Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System

June 30, 2003

Terry Jones,
Shawn Dawson,
Rob Neely

Lawrence Livermore National Laboratory
Livermore, CA, USA
94550

William Tuel,
Larry Brenner,
Jeffrey Fier,
Robert Blackmore,
Patrick Caffrey

International Business Machines Corporation
Armonk, NY, USA
10504

Brian Maskell,
Paul Tomlinson,
Mark Roberts

Atomic Weapons Establishment
Aldermaston Reading, UK
RG7 4PR

Abstract

A parallel application benefits from scheduling policies that include a global perspective of the application's process working set. As the interactions among cooperating processes increase, mechanisms to ameliorate waiting within one or more of the processes become more important. In particular, collective operations such as barriers and reductions are extremely sensitive to even usually harmless events such as context switches among members of the process working set. For the last 18 months, we have been researching the impact of random short-lived interruptions such as timer-decrement processing and periodic daemon activity, and developing strategies to minimize their impact on large processor-count SPMD bulk-synchronous programming styles. We present a novel co-scheduling scheme for improving performance of fine-grain collective activities such as barriers and reductions, describe an implementation consisting of operating system kernel modifications and run-time system, and present a set of empirical results comparing the technique with traditional operating system scheduling. Our results indicate a speedup of over 300% on synchronizing collectives.

1. Introduction

Traditional operating systems based on UNIX® and its variants (including AIX® and Linux) have serious deficiencies for large-scale parallel environments such as those of interest at national laboratories and supercomputing centers. This is largely a historical artifact: when UNIX was developed over 30

SC'03, November 15-21, 2003, Phoenix, Arizona, USA

years ago, no consideration was given to the types of issues that arise with parallel applications spanning multiple computers and operating system instances [Thompson74]. Today's operating systems lack any awareness of large parallel applications, instead viewing them as thousands of independent processes. A primary example of UNIX deficiencies with respect to parallel applications can be found in the way processes are dispatched by the multi-tasking scheduler on SMP nodes. Our findings indicate that serialization caused by uncoordinated scheduler activity dramatically impacts parallel applications, especially during synchronization or fine-grain parallelism.

We will discuss kernel scheduling improvements and an external time-based co-scheduler prototype and show their effectiveness.

Previous research activities [Ousterhout82; Sobalvarro97; Karl97] have not resulted in capabilities available to production super-computing facilities, in which, typically, a single job consisting of thousands of cooperating processes occupies a dedicated portion of the computing complex. Since the machines are typically SMP nodes, a node is assigned as many processes as there are processors on the node, and each process acts as if it has exclusive use of the processor. In this environment, fair share CPU scheduling and demand-based co-scheduling required for networks of workstations (NOWs) are not necessary. What is needed is collaborative scheduling of the job processes both within a node and across nodes, so that fine grain synchronization activities can proceed without having to experience the overhead of making scheduling requests.

The remainder of this paper describes our efforts to improve the scalability of large processor count parallel applications. Our findings show that even when interference present in full-featured operating systems such as daemons and interrupts cannot be removed, their impact to fine-grain synchronization can be greatly diminished through an operating system employing parallel aware co-scheduling techniques. The format for this paper is as follows: In section 2, we describe how common system operations result in scaling issues and discuss why this is of paramount importance to the common supercomputer class of large processor-count bulk-synchronous SPMD applications. In section 3, we describe the intra-node techniques we developed. In a similar vein, Section 4 describes the inter-node techniques we developed. In section 5, we present our performance results from empirical measurements. Sections 6 and 7 present related work and conclusions respectively. Finally, section 8 contains acknowledgements.

2. Why failure to coordinate routine system operations with the parallel workload degrades scalability

Synchronizing collective operations are operations in which a set of processes (frequently every process) participates and no single process can continue until every process has participated. Examples of synchronizing collective operations from the MPI interface are MPI_Barrier, MPI_Allreduce, and MPI_Allgather. These operations pose serious challenges to scalability since a single instance of a laggard process will block progress for every other process. Unfortunately, synchronizing collective operations are required for a large class of parallel algorithms and are quite common.[Gupta91]

A *cascading effect* results when one laggard process impedes the progress of every other process. The cascading effect has significant operating system implications and it proves especially detrimental in an HPC context: while operating systems may be considered very efficient in a serial context, even minimal system and/or daemon activity proves disastrous due to the cascading effect in the large processor count parallel environment common in HPC centers. Experiments show that typical operating system and daemon activity consumes 0.2% to 1.1% of each CPU for large dedicated RS/6000® SP® systems with 16 processors per node. [Jones03] Examples of these serializing system activities include daemons associated with file system activity, daemons associated with membership services, monitoring daemons, cron jobs, and so forth. For example, the AIX syncd daemon flushes file system buffers in memory to

rotating media, and the mld daemon manages the IP traffic for the dual switch fabric.

The degree of overlap is a key component in determining the impact of random events on synchronizing collectives. Figure 1 graphically portrays two separate runs of a parallel application. In the top instance, system activity (denoted as red rectangles) occurs at purely random times. As a result, operations that require every processor can make progress only when blue is present across all eight. The green rectangles show those periods in time when the application is running across all 8 processors. In the bottom portrayal, the same amount of system activity occurs (there is the same total amount of red) but it is largely overlapped. This means much more time is available for parallel application activities that require all processors, as shown by the larger green rectangles.

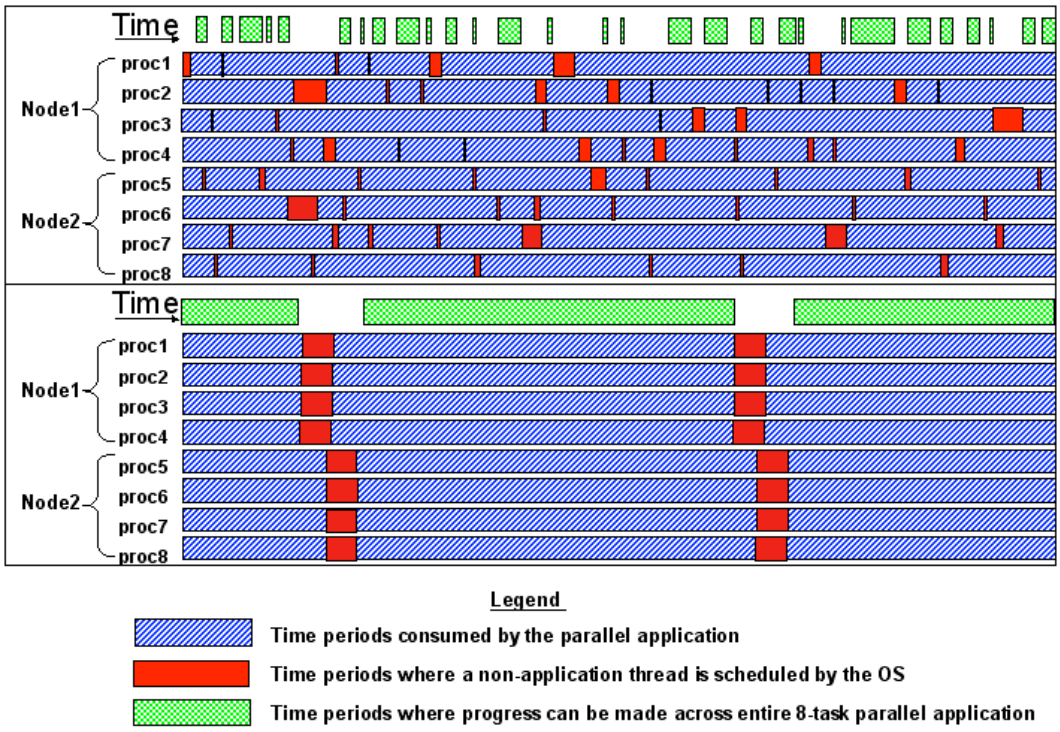

Figure 1: The above figure depicts two schedulings of the same eight-way parallel application. In the lower depiction, co-scheduling increases the efficiency of the parallel application as indicated by the larger amount of time periods where progress can be made across the entire 8-task parallel application. The top legend is blue; the middle legend is red, and the bottom legend is green. Adapted from Jones02.

