SANDIA REPORT

SAND2012-1750
Unlimited Release
Printed March 2012

Report of Experiments and Evidence for ASC L2 Milestone 4467 - Demonstration of a Legacy Application's Path to Exascale

Brian Barrett, Richard Barrett, James Brandt, Ron Brightwell, Matthew Curry, Nathan Fabian, Kurt Ferreira, Ann Gentile, Scott Hemmert, Suzanne Kelly, Ruth Klundt, James Laros III, Vitus Leung, Michael Levenhagen, Gerald Lofstead, Ken Moreland, Ron Oldfield, Kevin Pedretti, Arun Rodrigues, David Thompson, Tom Tucker, Lee Ward, John Van Dyke, Courtenay Vaughan, and Kyle Wheeler

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2012-1750
Unlimited Release
Printed March 2012

Report of Experiments and Evidence for ASC L2 Milestone 4467 - Demonstration of a Legacy Application's Path to Exascale

Brian Barrett*, Richard Barrett*, James Brandt,° Ron Brightwell*,
Matthew Curry*, Nathan Fabian*, Kurt Ferreira*, Ann Gentile°,
Scott Hemmert*, Suzanne Kelly*, Ruth Klundt*, James Laros III*,
Vitus Leung*, Michael Levenhagen*, Gerald Lofstead*, Kenneth Moreland*,
Ron Oldfield*, Kevin Pedretti*, Arun Rodrigues*, David Thompson,°
Tom Tucker,◇ Lee Ward*, John Van Dyke*, Courtenay Vaughan*, and Kyle Wheeler*

*°Sandia National Laboratories
P.O. Box *5800/°9169
*Albuquerque, NM 87185-1319/°Livermore, CA 94550
◇ Open Grid Computing, Inc.
Austin, TX 78759
{smkelly,others}@sandia.gov

Abstract

This report documents thirteen of Sandia's contributions to the Computational Systems and Software Environment (CSSE) within the Advanced Simulation and Computing (ASC) program between fiscal years 2009 and 2012. It describes their impact on ASC applications. Most contributions are implemented in lower software levels allowing for application improvement without source code changes. Improvements are identified in such areas as reduced run time, characterizing power usage, and Input/Output (I/O). Other experiments are more forward looking, demonstrating potential bottlenecks using mini-application versions of the legacy codes and simulating their network activity on Exascale-class hardware.

Acknowledgments

The authors would like to thank the Red Storm, Cielo, and Cielo Del Sur operations teams for their support during the dedicated experiments. These systems are valuable resources and the teams' prompt response ensured that we maximized our time on them.

Part of this work used resources of the Oak Ridge Leadership Computing Facility (OLCF), located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under DE-AC05-00OR22725. An award of the computer time at the OLCF was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program.

The authors would also like to thank the following teams who provided access to and insight into the applications used in the execution of this work: CTH, Charon, SIERRA/Aria, SIERRA/Fuego, and Zoltan.

Contents

List of Figures

List of Tables

Nomenclature

ASC Advanced Simulation and Computing

BSP Bulk Synchronous Processing

Core A computational unit that reads and executes program instructions.

CSSE Computational Systems and Software Engineering

FC Fiberchannel

HPC High Performance Computing

HSN High Speed Network

jMTTI Job Mean Time To Interrupt - any hardware or system software error or failure that results in a job failure.

MPP Massively Parallel Processing

NFS Network File system

NIC Network Interface Chip

Node A network endpoint containing one or more cores that share a common memory. For HPC, batch schedulers often reserve nodes as a minimum unit of allocation.

PTAS Polynomial-time approximation scheme

RDMA Remote Direct Memory Access

sMTTI System Mean Time To Interrupt - any hardware or system software error or failure, or cumulative errors or failures over time, resulting in more than 1% of the system being unavailable at any given time.

Chapter 1

Introduction and Executive Summary

As part of its mission, the CSSE sub-program within ASC is responsible for deploying key software components. These include system software and tools, input/output, storage systems, networking, and post-processing tools. Some work is on-going in nature, some activities address short term requirements, and others address the need to invest in technology development for anticipated future mission requirements. This milestone report covers the work done within Sandia's CSSE program to support the ASC applications as computer technology evolves.

We begin with the milestone description:

> Cielo is expected to be the last capability system on which existing ASC codes can run without significant modifications. This assertion will be tested to determine where the breaking point is for an existing highly scalable application. The goal is to stretch the performance boundaries of the application by applying recent CSSE RD in areas such as resilience, power, I/O, visualization services, SMARTMAP, lightweight LWKs, virtualization, simulation, and feedback loops. Dedicated system time reservations and/or CCC allocations will be used to quantify the impact of system-level changes to extend the life and performance of the ASC code base. Finally, a simulation of anticipated exascale-class hardware will be performed using SST to supplement the calculations.

