Performance of Multicore Systems on Parallel Datamining Services

Xiaohong Qiu, Geoffrey C. Fox, Huapeng Yuan, Seung-Hee Bae, George Chrysanthakopoulos, Henrik Frystyk Nielsen[1]

Abstract*—Multicore systems are of growing importance and 64-128 cores can be expected in a few years. We expect datamining to be an important application class of general importance and are developing such scalable parallel algorithms for managed code (C#) on Windows. We present a performance analysis that compares MPI and a new messaging runtime library CCR (Concurrency and Coordination Runtime) with Windows and Linux and using both threads and processes. We investigate effects of cache lines and memory bandwidth and fluctuations of run times of loosely synchronized threads. We give results on message latency and bandwidth for two processor multicore systems based on AMD and Intel architectures with a total of four and eight cores. Generating up to a million messages per second on a single PC, we find on an Intel dual quadcore system, latencies from 5µs in basic asynchronous threading to 20 µs for a full MPI_SENDRECV exchange with all threads (one per core) sending and receiving 2 messages at a traditional MPI style loosely synchronous rendezvous. We compare our C# results with C using MPICH2 and Nemesis and Java with both mpiJava and MPJ Express. We are packaging our core algorithms not as traditional libraries but as services and use DSS (Decentralized System Services built on CCR) to compose workflows or mashups for complete applications. We show initial results from GIS and Cheminformatics clustering problems. Our results suggest that the service composition model and Windows/C# thread programming model will be a flexible parallel programming environment for important commodity applications. C

***Index Terms*—Cache, Datamining, MPI, Multicore, Parallel Computing , Performance, Threads, Windows**

I. INTRODUCTION

Multicore architectures are of increasing importance and are impacting client, server and supercomputer systems [1-6]. They make parallel computing and its integration with large systems of great importance as "all" applications need good performance rather than just the relatively specialized areas covered by traditional high performance computing. We suggest that one needs to look again at parallel programming environments and runtimes and examine how they can support a broad market. In this paper we consider datamining as an application that has broad applicability and could be important on tomorrow's client systems as one supports "user expert assistants" that help the user by analyzing the "deluge" of data from sensors or just the internet connection. Perhaps on the 128 core client PC of 7 years hence, most of the cores would be spent on speculative and directed data analysis. Such applications are likely to be written in managed code (C#, Java) and run on Windows (or equivalent client OS for Mac) and use threads. This scenario is suggested by the recent RMS analysis by Intel [5]. It is interesting that the parallel kernels of most datamining algorithms are similar to those well studied by the high performance (scientific) computing community and often need the synchronization primitives supported by MPI.

In other papers [7-9] we have explained our hybrid programming model SALSA (Service Aggregated Linked Sequential Activities) that builds libraries as a set of services and uses simple service composition to compose complete applications [10]. Each service then runs on parallel on any number of cores – either part of a single PC or spread out over a cluster. The performance requirements at the service layer are less severe than at the "microscopic" thread level for which MPI is designed and where this paper concentrates. We use DSS (Decentralized System Services) which offers good performance with messaging latencies of 35 μs between services on a single PC [9]. Each service consists of parallel threads or processes that are synchronized in our case by Locks, MPI or a novel messaging runtime library CCR (Concurrency and Coordination Runtime) developed by Microsoft Research [11-15]. In this paper we explore these different synchronization overheads and the effects of operating system and the use of threads or processes.

In this paper, we present the performance analysis for C# and Java on both Windows and Linux and identify new features that have not been well studied for parallel scientific applications. This worked was performed on a set of multicore commodity PC's summarized in Table I. The results can be extended to clusters as we are using similar messaging runtime but we focus in this paper on the new results seen on the multicore systems.

[1] This work was partially supported by Microsoft Corporation. X. Qiu xqiu@indiana.edu is with Research Computing UITS, Indiana University Bloomington. G. C. Fox gcf@indiana.edu, H. Yuan yuanh@indiana.edu, and S. Bae sebae@indiana.edu are with Community Grids Laboratory Indiana University Bloomington. G. Chrysanthakopoulos georgioc@microsoft.com, and H. Frystyk Nielsen henrikn@microsoft.com are from Microsoft Research Redmond WA.

TABLE I: MULTICORE PC'S USED IN PAPER

AMD4: HPxw9300 workstation, 2 AMD Opteron CPUs Processor 275 at 2.19GHz, L2 Cache 2x1MB (for each chip), Memory 4GB. XP Pro 64bit and Windows Server 2003
Intel4: Dell Precision PWS670, 2 Intel Xeon CPUs at 2.80GHz, L2 Cache 2x2MB, Memory 4GB. XP Pro 64bit
Intel8a: Dell Precision PWS690, 2 Intel Xeon CPUs E5320 at 1.86GHz, L2 Cache 2x4M, Memory 8GB. XP Pro 64bit
Intel8b: Dell Precision PWS690, 2 Intel Xeon CPUs x5355 at 2.66GHz, L2 Cache 2X4M, Memory 4GB. Vista Ultimate 64bit and Fedora 7
Intel8c: Dell Precision PWS690, 2 Intel Xeon CPUs x5345 at 2.33GHz, L2 Cache 2X4M, Memory 8GB. Redhat

Section II discusses the CCR runtime and section III our motivating clustering datamining applications. These results identify some important benchmarks covering cache and memory effects, runtime fluctuations and synchronization costs discussed in sections IV-VII. Conclusions are in Section VIII.

