High Performance Multi-Paradigm Messaging Run Time on Multicore Systems

Xiaohong Qiu
Research Computing UITS
Indiana University Bloomington

Geoffrey Fox
Community Grids Laboratory
Indiana University Bloomington

George Chrysanthakopoulos, Henrik Frystyk Nielsen
Microsoft Research
Redmond WA

Abstract

Broad deployment of multicore systems in commodity situations has highlighted the need for parallel environments that support a wider range of application than those on traditional parallel supercomputers. Further we need to build composite jobs made of heterogeneous applications. A common underlying runtime that supports multiple programming paradigms appears to be one important part of future parallel computing environments integrating cluster and multicore parallelism. Here we examine CCR or the Concurrency and Coordination Run time from Microsoft as a multi-paradigm run time supporting three key parallel models: full MPI collective messaging, asynchronous threading and coarse grain functional parallelism or workflow. It is an attractive multi-paradigm candidate as it was designed to support general dynamic message patterns with high performance and already has a distributed, REST oriented runtime known as DSS built on top of it. We present 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 the AMD based PC, latencies from 4μs in basic asynchronous threading and point to point mode to 14 μs for a full MPI_SENDRECV exchange with all threads (one per core) sending and receiving 2 messages at a traditional MPI loosely synchronous rendezvous. Workflow latencies are measured as less than 40 μs with all results coming from the CCR and DSS freely available as part of the Microsoft Robotics Studio distribution. We present Intel PC latencies that are significantly higher but whose optimization has not been studied yet by us. Looking to the future, we suggest that the ease of programming custom collectives using the CCR primitives make it attractive to consider building a full MPI run time on top of it. This would have fully asynchronous queued messaging,

integration with workflow and thread-based programming, a rendezvous and active message mode, support of managed code (C#) and ability to run on cluster and multicore systems. Although the current CCR has reasonable performance for MPI primitives, it would be important to improve this and current core CCR plans should lead to a factor of 2 lower latencies. We also will investigate the origins of the differences between Intel and AMD results.

1. Introduction

Multicore architectures are bringing parallel computing to a broad range of applications with profound impact on hardware, systems software and applications [1-3]. The programming models and runtime that will be used on multicore architectures are the subject of substantial academic and industry research and development as they must bridge between current commercial desktop and server systems, commercial parallel databases, distributed Grid environments and the massively parallel supercomputers largely aimed at science and engineering [4]. Intel [5] has examined classes of possible future desktop applications which they term RMS – Recognition, Mining and Synthesis. This can be illustrated by a sophisticated datamining application that first accesses a parallel database and then runs analysis algorithms including perhaps a multithreaded branch and bound search, and a SVM Support Vector Machine built on parallel linear algebra followed by sophisticated visualization. This composite application would need to be controlled by a coarse grain executive similar to Grid workflow [6] or Google MapReduce [7]. The individual datamining filters could use either the dynamic thread parallelism appropriate for search algorithm or an MPI style messaging for parallel linear algebra used in the SVM filter. Further we would like this job to run efficiently and seamlessly either internally to a single CPU or across a tightly coupled cluster or distributed Grid. In this paper we address a small part of the multicore puzzle – namely what could be the runtime that could span these different environments and different platforms that would be used by the heterogeneous composite applications that could be common on future multicore applications for personal, commercial or research use. Managed code will be important for desktop applications and so C# (used here) and Java could become more important for parallel programming.

We examine the issues of building a multi-paradigm runtime with a particular example CCR which is a runtime [8-9] designed for robotics applications [10] but also investigated [11] 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. However this paper is not really proposing a 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 [12] from Darpa's HPCS High Productivity Computing System program supports the three execution styles (dynamic threading, MPI, coarse grain functional parallelism) we investigate here.

In the next section, we discuss CCR and DSS briefly while section 3 defines more precisely our three execution models. Section 4 presents our basic performance results that suggest one can build a single runtime that supports the different execution models. Future work and conclusions are in section 5.

2. 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.

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 first 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, 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.

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 [27] which is a SOAP-based protocol for managing structured data manipulations and event notifications. The combination of HTTP and DSSP allows services to expose their UI through traditional Web infrastructure mechanisms such as a browser as well as interacting efficiently with each other.

3. 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 [13] and double this number of interfaces in the more recent MPI-2 including support of parallel external I/O [14-15]. MPI like CCR is built around the idea of concurrently executing threads (processes, programs) that exchange information by messages. In the classic analysis [16-19], 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.

