Performance of Multicore Systems on Parallel Data Clustering with Deterministic Annealing

Xiaohong Qiu[1], Geoffrey C. Fox[2], Huapeng Yuan[2], Seung-Hee Bae[2],
George Chrysanthakopoulos[3], and Henrik Frystyk Nielsen[3]

[1] Research Computing UITS, Indiana University Bloomington
xqiu@indiana.edu
[2] Community Grids Lab Indiana University Bloomington
{gcf,yuanh,sebae}@indiana.edu
[3] Microsoft Research Redmond WA
{georgioc, henrikn}@microsoft.com

Abstract. We present a performance analysis of a scalable parallel data clustering algorithm with deterministic annealing for multicore systems that compares MPI and a new C# messaging runtime library CCR (Concurrency and Coordination Runtime) with Windows and Linux and using both threads and processes. We investigate effects of 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. We compare our C# results with C using MPICH2 and Nemesis and Java with both mpiJava and MPJ Express. We show initial speedup results from Geographical Information Systems and Cheminformatics clustering problems. We abstract the key features of the algorithm and multicore systems that lead to the observed scalable parallel performance.

Keywords: Data mining, MPI, Multicore, Parallel Computing, Performance, Threads, Windows.

1 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. In this paper we consider data mining as a class of applications that has broad applicability and could be important on tomorrow's client systems. 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 (Recognition, Mining and Synthesis) analysis by Intel [5]. In our research, we are looking at some core data mining algorithms and their application to scientific areas including cheminformatics, bioinformatics and demographic studies using GIS (Geographical Information Systems). On the computer science side, we are

looking at performance implications of both multicore architectures and use of managed code. Our close ties to science applications ensures that we understand important algorithms and parameter values and can generalize our initial results on a few algorithms to a broader set.

In this paper we present new results on a powerful parallel data clustering algorithm that uses deterministic annealing [20] to avoid local minima. We explore in detail the sources of the observed synchronization overhead. 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 research was performed on a set of multicore commodity PC's summarized in Table 1; each has two CPU chips and a total of 4 or 8 CPU cores. The results can be extended to computer clusters as we are using similar messaging runtime but we focus in this paper on the new results seen on the multicore systems.

Table 1. Multicore PC's used in paper

AMD4: 4 core 2 Processor HPxw9300 workstation, 2 AMD Opteron CPUs Processor 275 at 2.19GHz, L2 Cache 2x1MB (for each chip), Memory 4GB. XP 64bit & Server 2003
Intel4: 4 core 2 Processor Dell Precision PWS670, 2 Intel Xeon CPUs at 2.80GHz, L2 Cache 2x2MB, Memory 4GB. XP Pro 64bit
Intel8a: 8 core 2 Processor Dell Precision PWS690, 2 Intel Xeon CPUs E5320 at 1.86GHz, L2 Cache 2x4M, Memory 8GB. XP Pro 64bit
Intel8b: 8 core 2 Processor Dell Precision PWS690, 2 Intel Xeon CPUs x5355 at 2.66GHz, L2 Cache 2X4M, Memory 4GB. Vista Ultimate 64bit and Fedora 7
Intel8c: 8 core 2 Processor Dell Precision PWS690, 2 Intel Xeon CPUs x5345 at 2.33GHz, L2 Cache 2X4M, Memory 8GB. Redhat 5

Sect. 2 discusses the CCR and SALSA runtime described in more detail in [7-9]. Sect. 3 describes our motivating clustering application and explains how it illustrates a broader class of data mining algorithms [17]. These results identify some important benchmarks covering memory effects, runtime fluctuations and synchronization costs discussed in Sections 4-6. There are interesting cache effects that will be discussed elsewhere [8]. Conclusions are in Sect. 8 while Sect. 7 briefly describes the key features of the algorithm and how they generalize to other data mining areas. All results and benchmark codes presented are available from `http://www.infomall.org/salsa` [16].

2 Overview of CCR and SALSA Runtime Model

We do not address possible high level interfaces such as OpenMP or parallel languages but rather focus on lower level runtime to which these could map. 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]. Applications are built from services; services are built as parallel threads or processes that are synchronized with low latency by locks, MPI or a novel messaging runtime library CCR (Concurrency and Coordination Runtime) developed by Microsoft Research [11-15].

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 some 8 core capabilities given in more detail in [9]. CCR 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 discussed in this paper.

For our performance comparisons with MPI, we needed rendezvous semantics which are fully supported by CCR and we chose to use the Exchange pattern corresponding to the MPI_SENDRECV interface where each process (thread) sends and receives two messages equivalent to a combination of a left and right shift with its two neighbors in a ring topology. 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.