For clusters comprised of SMP nodes, both inter- and intra-node overlap is an issue. Notice that if the eight processors in Figure 1 are spread across two 4-way SMP nodes, it is desirable to ensure overlap between nodes as well as on-node. The bottom run shows very good on-node overlap of operating system interference but little cross-node overlap of operating system interference.

Parallel applications are most susceptible to operating system interference during fine-grain operations such as ring communication patterns, barriers, and reductions. For the Bulk-Synchronous SPMD model, each cycle contains one or more such fine-grain operations (see Figure 2). The importance of these collective synchronizing operations is dependent on the duration of computation and communication periods. Typical cycles last anywhere from a few milliseconds to many seconds.

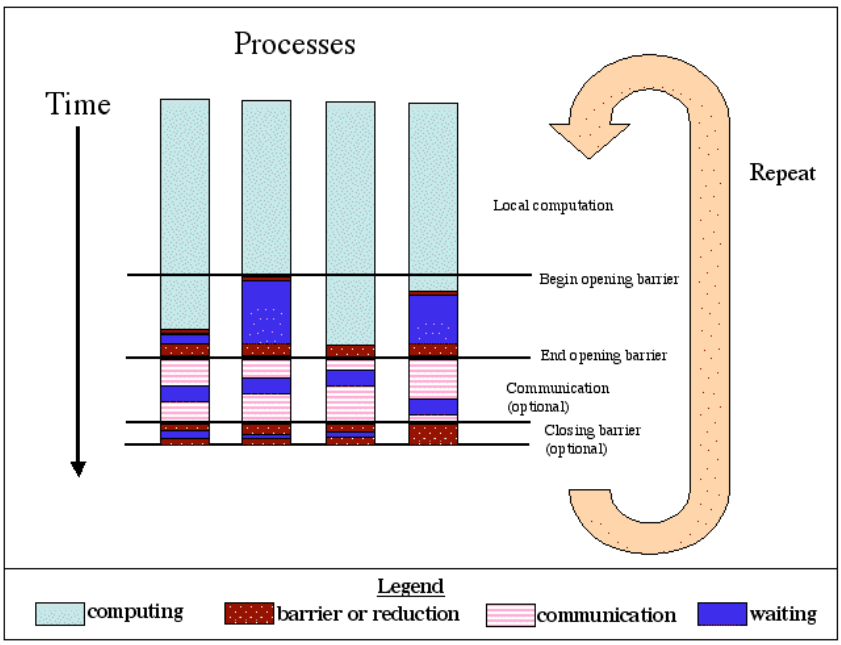

Figure 2: Bulk-Synchronous SPMD Model of parallel application. Each process of a parallel job executes on a separate processor and alternates between computation and communication phases. Adapted from Dusseau96.

The ability of any large processor count cluster to perform parallel applications with synchronizing collectives will depend heavily upon the degree of interference introduced by the operating system. Measurements taken from jobs run on ASCI White and ASCI Q at LLNL and LANL indicate Allreduces consume more than 50% of total time at 1728 processors. [Dawson03] A second study conducted by different researchers on ASCI Q measured Allreduces to consume around 50% of total time at 1728 processors, and over 70% of total time at 4096 processors. [Hoisie03]

Problematic interference such as timer decrement interrupt processing and daemon activities are inherent in typical UNIX derivatives and are not specific to AIX. This would suggest that the problem is common to UNIX derivatives. As expected, recent studies have shown both large variability and reduced synchronous collective performance in large Tru64 UNIX systems [Hoisie03]. Furthermore, anecdotal evidence collected at LLNL suggests Linux is also susceptible.

Developers and users of parallel applications have learned to deal with poor Allreduce performance by leaving one CPU idle on a multi-CPU (MP) node. This approach leaves a reserve CPU for processing daemons which would otherwise interfere with fine-grain activities. However the approach is undesirable since such strategies enforce a ceiling on machine efficiency. In addition, the approach does not handle the occasional event of two concurrent interfering daemons. Finally, it artificially limits the maximum scalability of the machine as one CPU is forfeited for every node on the machine.

Our work demonstrates that it is possible to mitigate the effects of system software interference without the drawbacks of underutilized MP nodes.

3. Adding Parallel Awareness to Intra-Node Scheduling

While the AIX operating system is able to run work simultaneously on multiple processors, it is not designed to start work simultaneously on multiple processors. There is no issue when processors are idle: if two threads are readied almost simultaneously, two idle processors will begin running them essentially immediately. AIX handles the busy processor case differently. When work is made ready in the face of busy processors, it must wait for the processor to which it is queued. (Should another processor become

idle, it may beneficially “steal” the thread, but this is atypical when running large parallel applications.) If the newly ready thread has a better execution priority than the currently running thread on its assigned processor, the newly ready thread pre-empts the running thread. If the processor involved is the one on which the readying operation occurred, the pre-emption can be immediate. If not, the other, busy, processor must notice that a pre-emption has been requested.

This happens whenever its running thread:

- Enables for interrupts in the kernel, as during a system call
- Takes an interrupt, as when an I/O completes or a timer goes off
- Blocks, as when waiting for I/O completion such as for a page fault

The problem is that this can represent a significant delay. In the worst case, it may be 10 msec until the next routinely scheduled timer interrupt gives the busy processor's kernel the opportunity to notice and accomplish the pre-emption.

The AIX kernel already contains a capability called the "real time scheduling" option, which solves a large part of this problem. When this option is invoked, the processor causing a pre-emption will force a hardware interrupt to be generated for the processor on which the pre-emption should occur. While this is not immediate, we have observed that the pre-emption is typically accomplished in tenths of a millisecond, as opposed to several milliseconds without this option.

The existing "real time scheduling" option, however, was deficient in two critical areas for our purposes, which we improved:

1. It only forced an interrupt when a better priority thread became runnable. It did not force an interrupt for a "reverse pre-emption," which occurs when the priority of a running thread is lowered below that of a runnable, waiting thread.
2. It forced an interrupt to only one processor at a time. Once such an interrupt was "in flight," it did not generate further interrupts if the processor involved would be eligible to run the thread on whose behalf the previous interrupt had been generated.

The ability to effect rapid pre-emptions and reverse pre-emptions across processors is a major building block in our approach to improving MPI scalability.

3.1. Reducing the total amount of system overhead cycles

Given that uncoordinated system overhead was causing scalability problems, we first looked for ways to reduce that overhead.

3.1.1. Generate fewer routine timer interrupts

The periodic (100 times a second on every CPU) generation of timer interrupts is an ongoing source of system overhead, albeit a relatively small one. Nonetheless, we wanted to eliminate as much of this overhead as practical. To this end, we modified the AIX kernel by adding a "big tick" capability, so named because the periodic interrupts are generally known as tick interrupts in the UNIX world.

This is really a fairly straightforward objective to accomplish. We simply found every place in the kernel that was involved with tick processing, and, instead of adding 1 to various counters, we added the "big tick" constant. We generally chose a big tick constant value of 25, which means that we generated the physical tick interrupts only once where the default kernel would have generated 25. By taking just 4 of these interrupts a second instead of 100, substantial overhead is saved.

Taking less frequent ticks yields a secondary benefit as well. Since the major consumers of system overhead are timer triggered, this mechanism causes a larger number of overhead tasks (daemons) to be made ready simultaneously. This natural batching saves overhead but, more importantly, increases the

likelihood of multiple processors executing these overhead tasks simultaneously, reducing the net impact of this overhead on a parallel job.

3.1.2. Execute overhead tasks with maximum parallelism

In the AIX kernel, work may be queued either to a single processor (to maximize storage locality), or to all processors (to minimize dispatching latency). Normal operation is to queue work to specific processors for best overall throughput.

We overrode this behavior, allowing only the parallel job's threads to be queued to specific processors. We forced everything else (namely, the system daemons) to be queued to all processors. While this adds significant overhead to the daemons as they execute, it also maximizes the parallelism with which they execute. The goal was to minimize the total time that daemons were running, which, as a source of interference to a parallel job, is much more desirable than ensuring that each daemon runs with maximum efficiency.