All results and benchmark codes presented are available from http://www.infomall.org/salsa [16]

II. OVERVIEW OF CCR

CCR provides a framework for building general collective communication where threads can write to a general set of ports and read one or more messages from one or more ports. The framework manages both ports and threads with optimized dispatchers that can efficiently iterate over multiple threads. All primitives result in a task construct being posted on one or more queues, associated with a dispatcher. The dispatcher uses OS threads to load balance tasks. The current applications and provided primitives support a dynamic threading model with capabilities that include:

1) *FromHandler:* Spawn threads without reading ports
2) *Receive:* Each handler reads one item from a single port
3) *MultipleItemReceive:* Each handler reads a prescribed number of items of a given type from a given port. Note items in a port can be general structures but all must have same type.
4) *MultiplePortReceive:* Each handler reads a one item of a given type from multiple ports.
5) *JoinedReceive:* Each handler reads one item from each of two ports. The items can be of different type.
6) *Choice:* Execute a choice of two or more port-handler pairings
7) *Interleave:* Consists of a set of arbiters (port -- handler pairs) of 3 types that are Concurrent, Exclusive or Teardown (called at end for clean up). Concurrent arbiters are run concurrently but exclusive handlers are not.

One can spawn handlers that consume messages as is natural in a dynamic search application where handlers correspond to links in a tree. However one can also have long running handlers where messages are sent and consumed at a rendezvous points (yield points in CCR) as used in traditional MPI applications. Note that "active messages" correspond to the spawning model of CCR and can be straightforwardly supported. Further CCR takes care of all the needed queuing and asynchronous operations that avoid race conditions in complex messaging. CCR is attractive as it supports such a wide variety of messaging from dynamic threading, services (via DSS described in [9]) and MPI style collective operations.

CODE SAMPLE 1: MPI EXCHANGE PATTERN IN CCR

Main Routine for Exchange Pseudocode {
Create CCR dispatchers to control threads
Create a queue to hold tasks
Set up start ports with MPI initialization data such as thread number
Invoke handlers (MPI threads) on start ports
} **End Main Routine**

MPI logical thread Pseudocode (Arguments are start port contents) {
Calculate nearest neighbors for exchange collective
Loop over stages { Post information to 2 ports that will be read by left and right neighbors
yield return on CCR MultipleItemReceive will wait till this thread's information is available in its ports and continue execution after reading 2 ports

Do computation for this stage
} *End loop over stages*

Each thread sends information to ending port and thread 0 only does *yield return* on CCR MultipleItemReceive to collect information from all threads to complete run after reading from one port for each thread (this is a reduction operation).
} End MPI Thread

For our performance comparisons with MPI, we needed rendezvous semantics which are fully supported by CCR and we chose to use patterns corresponding to the MPI_SENDRECV interface with either toroidal nearest neighbor shift or the combination of a left and right shift, namely an Exchange where each process (thread) sends and receives two messages. Note that posting to a port in CCR corresponds to a MPISEND and the matching MPIRECV is achieved from arguments of handler invoked to process the port. MPI has a much richer set of defined methods that describe different synchronicity options, various utilities and collectives. These include the multi-cast (broadcast, gather-scatter) of messages with the calculation of associative and commutative functions on the fly. It is not clear what primitives and indeed what implementation will be most effective on multicore systems [2, 17] and so we only looked at a few simple but representative cases in this initial performance study. In fact it is possible that our study which suggests one can support in the same framework a set of execution models that is broader than today's MPI, could motivate a new look at messaging standards for parallel computing. We used built in CCR primitives for the shift and reduction operations but exploited CCR's ability to construct customized collectives sketched in Code Sample I to implement the MPI Exchange pattern. An important innovation of the CCR is to allow sequential, asynchronous

computation without forcing the programmer to write callbacks, or continuations, and at the same time not blocking an OS thread. This allows the CCR to scale to tens of millions of pending I/O operations, but with code that reads like synchronous, blocking operations.

Note that all our work was for managed code in C# which is an important implementation language for commodity desktop applications although slower than C++. In this regard we note that there are plans for a C++ version of CCR which would be faster but prone to traditional un-managed code errors such as memory leaks, buffer overruns, memory corruption. The C++ version could be faster than the current CCR but eventually we expect that the C# CCR will be within 20% of the performance of the C++ version. CCR has been extensively applied to the dynamic threading characteristic of today's desktop application but its largest use is in the Robotics community. One interesting use is to add an efficient port-based implementation of "futures" to C#, since the CCR can easily express them with no modifications in the core runtime. CCR is very portable and runs on both CE (small devices) and desktop windows.

DSS sits on top of CCR and provides a lightweight, REST oriented application model that is particularly suited for creating Web-style applications as compositions of services running in a distributed environment and its use in SALSA is described in [9].