3 Deterministic Annealing Clustering Algorithm

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 demographic information derived from the US Census data and other sources. Our software successfully scales to cluster the 10 million chemicals in NIH PubChem and the 6 million people in the state of Indiana. Both applications will be published elsewhere and the results given here correspond to realistic applications and subsets designed to test scaling. We use a modification of the well known K-means algorithm [19], using deterministic annealing [20], that has much better convergence properties than K-means and good parallelization properties.

For a set of data points $\underline{X}$(labeled by x) and cluster centers $\underline{Y}$(labeled by k), one gradually lowers the annealing temperature T and iteratively calculates:

$$\begin{aligned} &\underline{Y}(k) = \sum_x p(\underline{X}(x),\underline{Y}(k))\, \underline{X}(x)\,. \\ &p(\underline{X}(x),\underline{Y}(k)) = \exp(-d(\underline{X}(x),\underline{Y}(k))/T)\; p(x) / Z_x\,. \\ &\text{with } Z_x = \sum_k \exp(-d(\underline{X}(x),\underline{Y}(k))/T)\,. \end{aligned} \tag{1}$$

Here $d(\underline{X}(x),\underline{Y}(k))$ 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 the calculation in (1). Rather than plot speed-up, we focus in more detail on the deviations from "perfect speed-up (of P)". Such parallel applications have a well understood performance model that can be expressed in terms of a parallel overhead $f(n,P)$ (roughly 1-efficiency) where different overhead effects are naturally additive. Putting $T(n,P)$ as the execution time on P cores or more generally processes/threads, we can define

$$\begin{aligned} &\text{Overhead } f(n,P) = (PT(n,P)-T(Pn,1))/T(Pn,1)\,. \\ &\text{and efficiency } \varepsilon = 1/(1+f) \text{ and Speed-up} = \varepsilon\,. \end{aligned} \tag{2}$$

For the algorithm of eqn. (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, which shows good speed-up on 8 cores of around 7.5 ($f(n,P)$~ .05) for large problems. However we do not find $f(n,P)$ going fully 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. This is expected from (1) as the ratio of computation to memory access is proportional to 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

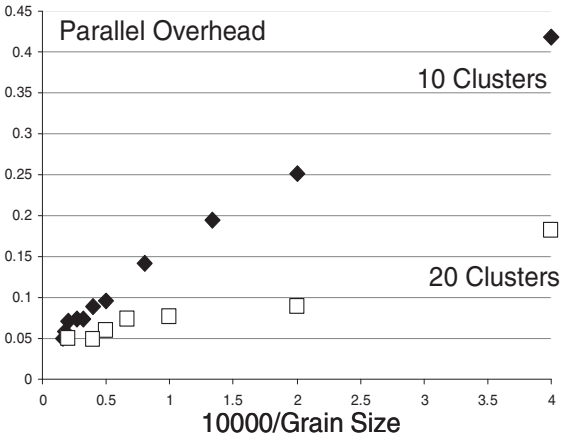

Fig. 1. Parallel Overhead for GIS 2D Clustering on Intel8b using C# with 8 threads (cores) and CCR Synchronization. We use two values (10, 20) for the number of clusters and plot against the reciprocal of the number of data points.

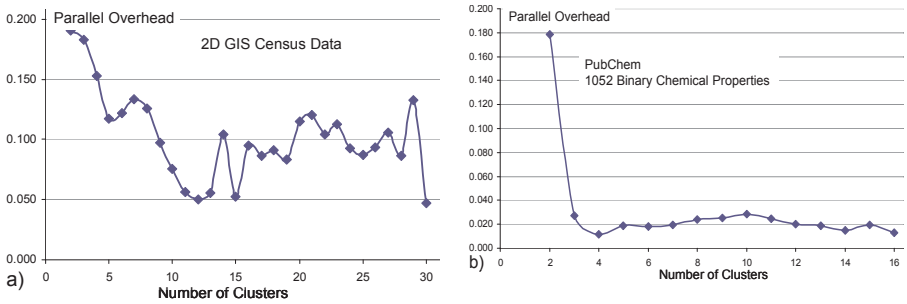

Fig. 2. Parallel Overhead defined in (2) as a function of the number of clusters for a) 2 dimensional GIS data for Indiana in over 200,000 blocks and 40,000 compounds each with 1052 binary properties

data used in Chemistry clustering. This problem showed fluctuations similar in size 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.