For example, if we have two daemons that inherently take 3 msec each to run, we would rather run them simultaneously on two CPUs without regard for storage locality, and take perhaps 3.1 msec total time, than run them serially on a single CPU, and take 6 msec. The degradation to the daemons is insignificant compared to the benefit to the parallel job.

3.1.3. Run the various system daemons less frequently

As mentioned above, triggering the daemons less frequently as a side effect of "big ticks" is desirable. But this was not the main approach to reducing daemon overhead. We wrote a "co-scheduler" (described in Section 4) that aggressively and (almost) simultaneously sets the relative priorities of the daemons and the parallel job, with the deliberate intent of denying the daemons processor time for much longer than just the big tick interval. By allowing daemon work to "pile up" for seconds at a time, and only then allowing it access to processors, we can ensure that we force daemon work to be executed simultaneously. In this fashion, the impact of the collective overhead generated by these daemons on the parallel job is minimized.

For this priority-swapping scheme to work best, we need to be able to pre-empt simultaneously across CPUs, as described above.

3.2. Coordinating system interference

There is a lot to be gained by coordinating system interference as well as minimizing it.

3.2.1. Take timer "tick" interrupts "simultaneously" on each CPU

By deliberate design, the AIX kernel schedules its timer ticks in a "staggered" manner across the processors of an MP system. For example, on a 10-way MP, these interrupts are scheduled (in absolute msec) at times x, x+10, x+20, etc. on CPU 0. On CPU 1, they are scheduled at times x+1, x+11, x+21, etc., and so on. The underlying idea is to keep the timer and timer-driven code from running simultaneously on multiple processors, since they are likely to contend for the same resources, and the same locks.

With AIX 5.1, the timer code was changed to require only a shared (or read) lock for much of its operation. This allowed us to make the tick interrupts essentially simultaneous across the processors of an MP system. As was the case with the daemons, this trades a little absolute efficiency in the timer interrupt handlers for a lot of parallelism between them, reducing the overall impact of timer tick handling on a parallel job.

Incidentally, implementing these changes *as options* in a production operating system such as AIX requires some mechanism for selecting these options. We accomplished this by adding options to the "schedtune" command of AIX, which provides a consistent mechanism for invoking kernel options.

4. Adding Parallel Awareness to Inter-Node Scheduling

It has long been recognized that parallel applications running across multiple nodes may perform poorly and inconsistently at high task counts [Mraz94]. One reason is that tasks involved in a data exchange may not happen to run at the same time, so that one task has to wait for the other. The solution proposed by several authors is to provide a "co-scheduler" that runs on each node and ensures that the parallel tasks are scheduled at the same time. As mentioned in the Introduction, our requirements are somewhat different than those for a Network of Workstations (NOW) co-scheduler, and we focus our implementation on satisfying the requirements of a large parallel job running on a network of SMP nodes in a super-computing center.

There are several elements to the co-scheduler:

- obtaining a globally synchronized time
- determining which processes on a node are to be scheduled
- providing a schedule that provides good application performance but does not starve system tasks
- providing an administrative interface that controls which jobs are eligible for co-scheduling
- providing an "escape" mechanism that allows a task to request that system tasks be allowed to run--particularly distributed I/O services such as IBM General Parallel File System (GPFS) for AIX5L®.

Everything that was beneficially done to coordinate system interference within an MP system can also be done across a clustered system. In particular:

1. If we can synchronize the time of day across all CPUs of the cluster, we can schedule our tick interrupts at the same time cluster wide. This requires that tick interrupts not only occur with a given interval, but that these interrupts be scheduled at tick boundaries in real time. Thus, a 10 msec tick must be forced to occur at a time an exact multiple of 10 msec from a global reference time.
2. Similarly, we can extend the priority-based coordination of daemons such that they are run at the same times not just on the processors of an MP node, but on all processors of a clustered system.

Elements of the co-scheduler described above are available as part of the IBM Parallel Environment (PE) running on the IBM SP [IBM01]. Its operation, and that of a prototype co-scheduler based on it, are described below.

Briefly, when a parallel job starts under the Parallel Operating Environment (POE) component of PE, and requests that it be controlled by the co-scheduler, a daemon process is started on each node for the exclusive purpose of scheduling the dispatching priorities of the tasks of the job running on that node. It does this by cycling the process priority of the tasks between a favored and unfavored value at periodic intervals. The actual priorities, favored priority duty cycle, and adjustment period are obtained from an administrative file, /etc/poe.priority, on each node. Setting a process priority to a fixed favored priority value causes AIX to assign a processor to this process (assuming that there aren't higher priority processes already running), and hence to put the application task into a running state. Similarly, setting a process priority to a fixed unfavored priority causes AIX to assign some other process to the processor, if

there are processes with more favored priority waiting to be run. Naturally, the co-scheduler itself runs with an even more favored priority, but sleeps most of the time.

On the IBM SP with its switch interconnect, the switch provides a globally synchronized time that is available by reading a register on the switch adapter. The communication subsystem component of PSSP, the SP System software, provides a function that allows an ordinary user program to access the time register, and thus allows programs running on the entire SP to have a common synchronized time base. On startup, the daemon compares the low order portion of the switch clock register with the low order bits of the AIX time of day value, and changes the AIX time of day so that the low order bits of AIX and the switch clock match. Thus, after startup has been completed on each node, the low order portions of the AIX clock are synchronized with the switch clock and with each other. It is not necessary to match the high order portions, as long as the AIX dispatching changes described above are affected only by the low order portion of the clock. Naturally, NTP must be turned off, since it is also trying to adjust the AIX clock. Furthermore, the co-scheduler adjusts its operation cycle so that the period ends on a second boundary (i.e. the co-scheduler period ends at 10:01:00, 10:01:01, 10:01:02, etc., and not at 10:00:00.27, 10:01:01.27, etc.) This way, all of the co-scheduler favored priority adjustments are made at a synchronized time across all nodes, with no inter-node communication required between the co-scheduler daemons.

At startup, the co-scheduler knows the user ID and the schedule to be used, but doesn't know the process ID's of the parallel application, which could be started by a script called by the process that forked the co-scheduler. Identifying the processes to be scheduled is done by the MPI library component of Parallel Environment. The library provides a "control pipe" between the library and the Partition Manager Daemon (pmd) that is the common ultimate parent of all tasks for a specific job on that node. The control pipe is used to send initialization/termination messages to POE. In addition, when a task calls the MPI initialization routine, its process ID is sent as a message to the pmd, which forwards it to the co-scheduler via a pipe that remains open for the duration of the job. The co-scheduler reads any data from its pipe, and from that builds a list of processes on which to adjust scheduling priority. Normally, as soon as a process registers, it is actively co-scheduled. When the parallel job ends, the co-scheduler knows that the processes have gone away, and exits.

The Parallel Environment co-scheduler implements a schedule that consists of alternate intervals of favored and non-favored priority. The administration file specifies the favored and non-favored priority values, the overall scheduling period in seconds, and the percent of time that the application is to be dispatched with the favored priority value. The administrator is given wide latitude in choosing the values – it is possible to give the tasks priority over all other processes running on the node for a very long time. This can starve system daemons and make the node unusable – in our experiments we encountered situations in which the only way to recover control was to reboot the node. A period of about 10 seconds, with a duty cycle of 90 or 95% seems to work pretty well on the 16-processor SMP nodes used in the SP. As for the favored priority value, normal priority is 60, and "real-time" processes usually run with priorities between 40 and 60. A favored value of less than 40 will defer most system daemon activity to the end of the period. Some daemons may time out in that time, and may have to have parameter adjustments to extend their timeout tolerance. Alternatively, for example, with GPFS, one could set the priority of the GPFS (mmfsd) daemon to 40 and the favored task priority to 41, thus allowing GPFS to run whenever it needed to, but making the application more favored than anything else. It is important to profile the background workload to determine how much system resource is being used when the application is not running

The POE administrative interface is a file (/etc/poe.priority) that is root-only writable, and is assumed to be the same on each node. Each record in the file identifies a priority class name, user ID, and scheduling parameters as mentioned above. A user wishing to have a job controlled by the co-scheduler sets the POE environment variable MP_PRIORITY=<priority class name>. At job start, the administrative file is searched for a match of priority class and user ID. If there is a match, the co-scheduler is started. Otherwise, an attention message is printed and the job runs as if no priority had been requested. There has been some dissatisfaction with this particular way of specification, and alternatives (perhaps using group ID instead of user ID) are being considered.