The milestone team collected Sandia's recent CSSE project activities that strive to improve application performance without (or with minimal) source code changes to the application. Work is from FY09, when the milestone was originally developed, through the second quarter of FY12. Thirteen activities are highlighted herein. They are:

- Catamount enhancements to fully utilize cores per node
- Utility to reduce debugging time on Red Storm
- Optimizations to reduce core-to-core communication overhead
- Improved node allocation algorithms
- Enhancements to increase power efficiency during application execution on Red Storm

- Application-level data services to reduce effective I/O costs
- In situ and in transit visualization and analysis
- Dynamic shared libraries on Cielo
- Process replication for reliability
- Enabling dynamic resource-aware computing
- Scalable virtualization environment
- File system for high bandwidth checkpoints
- Exascale simulation to enable collective communication offload

For each area, we identified and measured the performance impact using whichever unit of measure was most applicable. Performance was not constrained to the most narrow definition of "runs faster". In addition to improved/decreased runtime, some technologies offers better throughput of jobs, some improve user productivity, some offer efficiency improvments, others increase job resiliency, etc. Some of the work targeted future technologies and provided design data with an estimate for the expected improvement.

The milestone text references "**an** existing highly scalable application". We gathered data primarily using the CTH [44] and/or Charon [83] ASC applications. In a few cases, neither application exhibited the feature or issue being addressed by the milestone area. In that case, one or more different applications were used. Over time, we decided it was not necessary to down-select to *an* existing application. Each application provided interesting data that added value to the milestone results. All results are provided.

To address the "breaking point" portion of the milestone, we analyzed both CTH and Charon to determine where the scalability fell off, even with the available CSSE enhancements. We also estimated, using CTH's mini-app, the cross over point when the current CTH mpi-everywhere programming model was surpassed by on-node OpenMP and inter-node MPI. This analysis can be found in chapter 15.

Chapters 2 through 14 each address one of the thirteen development areas listed above. The chapters are sequenced primarily by platform, with the oldest targeted platform first. This translates to Red Storm, Cielo, and future platforms. Although the system is now retired, the Red Storm work offered interesting techniques and lessons learned as we continue on the path to exascale. A number of these early chapters compare the results with the equivalent Cielo tool giving us insight into the value of lightweight kernels. Each chapter begins with an abstract that presents the problem, describes the solution approach, and gives a very brief summary of the results of the work. The reader is encouraged to review just the chapter abstracts if only the highlights are desired. However, it is best to read the full content of each chapter in order to understand the environment under which the data was collected.

The remainder of this chapter gives an executive summary. Instead of the historical order used for sequencing the chapters, we summarize the work by the areas of improvement. The same caution about the risk of taking the results out of context from the test setup applies here as well.

Reduced runtime We document six different R&D contributions that can reduce the runtime for applications.

The SMARTMAP optimization was installed into the Red Storm's lightweight kernel operating system, Catamount. The production MPI library was then modified to take advantage of this very efficient on-node shared memory capability to reduce completion times for collective communication operations. Results indicate that SMARTMAP provides a 4.5% improvement in performance when running Charon on 2,048 quad-core compute nodes (8,192 cores) of Red Storm. See chapter 4 for more information.

Smart algorithms for allocating available nodes to jobs can have a significant effect on their runtime. For networks configured as a mesh or torus, a cube-like allocation of nodes reduces distance between communicating pairs and minimizes the impact of traffic from other jobs on the system. The initial Sandia-developed algorithm has been implemented in the Cray scheduler by Cray. The experiment documented herein shows that the magnitude of the improvement increased by tenfold when the average job running time of the sample increased by less than twenty percent. See chapter 5 for more information.

The Trilinos I/O Support (Trios) capability has been released as part of the Trilinos project. It includes the infrastucture for application-level data services. When combined with an I/O service, such as PnetCDF and netCDF, it can provide an effective I/O rate that is 10X higher for a single shared file. The application is free to continue its computation while the staged data is written to disk. See chapter 7 for more information.

The ability to adapt computation based on dynamically available resource utilization information has been identified as an enabler for applications running at exascale. We demonstrate such a mechanism on Cielo, proving the feasibility of dynamic resource-aware computing. The capability is very lightweight consuming less than a hundredth of a percent of the computing resource. It has a scalable implementation and was demonstrated on 10,000 cores. It provides both static and dynamic feedback mechanisms for the application or a library. See chapter 11 for more information.

We introduce a new file system concept and project, currently called GoofyFS. It is targeted for exascale-class systems. Quality of service is maximized by supporting multiple data storage devices and by allowing for local decision making. The first large-scale experiment on Cielo used off-node memory to write checkpoint files, which resulted in a 10-60x speedup in effective I/O rates. See chapter 13 for more information.

Reduced runtime was a goal of an exascale hardware design as well. The design offloads collective communication processing to a potential new NIC architecture. Under simulation, when coupled with the new version 4 of the Portals networking protocol, it is possible to achieve lower latency and higher tolerance to system noise. The simulations, run up to

32,768 nodes, show a 40% reduction in latency over host-based collectives processing. See chapter 14 for more information.

Improved Job Throughput: Three chapters report on contributions that can increase the job throughput on a system. The jobs do not necessarily run faster, but system utilization is improved.

