Performance of Windows Multicore Systems on Threading and MPI

Judy Qiu[1], Scott Beason[1], Seung-Hee Bae[1,2], Saliya Ekanayake[1,2], Geoffrey Fox[1,2]
[1]*Pervasive Technology Institute,* [2]*School of Informatics and Computing*
Indiana University, Bloomington IN, 47408 USA
Email: {xqiu, smbeason, sebae, sekanaya, gcf@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 use a simple matrix multiplication kernel as well as a significant bioinformatics gene clustering application. We find that the two threading models offer similar performance with MPI outperforming both at low levels of parallelism but threading much better when the grain size (problem size per process) is small. We find better performance on Intel compared to AMD on comparable 24 core systems. We develop simple models for the performance of the clustering code.**

Multicore, Performance, Threading, MPI, and Windows

I. 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 a significant production datamining code 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) used for the last 3 years with Microsoft's new TPL Task Parallel Library.

Section II briefly presents the clustering application used in this paper while section III summarizes the three approaches parallelism – CCR, TPL and MPI – used here. Section IV is the heart of paper and 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 V, we extend study to a matrix multiplication kernel running on single node Intel and AMD 24 core systems where CCR outperforms TPL. 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 16 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.

II. Applications

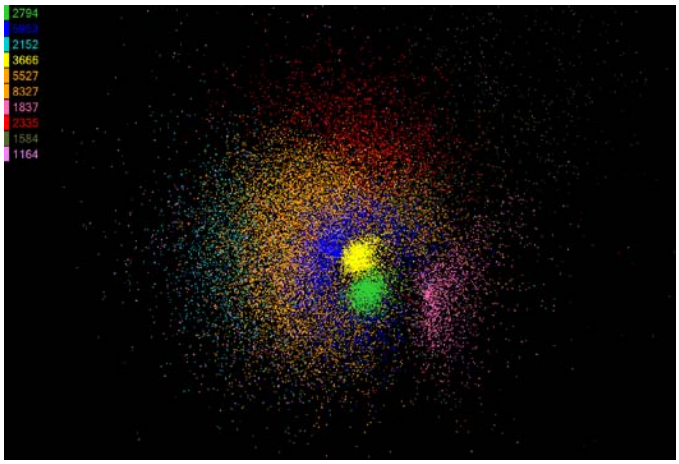

Figure 1. Clustering by Deterministic Annealing for 35339 AluY Sequences

We have described in earlier publications [1, 2, 4], our approach to clustering using deterministic annealing. This was introduced by Rose [5, 6] with Hofmann and Buhmann [7] 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 has used Microsoft's CCR threading library [8, 9].

The current paper uses two samples of Alu repeats [10-12] 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 [2]. The algorithm is

compute intensive as it is of $O(N^2)$ for N sequences and so we are motivated to seek both improved algorithms [1] and understand the performance of the current code [13-15].

III. Software Models

A. CCR (Concurrency and Coordination Runtime)

CCR [8, 9] has been a very reliable tool used in our group for several years and giving good performance. We have discussed its syntax and capabilities in previous papers [4, 13-15]. It offers high performance ports with queues to support messaging between threads and much of its sophistication is not needed in this application. As shown in Fig. 2, there is a non trivial amount of overhead in implementing a simple parallel loop that is needed 22 times in our application. This does produce relatively ugly code and in fact the MPI version of this is much simpler as it at most requires barrier 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();
```

Figure. 2 Typical Structure of CCR code used in Clustering code

B. TPL (Task Parallel Library) MPI (Message Passing Interface)

TPL [16] supports a loop parallelism model familiar from OpenMP [17]. 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.

```
ParallelOptions parallelOptions = new ParallelOptions();

parallelOptions.MaxDegreeOfParallelism = threadCount;

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

Figure 3. Typical Structure of TPL code used in Clustering code

C. 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 (i-Mapreduce) [1] [3] can be used to replace MPI but that is not the focus here. i-MapReduce has a more flexible communication model than MPI and that will lead to poorer performance.