III. CLUSTERING APPLICATION

We are building a suite of data mining services to test the runtime and two layer SALSA programming model. We start with data clustering which has many important applications including clustering of chemical properties which is an important tool [18] for finding for example a set of chemicals similar to each other and so likely candidates for a given drug. We are also looking at clustering of spatial information and in particular properties derived from the US Census data. We use a modification of the well known K-means algorithm [19], deterministic annealing [20], that has good convergence and parallelization properties. For a set of data points $\underline{x}$ and cluster centers $\underline{y}$, one gradually temperature T and iteratively calculates:

$$\underline{y} = \sum_x p(x,y)\, \underline{x}$$
$$p(x,y) = \exp(-d(x,y)/T)\, p(x) / Z_x \qquad (1)$$
$$\text{with } Z_x = \sum_y \exp(-d(x,y)/T)$$

Here d(x,y) is the distance defined in space where clustering is occurring. Parallelism can be implemented by dividing points $\underline{x}$ between the cores and there is a natural loosely synchronous barrier where the sums in each core are combined in a reduction collective to complete (1). Such parallel applications have a well understood performance model that can be expressed in terms of a parallel overhead *f(n,P)* where putting *T(n,P)* as the execution time on *P* cores or more generally processes/threads, we can define:

$$\textit{Overhead } f(n,P) = (PT(n,P)-T(Pn,1))/T(Pn,1) \qquad (2)$$

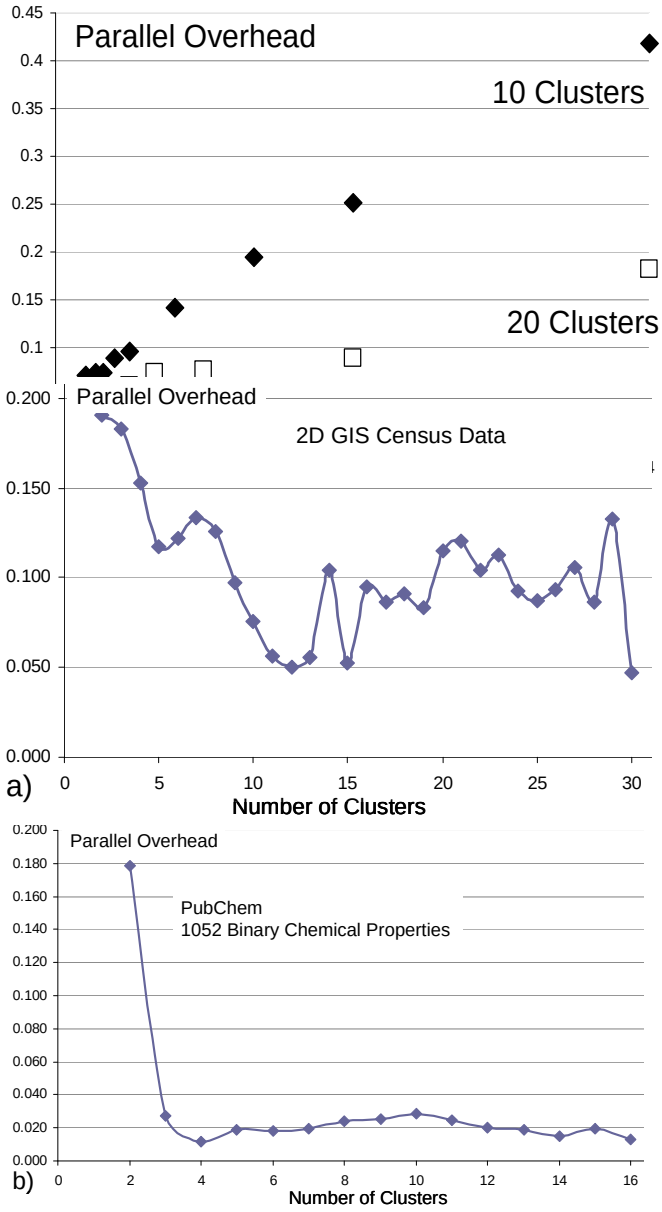

Fig. 2. Parallel Overhead defined in (2) as a function of the number of clusters for a) 2 dimensional GIS data for Indiana in over 200,000 blocks and 40,000 chemical compounds each with 1052 binary

$$\textit{Efficiency } \varepsilon = 1/(1+f)$$

For the algorithm of (1), *f(n,P)* should depend on the grain size *n* where each core handles n data points and in fact *f(n,P)* should decrease proportionally to the reciprocal of the grain size with a coefficient that depends on synchronization costs [6, 21-23. This effect is clearly seen in fig. 1 presented already in [9] although surprisingly we do not find *f(n,P)* tending to zero as *n* increases. Rather it rather erratically wanders around a small number 0.02 to 0.1 as parameters are varied. The overhead also decreases as shown in fig. 1 as the number of clusters increases as it is easy to show from (1) as the basic

computation is proportional to number of points *n* multiplied by the number of clusters. In fig. 2 we plot the parallel overhead as a function of the number of clusters for two large real problems coming from Census data and chemical property clustering. These clearly show the rather random behavior after *f(n,8)* decreases to a small value corresponding to quite good parallelism – speedups of over 7 on 8 core systems. The results in fig. 2(b) show lower asymptotic values which were determined to correspond to the binary data used in Chemistry clustering. This problem showed fluctuations similar to 2(a) if one used floating point representation for the Chemistry "fingerprint" data. Of course the binary choice shown in fig. 2(b) is fastest and the appropriate approach to use.