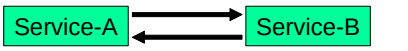

Fig. 1(a) 2-way Inter Service message Implemented in DSS

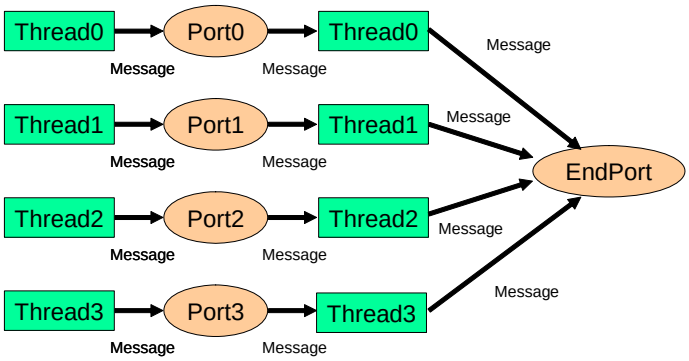

Fig. 1(b) Pipeline of Spawned Threads followed by a Reduction implemented using CCR Interleave

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 in this paper and will use DSS already developed for Robotics [10] on top of CCR for this case and idealized in fig. 1(a).

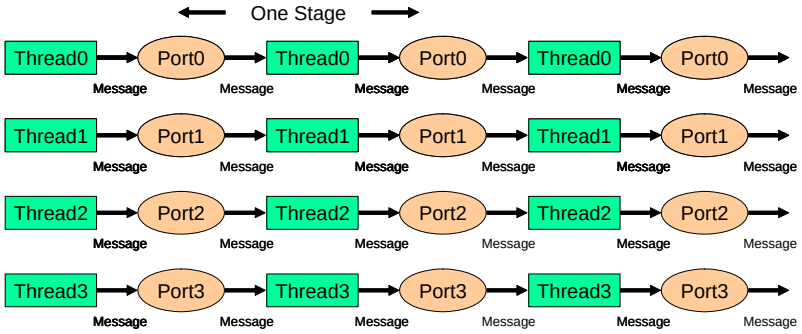

Fig. 2: Illustration of multiple stages used in CCR Performance Measurements

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 fig. 1(b). This also illustrates the simple Pipeline Spawn execution that we used for basic performance measurements of the

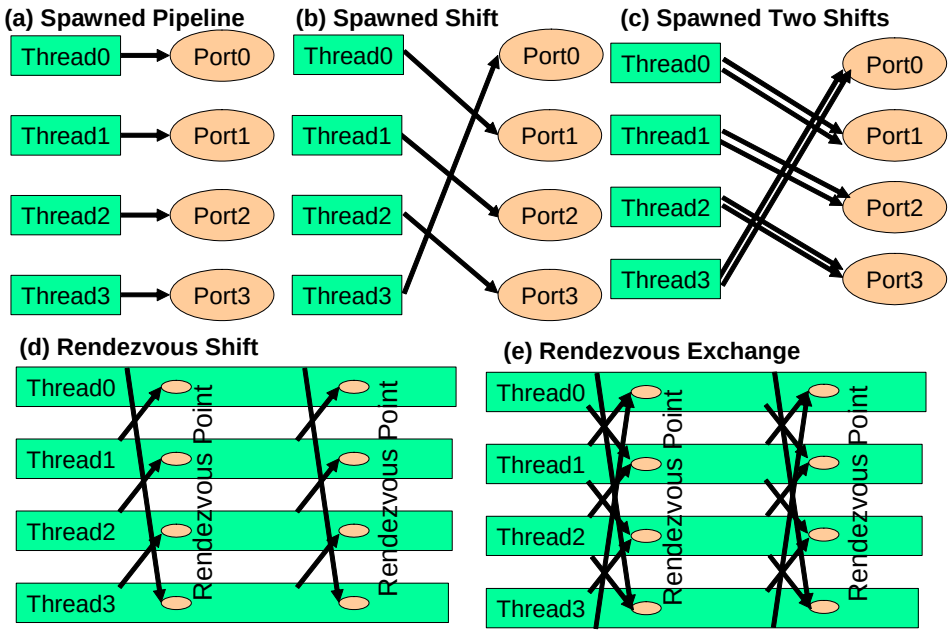

Fig. 3: Five CCR Communication Patterns used to test spawned dynamic threading (a,b,c) and MPI style Rendezvous's (d,e) and shown for 4 cores

dynamic threading performance. Each thread writes to a single port that is read by a fresh thread as shown in more detail in fig. 2.

We take a fixed computation that takes from 8 to 17 seconds 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 a 0.8 to 1.7 μs execution time and 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 different execution models. These spawned patterns are illustrated in fig. 3(a-c) and augment the pipeline of fig. 1(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 1 to 3 are drawn for 4 cores while our tests used both 4 and 8 core systems.

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 fig. 3(d) or the combination of a left and right shift, namely an exchange, shown in fig. 3(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 [2, 20] 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.