The job management software on Red Storm was modified to allow the user to specify resources based on units more intuitive to their problem setup. Instead of nodes, the user specifies the number of needed processing elements (aka MPI ranks) and optionally, the amount of memory needed per procesing element. This more natural specification was motivated initially by the fact that nodes on Red Storm are heterogeneous. Nodes could have either two or four processing elements and the amount of memory per processing element varied as well. Without the change, an allocation based on nodes had to assume only two processing elements and the smallest amount of memory per node. Allocation based on the most conservative node architecture wastes processors and memory. An experiment was run that showed the modification to the allocation specification improved overall job throughput by 10% as well as enhanced the user interface. See chapter 2 for additional information.

Traditionally, the computation and visualization portions of a problem analysis have been performed separately. Data files are written to disk during the computation phase. The files are then read during a separate visualization phase. Computational capabilities continue to outpace I/O and disk technological advances. To address this ever widening gap, in situ and in transit visualization techniques are being developed to reduce the amount of required I/O to disk. Modifying the standard HPC workflow can shrink the time from initial meshing to final results. Chapter 8 describes the progress to date, which will be further documented in an FY13 L2 milestone.

Although counter-intuitive, process replication can provide job throughput improvements depending on job size and job mean time to interrupt. We describe a prototype implementation to understand runtime overhead of replication. Modeling, empirical analysis and simulation are used to find the cross over curves where replication can be more efficient than traditional file-based checkpoint/restart solutions for resiliency. This data is extremely useful for exascale regimes. See chapter 10 for more information.

Increased User Productivity: Four of the R&D contributions improve user productivity. Depending on the specific area, these contributions may cause an increase in job runtime. For these situations, the increase in runtime is quantified, as user productivity can be a subjective measure.

A debugging utility called fast_where was provided on Red Storm to help understand what might be going on when a job appears hung. The TotalView debugger can perform the same function. However on Red Storm it was taking 30 minutes for TotalView to attach to 1024 processes. The fast_where utility was able to collect the required information in a few minutes on 31,600 cores. See chapter 3 for more information.

The in situ and in transit visualization capabilities described above are also anticipated

to improve user productivity because of the compressed workflow.

Statically linked application binaries have traditionally been mandated for the jobs running at the largest scales. However, dynamically linked binaries introduce productivity enhancements for application developers. It is not necessary to relink the application every time a library is changed. Also, some applications are so large they are forced to statically relink for each combination of modules needed for the problem being analyzed. At a minimum, this is a difficult bookkeeping task. The Cielo team implemented a hierarchical cache for active shared libraries and ran an experiment to assess its efficiency. Based on benchmarks, the setup appeared optimal. However, the dynamically linked version of Charon ran 34% longer than the statically linked version. The traditional wisdom of statically linked binaries prevails, unless circumstances dictate the use of a dynamically linked binary. See chapter 9 for more information.

Application developers are often concerned about the portability of their application. Virtualization can provide multiple operating systems on a single hardware platform. This allows the application to operate in an environment most suitable to its needs. The Kitten operating system, working in concert with the Palacios virtual machine monitor, can provide a highly scalable lightweight kernel environment and also support applications requiring a richer set of functionalty. Experiments shows a 5% increase in runtime when running in a virtual machine, rather than directly on the hardware itself. See chapter 12 for more information.

Reduced Power Consumption: In 2008, the Red Storm lightweight kernel operating system, Catamount, was modified to automatically transition to lower power states when idle. As part of this work, we developed scalable measuring techniques to quantify the power savings of the change. This somewhat ancillary activity created a capability that can identify additional opportunities for reduced power consumption. In this report we discuss research that identifies the impact of reduced power states on a per job basis. In a series of experiments we characterize the effect of CPU frequency and network bandwidth tuning on power usage and demonstrate energy savings of up to 39% with little to no impact on runtime performance.

Data for Co-Design: Most of the activities discussed so far inform our exascale co-design efforts. Of particular note are 1) power efficiency, 2) new visualization workflow techniques, 3) process replication for reliability, and 4) virtualization.

Chapter 15 is specifically targeted at scalability. We studied the CTH and Charon applications to identify bottlenecks that we expect to be issues as machines and applications approach exascale. There was no single breaking point found. In fact, several different ones were found for CTH and Charon. System software scalability issues remain, even with the enhancements documented herein. An analysis was performed estimating the breaking point for the MPI-everywhere programming model within CTH. Algorithmic improvements were identified to improve Charon scalability.

Given this introduction and summary of the work, we hope you are inspired to read more

of the document in the areas that interest you.

Chapter 2

Red Storm Catamount Enhancements to Fully Utilize Additional Cores per Node

Abstract: Enhancements to the job submission interface are described here which enable more efficient core utilization by accepting requests for compute resources using parameters that match the requirements of the application. This spares the user the task of remapping those requirements to the peculiarities of the hardware configuration, which may be complicated by heterogeneity and may be evolving. Experiments are described that demonstrated a 10% improvement in core utilization using sampled actual job mixes from Red Storm.