In the following section, we follow up on various details of these measurements examining the different components that effect performance

IV. CACHE EFFECTS ON PERFORMANCE

We found it quite hard to get reliable timing and identified two sources addressed here and in Sect. VII. The largest effect which is straightforward to address comes from the nature of the cache on all machines listed in table I. If different cores access different variables but those stored in the same cache line, then wild execution timer fluctuations can occur. These are documented by a simple computation that calculates concurrent summations and stores them in an array element A(*i*) for thread *i*. The summation used a random number generator to avoid being compiled away and can be downloaded from our web site [16]. This natural implementation leads to an order of magnitude increase in run time over an implementation that stores results in A(S*i*) where the separator S is chosen so that adjacent elements of A are separated by 64 bytes or more. These results are documented in Table II that records the execution time as a function of S and as a function of several machine and operating system choices. One sees good performance with modest fluctuations as long as S corresponds to a separation of 64 bytes or more. On the other hand in most cases the performance is dreadful and fluctuations sometimes large for separations S less than 64 bytes (the columns labeled 1 and 4 in units of double variables – 8 bytes – in Table II). This effect is independent of synchronization used (compare CCR and Locks in Table II) and is presumably due to the cache design on these modern multicore systems. Looking at the separation of 8 or 1024 doubles in Table II, one can see that with compilers we used, C was much faster than C# and Linux faster than Windows. Most remarkably the Redhat Linux results do not show the degradation of performance seen for Windows for separation of 1 or 4 doubles.

TABLE II: CACHE LINE COMPUTATION TIMES

Machine	**OS**	**Run Time**	**Thread Array Separation (unit is 8 bytes)**							
			1		**4**		**8**		**1024**	
			Mean (µs)	Std Dev Mean (µs)	Mean (µs)	Std Dev Mean (µs)	Mean (µs)	Std Dev Mean (µs)	Mean (µs)	Std Dev Mean (µs)
Intel8b	Vista CCR	C# CCR	8.03	.029	3.04	.059	0.884	.0051	0.884	.0069
	Vista	C# Locks	13.0	.0095	3.08	.0028	0.883	.0043	0.883	.0036
	Vista	C	13.4	.0047	1.69	.0026	0.66	.029	0.659	.0057
	Fedora	C	1.50	.01	0.69	.21	0.307	.0045	0.307	.016
Intel8a	XP CCR	C#	10.6	.033	4.16	.041	1.27	.051	1.43	.049
	XP Locks	C#	16.6	.016	4.31	.0067	1.27	.066	1.27	.054
	XP	C	16.9	.0016	2.27	.0042	0.946	.056	0.946	.058
Intel8c	Redhat	C	0.441	.0035	0.423	.0031	0.423	.0030	0.423	.032
AMD4	WinServer 2003	C# CCR	8.58	.0080	2.62	.081	0.839	.0031	0.838	.0031
		C# Locks	8.72	.0036	2.42	.01	0.836	.0016	0.836	.0013
		C	5.65	.020	2.69	.0060	1.05	.0013	1.05	.0014
	XP	C# CCR	8.58	.0080	2.62	.081	0.839	.0031	0.838	.0031
		C# Locks	8.72	.0036	2.42	.01	0.836	.0016	0.836	.0013
		C	5.65	.020	2.69	.0060	1.05	.0013	1.05	.0014

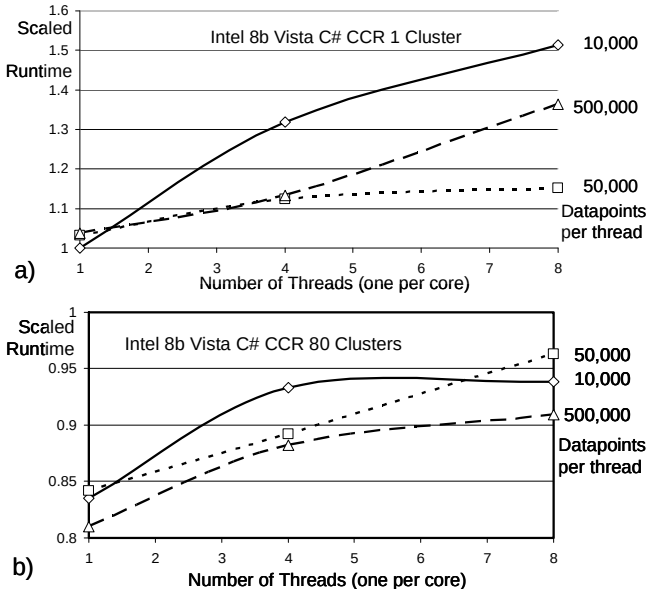

Fig. 3. Scaled Run time on Intel8b using Vista and C# with CCR for synchronization on Clustering Kernel for three dataset sizes with 10,000 50,000 or 500,000 points per thread(core). Each measurement involved averaging over at least 1000 computations separated by synchronization whose cost is not included in results