Many applications have I/O phases, particularly at job start, but also at job checkpoints and termination, during which the application depends on system daemon activity (GPFS, syncd, NFS daemons, etc.) to complete the I/O. During these periods, it is desirable for the application to have normal dispatching priority, not favored, as there is nothing the application can do while waiting for the I/O daemons to complete. Since the co-scheduler is unaware of the details of the application, the prototype MPI library provides an API that allows the application to request that it be detached from the co-scheduler, and another that allows the application to request that it be attached to the co-scheduler. These API calls are implemented via messages to the co-scheduler passing through the control pipe to the pmd, and then on the pipe from the pmd to the co-scheduler, which acts on the request when it sees it. Since there is no communication between the co-schedulers, if the application wants these calls to be synchronized, it will need to provide its own barrier calls as well.

5. Performance Results

In this section we describe the test runs undertaken to determine the effectiveness of the kernel modifications and co-scheduler in addressing performance variability and scaling problems encountered with the standard AIX kernel. The tools employed were a small benchmark code, a production application, and the AIX trace utility.

5.1. Benchmarks

Anomalously poor scaling behavior of MPI_Allreduce was noted when running more than 1024 task jobs. The standard tree algorithm for MPI_Allreduce does no more than $2*\log_2(N)$ separate point to point communications to complete the reduction where N is the number of tasks in the MPI communicator. The time to complete a call to MPI_Allreduce should increase in proportion to the log of the number of tasks participating in the call. For a relatively small number of tasks, the scaling exhibited nearly the expected logarithmic behavior. Starting with about 512 tasks the call time began to deviate from the expected logarithmic behavior and soon became nearly linear with the number of tasks.
In order to isolate the scaling problem a synthetic benchmark, aggregate_trace.c, was created. This program was intended to simulate the sorts of tasks programs may perform in the section of code where they use MPI_Allreduce. In this particular code, three loops are done where the timings of 4096 MPI_Allreduce calls were measured.

In addition to the overall timings, a call to AIX trace was done before and after every 64th call to MPI_Allreduce. The trace calls provide two important benefits. First, when other AIX trace hooks are turned on, it allows us to determine what other processes are running while the aggregate_trace.c program is delayed. Second, since there are 64 blocks of 64 MPI_Allreduce calls in the 4096 calls to this function, it allows a much better statistical picture of the MPI_Allreduce performance. Whereas it is very likely that some of the MPI calls will be delayed in a loop of 4096 calls, it is possible that sometimes no calls will be delayed. The AIX tracing was enabled only during the time that the loop of calls to MPI_Allreduce was active.

Since the kernel modifications and co-scheduler significantly change the scheduling behavior of AIX, we knew tests using real applications were needed. Although the aggregate_trace benchmark replicates the fine-grain synchronization that is common in many of our production codes, it uses only a minimal subset of the MPI interface. Furthermore, other than occasional writes to standard output, the benchmark does no I/O. I/O is one of the key features a full service OS such as AIX provides, so we needed to make sure that the co-scheduling didn't adversely impact I/O. This led to testing ALE3D which is a key LLNL application.

ALE3D is a large three-dimensional (3D) multi-physics code developed at LLNL. It uses an arbitrarily connected hexahedral element mesh, and each timestep involves a Lagrange step with a mesh remap and subsequent advection of materials through the relaxed mesh. This hydrodynamic algorithm is referred to as *Arbitrary Lagrange-Eulerian*, or ALE. ALE3D also supports explicit or implicit time integration, interaction between discontinuous portions of the mesh (slide surfaces), advanced material modeling

capabilities, chemistry, thermal transport, incompressible flow, and an impressive set of 3rd party solver libraries to support various matrix-based algorithms.

Because of the wide variety of physics packages available in ALE3D, the large number of user-settable options, and the variety of meshes that the code can run simulations on, ALE3D makes an interesting testbed for performance analysis. Simply by running different test problems, one can change the performance characteristics of the simulation vastly, which thus makes it is difficult to tune kernel performance improvements to the code, as can sometimes be a pitfall with smaller benchmark applications. For example, by using implicit hydrodynamics with slide surfaces, one must use iterative linear solvers and preconditioners, with thousands of matrix-vector multiplies and tens or hundreds of reductions per timestep. Explicit hydrodynamics, on the other hand, does not involve the production of a matrix, and is primarily nearest neighbor (element) communication.

The test problem used initially for this study was an explicit time integrated hydrodynamics ALE calculation on a simple cylindrical geometry with slide surfaces. A shock generated by detonation of high explosive provided hydrodynamic motion over a timescale of tens of microseconds. The problem ran for approximately 50 timesteps, and each timestep involved a large amount of point-to-point MPI message passing, as well as several global reduction operations. The problem performed a fair amount of I/O by reading an initial state file at the beginning of the run, and dumping a restart file at the calculation's terminus.

5.2. Methodology

The objective in testing the "aggregate" MPI application, containing several loops of MPI Allreduce calls, was aimed at identifying the top contributors to system overhead, and then examining ways of reducing such overhead, in the hope of improving performance.

There were several components utilized in the testing and analysis of the system and application behaviors:

- Use of AIX trace events and trace data collection.
- Establishing a base line for "optimal" performance in a dedicated system environment, without "normal" system & application activity present.
- Performing a series of tests gradually increasing the number of tasks, to collect as many data points as possible.
- Adjusting for various miscellaneous activities (e.g. use of GPFS, co-scheduler/priority adjustment intervals, kernel/scheduler tuning modifications, etc.)
- Large scale system availability.

5.2.1. AIX TRACING

The use of the AIX trace facility was the primary means used for determining outliers of system overhead. By establishing a series of trace events, including a number of event records written at key code points inside of the "aggregate" application, we used tracing to record the activity of both the application and system daemons.

5.2.2. BASE LINE MEASUREMENTS

The establishment of a "best case" *baseline* was important in comparing progress and improvements made with the problem. These consisted of testing under optimal system conditions where we eliminated non-essential system activity through the use of a dedicated system, with no other users or application activity running at the time. Such tests were conducted using 15 of the 16 processors on each node, and utilizing the co-scheduler to give the application increased run priority over everything but the bare minimum of system daemons required to keep the system functioning.

5.2.3. SCALING TESTS

Once a baseline was established, we focused on testing with the system configured for a "normal" load, allowing applications that would be expected to be operational while the system was in production use, utilizing all 16 processors per node. In both cases for the baseline and normal configurations, we would begin testing at small node/task counts, gradually increasing the number of nodes/tasks, to the maximum configuration possible, collecting data at each point. In the early stages of the tests, before improvements were identified, the rate of increase was more gradual. As more positive progress was achieved, we made the rate of increase steeper, to focus on obtaining more data at the higher node counts. In all cases, we strived to choose a consistent set of data points for all of the tests, to maintain a comparison of like data among the tests.

5.2.4. MISCELLANEOUS ADJUSTMENTS

In performing the test runs, there were a number of items that required some interaction in order for the specific test to be performed. AIX, POE, and MPI all have a high degree of options, parameters and tunables that were integral in ensuring the tests ran as necessary. For instance, early on it was discovered that the use of GPFS had played a significant role in the number of I/O related interrupts occurring during the MPI Allreduce "aggregate" runs, therefore it was found necessary to limit its use in most cases.

Other test variables and adjustments needed were:

- Application and removal of a prototype AIX kernel containing changes for additional system tunables for interrupt scheduling.
- Activation of the POE co-scheduler, with specific and varied priorities and durations.
- MPI tunables, such as for using shared memory, CPU utilization, and interrupt handling.
- Saving and restoring the system for production use when the tests were concluded.