IV. Performance of Clustering Code on Tempest Cluster

A. CCR (Concurrency and Coordination Runtime)

In Fig. 4, we show typical execution time measurements with the parallelism between nodes implemented using MPI and that internal to node implemented with either threading or MPI. One sees approximate equality in performance at low parallelism 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

$$\varepsilon = S(p)/p \text{ or } (p_{ref}T(p_{ref}))/(pT(p)) \quad (1)$$

$$f = 1/\varepsilon - 1 = pT(p)/ (p_{ref}T(p_{ref})) - 1 \quad (2)$$

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. Efficiency is usually between 0 and 1 but the overhead is unbounded and so can confuse plots. However *f* is linear in overheads as it is linear in execution time and so it can be simpler to model as we see below. Note that we label parallelism as *t*X*m*X*n* where

$$p = t\ m\ n \quad (3)$$

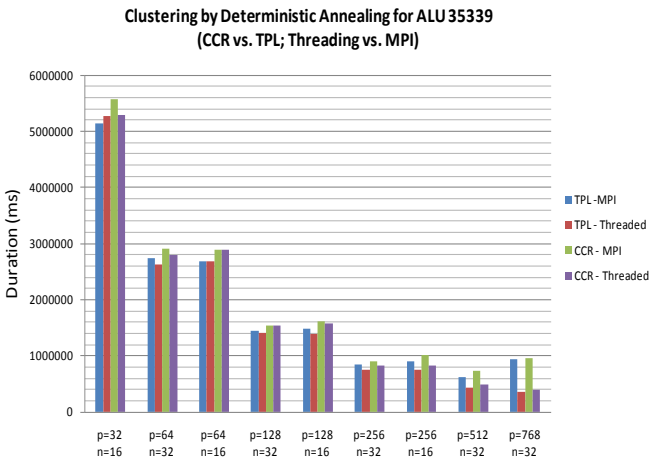

Figure 4. Execution Time of Clustering Code for selected parallelism *p* and node counts *n*. Shown are CCR and TPL internal threading as well as intra-node MPI implemented in these two frameworks

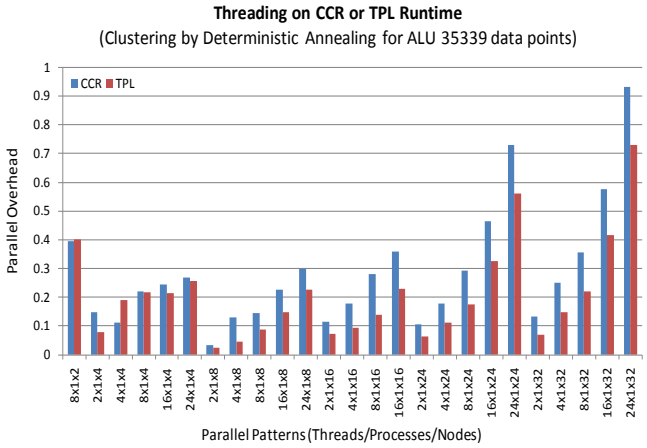

Figure 5. Parallel Overhead for 35399 AluY sequence clustering for cases of pure threading in the node

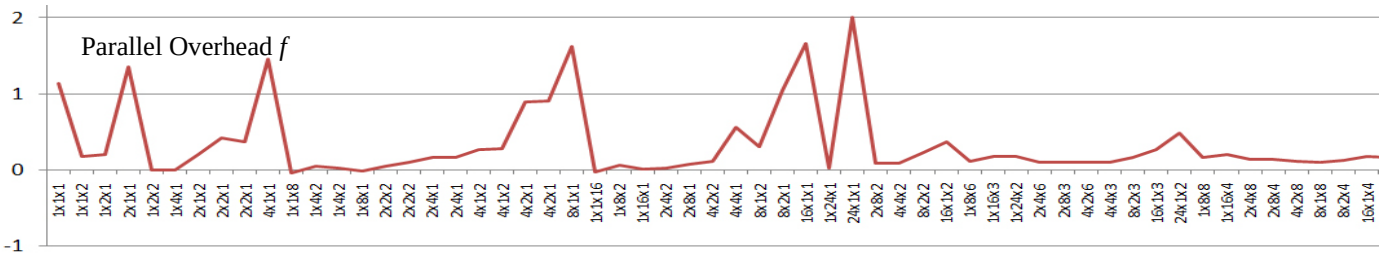

Figure 6. Parallel Overhead *f* as a function of pattern *t*X*m*X*n* for a sample of 30,000 Metagenomics sequences. [2] (Selected on small parallel counts $p \leq 64$)

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.

B. 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. 4 showing the overhead for many cases of low parallelism counts. This figure shows that here MPI internal (or external) to the node outperforms threading as it reduces the process memory size. 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 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 p while the two factor fit uses p and n. The two fits are indistinguishable and also simultaneously describe three dataset sizes with 12.5K, 25K and 50K points. Precisely we used a factor that was Parallelism p/(Data set size) that is precisely the inverse of grain size.