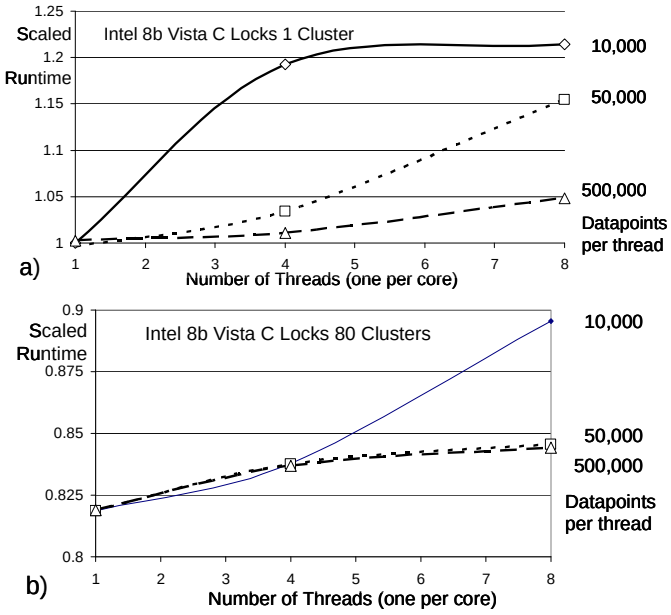

Fig. 4. Scaled Run time on Intel8b using Vista and C with locks for synchronization on Clustering Kernel for three dataset sizes with 10,000 50,000 or 500,000 points per thread (core).

The Fedora Linux results on Intel 8b rather remarkably lie in between those of Windows and Redhat in Table II showing a factor of 5 difference between separation 1 and 8 whereas Redhat has only a 5% effect while Windows varies widely with an upto factor of 15 effect. Thus although the cache hardware architecture produces this effect, its impact is very systems software dependent. We are obviously able to program around this feature but it is unfortunate as using A(*i*) to store results from thread *i* is surely a natural strategy. This effect is in fact known [24] but its implications are often not properly implemented. For example the C# math random number generator makes the mistake of using such an array and so has unnecessarily poor performance.

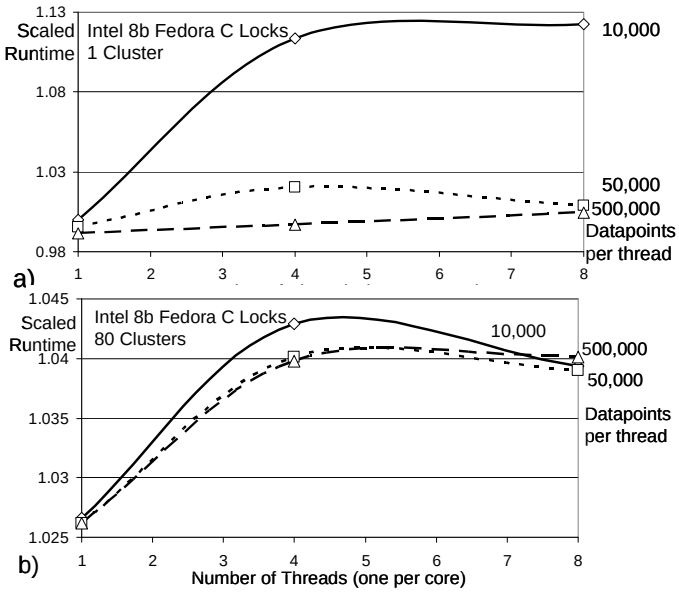

Fig. 5. Scaled Run time on Intel8b using Fedora Linux and C with locks for synchronization on Clustering Kernel for three dataset sizes with 10,000 50,000 or 500,000 points per thread (core).

V. MEMORY AND THREAD PERFORMANCE

In figs. 3, 4 and 5 we isolate the kernel of the clustering algorithm of sec. II and examine its performance as a function of grain size *n*, number of clusters and number of cores. In all cases we use the scaled speed up scenario and measure thread dependence at three fixed values of grain size *n* (10,000, 50,000 and 500,000). All results are divided by the number of clusters, the grain size, and the number of cores and scaled so the 10,000 data point, one cluster, one core result becomes 1. These figures then immediately allow us to identify key features of the computation. We display cases for 1 cluster where memory bandwidth effects could be important and also for 80 clusters where such effects are small as one performs 80 floating point operations on every variable fetched from memory. The three figures have typical results covering respectively Windows and C#, Windows and C and finally Linux and C. Always we use threads not processes and C uses Locks and C# uses CCR synchronization. Data is stored so as to avoid any of cache line effects discussed in the previous section.

The results foe one cluster clearly show the effect of memory bandwidth with scaled run time increasing significantly as one increases the number of cores used. In this benchmark the memory demands scale directly with number of cores. Indeed a major concern with multicore system is the need for a memory bandwidth that increases linearly with the number of cores. In fig. 4 we see a 50% increase in the run time for a grain size of 10,000. This is for C# and Windows and the

overhead is reduced to 22% for C on Windows and 13% for C on Linux. Further we note that naively the 10,000 data point case should get excellent performance as the dataset can easily fit in cache and minimize memory bandwidth needs. These results again (see section IV) illustrate that the current multicore hardware and software cache architecture is highly unsuitable for this style of application. As we believe datamining is likely to be a critical type of application on future machines, we suggest serious attention be given to these problems.

We get modest overheads for 80 clusters in all cases which is in fact why the applications of sect. II run well. There are no serious memory bandwidth issues in cases with several clusters and in this case that dominates the computation. This is usual parallel computing wisdom; real size problems run with good efficiency as long as there is plenty of computation. [6, 21-23] The datamining cases we are studying (Clustering, EM based Mixture models, Hidden Markov Methods) satisfy this and will run well on machines expected in next 5 years.

VI. SYNCHRONIZATION PERFORMANCE

The synchronization performance has been discussed in detail earlier for CCR where we discussed both dynamic threading showing it had an approximate 5µs overhead and MPI style behavior [9]. Here we detail in table III some comparisons for the MPI Exchange operation running on the maximum number of cores (4 or 8) available on the systems of Table I. Results for the older Intel8a are also in [16]. In each we use a zero size message.

