High Performance Multi-Paradigm Messaging Runtime Integrating Grids and Multicore Systems

Xiaohong Qiu
xqiu@indiana.edu
Research Computing UITS
Indiana University Bloomington

Geoffrey C. Fox, Huapeng Yuan, Seung-Hee Bae
gcf@indiana.edu yuanh@indiana.edu sebae@indiana.edu
Community Grids Laboratory
Indiana University Bloomington

George Chrysanthakopoulos, Henrik Frystyk Nielsen
georgioc@microsoft.com henrikn@microsoft.com
Microsoft Research
Redmond WA

Abstract

eScience applications need to use distributed Grid environments where each component is an individual or cluster of multicore machines. These are expected to have 64-128 cores 5 years from now and need to support scalable parallelism. Users will want to compose heterogeneous components into single jobs and run seamlessly in both distributed fashion and on a future "Grid on a chip" with different subsets of cores supporting individual components. We support this with a simple programming model made up of two layers supporting traditional parallel and Grid programming paradigms (workflow) respectively. We examine for a parallel clustering application, the Concurrency and Coordination Runtime CCR from Microsoft as a multi-paradigm runtime that integrates the two layers. Our work uses managed code (C#) and for AMD and Intel processors shows around a factor of 5 better performance than Java. CCR has MPI pattern and dynamic threading latencies of a few microseconds that are competitive with the performance of standard MPI for C.

1. Introduction

Grids and multicore [4-6] systems are contemporary technologies that will have great impact on eScience. Grids integrate the myriad of distributed resources needed by modern science and multicore machines will certainly allow very high performance simulation engines. However multicore technology is also a dominant feature of the next round of commodity systems and this will motivate a new generation of software environments that will address commodity client and server applications. Thus we need to examine the possibility of common software models that support both these new commodity applications and traditional scientific applications.

Future client systems are not likely to need large scale simulations. Rather an analysis by Intel [7] identifies gaming and more broadly datamining as critical client-side applications with latter including speech, image and video analysis as well processing of local sensors and data fetched from the web. These datamining algorithms often use "classic" scientific algorithms for matrix arithmetic or optimization at their core and will need good support for both dynamic threads and MPI style loosely synchronous applications [8]. We note that it will be the data deluge that drives eScience in the large and the commodity software on the multicore "small". Note that almost certainly the software environments developed for commodity multicore systems will be the most attractive for (specialized) eScience applications due to the excellent support that is provided for commodity software.

We follow the model proposed by Berkeley [1] and indeed used traditionally in parallel computing with two layers called by them "productivity" and "efficiency". One needs a modest number of experts working at the "efficiency" layer building libraries (packaged as services in our approach) that implement good parallel algorithms. We see several excellent approaches to the "productivity" layer which can and

will be used by a broad range of programmers. This layer was often termed coarse grained functional parallelism in the old literature and supported by systems like AVS, Khoros, SciRUN and HeNCE [9]. However our key idea is to use modern distributed system (Grid) approaches at this layer. In fact Grid workflow, Mashups, MapReduce [2], or just scripting languages provide popular productivity models today and we expect these to improve. However we believe that no breakthroughs are needed to provide a good programming model at this layer. We suggest that the efficiency layer still needs much work and we need to integrate support for it with the productivity layer and between the different models in the efficiency layer; these include MPI style collective synchronizations and dynamic threading used in for example dynamic graph searches and discrete event simulations.

In this paper we focus on one part of this puzzle – namely investigating the runtime that could span these different environments and different platforms that would be used by the expected heterogeneous composite applications. Note that managed code will be important for desktop commodity applications and so C# (mainly used here) and Java could become more important than now for parallel programming.

Our research is looking at a variety of datamining algorithms and the underlying runtime which allow jobs to be composed from multiple, data sources and visualizations and to run efficiently and seamlessly either internally to a single CPU or across a tightly coupled cluster or distributed Grid. We use Cheminformatics and Geographical Information System GIS examples built around a parallel clustering datamining service. The parallelization of this would traditionally use MPI but here we use the CCR runtime running in MPI mode. CCR [10-11] was designed for robotics applications [12] but also investigated [13] as a general programming paradigm. CCR supports efficient thread management for handlers (continuations) spawned in response to messages being posted to ports. The ports (queues) are managed by CCR which has several primitives supporting the initiation of handlers when different message/port assignment patterns are recognized. Note that CCR supports a particular flavor of threading where information is passed by messages allowing simple correctness criteria. Further CCR already has a distributed "productivity layer" runtime known as DSS (Decentralised Software Services) built on top of it and we will use this as our productivity layer.