Typical overheads in parallel computing are proportional to a ratio of communication and computation times. These involve primitive times multiplied by the complexity of the calculation divided by the computation. Pairwise Clustering is an $O(N^2)$ algorithm (where N is number of points) for which the complexity is just proportional to the inverse of the grain size (number of points in each thread or process). This leads to the expectation that the overhead *f* is linear in the parallelism measure *p*. Here we adopt a phenomenological two-factor model

$$f = a_1 x_1 + a_2 x_2 \qquad (4)$$

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. 5 for the choices *p* and node count *n* 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 < 8 where we had already suggested that the overhead was coming from a totally different effect (large

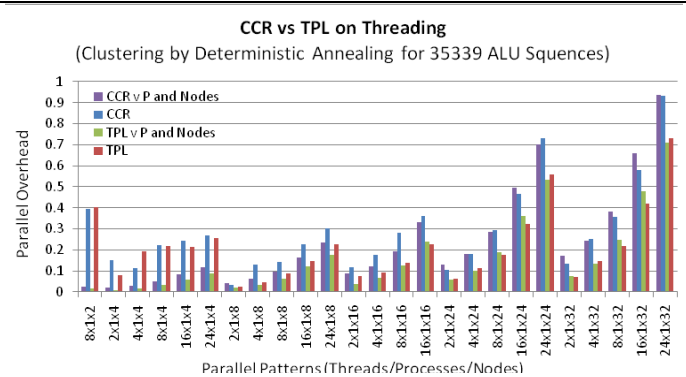

Figure 7. The data of Fig. 5 compared with a simple model described in text for MPI and threading. For each pattern, we show in order the model CCR prediction, the measured CCR, the model TPL prediction and finally the measured TPL

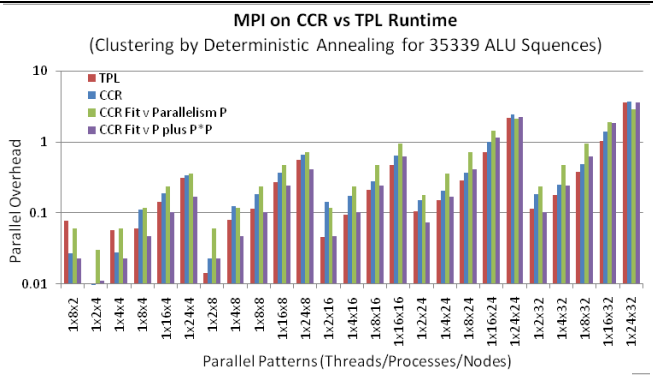

Figure 10. Parallel Overhead for 35399 AluY sequence clustering for cases of pure MPI internal to the node. For each pattern, we show in order the measured TPL, the measured CCR, the single factor model CCR prediction and finally the two factor model CCR prediction.

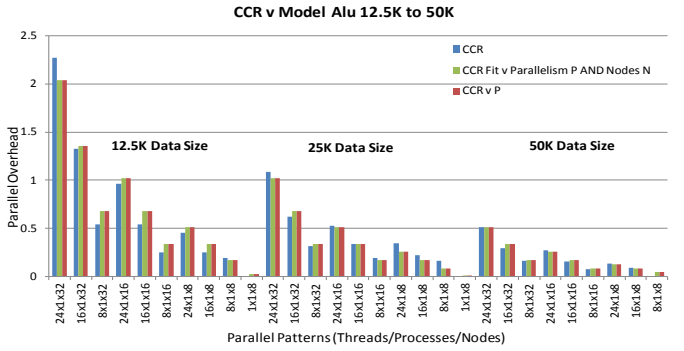

Figure 8. Parallel Overhead *f* as a function of pattern *t*X*m*X*n* for three samples of respectively 12,500 25.000 and 50,000 AluY sequences in the case *m*=1 of threading internal to node. We show for each pattern, the CCR measurement followed by the two factor and single factor model.