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

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. We illustrate the CCR structure with the Pseudocode corresponding to an MPI exchange pattern given above in Code sample 1.

We performed measurements on 3 machines labeled AMD, INTEL4 and INTEL8. The machine termed AMD had 2 gigabytes of memory and two AMD Opteron chips – each with two cores running at 2.4 GHz speed. The INTEL4 machine was a Dell Precision Workstation 670 with two dual-Core Intel® Xeon™ Processors running at 2.80GHz with 2x2MB L2 cache. The INTEL8 machine was a workstation with two 2.64 GHz 4-core Intel® Xeon™ processors and a total of 4 gigabytes of memory. The AMD and INTEL8 run the Windows Vista Enterprise operating system in 32bit and 64bit mode respectively. The INTEL4 machine ran Windows XP Professional 64 bit edition and this was also used on earlier models of AMD and INTEL8 machines we used in our first tests reported below on the patterns in figs. 3(a-c). This initial AMD platform had a 2.19 Ghz clock while the INTEL8 ran at 1.86 GHz in our first work. For the final version of the paper, we will rerun all our measurements on the newer machines which will improve the performances reported below for pipeline, shift and "2 shifts" dynamic threading modes.

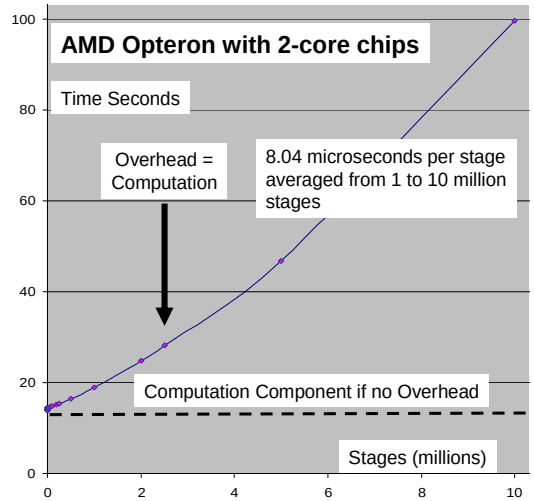

Fig. 4(a): Fixed amount of computation (4.10^7 units) divided into 4 cores and from 1 to 10^7 stages on HP Opteron Multicore termed AMD. Each stage is separated by reading and writing CCR ports in Pipeline mode

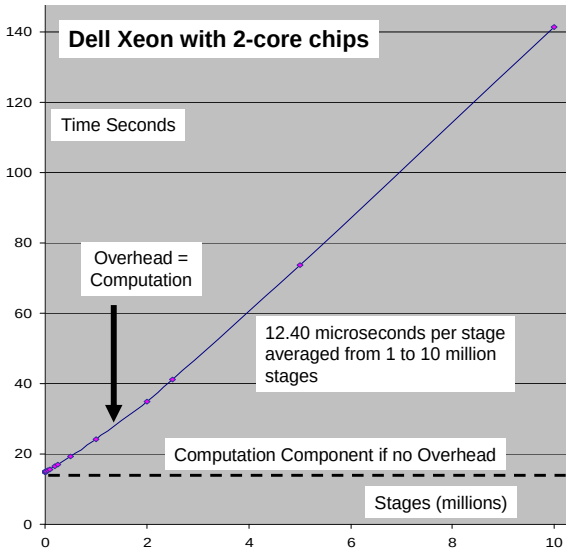

Fig. 4(b): Fixed amount of computation (4.10^7 units) divided into 4 cores and from 1 to 10^7 stages on Dell Xeon Multicore termed INTEL4. Each stage separated by reading and writing CCR ports in Pipeline mode

4. Performance of CCR in 3 Execution Models

4.1 CCR Message Latency and Overhead

We illustrate our approach by discussing the simple pipeline pattern of fig. 3(a) for the AMD and INTEL4 machines. In figs 4(a) and 4(b), we plot the total execution time for a series of computations. Each ran 4.10^7 repetitions (10^7 repetitions on each of four cores) of the basic 1.4 microsecond compute activity (it is this long on the 2.19 Ghz AMD, it was 1.5 microsecond on INTEL4) on 4 cores. The repetitions are achieved by either a simple loop inside the thread of basic computation unit or by splitting into separate stages separated by writing and reading CCR ports. This simple strategy ensures that without threading overhead the execution time will be identical whether one divides computation by loops or by CCR stages; this way we can get accurate estimates of the overhead incurred by the port messaging interface.