However this paper is not really proposing the ultimate programming model but examining a possible low level runtime which could support a variety of different parallel programming models that would map down into it. In particular the new generation of parallel languages [14] from Darpa's HPCS High Productivity Computing System program supports the three execution styles (dynamic threading, MPI, coarse grain functional parallelism) we investigate here and our runtime could be used by these and other high level programming approaches.

Table 1. Machines used

AMD4: HPxw9300 workstation, 2 AMD Opteron CPUs Processor 275 at 2.19GHz, L2 Cache 2x1MB (for each chip), Memory 4GB, XP Pro 64bit Benchmark Computational unit: **1.388 *μs***
Intel4: Dell Precision PWS670, 2 Intel Xeon CPUs at 2.80GHz, L2 Cache 2x2MB, Memory 4GB, XP Pro 64bit Benchmark Computational unit: **1.475 *μs***
Intel8a: Dell Precision PWS690, 2 Intel Xeon CPUs E5320 at 1.86GHz, L2 Cache 2x4M, Memory 8GB, XP Pro 64bit Benchmark Computational unit: **1.696 *μs***
Intel8b: Dell Precision PWS690, 2 Intel Xeon CPUs x5355 at 2.66GHz, L2 Cache 2X4M, Memory 4GB, Vista Ultimate 64bit Benchmark Computational unit: **1.188 *μs***

We used four machine types in this study with details recorded above in Table 1. Each machine had two processors; one used the AMD dual core Opteron, one dual core Intel Xeons and there were two models of quad core Intel Xeons. As well as basic hardware, the table indicates performance on a computational unit used in performance test.

In the next section, we discuss the clustering application and then in section 3 CCR and DSS. Section 4 defines more precisely our three execution models. Section 5 presents our basic performance results that suggest one can build a single runtime that supports the different execution models and so implement the hybrid productivity-efficiency environment. The CCR results are briefly compared with Java [3] and MPICH2 while the DSS results are contrasted with those of the Java Axis2. Future work and conclusions are in section 6.

2. Clustering

We are building a suite of data mining services to test the runtime and two layer programming model. We are starting with data clustering which has many important applications including clustering of chemical properties which is an important tool [15] for finding for example a set of chemicals similar to each other and so likely candidates for a given drug. In GIS

applications, clustering is commonly used on selected samples from population census data.

We present in Figure 1, initial results from the parallel clustering service implemented on the 8 core Intel machine labeled Intel8b in table 1. We chose an improved K-means [16] algorithm whose structure can be straightforwardly and efficiently parallelized using MPI style programming. This method [17] uses a multi-scale approach to avoid false minima and has a parallel overhead [18] that decreases asymptotically like 1/grain size as the data set increases. Here grain size is the dataset size divided by the number of processors (cores) which is here 8. Putting T(n) as the execution time on n cores, we can define

$$Overhead\ f = (PT(P)-T(1))/T(1)$$
$$Efficiency\ \varepsilon = 1/(1+f)$$

Note that the advent of multicore systems is likely to prompt a re-examination of algorithms with preference being given to those that can be efficiently parallelized. Our results required arrangement of data in memory to avoid any interference in cache lines accessed by different cores; such interference increased memory traffic and produced factors of 2 variations in runtime [33] for our initial implementation. We present typical results in figure 1 for Intel8b. They show an overhead decreasing (efficiency tending to 1 and speedups to 8) as the grain size increases.