TABLE III: MPI EXCHANGE LATENCY

Machine	OS	Runtime	Grains	Latency µs
Intel8c: gf12	Redhat	MPJE	8 Procs	181
		MPICH2	8 Procs	40.0
		MPICH2 Fast Option	8 Procs	39.3
		Nemesis	8 Procs	4.21
Intel8c: gf20	Fedora	MPJE	8 Procs	157
		mpiJava	8 Procs	111
		MPICH2	8 Procs	64.2
Intel8b	Vista	MPJE	8 Procs	170
	Fedora	MPJE	8 Procs	142
		mpiJava	8 Procs	100
	Vista	CCR	8 Thrds	20.2
AMD4	XP	MPJE	4 Procs	185
	Redhat	MPJE	4 Procs	152
		mpiJava	4 Procs	99.4
		MPICH2	4 Procs	39.3
	XP	CCR	4 Thrds	16.3
Intel4	XP	CCR	4 Thrds	25.8

Note that the CCR Exchange operation timed above has the full messaging transfer semantics of the MPI standards but avoids the complexity of some MPI capabilities like tags [25-33]. We expect that future simplified messaging systems that like CCR span from concurrent threads to collective rendezvous's will chose such simpler implementations. Nevertheless we think that Table III is a fair comparison. Note that in the "Grains" column, we list number of concurrent activities and if they are threads or processes. These measurements correspond to synchronizations occuring roughly every 30µs and were averaged over 500,000 such synchronizations in a single run. The optimized Nemesis version of MPICH2 gives best performance while CCR with for example 20µs latency on Intel8b, outperforms "vanilla MPICH2". We see from Table I that C on Linux is much faster on computation than C# and we can expect as discussed in Section II, CCR and C# to improve and compete in performance with the better optimized (older) languages.

We were surprised by the uniformly poor performance of MPI with Java. Here the old mpiJava invokes MPICH2 from a Java-C binding wile MPJ Express [28] is pure Java., It appears threads in Java currently are not competitive in performance with those in C#. Perhaps we need to revisit the goals of the old Java Grande activity [34]. As discussed earlier we expect managed code to be of growing importance as client multicores prolifergate.

VII. PERFORMANCE FLUCTUATIONS

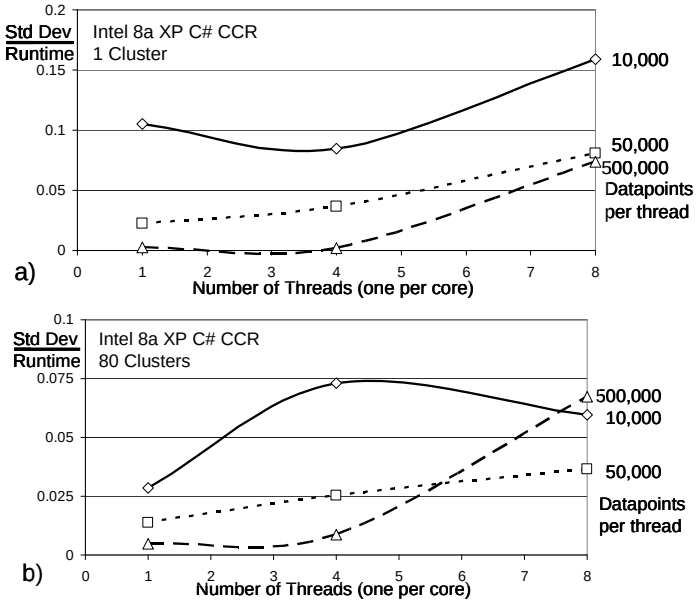

Fig. 6. Ratio of Standard Deviation to mean of thread execution time averaged over 1000 instances using XP on Intel 8a and C# with CCR for synchronization on Clustering Kernel for three dataset sizes with 10,000 50,000 or 500,000 points per thread (core).

We already noted in Sect III that our performance was impacted by flluctuations in run time that were bigger than seen in most parallel computing studies that typically look at Linux and processes whereas our results are mainly for Windows and threads. In figures 6, 7 and 8 we present some results quantifying this using the same "clustering kernel" introduced in Sect V. We average results over 1000

synchronization points in a single run. In figs. 6 and 7 we calculate the standard deviation of the 1000P samples if P cores are used. In the final fig. 8, we calculate the the value of the Maximum minus the Minimum execution time of the P concurrent threads between synchronization points and average over the 1000 samples. By definition, this measurement must give zero for one thread whereas the that used in figs. 6 and 7 can be nonzero even for one thread. Our results show much larger run time fluctuations for Windows than for Linux. And we believe this effect leads to the 2-10% parallel overheads seen already in fig. 2.