2.1 Motivation for the Change

The Red Storm computer and Catamount, its lightweight compute node operating system, evolved over the years through multiple hardware upgrades. Throughout most of Red Storm's lifetime, as a result of partial system hardware upgrades, its hardware has been heterogeneous with a varying number of compute cores per node and a varying amount of memory per node. The software enhancements described and evaluated here were developed and installed to allow seamless full utilization of the processors in such a heterogeneous environment.

Before these changes, the user requested some number of nodes from Moab, the batch job scheduler, but had to specify the number of cores (MPI ranks) per node to use to Yod, the Catamount program that loads and oversees a particular parallel job. There was a quad-core batch queue available, restricting jobs to only run on quad-core nodes. The standard queue included all the nodes. Thus unless one submitted their job to the quad-core queue, they could only assume there were 2 cores per node available. This conservative assumption could mean that not all cores were fully utilized on each allocated node. The changes described here allow the user to request the more natural number, the number of ranks needed and optionally the amount of memory per rank that the job requires. It is then up to the system to load the job on dual-core nodes, quad-core nodes or a mixture to fully utilize the processors. It's easy and in the language of the application to specify to Moab the number of ranks needed and optionally the memory per rank needed. It's more cumbersome to specify

to Yod how many cores per node are consistent with the memory requirements per rank and with the queue choice and then calculate how many nodes need to be requested from Moab. The enhancement described here required change to both Moab and to Yod.

2.2 Characterizing the effect of the Enhancements

To measure the performance impact of these changes, three run streams were created. The core utilizations were compared by running each of these streams twice on Red Storm, once with the changes ("new way") and once emulating the environment without ("old way"). To do this, three two-week windows of actual Red Storm usage were selected from before these changes were introduced. For each of those windows, the number of nodes used, the elapsed time of job execution (truncated to minutes) and a flag to indicate whether the job ran in the quad-core batch queue was collected from the Accounting Database. Very small jobs, less than 9 nodes or less than 20 minutes, were excluded.

2.2.1 Details of the test

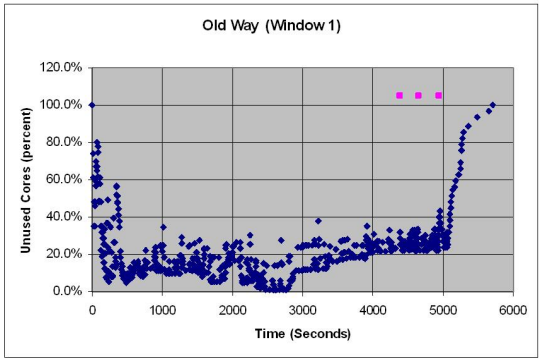
The creation, submitting and running in the two modes differed in three ways. (1) Separate Bourne shell scripts were used for the each mode to read the job list file and submit the jobs to Moab. Each submission requested a Yod enforced time limit of the scaled run time. Since Moab now needs requested cores instead of requested node, for the new way, the number of requested cores was two or four times the number of nodes depending on the quad-queue flag. For the emulation mode, the number of requested cores from Moab was always artificially set to twice the number of nodes. If the quad-queue flag was set, that information was passed on to Moab and used in creating the Yod command line. (2) The currently installed Moab program was used for all runs, but for emulation mode, the node description in the Moab configuration file was changed by specifying that all nodes had two cores per node. Fortunately the current Moab has a separate specification tag of node type. Thus specify "quad" could be used to force a job to quad-core nodes. (3) For emulation mode only, a modified Yod was then used that tweaked the loading information so that non-quad-queue jobs were forced to use 2 cores per node no matter which node they ran on and to enable quad-queue jobs to utilize 4 cores. The shock physics code, CTH, was used for all of the jobs. The submission scripts configured the run to use the appropriate number of ranks.

2.2.2 Results

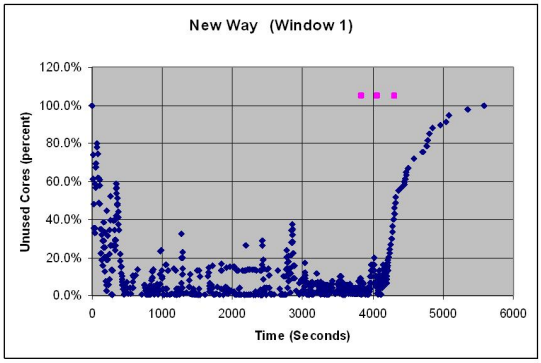
The results for core utilization are shown in Figure 2.1, which plot the percent of unused cores throughout the runs of the job streams derived from window one. Visually it appears that the average number of unused cores is less the new way, i.e. the core utilization is better. To reduce the end effects as the queue emptied, the comparisons were done using the

average core utilization computed over the period up until the work done in the job stream was 90 percent completed. In the figures the three squares in the upper right mark the 85, 90 and 95 percent core-hours point. The results would not be particularly different if 85 percent or 95 percent had been chosen.

