Hee Bae, George Chrysanthakopoulos, and Henrik Frystyk Nielsen, "Parallel Clustering and Dimensional Scaling on Multicore Systems," in *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)*. June 3 - 6, 2008. Nicosia, Cyprus.
9. Xiaohong Qiu, Geoffrey C. Fox, Huapeng Yuan, Seung-Hee Bae, George Chrysanthakopoulos, and Henrik Frystyk Nielsen, "Performance of Multicore Systems on Parallel Data Clustering with Deterministic Annealing," in *Proceedings of International Conference on Computational Science (ICCS 2008),* Kraków, POLAND. pages. 407-416.
10. *Concurrent Affairs: Concurrency and Coordination Runtime. Jeffrey Richter, Mirosoft. Available from: http://msdn.microsoft.com/en-us/magazine/cc163556.aspx*.
11. Kruskal, J.B. and M. Wish, *Multidimensional Scaling*. 1978: Sage Publications Inc.
12. Borg, I. and P.J. Groenen, *Modern Multidimensional Scaling: Theory and Applications*. 2005: Springer
13. Kruskal, J.B., "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis," *Psychometrika*, 1964. ***29***: p. 1-27.
14. Jan de Leeuw, "Applications of convex analysis to multidimensional scaling," *Recent Developments in Statistics*, 1977: p. 133-145.
15. Batzer MA and Deininger PL, "Alu repeats and human genomic diversity," *Nature Reviews Genetics*, 2002. **3**(5): p. 370-379.
16. Smit, A., R. Hubley, and P. Green. *RepeatMasker Open-3.0.* 1996-2010 [accessed 2010 November 26]; Available from: http://www.repeatmasker.org.
17. Judy Qiu, Scott Beason, Seung-Hee Bae, Saliya Ekanayake, and Geoffrey Fox, "Performance of Windows Multicore Systems on Threading and MPI," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. 2010, IEEE Computer Society. pages. 814-819. DOI: 10.1109/ccgrid.2010.105.

18. Georgio Chrysanthakopoulos and Satnam Singh, "An Asynchronous Messaging Library for C#," in *Proceedings of Synchronization and Concurrency in Object-Oriented Languages (SCOOL) workshop at OOPSLA*. October, 2005. San Diego, CA.
19. Xiaohong Qiu, Geoffrey Fox, Huapeng Yuan, Seung-Hee Bae, George Chrysanthakopoulos, Henrik Frystyk Nielsen, "High Performance Multi-Paradim Messaging Runtime Integrating Grids and Multicore Systems", in *proceedings of eScience 2007 Conference*, Bangalore India Dec. 10-13 2007.
20. Daan Leijen and Judd Hall. *Parallel Performance: Optimize Managed Code For Multi-Core Machinesl*. 2007 October [accessed 2010 November 26]; Available from: http://msdn.microsoft.com/en-us/magazine/cc163340.aspx.
21. *OpenMP API specification for parallel programming*. [accessed 2010 November 26]; Available from: http://openmp.org/wp/.
22. Bingjing Zhang, Yang Ruan, Tak-Lon Wu, Judy Qiu, Adam Hughes, and Geoffrey Fox, "Applying Twister to Scientific Applications," in *CloudCom 2010*. Nov. 30-Dec. 3, 2010. Indianapolis, Indiana.
23. J.Ekanayake, H.Li, B.Zhang, T.Gunarathne, S.Bae, J.Qiu, and G.Fox, "Twister: A Runtime for iterative MapReduce," in *Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010*. 2010, ACM. Chicago, Illinois.
24. SALSA Group. *Iterative MapReduce*. 2010 [accessed 2010 November 7]; Twister Home Page Available from: http://www.iterativemapreduce.org/.
25. Seung-Hee Bae, "Parallel Multidimensional Scaling Performance on Multicore Systems," in *Proceedings of the Advances in High-Performance E-Science Middleware and Applications workshop (AHEMA) of Fourth IEEE International Conference on eScience*. 2008. Indianapolis: IEEE Computer Society. pages. pp. 695-702.