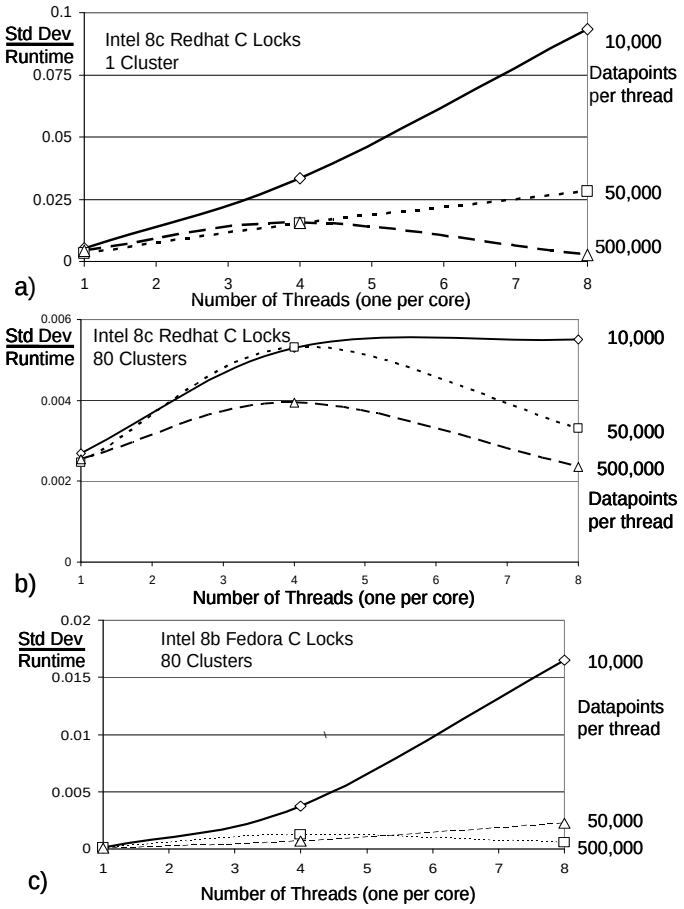

Fig. 7. Ratio of Standard Deviation to mean of thread execution time using Redhat on Intel8c (a,b) or Fedora on Intel8b (c) Linux and C with locks for synchronization on Clustering Kernel for three dataset sizes with 10,000 50,000 or 500,000 points per thread (core).

These figures also show many of the same trends of earlier results. The smallest dataset (10,000) which should be contained in cache has the largest fluctuations. Further Redhat has significantly lower fluctuations than Fedora with C and Linux seeing lower numbers than C# and Windows. Even Redhat has quite large fluctuations for 1 cluster (fig. 7a)) which reduce to about 0.4% for the compute bound 80 cluster case. C# in fig. 6 has rather large (5% or above) fluctuations in all cases considered.

Note our results with Linux are all obtained with threads and so are not directly comparable with traditional MPI measurements that use processes. Processes are better isolated from each other in both cache and system effects and so it is possible that these fluctuations are quite unimportant in past Scientific programming studies but significant in our case.

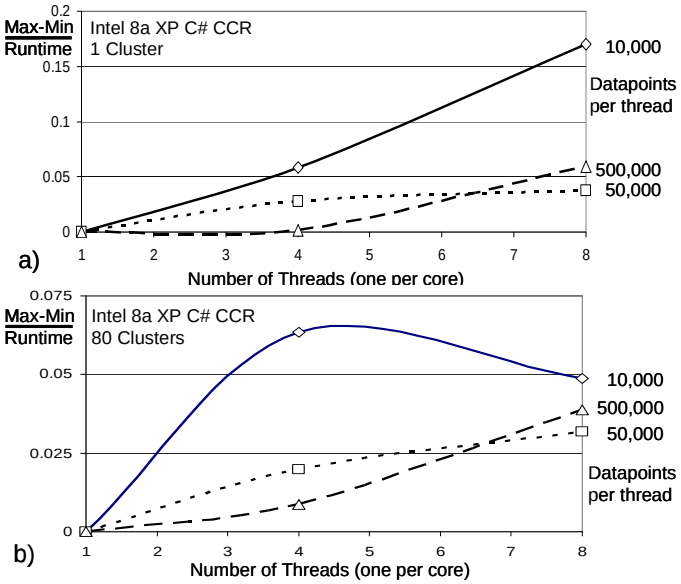

Fig. 8. Ratio of Maximum minus Minimum to mean of concurrent thread execution time averaged over 1000 instances using XP on Intel 8a and C# with CCR for synchronization on Clustering Kernel for three dataset sizes with 10,000 50,000 or 500,000 points per thread (core).

VIII. CONCLUSIONS

Our results are basically positive. We have initial results that suggest a class of datamining applications run well on current multicore architectures. We have looked in detail at overheads due to memory, cache, run time fluctuation and synchronizations. Some of these are surprisingly high in Windows/C# environments but further work is likely to address this problem as the best Linux systems only show small effects. C# appears to have much better thread synchronization effects than Java and it seems important to investigate this. Current cache architectures are unsuitable for this application class and it would be useful to allow a cache operation mode that avoided unnecessary overheads such as those studied in Sect. IV.

REFERENCES

[1] David Patterson *The Landscape of Parallel Computing Research: A View from Berkeley 2.0* Presentation at Manycore Computing 2007 Seattle June 20 2007 http://science.officeisp.net/ManycoreComputingWorkshop07/Presentations/David%20Patterson.pdf

[2] Jack Dongarra Editor *The Promise and Perils of the Coming Multicore Revolution and Its Impact*, CTWatch Quarterly Vol 3 No. 1 February 07, http://www.ctwatch.org/quarterly/archives/february-2007

[3] Herb Sutter, *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*, Dr. Dobb's Journal, 30(3), March 2005.