Figure 2.1: Node Utilization - Unused Core Percentage



Unused Core Percentage Old Way



Unused Core Percentage New Way

Table 2.1: Summary of Node Utilization Test Details

Window	Start Date of 2 week window	Number of Jobs	Time Scale Factor	Average core Utilization Old Way	Average core Utilization New Way	Improvement
#1	Jan. 18, 2009	322	180	83.4%	92.4%	10.8%
#2	Feb. 22, 2009	264	240	80.4%	88.2%	9.7%
#3	May 3, 2009	245	240	82.0%	91.3%	11.4%

For the first run stream the time was scaled by 180. That is, seconds are used for minutes and the time is divided by an additional factor of 3. The elapsed time for the first run indicated that using a scale factor of 180 would make the full experiment run too long and so a time scale factor of 240 was employed with the 2nd and 3rd run streams. The time to complete each experiment was highly dependent on the order in which jobs ran, i.e. which jobs were left to finish in the tail.

If there were no quad-core queue or no jobs requested the quad queue, there would be theoretically improvement available of about 33 percent[1]. The improvement of only 10 percent is consistent with the fact that a significant fraction of the jobs in the run streams had been submitted to quad queue.

2.3 Conclusion

The enhancement described here allows the user to request compute resources directly using parameters that match the requirements of the application without having to remap those requirements to the peculiarities of the hardware configuration. An added bonus is the potential for improvement in core utilization which was measured here at about 10 percent on actual job mixes.

[1]In large mode, Red Storm had 3,360 dual-core nodes and 6,240 quad-core nodes for a total of 31,680 cores. If only two cores per node are used that leaves 12,480, about 33% of the cores, idle.

Chapter 3

Fast_where: A Utility to Reduce Debugging Time on Red Storm

Abstract: A simple debugging utility for Red Storm called fast_where was developed and deployed. It addressed the need to identify where in a running application each MPI process is executing. While Red Storm's Totalview debugger can provide this information, it only effectively scaled to about 1024 processes. Red Storm is a Massively Parallel Processing (MPP) system with 9600 nodes and 31680 processing cores for computation. The 1024 process effective limit for Totalview is far below the size of the running jobs on Red Storm. Above that process count, Totalview startup time exceeded 30 minutes, which is not tolerable or practical for the standard, non-desperate user. Fast_where was able to collect and summarize this information in a manner of minutes on up to 31,600 MPI processes. A small study was done to compare timing results of fast_where with the more recently available Stack Trace Analysis Tool (STAT) [8] on Cray systems. Fast_where's text-based output was naively basic in comparison to STAT. However, performance was comparable, while robustness and ease of use, were better with fast_where.

3.1 Motivation for the Implementation

A common question posed in emails to HPC system help lists, including Red Storm's, is, "I think my job is hung. How can I tell? In what function(s) are the processes executing?" While full featured parallel debuggers can answer this question, they also provide additional, powerful debugging capabilities often at the expense of good scalability for any query. Responses at high scale can take several tens of minutes.

Prompted by the email appeals for help, Red Storm staff brainstormed ways to use existing health checking and ptrace-like capabilities provided by the lightweight kernal operating system, Catamount, running on Red Storm's compute nodes. The result was a 400-line shell script that could provide a summary of which ranks were executing in which functions. The vision was that this script could pinpoint three likely scenarios: 1) one node does not respond and is likely hung, 2) all but one process are waiting in an MPI barrier, with one lagging behind or 3) processes are stuck in I/O routines, indicating a possible file system problem.

3.2 Evaluating the Impact of the New Utility

To measure the performance impact of fast_where, we look at three facets related to usability. Are the results provided sufficiently quickly? Is the output understandable and scalable? Is the utility robust? Performance results from a scaling study are provided to answer the first question. The latter two questions are addressed by empirical observations made during the scaling study. Data was obtained using the two legacy applications, CTH and Charon.

3.2.1 Fast_where Performance Results

Timing data for fast_where on Red Storm was obtained during dedicated time to minimize variability in the results. Thus, fewer data points were needed to smooth out possible impact of other users' jobs running on the mesh. Batch jobs were submitted at various sizes for both CTH and Charon. Once the jobs were determined to be started, a simple script was run. The script used the Unix "time" command to measure how long the fast_where utility took to execute. The results are shown in Figure 3.1.

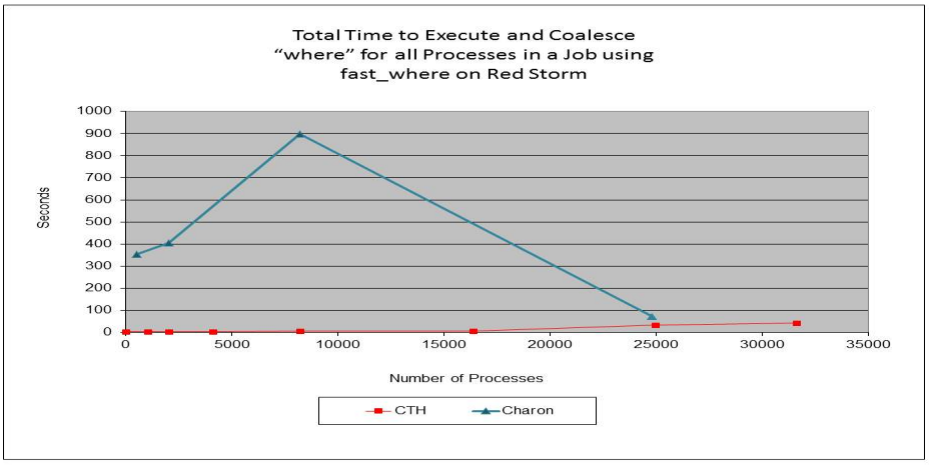


Figure 3.1: Based on the limited data points collected, fast_where execution times scaled very well with CTH. Charon results were acceptable, but difficult to interpret.