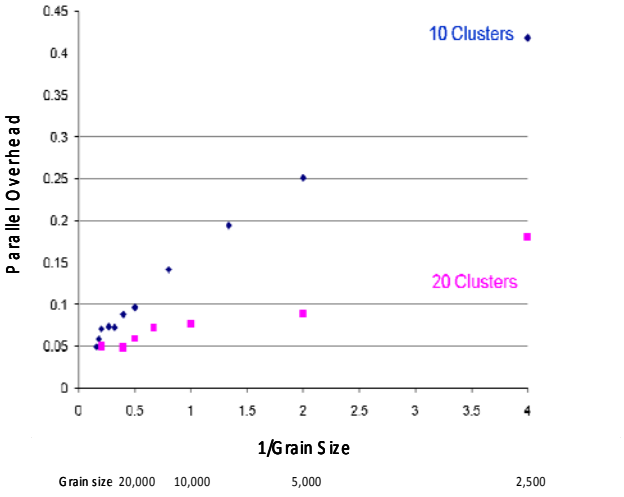

Figure 1. Parallel overhead on clustering algorithm

The detailed analysis [33] is affected by cache issues and fluctuations in run time that are much larger for Windows than the usual HPC Linux OS. In particular run time fluctuations give an overhead of 0.05 to 0.1 that is present even for very large grain sizes where the traditional analysis predicts zero overhead and an efficiency of 1. These results use CCR for all inter process communication and messaging and will improve when we have finished primitives optimized on multicore systems for the equivalent of MPI reduction operations. These initial results confirm that we can get good performance with CCR and are encouraging for the basic strategy of building a suite of high performance datamining algorithms with CCR. Now we look in detail at CCR and DSS performance (DSS encapsulates this clustering application in the "productivity" layer) so we can compare our new approach with more familiar technology.

3. Overview of CCR and DSS

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 what we call the 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.

Our current work uses the first three a) b) c) efficient capabilities of CCR. 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. For this paper, we did use the CCR framework to build a custom optimized collective operation corresponding to the MPI "exchange" operation but used existing capabilities for the "reduce" and "shift" patterns. We believe one can extend this work to provide all MPI messaging patterns.

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 and memory corruption. 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 coarse grain applications in the Web-style as compositions of services running in a distributed environment. Services are isolated from each other, even when running within the same node and are only exposed through their state and a uniform set of operations over that state. The DSS runtime provides a hosting environment for managing services and a set of infrastructure services that can be used for service creation, discovery, logging, debugging, monitoring, and security. DSS builds on existing Web architecture and extends the application model provided by HTTP through structured data manipulation and event notification. Interaction with DSS services happen either through HTTP or DSSP [19] which is a SOAP-based protocol for managing structured data manipulations and event notifications.

4. MPI and the 3 Execution Models

MPI – Message Passing Interface – dominates the runtime support of large scale parallel applications for technical computing. It is a complicated specification with 128 separate calls in the original specification [20] and double this number of interfaces in the more recent MPI-2 including support of parallel external I/O [21-22]. MPI like CCR is built around the idea of concurrently executing threads (processes, programs) that exchange information by messages. In the classic analysis [18, 23-25], parallel technical computing applications can be divided into four classes:

a) **Synchronous** problems where every process executes the same instruction at each clock cycle. This is a special case of b) below and only relevant as a separate class if one considers SIMD (Single Instruction Multiple Data) hardware architectures.
b) **Loosely Synchronous** problems where each process runs different instruction streams (often using the same program in SPMD mode) but they synchronize with the other processes every now and then. Such problems divide into stages where at the beginning and end of each stage the processes exchange messages and this exchange provides the needed synchronization that is scalable as it needs no global barriers. Load balancing must be used to ensure that all processes execute for roughly (within say 5%) the same time in each phase and MPI provides the messaging at the beginning and end of each stage. We get at each loose synchronization point a message pattern of many overlapping joins that is not usually seen in commodity applications and represents a new challenge.
c) **Embarrassingly or Pleasingly parallel** problems have no significant inter-process communication and are often executed on a Grid.
d) **Functional** parallelism leads to what were originally called metaproblems that consist of multiple applications, each of which is of one of the classes a), b), c) as seen in multidisciplinary applications such as linkage of structural, acoustic and fluid-flow simulations in aerodynamics. These have a coarse grain parallelism.

Classes c) and d) today would typically be implemented as a workflow using services to represent the individual components. Often the components are distributed and the latency requirements are typically less stringent than for synchronous and loosely synchronous problems. We view this as functional parallelism corresponding to the "productivity layer" and use DSS already developed for Robotics [10] on top of CCR for this case and idealized in Figure 2(a). Note in this paper, we only discuss run-time and do not address the many different ways of expressing the "productivity layer" i.e. we are discussing runtime and not languages.