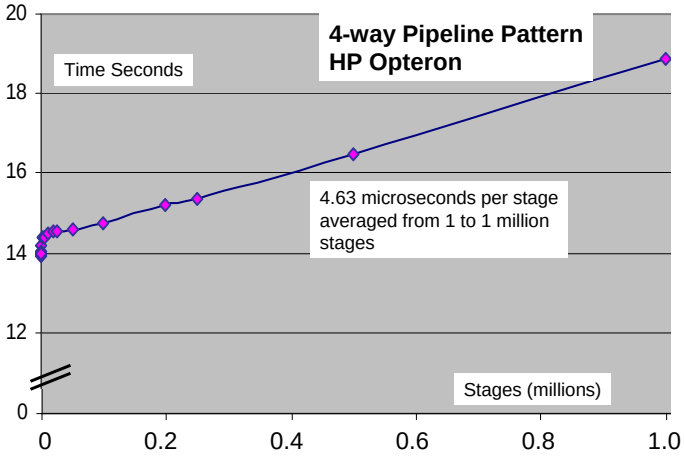

Fig. 5(a): Detail from Fig.4(a) for 1 to 1 million stages on AMD Machine

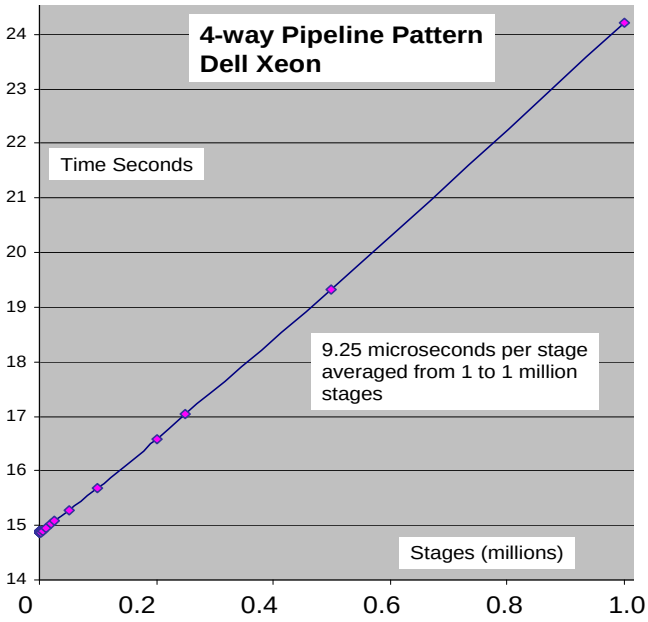

Fig. 5(b): Detail from Fig.4(b) for 1 to 1 million stages on INTEL4 multicore.

We first analyze the AMD results where without overhead the execution time will be about 14 seconds and is shown as a dashed line in figure 4(a). The figure takes these 4.10^7 repetitions and plots their execution time when divided into stages of the type shown in figure 2. Each measurement was an average over at least 10 runs with a given set of parameters. Figure 4 shows the results plotted up to 10

million stages while figure 5 shows the detail for up to one million stages. Always we use the term overhead to represent the actual measured execution time with subtraction of the time that a single stage would take to execute the same computational load. Overhead corresponds to latency in typical MPI benchmarking parlance. Figure 4 marks as “overhead=computation”, the point where measured execution time is twice that taken by a single stage. For 10 million stages the overhead on the AMD is large – almost 85 seconds; this corresponds to a set of loosely synchronous stages lasting 9.9 microseconds which is mainly overhead as the “real” computation is just 1.4 microseconds per stage. The INTEL4 results show 125 seconds overhead in this extreme case.

Looking at the case of one million stages, the overhead is smaller – about 5 seconds (for AMD and 9 seconds for INTEL4); for the AMD, this corresponds to a set of loosely synchronous stages lasting 19 microseconds where the overhead is about 5 microseconds and the “real” computation is 14 microseconds (a loop of ten basic computation unit) per stage. Linear fits to the stage dependence leads to an overhead per stage of 4.63 microseconds from figure 5(a) while the behavior becomes somewhat nonlinear in the larger range of stages in figure 4(a). This overhead represents the CCR (and system) time to set up threads and process the ports. Our measurement says this overhead is linear in the number of invocations when the spawned threads execute for substantially more (14 microseconds) than the basic overhead (4.63 microseconds) but the overhead increases when the thread execution time decreases to a few microseconds. Turning to the INTEL4 results they are qualitatively similar but with significantly higher overhead. In figure 4(b), the average has increased from 8.04 for AMD to 12.66 microseconds for INTEL4 with the execution of 10 million stages taking 40% longer than the AMD even though the execution time with 1 stage is only 7% longer. Comparing figs 5(a) and 5(b), shows the discrepancy between AMD and INTEL4 CCR performance to be exacerbated if one restricts attention to just the first one million stages.