5.2.5. LARGE SCALE SYSTEM AVAILABILITY

The availability of a large scale test system was a key factor in making progress. Because the IBM Poughkeepsie and Austin locations lacked such a system, we were heavily reliant on systems at Lawrence Livermore National Laboratory and at AWE in the UK. Since the performance issues were more apparent at the higher node/task counts, it was imperative that the tests be conducted on large scale systems.

Because such systems are in production use and shared by the general user population, the tests were scheduled in regular intervals competing with other scheduled activities, and we were not afforded a high number of such opportunities. For these reason, when scheduling and performing the tests, we would strive to be efficient and economical with the allocated test time, to make the most of the test window.

In addition, since two of the machines are classified systems, we were limited with our ability to collect and analyze trace data collected on them. Therefore it was necessary to have someone with the proper security clearance engaged in our efforts to be our "eyes and ears" in such cases.

LLNL conducted tests on two machines: 'ASCI White,' a classified system that has a total of 512 nodes, all 16-way SMPs based on the 375 Mhz Power3® processor; and 'Frost' which has a total of 68 nodes, likewise 16-way SMPs based on the 375 Mhz Power3 processor. Both machines have 16 GB of memory per node, and run the IBM HPC software stack of AIX, PSSP, and POE. [ASCI-white]

The AWE machine, 'Blue Oak', has a total of 128 nodes, of which 120 are 16-way Nighthawk II compute nodes; thus the maximum number of Power3-II processors available to run the tests is 1920. Each node has at least 16 GBytes of memory (two have 64 GBytes), giving a total of 2 TBytes across the entire system. The software installed on the system consists of the usual HPC stack found on an IBM SP i.e. PSSP, AIX, and POE.

As with ASCI White, data cannot freely be extracted from the 'Blue Oak' system, so detailed results from the AIX trace facility were impractical; limiting the amount of data eased the problem of getting the data to IBM quickly after the tests had been completed. Also, Blue Oak being the only large parallel production system at AWE, the opportunities to take over the whole machine for test shots were limited to occasions when the impact was tolerable, viz. when a shutdown was already scheduled for some other essential purpose such as a software upgrade or power supply engineering work.

5.3. Measurements and Interpretation

As a baseline, extensive runs were made with aggregate_trace and the standard AIX kernel using 15 tasks per node. Runs using 16 tasks per node were also made for comparison, and they are presented first since they highlight many of our findings. These results are shown in Figure 3. Here, the average wall clock time per Allreduce (in µsec) is plotted against the number of processors. Each plotted datum is the average of at least 3 runs, and each run is the result of thousands of Allreduces. Instead of the expected smooth, logarithmic scaling, the performance is linear and exhibits extreme variability.

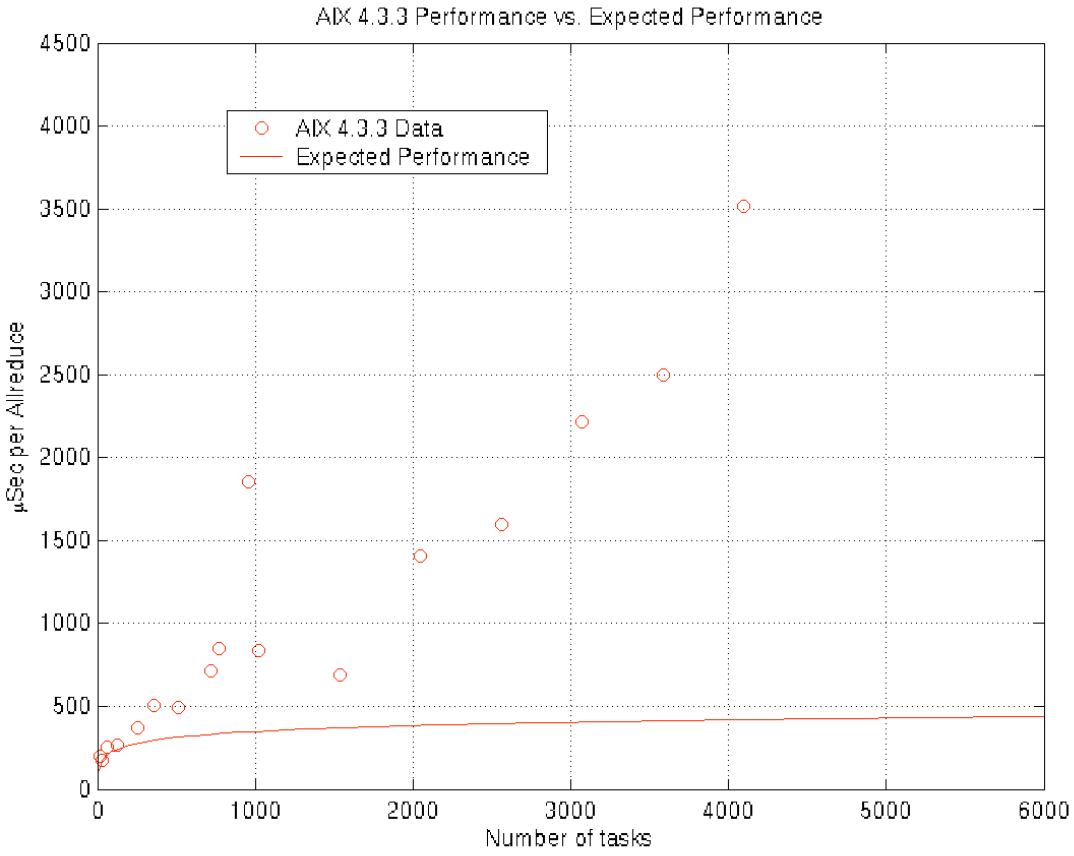

Figure 3: Allreduce µsecs vs. Processor count. Runs using 16 tasks per node and the standard AIX kernel exhibit considerable variability, and the performance scales linearly rather than logarithmically.

To investigate the cause of this poor performance, times for individual Allreduces were extracted from the AIX trace logs (without any kernel modifications or co-scheduler). A plot of 448 sorted Allreduce times, sampled from one node in a 944-processor run, is shown in Figure 4. Models of the benchmark predict an Allreduce should take approximately 350 µsec [Jones02]. The fastest Allreduces come to within about 10% of this, but the median time is another 25% higher, and the slowest 10% represent significant outliers.