[4] Annotated list of multicore Internet sites http://www.connotea.org/user/crmc/

[5] Pradeep Dubey *Teraflops for the Masses: Killer Apps of Tomorrow* Workshop on Edge Computing Using New Commodity Architectures, UNC 23 May 2006 http://gamma.cs.unc.edu/EDGE/SLIDES/dubey.pdf

[6] Geoffrey Fox tutorial at Microsoft Research *Parallel Computing 2007: Lessons for a Multicore Future from the Past* February 26 to March 1 2007 http://grids.ucs.indiana.edu/ptliupages/presentations/PC2007/index.html

[7] Xiaohong Qiu, Geoffrey Fox, and Alex Ho Analysis of Concurrency and Coordination Runtime CCR and DSS, Technical Report January 21 2007 http://grids.ucs.indiana.edu/ptliupages/publications/CCRDSSanalysis_jan21-07.pdf

[8] Xiaohong Qiu, Geoffrey Fox, H. Yuan, Seung-Hee Bae, George Chrysanthakopoulos, Henrik Frystyk Nielsen *Performance Measurements of CCR and MPI on Multicore Systems Summary* September 23 2007 http://grids.ucs.indiana.edu/ptliupages/presentations/MCPerformanceSept21-07.ppt

[9] 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* to be published in proceedings of eScience 2007 Conference Bangalore India December 10-13 2007 http://grids.ucs.indiana.edu/ptliupages/publications/CCRSept23-07eScience07.pdf

[10] 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 http://grids.ucs.indiana.edu/ptliupages/publications/Workflow-overview.pdf

[11] Henrik Frystyk Nielsen, George Chrysanthakopoulos, "Decentralized Software Services Protocol – DSSP" http://msdn.microsoft.com/robotics/media/DSSP.pdf

[12] "Concurrency Runtime: An Asynchronous Messaging Library for C# 2.0" George Chrysanthakopoulos Channel9 Wiki Microsoft http://channel9.msdn.com/wiki/default.aspx/Channel9.ConcurrencyRuntime

[13] "Concurrent Affairs: Concurrent Affairs: Concurrency and Coordination Runtime", Jeffrey Richter Microsoft http://msdn.microsoft.com/msdnmag/issues/06/09/ConcurrentAffairs/default.aspx

[14] 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/

[15] 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. http://urresearch.rochester.edu/handle/1802/2105

[16] SALSA Multicore research Web site, http://www.infomall.org/salsa

[17] J Kurzak and J J Dongarra, *Pipelined Shared Memory Implementation of Linear Algebra Routines with arbitary Lookahead - LU, Cholesky, QR,* Workshop on State-of-the-Art in Scientific and Parallel Computing, Umea, Sweden, June 2006 http://www.hpc2n.umu.se/para06/papers/paper_188.pdf

[18] Geoff M. Downs, John M. Barnard *Clustering Methods and Their Uses in Computational Chemistry*, Reviews in Computational Chemistry, Volume 18, 1-40 2003

[19] *K-means algorithm* at Wikipedia http://en.wikipedia.org/wiki/K-means_algorithm

[20] Rose, K. *Deterministic annealing for clustering, compression, classification, regression, and related optimization problems*, Proceedings of the IEEE Vol. 86, pages 2210-2239, Nov 1998

[21] "The Sourcebook of Parallel Computing" edited by Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, Morgan Kaufmann, November 2002.

[22] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker *Solving Problems in Concurrent Processors-Volume 1*, Prentice Hall, March 1988

[23] Fox, G. C., Messina, P., Williams, R., "Parallel Computing Works!", Morgan Kaufmann, San Mateo Ca, 1994.

[24] *How to Align Data Structures on Cache Boundaries*, Internet resource from Intel, http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/knowledgebase/43837.htm

[25] Message passing Interface MPI Forum http://www.mpi-forum.org/index.html

[26] MPICH2 implementation of the Message-Passing Interface (MPI) http://www-unix.mcs.anl.gov/mpi/mpich/

[27] High Performance MPI Message Passing Library http://www.open-mpi.org/

[28] Mark Baker, Bryan Carpenter, and Aamir Shafi. *MPJ Express: Towards Thread Safe Java HPC*, Submitted to the IEEE International Conference on Cluster Computing (Cluster 2006), Barcelona, Spain, 25-28 September, 2006. http://www.mpj-express.org/docs/papers/mpj-clust06.pdf

[29] mpiJava Java interface to the standard MPI runtime including MPICH and LAM-MPI http://www.hpjava.org/mpiJava.html

[30] Richard L. Graham and Timothy S. Woodall and Jeffrey M. Squyres "Open MPI: A Flexible High Performance MPI", Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics, 2005 http://www.open-mpi.org/papers/ppam-2005

[31] D.K. Panda "How will we develop and program emerging robust, low-power, adaptive multicore computing systems?" The Twelfth International Conference on Parallel and Distributed Systems ICPADS '06 July 2006 Minneapolis http://www.icpads.umn.edu/powerpoint-slides/Panda-panel.pdf

[32] Thomas Bemmerl "Pallas MPI Benchmarks Results" http://www.lfbs.rwth-aachen.de/content/index.php?ctl_pos=392

[33] Myricom *Myri-10G and Myrinet-2000 Performance Measurements* http://www.myri.com/scs/performance/

[34] Java Grande http://www.javagrande.org