We summarize in Table 1 several styles of runs in terms of the overheads on the case of 500,000 stages when the stage computations takes approximately 17 (AMD running at 2.4 GHz used in rendezvous measurements of figs. 3(d-e)) 28(AMD running at 2.19 GHz used in patterns of figs. 3(a-c)) 24(INTEL8 used in patterns of figs. 3(d-e)), 30(INTEL4) or 34(INTEL8 used in patterns of figs. 3(a-c)) microseconds. We only list overhead or latency per stage in the messaging by subtracting out the stage computations from

the run time. We look at 1-4 way parallelism for AMD and INTEL4 and 1-8 way parallelism for INTEL8. Not surprisingly when the requested parallelism is less than the maximum of cores, the system is able to use the "free" cores for port/message operations and reduce the stage overhead. We did use the AMD thread debugger to verify that the system made efficient use of cores but this did not have the microsecond resolution needed for an in depth study. Table 1 has all 5 patterns of fig. 3 including the three spawned thread and two rendezvous style. For the dynamic spawned thread case, shift is very similar to pipeline as one might expect as both have the same number of port read/write operations. The two shift case of fig. 3(c) shows that the overhead roughly doubles as we double the number of reads and writes.

Table 1: Stage overheads in microseconds for the five CCR patterns illustrated in Figure 3 and calculated from the 500,000 stage runtimes

a) AMD Stage Overhead (microseconds)	**Number of Parallel Computations**			
	1	**2**	**3**	**4**
Straight Pipeline	0.77	2.4	3.6	5.0
Shift	N/A	3.3	3.4	4.7
Two Shifts	N/A	4.8	7.7	9.5
Rendezvous Shift	N/A	7.1	9.0	9.7
Rendezvous Exchange	N/A	8.8	13.6	14.2

b) INTEL4 Stage Overhead (microseconds)	**Number of Parallel Computations**			
	1	**2**	**3**	**4**
Straight Pipeline	1.7	3.3	4.0	9.1
Shift	N/A	3.4	5.1	9.4
Two Shifts	N/A	6.8	13.8	13.4

c) INTEL8 Stage Overhead (microseconds)	**Number of Parallel Computations**					
	1	**2**	**3**	**4**	**7**	**8**
Straight Pipeline	1.33	4.2	4.3	4.7	*	6.5
Shift	N/A	4.3	4.5	5.1	*	7.2
Two Shifts	N/A	7.5	6.8	8.4	*	22.8
Rendezvous Shift	N/A	6.6	8.9	10.4	14.4	17.3
Rendezvous Exchange	N/A	8.9	15.6	15.2	22.3	28.3

Notes: * marks missing measurements that will be added before publication

Now we study in detail the Rendezvous measurements which exactly mimic MPI and are depicted in figs. 3(d) and 3(e). Their overheads for 500,000 stages are also recorded in table 1 and the variation of performance with number of stages is given in figures 6(a) for AMD up to ten million and in fig. 6(b) for INTEL8 up to one million stages. We looked at two implementations of rendezvous exchange – in the first

each thread issued writes to the left and right neighboring ports and then waited till its two messages were available. This uses a new version of CCR's *MultipleItemReceive*. The second approach (which is shown in figs. 6 and 7 but not table 1) simple achieves an exchange as a right shift followed by a left right with no computation in between. This uses two calls per stage to CCR's *Receive* primitive; the same call used once per stage by spawned pipeline and shift.

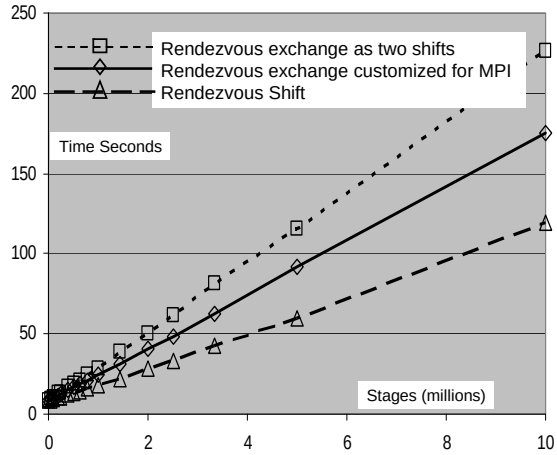

Fig. 6(a): Performance of AMD PC with 4 execution threads on MPI style Rendezvous Messaging for Shift and Exchange implemented either as two shifts or as custom CCR pattern