Looking at this performance in more detail we identified effects from memory bandwidth, fluctuations in thread run time and cache interference [24]. We present a summary of the first two here and present cache effects and details in [7, 8].

4 Memory Bandwidth

In Fig. 3, we give typical results of a study of the impact of memory bandwidth in the different hardware and software configurations of Table 1. We isolate the kernel of the clustering algorithm of Sect. 2 and examine its performance as a function of grain size n, number of clusters and number of cores. We employ the scaled speed up strategy 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 and deviations from this value represent interesting performance effects. 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 steps on every variable fetched from memory. Although we studied C, C#, Windows and Linux, we only present Windows C# results in Fig. 3. C with Windows shows similar effects but of smaller magnitude while Linux shows small effects (the results for all n and cluster counts are near 1). Always we use threads not processes and C uses locks and C# uses CCR synchronization. Data is stored so as to avoid any cache line (false sharing) effects [8, 24]. The results for one cluster in Fig. 3(a) clearly show the effect of memory bandwidth with scaled run time increasing significantly as one increases the number of cores used. The performance improves in Fig. 3(b) (scaled runtime < 1) with more clusters when the memory demands are small. In this benchmark the memory demands scale directly with number of cores and inversely with number of clusters. A major concern with multicore system is the need for a memory bandwidth that increases linearly with the number of cores. In Fig. 3(a) we see a 50% increase in the run time for a grain size of 10,000 and 1 cluster. This is for

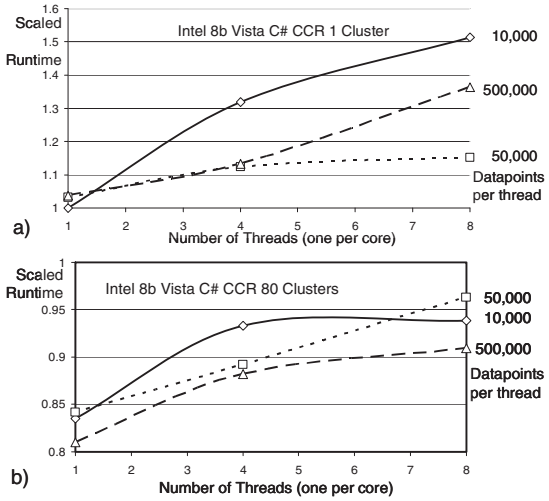

Fig. 3. Scaled Run time on Intel8b using Vista and C# with CCR for synchronization on Clustering Kernel for three dataset sizes with 10,000 50,000 or 500,000 points per thread(core). Each measurement involved averaging over at least 1000 computations separated by synchronization whose small cost is not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

C# and Windows and the overhead is reduced to 22% for C on Windows and 13% for C on Linux. Further we note that one expect the 10,000 data point case to get excellent performance as the dataset can easily fit in cache and minimize memory bandwidth needs. However we see similar results whether or not dataset fits into cache. This must be due to the complex memory structure leading to cache conflicts. We get excellent cache performance for the simple data structures of matrix multiplication.

In all cases, we get small overheads for 80 clusters (and in fact for cluster counts greater than 4), which explains why the applications of Sect. 2 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 data mining cases we are studying satisfy this and we expect them to run well on multicore machines expected over the next 5 years.

5 Synchronization Performance

The synchronization performance has been discussed in detail previously [9] for CCR where we discussed dynamic threading in detail showing it had an approximate 5μs overhead. Here we expand the previous brief discussion of the rendezvous (MPI) style performance with Table 2 giving some comparisons between C, C# and Java for the MPI Exchange operation (defined in Sect. 2) running on the maximum number of cores (4 or 8) available on the systems of Table 1. Results for the older Intel8a are available online [16]. In these tests we use a zero size message. Note that the CCR Exchange operation timed in Table 2 has the full messaging transfer semantics of the MPI standards but avoids the complexity of some MPI capabilities like tags [25-27]. 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 2 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.

Table 2. MPI Exchange Latency

Machine	OS	Runtime	Grains	Latency µs
Intel8c	Redhat	MPJE	8 Procs	181
		MPICH2	8 Procs	40.0
		MPICH2 Fast Option	8 Procs	39.3
		Nemesis	8 Procs	4.21
Intel8c	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

The optimized Nemesis version of MPICH2 gives best performance while CCR with for example 20µs latency on Intel8b, outperforms "vanilla MPICH2". We can expect CCR and C# to improve and compete in performance with systems like Nemesis using 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 while MPJ Express [27] 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 [29]. As discussed earlier we expect managed code (Java and C#) to be of growing importance as client multicores prolifergate so good parallel multicore Java performance is important.