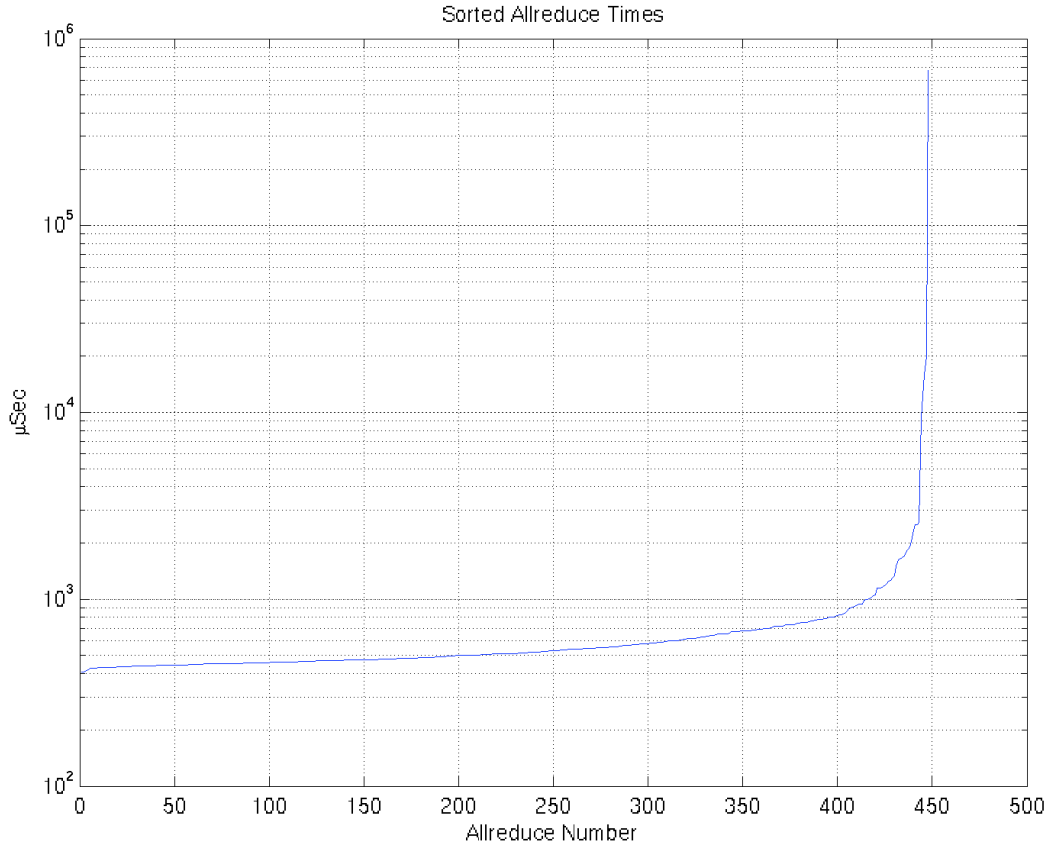

Figure 4: Plot of sorted Allreduce times. The main contributors to the poor average time are the handful of outliers.

The average Allreduce time for this sample is 2240 µsec, about 6 times slower than expected. The outliers are the main contributors to the poor average Allreduce time: the slowest one accounts for more than half the total time. In examining the traces to determine what caused the outliers, we found that an administrative cron job ran during the slowest Allreduce. This cron job is run every 15 minutes to check on the health of the system. Its various components Perl scripts and a variety of utility commands run at a higher priority than user processes and steal CPU resources. We observed that on multiple nodes, one CPU had over 600 msec of wall clock time consumed by these components, blocking a single MPI task per node from making progress.

This worst outlier was an extreme case, but system daemons and interrupt handlers in other outliers similarly blocked user tasks. A variety of AIX daemons, such as syncd, mmfsd, hatsd, hats_nim, inetd, LoadL_startd, mld, and hostmibd, which run at higher priority than user processes, and interrupt handlers such as caddpin and phxentdd, commandeered CPUs to carry out their tasks. In addition, the execution of these processes was often accompanied by page faults, increasing their run time and further impacting the Allreduce performance. Most of the outliers could be attributed to these interferences.

But for the remainder, one other source of interference was observed. In these cases, the trace records showed auxiliary threads of the user processes sharing the CPUs with the main threads. The total run time of the auxiliary threads amounted to a significant fraction of the Allreduce time. For example, in the case of one Allreduce that took 6.7 msec, the auxiliary threads consumed 4.5 msec of run time spread over several nodes.

These auxiliary threads were identified as the MPI timer threads. They are the "progress engine" [MPICH02] in IBM's MPI implementation. The default behavior is that these threads run every 400 msec, and, apparently, even with that relatively long period, their influence was strong enough to disrupt the tightly synchronized Allreduce code.

For non-outliers, only minor interference from daemons was observed. And for the fastest Allreduces, only events marking the beginning and end of the Allreduce in each task and some decrementer interrupts showed up in the traces.

For the 15 tasks per node runs, our results agree with what many users have found in running their applications on these systems: absolute performance is improved and there is much less variability using 15 tasks per node. In spite of the improved performance, the scaling is still linear rather than logarithmic. The traces show that daemons aren't a problem in the 15 tasks per node runs since they make use of the available CPU. The MPI timer threads, however, still cause some interference. Besides them, decrementer interrupts are the only other potential source of interference contained in the traces, although these don't totally explain the observed linear scaling.

With potential sources of interference identified, we next sought ways to eliminate them. Results of 16 tasks per node runs of aggregate_trace using the prototype kernel and co-scheduler are shown in Figures 5 and 6.

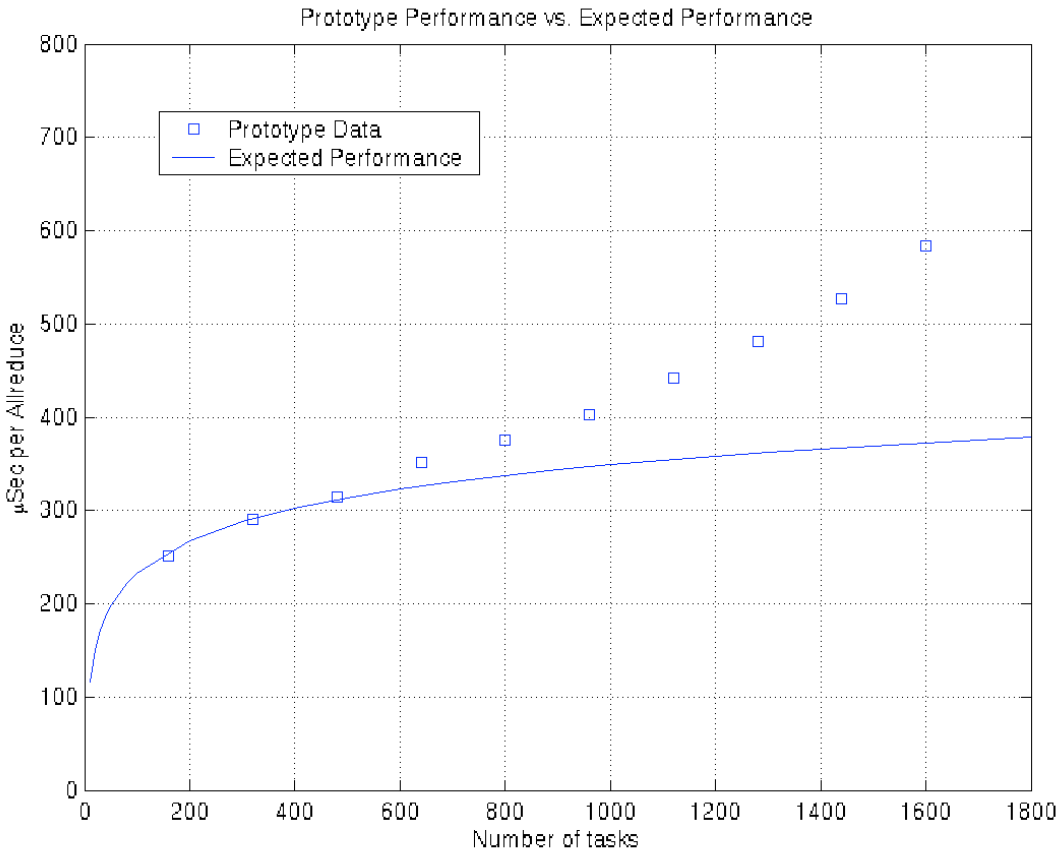

Figure 5: Allreduce µsecs vs. Processor count. Runs using 16 tasks per node and the prototype AIX kernel exhibit improved performance and smaller variability compared to 16 tasks per node runs on the standard AIX kernel. The performance, however, still scales linearly rather than logarithmically.

Addressing the problems caused by the MPI timer threads was relatively easy. We increased their period by a large amount and found this removed the interference. This is done by setting the environment variable MP_POLLING_INTERVAL to, say, 400,000,000 (period = 400 seconds).

Such a large setting is not advisable for all programs. For example, problems could arise in programs using non-blocking calls such as MPI_Isend, the non-blocking MPI_IO calls or the non-standard non-blocking collective communication if their completion is tested with MPI_Test. But programs that use blocking calls, and those using non-blocking calls that wait for non-blocking requests with MPI_Wait will run fine.

Addressing interference caused by system daemons and decrementer interrupts is, of course, what led to the development of the prototype kernel and co-scheduler. After establishing baselines, their effectiveness was tested via a series of benchmark runs on the machines at LLNL and AWE. After some initial runs used to debug the new software and test tunables, we settled on co-scheduler parameter settings of favored priority = 30, unfavored priority = 100, and a window of 5 seconds with 90% of the window at the favored priority; the kernel was set to use a big tick interval of 250 msec. Then we began serious test runs.