The CTH runs show very nice scalable performance with a small time to solution of 42 seconds when running on 31600 MPI ranks/processes. The results for Charon were less predictable, with no apparant pattern. The largest run with 24800 processes took the least amount of time–71 seconds. The medium-sized run with 8192 processes took 15 minutes. And finally, the smallish runs using 512 and 2048 processes, took on the order of 6 minutes. The output of the runs gave a clue as to the cause. When the data was collected for the large 24800 process run, all processes were executing in the same function. The data coalescing was trivial. For the other runs, almost every process was in a different function and reporting the high volume of not-well-summarized results was very slow. CTH is primarily a FORTRAN application, while Charon is C++ with considerable use of templates.

All of the fast_where test cases ran on Red Storm without failure. In production use, there was an early bug due to (C++) function names longer than 4096 characters. That was quickly addressed, especially since the utility is a shell script. Anyone can debug the problem, make a copy of the script, code the fix and rerun from their personal copy. Due to the nature of the CTH logic, the fast_where output was easily understable. It provided a readable and useful text file enumerating which processes were executing in a relatively small list of functions. Charon output was far less useful. If there had been a hung node, that would have been readily apparent from the output. During the dedicated test time, there were no hung nodes. Output was voluminous making it hard to interpret. Since it was text output, it would be possible for a desperate user to parse the file and find a candidate for a rogue, run-away process. But that was not the intent of the utility.

3.2.2 STAT Results

Fast_where has clearly exceeded the performance of the Totalview debugger, which was 30 minutes for 1024 processing elements. A second set of data points was obtained using another debugging utility called STAT. STAT cannot be run on Red Storm since it requires an interface not provided by the compute nodes' Catamount light weight kernel. But STAT does run on later Cray systems, such as Cielo, which run Cray's Compute Node Linux light weight kernel. The rest of Cielo's software architecture is very similar to Red Storm's. STAT was run without performing a study of tunable options.

Timing data for version 1.1.3 of STAT on Cielo was also obtained during dedicated time to minimize variability in the results. Batch jobs were submitted at various sizes for both CTH and Charon. Once the jobs were determined to be started, a simple script was once again run. The script used the Unix "time" command to measure how long the STAT utility took to execute. The available results are shown in Figure 3.2. The CTH fast_where results on Red Storm are provided for comparison purposes.

The STAT execution times for smallish core counts (less than 5000) were acceptable. A response time of 30 seconds is certainly tolerable. However, when run on 8192 cores, STAT aborted with "too many open files". Therefore, no results were available for runs above 5000 cores. It should be noted, that on Cray systems, the STAT utility is only officially supported on up to 1024 cores. It is possible that tuning options could have addressed this problem and/or will be addressed in later releases by Cray.

The Charon runs were more disconcerting. The first test used 512 cores. The STAT program appeared to execute normally and took 47 seconds. However, the Charon application aborted with an out of memory error while STAT was interogating it. This error was repeatable at all attempted process counts. The Charon binary is approximately 1GB in size. Each of the 16 processes on a single Cielo node share 32GB of memory with no swap space.

The STAT utility was not robust at the vendor-supported limit of 1024 cores. It was

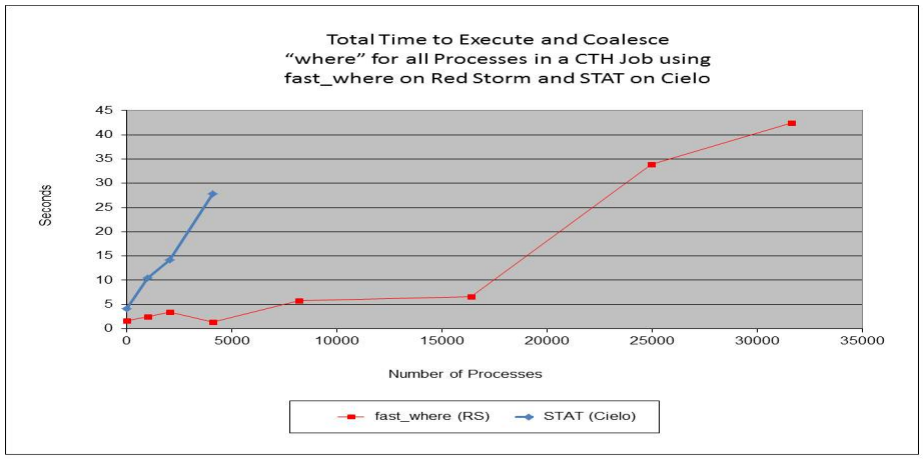


Figure 3.2: STAT timing results gave acceptable interactive response times for the data points obtained. However, fast_where exhibited a better scaling curve.