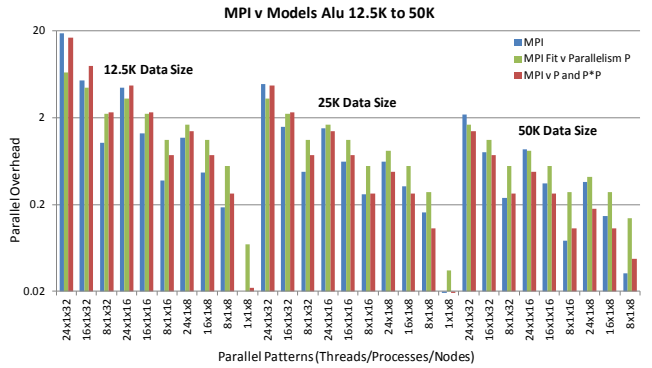

Figure 11. Parallel Overhead *f* as a function of pattern *t*X*m*X*n* for three samples of respectively 12,500 25.000 and 50,000 AluY sequences in the case *t*=1 of MPI internal to node. We show for each pattern, the CCR measurement followed by the single factor and two factor model.

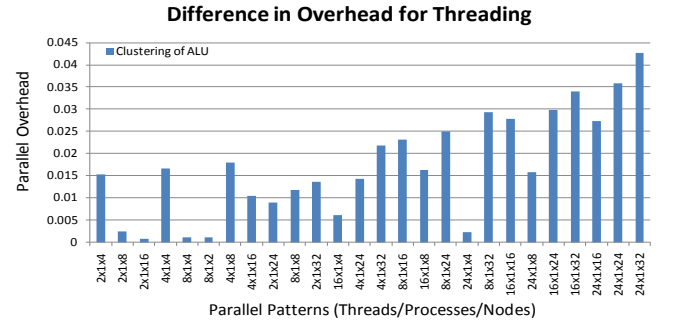

Figure 9. Parallel Overhead difference CCR minus TPL for threading internal to node with Clustering by Deterministic Annealing for 35339 AluY Sequences

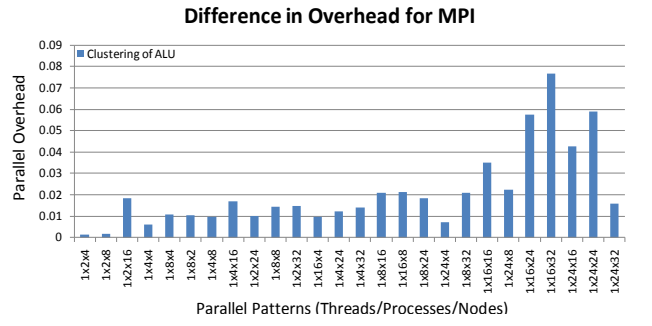

Figure 12. Parallel Overhead difference CCR minus TPL for MPI internal to node with Clustering by Deterministic Annealing for 35339 AluY Sequences

process memory) than the usual communication and synchronization overheads that (4) attempts 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 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 p while the two factor fit uses p and n. The two fits are indistinguishable and also simultaneously describe three dataset sizes with 12.5K, 25K and 50K points. Precisely we used a factor that was Parallelism *p*/(Data set size) that is 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.

C. 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 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 3 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.

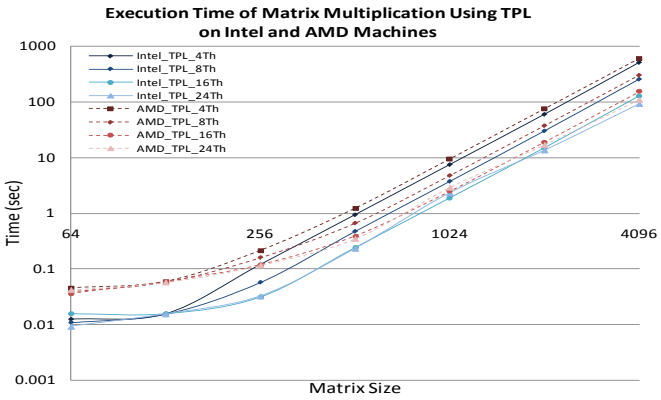

Figure 13. Execution time of TPL for matrix multiplication kernel on single node 24 core Intel and AMD machines for the cases of 4, 8, 16 and 24 concurrent threads.

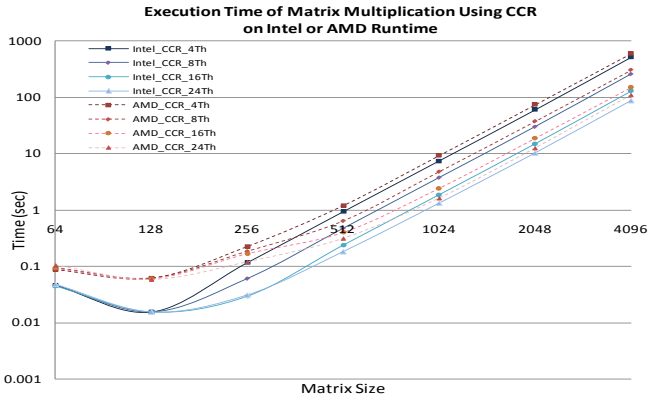

Figure14. Execution time of CCR for matrix multiplication kernel on single node 24 core Intel and AMD machines for the cases of 4, 8, 16 and 24 concurrent threads.