The Allreduce times have been cut by a factor of three and the performance no longer shows the extreme variability observed with the standard AIX kernel. In fact, 100 fully populated nodes running the prototype kernel yielded a 154% speedup over 100 nodes running at 15 tasks per node on the standard AIX kernel. This indicates that the delays caused by daemons have been dealt with effectively.

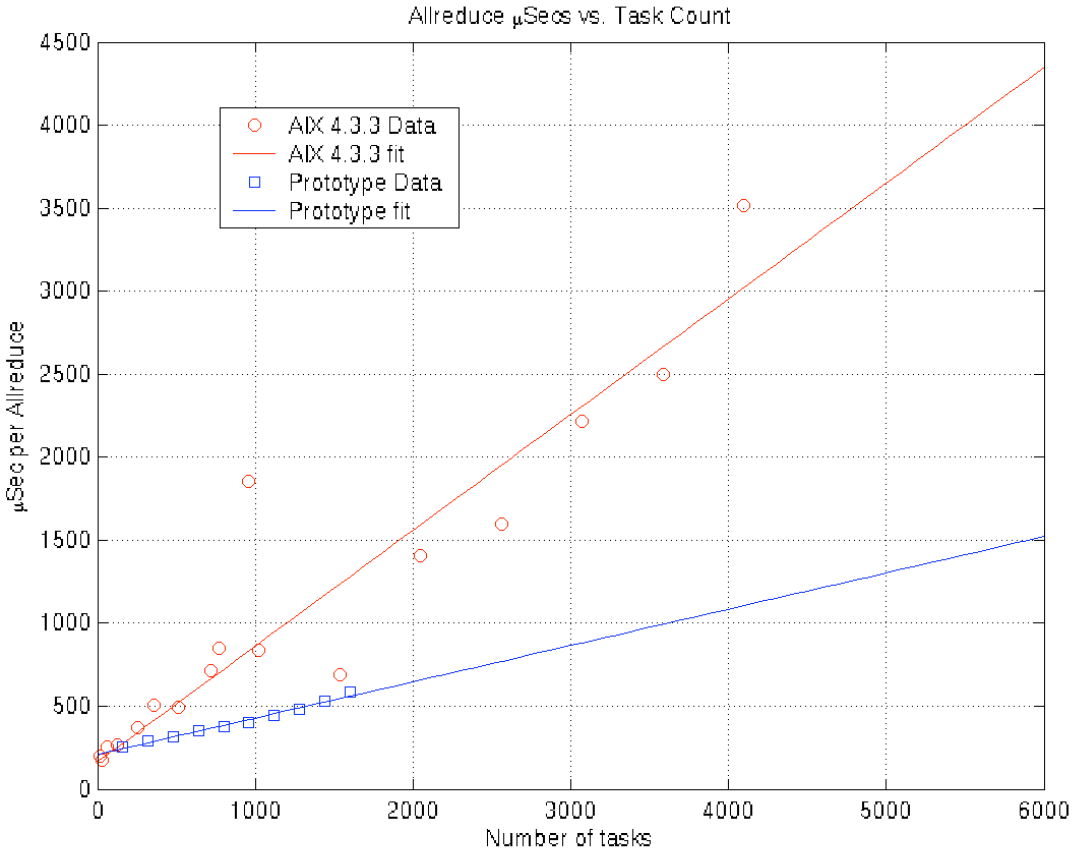

Figure 6: Allreduce µsecs vs. Processor count. The above graph shows the fruit of our labor! The upper points are from the 'vanilla' AIX 4.3.3 kernel, the lower points from the prototype kernel. Lines are fitted to the data: y_vanilla(x) = 0.70(x) + 166 y_prototype(x) = 0.22 (x) + 210. The slope indicates ~3x improvement from these testshots.

We then ran calculations using ALE3D to see if similar performance improvements would be seen with real applications. The first tests of ALE3D were very disappointing: the co-scheduler actually slowed it down. Profiling revealed that slower I/O was the cause of the reduced performance. Observations of system activity while the application was running revealed that limiting I/O daemons to just 10% of a 5 second window starved them.

To fix this problem we adjusted the favored priority to just above that of key I/O daemons. This means that I/O daemons are allowed to pre-empt the user tasks, thus providing a source of interference, but for real applications, this seems to be the best tradeoff. Note that the favored priority is still high enough to eliminate interference from other daemons.

With these relative priorities, the results improved. The ALE3D run time dropped 24% from 1315 seconds to 1152 seconds on 944 (59x16) processors using the prototype kernel and co-scheduler rather than the standard AIX kernel. We did traces of subsets of both runs. In looking at the traces for the ALE3D run on the standard kernel, we found some long running daemons that took up CPU resources during allreduce calls. This forced two MPI tasks to share a single CPU, delaying the allreduce. These daemons ran with a priority of 56, which is more favored than those for normal user processes, which range between 90 and 120. These daemons, and the consequent performance degradation they caused, were not observed in the traces of the run using the prototype kernel and co-scheduler since the priority of the user tasks had been boosted to 40. The elimination of this kind of interference, was, of course, what the co-scheduler was designed to do, and in our ALE3D runs, it was successful.

6. Related Work

The scheduling of processes onto processors of a parallel machine has been and continues to be an important and challenging area of research. Feitelson suggests that the domain of parallel machine scheduling may be divided into two dimensions of sharing: *space-slicing* which refers to how jobs are mapped onto partitions of the machine's processors, and *time-slicing* which refers to context switching within a given processor [Feitelson97]. Two other important factors to consider when mapping the domain of scheduling research are: *dedicated usage* which refers to whether or not more than one parallel application is mapped on to a collection of processors at any given time, and *cluster-machine* which refers to whether the size of the computer is allowed to span multiple SMPs.

When the above four scheduling dimensions (space-slicing, time-slicing, dedicated usage, and cluster-machine) are considered in a supercomputing context, scheduling research may be loosely grouped into four general areas: Spatial-schedulers, Gang-schedulers, Fair Share Co-schedulers, and Dedicated Job Co-Schedulers:

1. Gang-schedulers: Efforts directed at multi-programming two or more parallel applications. Typical scheduling time-quanta of minutes (the NQS GangScheduler default for the Intel Paragon is 10 minutes). Implies non-dedicated usage. Appropriate for cluster-machines. Examples include the Lawrence Livermore Gang Scheduler [Moreira99], Concurrent gang [Barbosa da Silva99]
2. Spatial-schedulers: Efforts directed at parallel job placement (e.g., NQS, IBM's LoadLeveler). Typical time-quanta of an entire job. Appropriate for dedicated and non-dedicated usage. Appropriate for cluster-machines. Examples include NQS [Kramer89], PBS [Henderson95], Condor [Litzkow88], and the Maui Scheduler [MHPCC].
3. Fair Share Co-Schedulers Efforts directed at improving global throughput of a multiprogram environment spanning multiple SMPs through utilizing more than local knowledge. (For example, attempts

		to run a webserver and a distributed database concurrently on a Network of Workstations.) Typical time-quanta of OS timer-decrement and/or communication interrupts. Implies non-dedicated usage. Appropriate for cluster-machines. Examples include [Ousterhout82] [Dusseau96] [Sobalvarro97] [Nager99].
4.	Dedicated Job Co-Schedulers	Efforts such as our own directed at optimizing a working-set of cooperating processes in an environment dedicated to a single parallel job spanning multiple SMPs. Typical time-quanta of OS timer-decrement and/or communication interrupts. For dedicated usage by definition. Appropriate for cluster-machines. Examples include [Bryant91] [Hoisie03].

Our work, which focuses on large MPP systems commissioned to solve grand-challenge scientific problems, falls naturally in the above category 4. Due to their time quanta, the Gang-schedulers of category 1 are not able to address context switch interference which is a key goal of our work. The Spatial-schedulers of category 2 are complementary to our work and our techniques may be applied between invocations of any of the aforementioned Spatial schedulers. Our work shares some of the same goals of category 3, but we always wish to optimize the turn-around time of the parallel application whereas Fair Share Co-schedulers typically seek to optimize the overall efficiency of the machine. This is a subtle but important difference: we are willing to have large inefficiencies in distributed daemons and cron jobs (e.g. membership service subsystems or parallel file systems) if the time-to-completion for the dedicated parallel application improves.