disappointing that the Charon application was aborted, rather than the STAT utility when no more memory was available. The scaling issues with STAT appear to be vendor-specific, since STAT has been demonstrated at very high scales on IBM Blue Gene systems. The graphical tree-shaped output provided by STAT is extremely intuitive and easy to understand. It is far superior to fast_where. The nature of Charon, with processes executing in many different functions at the same time was problematic for both STAT and fast_where. The graphical output was difficult to read. The graphical tree was more bush-like with function names overwriting each other on a relatively large monitor display.

3.3 Conclusion

The fast_where utility played a useful role on Red Storm. Its contribution to application performance and scalability was to reduce debugging time at high core counts. The techniques used in fast_where are not directly applicable to Unix/Linux based systems. The STAT utility provides the equivalent, plus additional features for other HPC systems. There are both Linux and Blue Gene CNK (Compute Node Kernel) [96] implementations. While STAT's implementation is immature on the Cray Linux Environment, it will likely improve to exceed that provided by fast_where.

Fast_where was a successful CSSE effort. It was used by Red Storm users and support staff to diagnose hung applications. Not every hung job ended up being one of the three scenarios it was designed to detect (hung/dead node, process "stuck in the weeds", or I/O problem), but it eliminated these problems as a possibility for the hang. The quantification of its value was reducing the time it takes to execute Totalview's "where" command. While Totalview took 30 minutes for 1024 processes, fast_where was capable of providing the equivalent information in five minutes or less on 31,600 processes. Now that's a *fast where* command!

Chapter 4

SMARTMAP Optimization to Reduce Core-to-Core Communication Overhead

Abstract: SMARTMAP [25] is a Sandia-developed operating-system (OS) memory-mapping technique that enables separate processes running on a multi-core processor to access each other's memory directly, without any intermediate data copies and without OS kernel involvement. This technique has been implemented in the Catamount OS on Red Storm and used to optimize MPI point-to-point and collective operations [21, 23]. The impact of SMARTMAP is demonstrated here by benchmarking the Charon semiconductor device simulation application running on Red Storm with and without SMARTMAP optimizations. Results indicate that SMARTMAP provides a 4.5% improvement in performance when running Charon on 2,048 quad-core compute nodes (8,192 cores) of Red Storm. Additionally, results are presented for Charon running on the newer Cielo system with and without Cray's SMARTMAP-like XPMEM optimization technique for Linux. Results indicate significantly reduced benefit compared to the Charon SMARTMAP results from Red Storm running Catamount.

4.1 SMARTMAP Overview

Sandia has recently developed a simple address space mapping capability, called SMARTMAP [25], and used it to implement multi-core optimized MPI point-to-point and collective operations [21, 23]. SMARTMAP enables cooperating processes in the same parallel job on the same multi-core processor to access each other's memory directly by using normal load and store instructions. This eliminates all extraneous data copies and operating system (OS) involvement, resulting in the minimum possible memory bandwidth utilization. Minimizing memory bandwidth is important on multi-core processors because the number of cores per processor is growing at a faster rate than per processor memory bandwidth.

SMARTMAP is implemented at the OS-level and uses the top-level page table entries of each process (i.e., address space) to map the memory of cooperating processes at fixed-offset virtual addresses. This enables a given source process to access a virtual address in a

```
static inline
void *
remote_address(unsigned rank,const void *vaddr)
{
  uintptr_t addr=(uintptr_t)vaddr;

  addr |= ((uintptr_t)(rank+1))<<39;

  return addr;
}
```

Figure 4.1: Catamount code for converting a local address to a remote address.

destination process by doing a straightforward address offset calculation, shown in Figure 4.1, and then using the resulting address directly (e.g., by casting to an appropriate pointer type and dereferencing). This operation is completely one-sided requiring no participation from the destination process. Once SMARTMAP has setup all of the required address space mappings, which typically occurs one time at job startup, there is no additional OS overhead for accessing the memory of a remote process.

SMARTMAP is a general-purpose technique. In addition to MPI, it maps very well to the partitioned global address space (PGAS) programming model and can be used to implement one-sided get/put operations such as those available in the OpenSHMEM model [24]. At the system software level, SMARTMAP functionality could be added to any OS kernel. Cray has recently developed a SMARTMAP-like technique for the Cray Linux Environment called XPMEM. SMARTMAP and XPMEM provide essentially the same user-visible capability, but have different internal implementations and runtime behavior. XPMEM performance is evaluated in Section 4.3.

4.2 SMARTMAP Performance Evaluation on Red Storm

To measure the performance benefit of SMARTMAP, the Charon ASC application was run on up to 2,048 nodes of Red Storm with and without SMARTMAP optimizations. Descriptions of Charon, the Red Storm test platform, the experiment setup, and results obtained are described in the following sections.