We use CCR in a mode where multiple identical stages are executed and the run is completed by combining the computations with a simple CCR supported reduction as shown in Figure 2(b). This also illustrates the simple Pipeline Spawn execution that we used for basic performance measurements of the dynamic threading performance. Each thread writes to

a single port that is read by a fresh thread as shown in more detail in Figure 3.

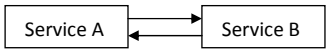

Figure 2(a). 2-way notification message using DSS

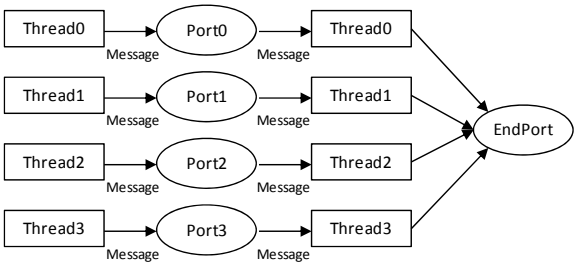

Figure 2(b). Pipeline of Spawned Threads followed by a Reduction using CCR Interleave

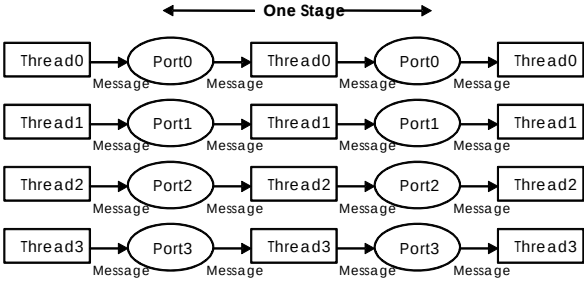

Figure 3. Multiple stages in CCR Performance Measurements

We take a fixed computation that takes from 12 to 17 seconds (10^7 stages of time complexity listed in table 1) depending on hardware and execution environment to run sequentially on the machines we used in this study. This computation was divided into a variable number of stages of identical computational complexity and then the measurement of execution time as a function of number of stages allows one to find the thread and messaging overhead. Note that the extreme case of 10^7 stages corresponds to the basic unit execution times of 1.188 to 1.696 μs given in Table 1 and is a stringent test for MPI style messaging which can require microsecond level latencies. We concentrated on small message payloads as it is the latency (overhead) in this case that is the critical problem. As multicore systems have shared memories, one would often use handles in small messages rather than transferring large payloads.

We looked at three different message patterns for the dynamic spawned thread case choosing structure that was similar to MPI to allow easier comparison of the rendezvous and spawned models. These spawned patterns are illustrated in Figure 4(a-c) and augment the pipeline of Figure 2(b) and 2 with a "nearest neighbor" shift with either one or two messages written to ports so we could time both the *Receive* and *MultiItemReceive* modes of CCR. We note that figures 2 to 4 are drawn for 4 cores while our tests used both 4 and 8 core systems.

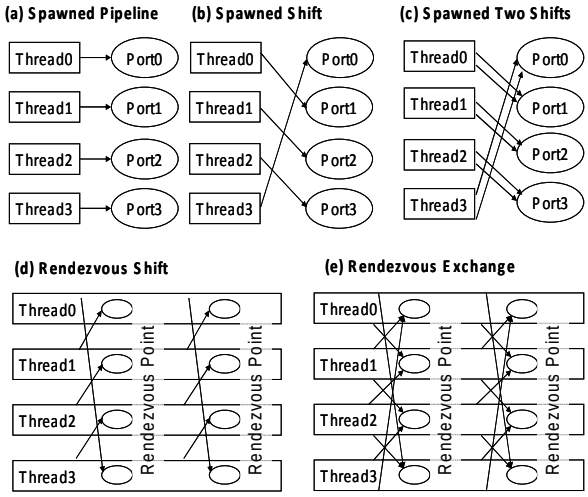

Figure 4. Five Communication patterns using CCR to test spawned dynamic threading (*a*,*b*,*c*) and MPI style Rendezvous's (*d*,*e*)

For our test of the final execution style, namely the MPI style runtime, 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 of Figure 4(d) or the combination of a left and right shift, namely an exchange, shown in Figure 4(e). 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 [1, 26] and so we only looked at a few simple but representative cases in this initial 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.