Bryant, Chang and Rosenburg present a category 4 scheme they call Family scheduling [Bryant91]. This threads-based approach differs from our work in the assumptions and restrictions placed on the underlying computer architecture–our approach is a general approach suitable for any cluster of SMPs. A research team headed by Adolfy Hoisie at Los Alamos has been investigating similar issues [Hoisie03]. Their work has focused on techniques applied to Network Interface Cards (NICs) and curtailing or removing system activities such as daemons. Their work differs from our work in that we attempt to mitigate the interference caused by system software that cannot be further removed or altered (e.g. parallel file systems).

To our knowledge, we are the first to develop and implement a general approach for dedicated job co-schedulers, an area we believe to be of considerable interest to the supercomputing community.

7. Conclusion and Future Work

Modifying the operating system to be specifically aware of, and provide coordinated scheduling of, fine-grain parallel processes leads to faster and more repeatable run times. This is further enhanced by providing a co-scheduler that synchronizes times across a cluster and provides preferred dispatch slots for large fine-grain parallel applications, while still allowing critical system daemons to perform their functions.

We developed a run-time system and accompanying kernel modifications that increase the overlap of interfering system activities. Both long-lived system activities such as daemon schedulings and short-lived events such as timer-decrement interrupt processing are overlapped when using the prototype kernel and co-scheduler. The forced overlap of system interference is accomplished both intra-node and inter-node.

Our results indicate a speedup of over 300% on synchronizing collectives. These fine-grain parallel operations have been shown to consume more than 50% of total time for typical bulk-synchronous SPMD applications at 1728 processors, and a larger percentage at higher processor counts.

Future work is needed to examine the benefit of this research on a wide range of parallel applications. The results presented in this paper have focused on a detailed understanding of the issues involved in a limited set of applications and evaluating additional applications at scale would provide added insight--at present we are collecting additional scaling and baseline data. Providing a mechanism for parallel applications to establish when they are entering and exiting fine-grain regions may be beneficial on systems supporting the described scheduling capabilities. Another interesting prospect is to combine the techniques described in this paper with complementary techniques designed to improve fine-grain parallel processing (e.g., hardware assisted collectives). Before these prototypes can become system infrastructure for production supercomputer machines, they will also need several improvements such as more sophisticated interfaces to system administrators. Our hope is that future system infrastructures intended for large processor-count machines will realize the potential of parallel aware scheduling techniques.

8. Acknowledgements

This work benefits from the helpful contributions and suggestions of Pythagoras Watson, David Fox, and Linda Stanberry from Lawrence Livermore National Laboratory, and Mike Cavanaugh and Dan McNabb at IBM. Assistance received from Paul Szepietowski is also appreciated. The authors would also like to thank Fabrizio Petrini of Los Alamos National Laboratory for helpful discussions.

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

9. References

[ASCI_White] ASCI White Information. http://www.llnl.gov/asci/platforms/white

[Barbosa da Silva99] Barbosa da Silva FA, Scherson ID. Concurrent gang: Towards a flexible and scalable gang scheduler. Proceedings *11th Symposium on Computer Architecture and High Performance Computing. Univ. Federal do Rio Grande do Sul. 1999, pp.243-7. Porto Alegre, Brazil.*

[Bryant91] R. M. Bryant, H-Y. Chang, and B. S. Rosenburg, "Operating system support for parallel programming on RP3". IBM J. Res. Dev. 35 (5/6), pp. 617-634, Sep/Nov 1991.

[Burger96] D.C. Burger, R.S. Hyder, B.P. Miller, and D.A. Wood, "Paging Tradeoffs in Distributed Shared-Memory Multiprocessors", Journal of Supercomputing 10, 1 (1996). Also appears in Supercomputing '94, Washington, DC, November 1994.

[Dawson03] Shawn Dawson, Mike Collette, Performance of Ares. LLNL Technical Report UCRL-, January 21, 2003.

[Dusseau96] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Efffective distributed scheduling on parallel workloads. In *ACM SIGMETRICS '96 Conference on the Measurement and Modeling of Computer Systems*, 1996.

[Feitelson89] Dror G. Feitelson and Larry Rudolph. *Gang Scheduling Performance Benefits for Fine-Grain Synchronization*. Journal of Parallel and Distributed Computing, 16(4), 1992.

[Feitelson97] Feitelson, D.: Job Scheduling in Multiprogrammed Parallel Systems IBM Research Report RC 19970, Second Revision (1997).

[Gupta91] A. Gupta, A. Tucker, and S. Urushibara, "*The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications*," In Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 120-132, May 1991.

[Henderson95] Henderson RL. Job scheduling under the Portable Batch System. Job Scheduling Strategies for Parallel Processing. *IPPS'95 Workshop. Proceedings. Springer-Verlag. 1995, pp.279-94. Berlin, Germany.*

[Hoisie03] Adolfy Hoisie, Darren Kerbyson, Scott Pakin, Fabrizio Petrini, Harvey Wasserman, Juan Fernandez-Peinador, Identifying and Eliminating the performance Variability on the ASCI Q Machine, LANL Technical Report UCRL-yyyyy. January 2, 2003.

[IBM 01] IBM Corp – IBM Parallel Environment for AIX: Installation, GA22-7418

[IBM 03] IBM Cluster solutions. http://www-1.ibm.com/servers/eserver/clusters

[Jones02] Terry Jones, "A Scaling Investigation on IBM SPs", ScicomP 6, Aug. 2002, Berkeley, CA.

[Jones03] Terry Jones, Jeff Fier, Larry Brenner, Observed Impacts of Operating Systems on the Scalability of Applications. LLNL Technical Report UCRL-, March 5, 2003.

[Karl97] Holger Karl – Co-scheduling through synchronized Scheduling servers – A prototype and experiments (unpublished).

[Kramer89] W.T.C. Kramer and J. M. Craw, "Effective use of Cray supercomputers". In Supercomputing '89, pp. 721-731, Nov 1989.

[Litzkow88] M. Litzkow, M. Livny, and M. Mutka. *Condor - a hunter of idle workstations*. In Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS'88), pages 108--111. IEEE Computer Society Press, June 1988.

[MHPCC] MHPCC, The Maui Scheduler, http://www.hpc2n.umu.se/doc/maui/index.html

[Moreira99] Moreira JE, Franke H, Waiman Chan, Fong LL, Jette MA, Yoo A. A gang-scheduling system for ASCT Blue-Pacific. *High-Performance Computing and Networking. 7th International Conference, HPCN Europe 1999.* Proceedings. Springer-Verlag. 1999, pp.831-40. Berlin, Germany.

[MPICH02] Abstract Device Interface Version 3.3 Reference Manual Draft of October 17, 2002.

[Mraz94] R. Mraz, Reducing the Variance of Point to Point Transfers in the IBM 9076 Parallel Computer, July, 1994. IBM Research Report RC-19675

[Nager99] S. Nager S., A. Banerjee, A. Sivasubramaniam, C.R. Das. A closer look at coscheduling approaches for a network of workstations. *Proceedings of ACM Symposium on Parallel Algorithms & Architectures, 96-105, 1999.*

[Ousterhout82] John K. Ousterhout – Scheduling Techniques for Concurrent Systems. In Third International Conference on Distributed Computing Systems, pp. 22-30, May 1982

[Sobalvarro97] Patrick G. Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien – Dynamic Co-scheduling on Workstation Clusters. Digital Systems Research Center Technical Note 1997-017, March, 1997

[Thompson74] Thompson, K. and D.M. Richie. July 1974. ``The UNIX Time-Sharing System''. *Communications of the ACM* Vol. 17, No. 7. pp. 365-375.

[Top500] Top 500 Supercomputer sites. http://www.top500.org/lists/2002/11/