6 Performance Fluctuations

We already noted in Sect. 3 that our performance was impacted by fluctuations 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 Figs. 4 and 5 we present some results quantifying this using the same "clustering kernel" introduced in Sect. 4. We average results over 1000 synchronization points in a single run. In Figs. 4 and 5 we calculate the standard deviation of the 1000*P* measured thread runtimes gotten if *P* cores are used. 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. 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. C and Linux show lower fluctutions

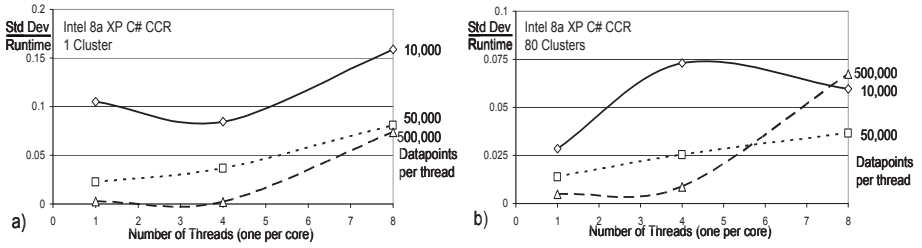

Fig. 4. Ratio of Standard Deviation to mean of thread execution time averaged over 1000 instances using XP on Intel 8a and C# with CCR for synchronization on Clustering Kernel for three dataset sizes with 10,000 50,000 or 500,000 points per thread (core)

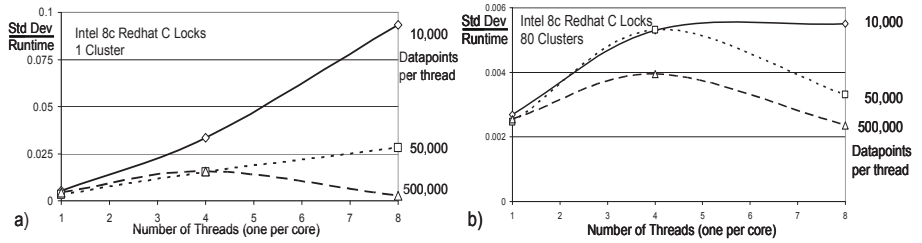

Fig. 5. Ratio of Standard Deviation to mean of thread execution time using Redhat on Intel8c (a,b) Linux and C with locks for synchronization on Clustering Kernel for three dataset sizes with 10,000 50,000 or 500,000 points per thread (core). Fedora shows larger effects than Redhat.

than C# and Windows. Further turning to Linux, Redhat outperforms Fedora (shown in [9]). C# in Fig. 4 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 Linux 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. Although these fluctuations are important in the limit of large grain size when other overheads are small, they are never a large effect and do not stop us getting excellent speedup on large problems.

7 Generalization to Other Data Mining Algorithms

The deterministic annealing clustering algorithm has exactly the same structure as other important data mining problems including dimensional scaling and Gaussian mixture models with the addition of deterministic annealing to mitigate the local minima that are a well known difficulty with these algorithms [17]. One can show [17] that one gets these different algorithms by different choices for $\underline{Y}(k)$, $a(x)$, $g(k)$, T and $s(k)$ in (3). As in Sect. 2, $\underline{X}(x)$ are the data points to be modeled and F is the objective function to be minimized.

$$F = -T\sum_{x=1}^{N} a(x)\ln\left[\sum\nolimits_{k=1}^{K} g(k)\exp\{-0.5(\underline{X}(x)-\underline{Y}(k))^2/(Ts(k))\}\right]. \quad (3)$$

Thus we can immediately deduce that our results imply that scalable parallel performance can be achieved for all algorithms given by (3). Further it is interesting that the parallel kernels of these data mining algorithms are similar to those well studied by the high performance (scientific) computing community and need the synchronization primitives supported by MPI. The algorithms use the well established SPMD (Single Program Multiple Data) style with the same decomposition for multicore and distributed execution. However clusters and multicore systems use different implementations of collective operations at synchronization points. We expect this structure is more general than the studied algorithm set.

8 Conclusions

Our results are very encouraging for both using C# and for getting good multicore performance on important applications. We have initial results that suggest a class of data mining applications run well on current multicore architectures with efficiencies on 8 cores of at least 95% for large realistic problems. We have looked in detail at overheads due to memory, run time fluctuation and synchronizations. Our results are reinforced in [8, 9] with a study of cache effects and further details of issues covered in this paper. Some overheads such as runtime fluctuations are surprisingly high in Windows/C# environments but further work is likely to address this problem by using lessons from Linux systems that show small effects. C# appears to have much better thread synchronization effects than Java and it seems interesting to investigate this.