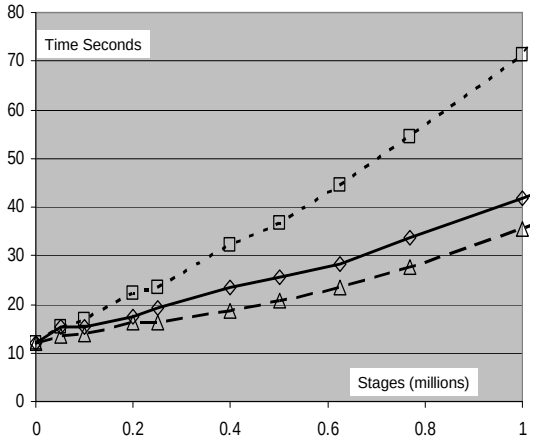

Fig. 6(b): Performance of INTEL8 PC with 8 execution threads on MPI style Rendezvous Messaging for Shift and Exchange implemented either as two shifts or as custom CCR pattern

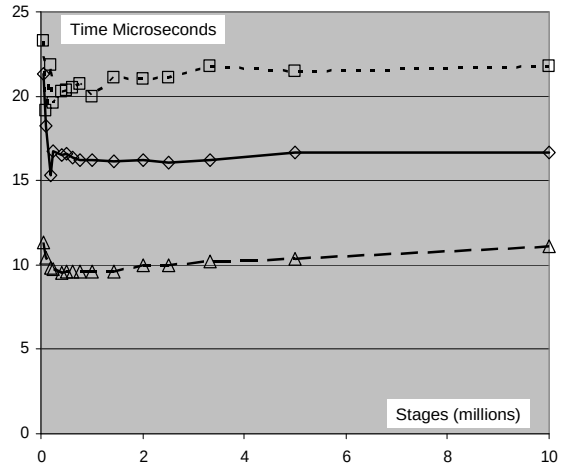

Fig. 7(a): Overhead (latency) of AMD PC with 4 execution threads on MPI style Rendezvous Messaging for Shift and Exchange implemented either as two shifts or as custom CCR pattern

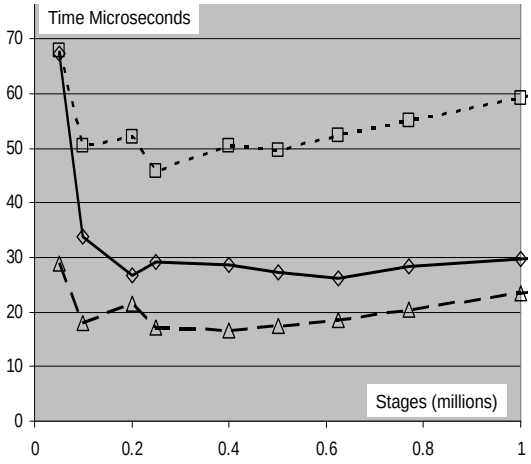

Fig. 7(b): Overhead (latency) of INTEL8 PC with 8 execution threads on MPI style Rendezvous Messaging for Shift and Exchange implemented either as two shifts or as custom CCR pattern

It is interesting to see that the special CCR primitive optimized for exchange easily outperforms the simpler "exchange as two shifts" and demonstrates the value of being to develop new patterns directly in the managed code environment.

The AMD PC always outperforms INTEL8 in both table 1 and figures 6 and 7 but this is somewhat unfair as the management of the 8 parallel threads in the INTEL8 case is more demanding than the 4 needed by AMD and this impacts both CCR and the underlying Windows scheduler. The scaling of multicore chip performance as the number of cores increase will be an important area for future study.

Figs. 7(a, b) show the messaging overhead as a function of the number of stages which is essentially the slope of the run time measurements in fig. 6. The AMD results are remarkably constant up to 10 million stages that illustrates that both CCR and the Windows O/S are able to schedule the 4 AMD cores in an efficient fashion with no interference between the stages. We note the early results were less good and the presented results come from optimizing the CCR-Windows scheduling interface. Especially for INTEL8, there are hints of an anomaly to the left of figs. 7(a, b) around 50,000 stages where the overhead surprisingly rises. We have understood this as due to non-optimal Windows scheduling and are working with the responsible Microsoft team on thread scheduling in this unusual (for commodity applications) case of highly correlated messaging. We believe that the MPI message pattern performance in figs. 6 and 7 and table 1 are excellent for the AMD and acceptable for INTEL8. We anticipate the results to improve with further analysis of results and optimizations.

4.2 CCR Message Bandwidth

We now briefly discuss some measurements of message bandwidth supported by CCR. We simulated a typical MPI CALL such as *subroutine mpisend (buf, count, datatype, dest, tag, comm)* by posting a structure made up of an array of doubles of length N and a set of six integers (with one extra integer for CCR control). These all used the full 4-way parallelism of AMD and INTEL4 and typical results are shown in fig. 8 with more detail available on request. One passes a reference for the data buffer *buf* and we used three distinct models for locations of final data termed respectively

a) **Inside Thread**: The buffer *buf* is copied using C# *Copy To* function to a new array inside the thread whose size is identical to that of *buf*.

b) **Outside Thread**: The buffer *buf* is copied using C# *Copy To* function to a fixed array (with distinct arrays for each core) outside the thread whose size is identical to that of *buf*.

c) **Stepped Array Outside Thread**: The buffer *buf* is copied using *element by element Copy* in C# to a fixed large array outside the thread whose size is two million double floating words. Again each core has its own separate stepped array.