Note we are using threads in our CCR runtime whereas traditional MPI would use processes even on a multicore system. We expect thread-based parallelism to become more important in the future as one moves to integrate the different paradigms.

5. Performance of CCR in 3 Execution Models

5.1 CCR Message Latency and Overhead

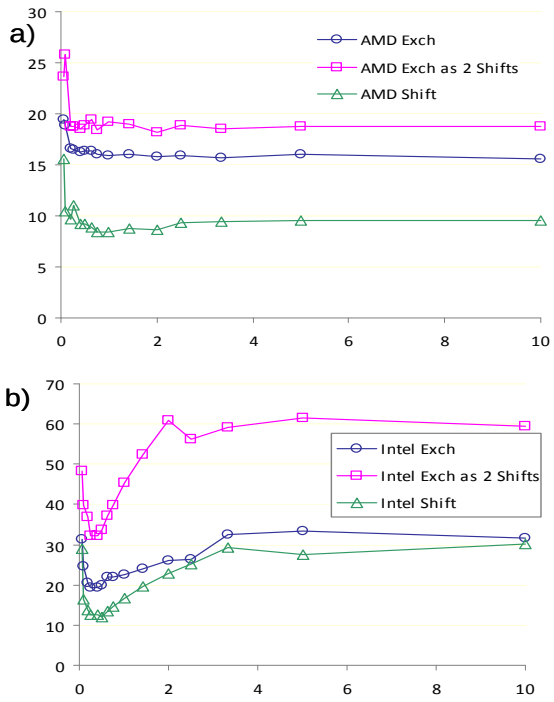

Figure 5. Overhead (latency) on MPI style Rendezvous Messaging for Shift and Exchange implemented either as two shifts or as custom CCR pattern for a) AMD4 with 4 threads and b) Intel8b with 8 threads

We present detailed benchmark measurements on 3 machines labeled AMD4 Intel4 and Intel8b in Table 1 with results shown in Table 2 and Figures 5(a) and (b). Table 2 looks at seven messaging modes shown in Figure 3 with rendezvous exchange implemented either as two separate shifts or as a custom CCR primitive which is faster especially on Intel8b. We also show pipeline implemented in both spawned and rendezvous fashion. The results correspond to a computation stage between messaging that is 20 times the values of table 1 i.e. from 23 (Intel8b) to 29 µs (Intel4). The newer Intel8b on 8 cores shows significantly lower overheads than the older Intel4 on 4 cores and the AMD4 Opteron on 4 cores has slightly lower overheads than Intel8b on 8 cores. Detailed results for the slower Intel 8a are also available [32].

We see the dynamic threading has an overhead that is around 5 µs for pipeline and shift for both AMD4 and Intel8b on 4 or 8 cores respectively. The MPI style rendezvous overheads increase to 9.36(11.74) µs for Shift and 16.3(20.16) µs for the optimized exchange operation on AMD4 (Intel8b) on 4(8) cores. We have performed identical measurements on the recent pure Java MPJE [3] for AMD4 which gives overheads of 185 µs for exchange and 104 µs for Shift. On the same machine, mpiJava (which invokes MPICH2 from Java) has for exchange a latency of 99.4 µs while the standard MPICH2 with C has 39.3 µs latency in this case. From table 2, CCR is faster than these but the optimized Nemesis version of MPICH is substantially faster than CCR. Our results show that C# with CCR is the by far the fastest managed code for messaging and gives competitive results to the best MPI's in common use for traditional scientific languages like C and C++.