References

1. Patterson, D.: The Landscape of Parallel Computing Research: A View from Berkeley 2.0 Presentation at Manycore Computing, Seattle, June 20 (2007)
2. Dongarra, J. (ed.): The Promise and Perils of the Coming Multicore Revolution and Its Impact, CTWatch Quarterly, February 2007, vol. 3(1) (2007), `http://www.ctwatch.org/quarterly/archives/february-2007`
3. Sutter, H.: 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 sites: `http://www.connotea.org/user/crmc/`
5. Dubey P.: Teraflops for the Masses: Killer Apps of Tomorrow Workshop on Edge Computing Using New Commodity Architectures, UNC (May 23, 2006), http://gamma.cs.unc.edu/EDGE/SLIDES/dubey.pdf
6. Fox G.: Parallel Computing 2007: Lessons for a Multicore Future from the Past Tutorial at Microsoft Research (February 26 to March 1 2007)
7. Qiu, X., Fox, G., Ho, A.: Analysis of Concurrency and Coordination Runtime CCR and DSS, Technical Report January 21 (2007)
8. Qiu, X., Fox, G., Yuan, H., Bae, S., Chrysanthakopoulos, G., Nielsen, H.: Performance Measurements of CCR and MPI on Multicore Systems Summary, September 23 (2007)

9. Qiu, X., Fox, G., Yuan, H., Bae, S., Chrysanthakopoulos, G., Nielsen, H.: High Performance Multi-Paradigm Messaging Runtime Integrating Grids and Multicore Systems. In: Proceedings of eScience 2007 Conference, Bangalore, India, December 10-13 (2007)
10. Gannon, D., Fox, G.: Workflow in Grid Systems Concurrency and Computation. Practice & Experience 18(10), 1009–1019 (2006)
11. Nielsen, H., Chrysanthakopoulos, G.: Decentralized Software Services Protocol – DSSP, http://msdn.microsoft.com/robotics/media/DSSP.pdf
12. Chrysanthakopoulos, G.: Concurrency Runtime: An Asynchronous Messaging Library for C# 2.0, Channel9 Wiki Microsoft, http://channel9.msdn.com/wiki/default.aspx/Channel9.ConcurrencyRuntime
13. Richter J.: Concurrent Affairs: Concurrent Affairs: Concurrency and Coordination Runtime, 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, http://msdn.microsoft.com/robotics/
15. Chrysanthakopoulos, G., Singh, S.: An Asynchronous Messaging Library for C#, Synchronization and Concurrency in Object-Oriented Languages (SCOOL) at OOPSLA Workshop, San Diego, CA (October 2005), http://urresearch.rochester.edu/handle/1802/2105
16. SALSA Multicore research Web site, http://www.infomall.org/salsa For Indiana University papers cited here, http://grids.ucs.indiana.edu/ptliupages/publications
17. Qiu X., Fox G., Yuan H., Bae S., Chrysanthakopoulos G., Nielsen H.: Parallel Clustering and Dimensional Scaling on Multicore Systems Technical Report (February 21 2008)
18. Downs, G., Barnard, J.: Clustering Methods and Their Uses in Computational Chemistry. Reviews in Computational Chemistry 18, 1–40 (2003)
19. K-means algorithm at, http://en.wikipedia.org/wiki/K-means_algorithm
20. Rose, K.: Deterministic annealing for clustering, compression, classification, regression, and related optimization problems. Proc IEEE 86, 2210–2239 (1998)
21. Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A. (eds.): The Sourcebook of Parallel Computing. Morgan Kaufmann, San Francisco (2002)
22. Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D.: Solving Problems in Concurrent Processors, vol. 1. Prentice-Hall, Englewood Cliffs (1988)
23. Fox, G., Messina, P., Williams, R.: Parallel Computing Work! 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/asmona/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. Baker, M., Carpenter, B., Shafi, A.: MPJ Express: Towards Thread Safe Java HPC. In: IEEE International Conference on Cluster Computing (Cluster 2006), Barcelona, Spain, September 25-28 (2006), http://www.mpj-express.org/docs/papers/mpj-clust06.pdf
28. mpiJava Java interface to the standard MPI runtime including MPICH and LAM-MPI, http://www.hpjava.org/mpiJava.html
29. Java Grande, http://www.javagrande.org