Note all measurements in this section involved 4-way parallelism and the bandwidth is summed over all four cores simultaneously copying message buffers.

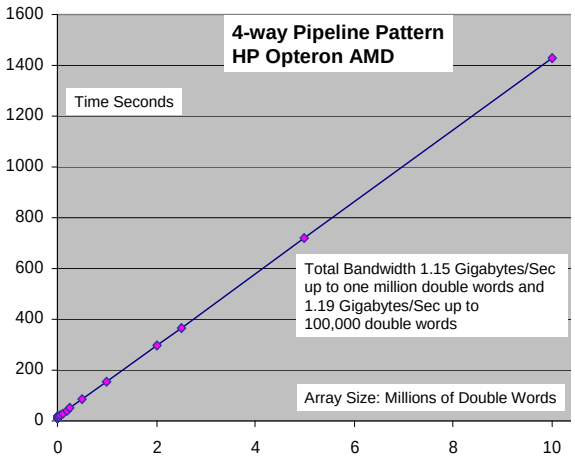

Fig. 8(a): Scenario from Fig. 4(a) for 5,000 stages with run time plotted against size of double array copied in each stage from thread to stepped locations in a large array on HP Opteron Multicore.

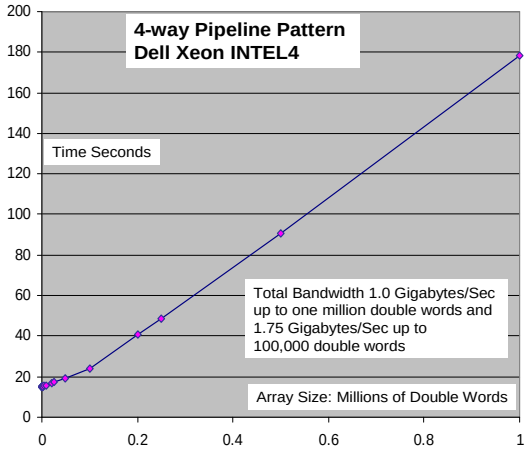

Fig. 8(b): Scenario from Fig.4(b) for 5,000 stages with run time plotted against size of double array copied in each stage from thread to stepped locations in a large array on Dell Xeon Multicore INTEL4 machine

In this bandwidth investigation, we fix the number of stages and look at the run time as a function of size of array stored in the port. One finds bandwidths that vary from 0.75 to 2 Gigabytes per second with the INTEL4 machine claiming both upper and lower values although it typically shows better bandwidth than the AMD machine. Note the best bandwidths are obtained when the destination arrays are outside the thread and when of course the copied data is small enough to make good use of cache. Also the bandwidth is higher for the cases where the computing component is significant; i.e. when it has a value of a few milliseconds rather than the lower limit of a few microseconds. Figure 8 illustrates our benchmarks with a stage computation of 2800 (AMD) to 3000 (Intel) microseconds. For the case (c) of a stepped array, the INTEL4 PC achieves a bandwidth of 1.75 gigabytes/second for 100,000 word messages which decreases to just 1 gigabyte/second for million word arrays. The AMD machine shows a roughly uniform bandwidth of

1.17 gigabyte/second independent of array size. Note typical messages in MPI would be smaller than 100,000 words and so MPI would benefit from the performance increase for small messages.

4.3 DSS Message Latency and Overhead

We now examine CCR for the third form of parallelism; namely the functional parallelism model represented in fig. 1(a). The Robotics release [10] includes a lightweight service environment DSS built on top of CCR and we performed an initial evaluation of DSS on the AMD machine. In fig. 9, we time groups of request-response two way messages running on (different) cores of the AMD system. For a group of 200

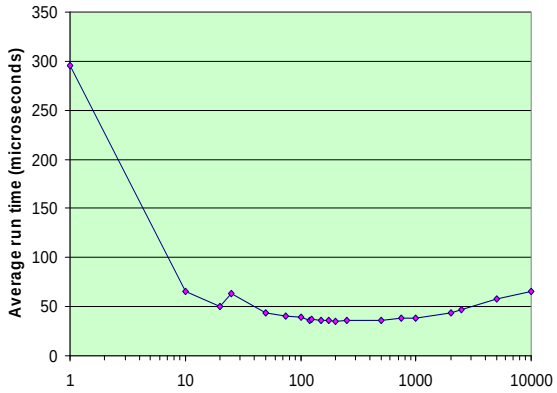

Fig. 9(a): Timing of HP Opteron Multicore AMD machineas a function of number of two-way service messages processed