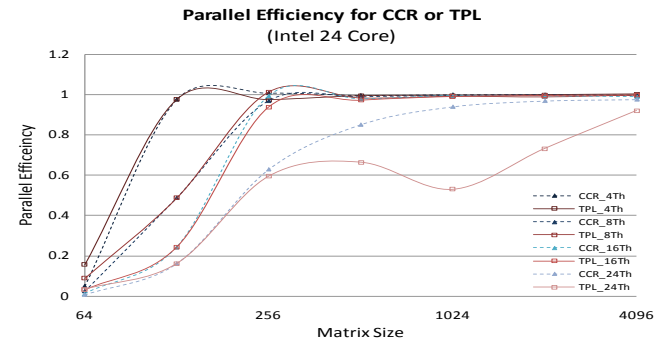

Figure 15. Parallel efficiency (1) comparing CCR and TPL on an Intel 24 core node with 4, 8, 16 and 24 concurrent threads.

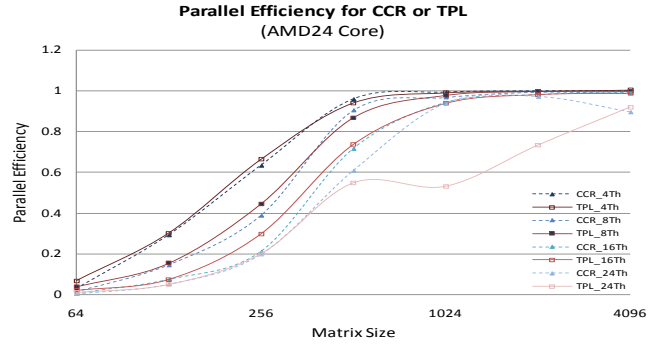

Figure 16. Parallel efficiency (1) comparing CCR and TPL on an AMD 24 core node with 4, 8, 16 and 24 concurrent threads

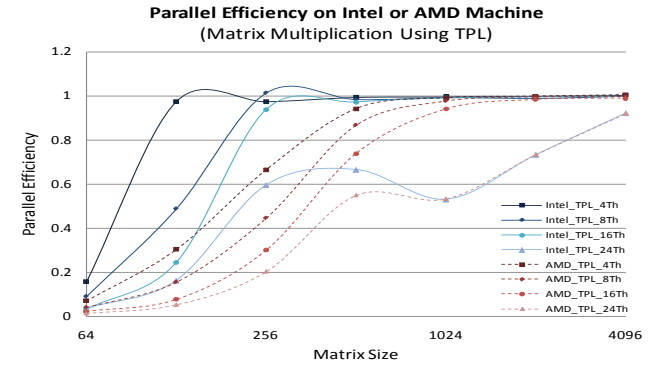

Figure 17. Parallel efficiency (1) comparing AMD and Intel 24 core nodes with 4, 8, 16 and 24 concurrent threads using TPL framework

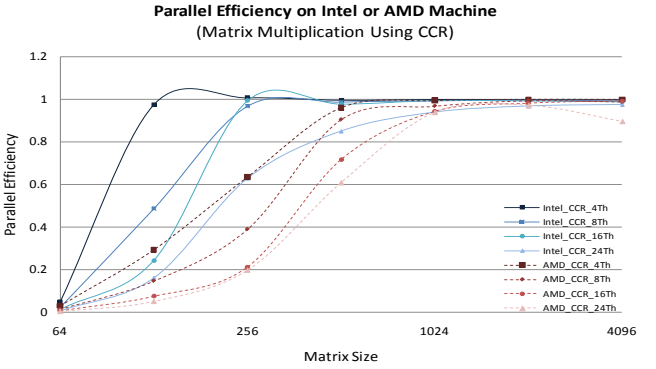

Figure 18. Parallel efficiency (1) comparing AMD and Intel 24 core nodes with 4, 8, 16 and 24 concurrent threads using CCR framework

V. Matrix Multiplication

A. Comparison of CCR and TPL

In section IV, we looked at a relatively complex "real" application and here we examine the comparison between CCR and TPL on a simple kernel matrix multiplication. This has been extensively studied under MPI and so here we just compare the two threading environments CCR and TPL on a single 24 core node. We compare the Intel and AMD models detailed in section 1 and consider four cases – 4 8 16 or 24 threads. The raw results in figures 13 and 14 show that the Intel machine slightly outperforms the AMD one.