Table 2: Messaging Overhead per stage for CCR patterns with 0.5 million stages

AMD4: 4 Core		Number of Parallel Computations					
(µs)		1	2	3	4	7	8
Spawned	Pipeline	1.76	4.52	4.4	4.84	1.42	8.54
	Shift		4.48	4.62	4.8	0.84	8.94
	Two Shifts		7.44	8.9	10.18	12.74	23.92
Rendezvous	Pipeline	3.7	5.88	6.52	6.74	8.54	14.98
	Shift		6.8	8.42	9.36	2.74	11.16
	Exchange As Two Shifts		14.1	15.9	19.14	11.78	22.6
	Exchange		10.32	15.5	16.3	11.3	21.38
Intel4: 4 Core		**Number of Parallel Computations**					
(µs)		1	2	3	4	7	8
Spawned	Pipeline	3.32	8.3	9.38	10.18	3.02	12.12
	Shift		8.3	9.34	10.08	4.38	13.52
	Two Shifts		17.64	19.32	21	28.74	44.02
Rendezvous	Pipeline	9.36	12.08	13.02	13.58	16.68	25.68
	Shift		12.56	13.7	14.4	4.72	15.94
	Exchange As Two Shifts		23.76	27.48	30.64	22.14	36.16
	Exchange		18.48	24.02	25.76	20	34.56
Intel8b: 8 Core		**Number of Parallel Computations**					
(µs)		1	2	3	4	7	8
Spawned	Pipeline	1.58	2.44	3	2.94	4.5	5.06
	Shift		2.42	3.2	3.38	5.26	5.14
	Two Shifts		4.94	5.9	6.84	14.32	19.44
Rendezvous	Pipeline	2.48	3.96	4.52	5.78	6.82	7.18
	Shift		4.46	6.42	5.86	10.86	11.74
	Exchange As Two Shifts		7.4	11.64	14.16	31.86	35.62
	Exchange		6.94	11.22	13.3	18.78	20.16

Figure 5 explores the dependence of the overhead on the number of stages for the MPI style rendezvous case i.e. on the amount of computation between each

message with the extreme case of 10^7 stages corresponding to the 1.388 (1.188) µs computation value between messages of table 1 for AMD4 (Intel8b). The overheads remain reasonable even in these extreme conditions showing that an intense number of small messages will not be a serious problem. This Figure emphasizes that the CCR custom exchange (marked just "Exch") can be much faster than the exchange implemented as a left followed by right shift (Marked "Exch as 2 Shifts").

5.2 DSS Message Latency and Overhead

We now examine CCR for the third form of parallelism; namely the functional parallelism model represented in Figure 1(a). The Robotics release [12] includes a lightweight service environment DSS built on top of CCR and we performed an initial evaluation of DSS on the AMD4 machine. We time groups of request-response two way messages running on (different) cores of the AMD system. We find average times of 30-50 microseconds or throughputs of 20,000 to 25,000 two-way messages per second after initial start up effects are past. This result of internal service to internal service can be compared with Apache Axis 2 where the AMD PC supports about 3,000 messages per second throughput. This is not an entirely fair comparison as the measurements are internal to one machine so each service end-point has effectively just two cores. The Axis measurements used external clients interacting on a LAN so there is network overhead but now the service can access the full 4 cores. We will give more complete comparisons later and also examine the important one-way messaging case.

6. Conclusions and Futures

This study shows that CCR and DSS form an interesting infrastructure for eScience supporting with uniformly acceptable performance the hybrid efficiency-productivity layered programming model from multicore through Grids. Current performance results are not as good as the best for MPI [28-31] but MPI has the benefit coming from many years of experience. CCR and the underlying Windows multicore scheduler have not before been applied to this style of messaging in intense environments and we expect significant improvements in CCR and DSS performance for both managed code and even more so C++ and native implementations. Further CCR supports the important dynamic threading and the coarse grain functional parallelism for which MPI is usually non optimal. In particular, we expect discrete event simulation to run well on a CCR base. We expect our work to suggest simplifications and extensions to MPI to support the rich messaging needed in hybrid Grid-multicore environments We have also finished extensive benchmarks on MPICH, and MPJ Express [3] (and mpiJava [27]) to cover the very best classic MPI's. Further details of this analysis can be found in [32, 33] which also study in depth memory bandwidth and the Intel8a machine.

7. 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] Jeffrey Dean and Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004 http://labs.google.com/papers/mapreduce.html

[3] 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

[4] 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

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

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

[7] 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

[8] 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

[9] 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

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

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

[12] 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/

[13] 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

[14] Internet Resource for HPCS Languages http://crd.lbl.gov/~parry/hpcs_resources.html

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

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

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

[18] "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.

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

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

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

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

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

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

[25] Geoffrey Fox "Messaging Systems: Parallel Computing the Internet and the Grid", EuroPVM/MPI 2003 Invited Talk September 30 2003. http://grids.ucs.indiana.edu/ptliupages/publications/gridmp_fox.pdf

[26] 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

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

[28] 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

[29] 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

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

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

[32] 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

[33] 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