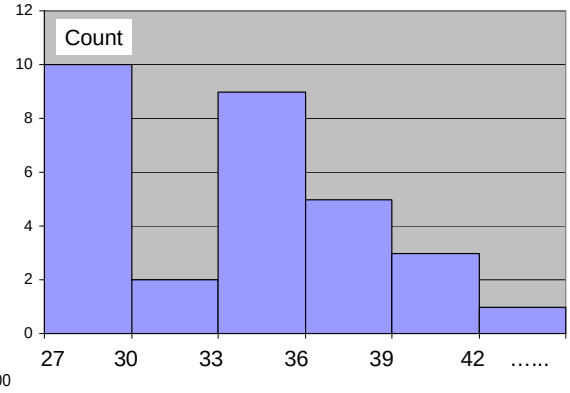

Fig. 9(b): Histogram of a set of 30 independent timings of HP Opteron Multicore AMD machine for 200 two-way service messages processed

messages we histogram the timings of 30 separate measurements. For low message counts DSS initialization bumps up the run time and for large groups of messages it increases – perhaps due to overheads like Garbage Collection. For message groups from about 50-1000 messages, we find average times of 30-50 microseconds or throughputs of 20,000 to 25,000 two-way messages per second. 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 a fair comparison as the measurements of fig. 9 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 complete fairer comparisons later and also examine the important one-way messaging case.

5. Conclusions and Futures

This preliminary study suggests that CCR and DSS form an interesting infrastructure for parallel computing. We addressed this by showing they can support the three basic messaging runtime models used in parallel computing. We found overheads on the AMD machine that varied from about 4 μs for dynamic threads to less than 40 μs for a flexible functional parallelism model with MPI style rendezvous' in between. The AMD machine had a 14 μs overhead for rendezvous exchange (MPI_SENDRECV) that was little changed as we varied the computation per stage in the range of 1 to 100 μs. This is not as good as the current best MPI [21-24] but MPI has the benefit of 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 over the next few months and would include updated figures in the published version of this paper. We will also complete the INTEL4 measurements and use our new faster INTEL8 and AMD PC's for the dynamic thread case. We are also doing comparable multicore benchmarks on MPICH, OpenMPI and MPJ Express [25] (and mpiJava [26]) to cover the very best classic MPI's with in addition Java implementations giving us another managed code example. Further details of current analysis can be found in [28].

REFERENCES

1. 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
2. Herb Sutter, *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*, Dr. Dobb's Journal, 30(3), March 2005.
3. Annotated list of multicore Internet sites http://www.connotea.org/user/crmc/
4. 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

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. 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
7. 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
8. "Concurrency Runtime: An Asynchronous Messaging Library for C# 2.0" George Chrysanthakopoulos Channel9 Wiki Microsoft http://channel9.msdn.com/wiki/default.aspx/Channel9.ConcurrencyRuntime
9. "Concurrent Affairs: Concurrent Affairs: Concurrency and Coordination Runtime", Jeffrey Richter Microsoft http://msdn.microsoft.com/msdnmag/issues/06/09/ConcurrentAffairs/default.aspx
10. Microsoft Robotics Studio is a Windows-based environment that provides easy creation of robotics applications across a wide variety of hardware. It 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/ with tutorials at http://msdn.microsoft.com/robotics/learn/On_Demand/default.aspx
11. 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
12. Internet Resource for HPCS Languages http://crd.lbl.gov/~parry/hpcs_resources.html
13. Message passing Interface MPI Forum http://www.mpi-forum.org/index.html
14. MPICH2 implementation of the Message-Passing Interface (MPI) http://www-unix.mcs.anl.gov/mpi/mpich/
15. High Performance MPI Message Passing Library http://www.open-mpi.org/

16. "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.
17. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker *Solving Problems in Concurrent Processors-Volume 1*, Prentice Hall, March 1988.
18. Fox, G. C., Messina, P., Williams, R., "Parallel Computing Works!", Morgan Kaufmann, San Mateo Ca, 1994.
19. 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
20. 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
21. 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
22. 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
23. Thomas Bemmerl "Pallas MPI Benchmarks Results" http://www.lfbs.rwth-aachen.de/content/index.php?ctl_pos=392
24. Myricom *Myri-10G and Myrinet-2000 Performance Measurements* http://www.myri.com/scs/performance/
25. 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

26. mpiJava Java interface to the standard MPI runtime including MPICH and LAM-MPI http://www.hpjava.org/mpiJava.html
27. Henrik Frystyk Nielsen, George Chrysanthakopoulos, “Decentralized Software Services Protocol – DSSP” http://msdn.microsoft.com/robotics/media/DSSP.pdf
28. 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