We now use (1) to calculate efficiencies and discuss them in Figs. 15 and 16 in a way that allows comparison of CCR and TPL with Fig. 15 comparing them on the Intel and Fig. 16 the AMD node. Unlike section IV, we see that CCR typically outperforms TPL although the effect is often small. Further TPL outperforms CCR on the AMD machine for smaller thread counts. We also see smooth results except for the 24 core case where efficiencies show a strange shape that we need to investigate further.

If we compare the code structure for the applications of sections IV and V, we see matrix multiplication is very structured and totally load balanced. Thus the dynamic tasking of TPL has no advantages over the static user generated decomposition used in CCR. However the clustering algorithm has some inhomogeneity and may be benefiting from the dynamic TPL features.

B. Comparison of Intel and AMD

Figs. 17 and 18 compare for TPL and CCR respectively the efficiencies for Intel and AMD that were shown on separate graphs in figures 15 and 16. We see that the Intel efficiencies are strikingly better than those on AMD machine achieving efficiencies near 1 at lower matrix sizes (by approximately a factor of 4).

VI. 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 and slightly better performance 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. In future work, we will extend our analysis to other applications including those that are memory bandwidth limited. In future work, we will extend our analysis to other applications including those that are memory bandwidth limited.

Acknowledgment

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.

References

[1] Judy Qiu, Jaliya Ekanayake, Thilina Gunarathne, Jong Youl Choi, Seung-Hee Bae, Yang Ruan, Saliya Ekanayake, Stephen Wu, Scott Beason, Geoffrey Fox, Mina Rho, Haixu Tang, "Data Intensive Computing for Bioinformatics", submitted as a book chapter for book "Data Intensive Distributed Computing", IGI Publishers, 2010.

[2] Fox, G., Qiu, X., Beason, S., Choi, J. Y., Rho, M., Tang, H., et al. (2009). "Biomedical Case Studies in Data Intensive Computing," in Proceedings of The 1st International Conference on Cloud Computing (CloudCom 2009). Springer Verlag.

[3] i-MapReduce Home Page. http://www.iterativemapreduce.org.

[4] G. Fox, S.H. Bae, J. Ekanayake, X. Qiu, H. Yuan Parallel Data Mining from Multicore to Cloudy Grids Proceedings of HPC 2008 High Performance Computing and Grids workshop. Cetraro, Italy. July 3 2008.

[5] K. Rose, "Deterministic Annealing for Clustering, Compression, Classification, Regression, and Related Optimization Problems", Proceedings of the IEEE, vol. 80, pp. 2210-2239, November 1998.

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

[7] T Hofmann, JM Buhmann "Pairwise data clustering by deterministic annealing", IEEE Transactions on Pattern Analysis and Machine Intelligence 19, pp1-13 1997

[8] Microsoft Robotics Studio is a Windows-based environment that includes end-to-end Robotics Development Platform, lightweight service-oriented runtime, and a scalable and extensible platform. For details, see http://msdn.microsoft.com/robotics/

[9] 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.

[10] M.A. Batzer, P.L. Deininger, 2002. "Alu Repeats And Human Genomic Diversity." Nature Reviews Genetics 3, no. 5: 370-379. 2002

[11] A. F. A. Smit, R. Hubley, P. Green, 2004. Repeatmasker. http://www.repeatmasker.org

[12] J. Jurka, 2000. Repbase Update: a database and an electronic journal of repetitive elements. Trends Genet. 9:418-420 (2000).

[13] 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) Nicosia, Cyprus June 3 - 6, 2008.

[14] 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 Kraków, Poland; June 23-25, 2008. Springer Lecture Notes in Computer Science Volume 5101, pages 407-416, 2008. DOI: http://dx.doi.org/10.1007/978-3-540-69384-0_46

[15] 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 .

[16] Daan Leijen and Judd Hall, "Optimize Managed Code For Multi-Core Machines" http://msdn.microsoft.com/en-us/magazine/cc163340.aspx

[17] The OpenMP parallel programming API http://openmp.org/wp/