4.2.1 Charon Application

Charon is an ASC semiconductor device simulation code [83] that models semiconductor devices via the drift-diffusion equations. These equations, which consist of a coupled system of nonlinear partial differential equations, are discretized using a stabilized finite element formulation and solved via a Newton-Krylov approach.

Charon has been designed to run on distributed memory massively parallel computers and uses MPI for all communication. Previous empirical observations have shown Charon to make MPI collective calls at high rate, and that collective performance is a major factor constraining its performance on a large number of processors [117]. This makes Charon a good candidate for performance and scalability improvement by SMARTMAP-optimized MPI collectives.

4.2.2 Red Storm Test Platform and System Software

Red Storm is the first instance of the Cray XT architecture, and was jointly designed by Cray Inc. and Sandia. At the time of the experiment, Red Storm contained a mix of dual and quad-core compute nodes, but only quad-core processors were used for testing.

Red Storm runs the Catamount operating system on its compute nodes, which is a special-purpose lightweight kernel designed to maximize performance for large-scale ASC applications. The Catamount OS kernel was modified by Sandia to support SMARTMAP, and this version of Catamount has been running in production for over a year.

The Red Storm MPI library was extended by Sandia to take advantage of the SMARTMAP optimizations for both intra-node point-to-point and collective operations. In the case of collective operations, a hierarchical approach is used where the first step is to perform the collective within each multi-core node using SMARTMAP, then one representative from each node communicates off-node to complete the inter-node portion of the collective. When SMARTMAP collectives are turned off, the MPI library uses shared-memory for MPI messages sent between cores on the same node, which is more efficient than using the network interface in loop-back mode, but less efficient than using SMARTMAP.

4.2.3 Experiment Setup

The Charon input problem solved was to find a 2D steady-state drift-diffusion solution for a bipolar junction transistor. The Charon application code was not modified in any way. The only difference between runs was enabling or disabling SMARTMAP optimizations in the MPI library.

Three different problem sizes were evaluated: 128, 512, and 2048 nodes. In each case, four MPI processes per node were used binding one process to each of the quad-core node's cores, resulting in 512, 2048, and 8192 total cores being used for each problem size. The Charon input problem was scaled so that each core had about 31,000 unknowns per core for each configuration. Dedicated system time was used to eliminate interference with other running jobs. One job at a time was run, ensuring predictable allocation of nodes.

4.2.4 SMARTMAP Results

Table 4.1 shows the Charon solution time measured for each test configuration. Solution time measures the time needed to perform the main calculation, which includes MPI messaging between processes. Solution time includes job startup overhead (job load time, one-time data structure initialization, etc.), and I/O overhead.

As can be seen, the SMARTMAP optimizations provide up to a 7.3% improvement in solution time compared to the baseline, which uses normal shared memory instead of SMARTMAP for intra-node communication. At the largest scale tested, 2048 nodes and 8192 cores, SMARTMAP provides a 4.5% benefit.

Table 4.1: SMARTMAP/Catamount on Red Storm, Charon Solution Time

Nodes	Cores	Baseline solution time (seconds)	With SMARTMAP solution time (seconds)	**Improvement**
128	512	394.0	386.0	**2.0%**
512	2048	331.0	307.0	**7.3%**
2048	8192	446.0	426.0	**4.5%**

Table 4.2 shows the overall end-to-end runtime for each configuration, including job startup/shutdown and file I/O. The results are slightly better than the results without startup and I/O time. A possible explanation for this is that when SMARTMAP is disabled, more intermediate communication memory buffers need to be allocated and freed at job startup and shutdown. More investigation is required to determine if this is the case. Nonetheless, SMARTMAP is providing a substantial benefit.

Table 4.2: SMARTMAP/Catamount on Red Storm, Charon Overall Elapsed Time

Nodes	Cores	Baseline overall time (seconds)	With SMARTMAP overall time (seconds)	**Improvement**
128	512	414.1	403.3	**2.6%**
512	2048	366.6	335.1	**8.6%**
2048	8192	527.3	487.7	**7.5%**

4.3 XPMEM Performance Evaluation on Cielo

Tables 4.3 and 4.4 present results for Charon running on the Cielo system at LANL with and without Cray's XPMEM optimization. The test is setup similarly to the SMARTMAP results presented in Section 4.2, with the following main differences:

1. Cielo is a much newer system than Red Storm (Cray XE6 vs. XT4).

2. Cielo compute nodes have 16 cores per node vs. 4 cores per node on Red Storm. Therefore, 16 MPI ranks per node were used for the Cielo experiments.

3. Cielo runs Cray's Linux OS on compute nodes. Red Storm runs the Catamount OS.

4. Cielo uses Cray's XPMEM optimization to provide SMARTMAP-like functionality, along with a proprietary closed-source MPI library that makes use of XPMEM for intra-node core-to-core communication.

Notably, the same Charon test problem used on Red Storm was used for the Cielo experiments. As with Red Storm, dedicated system time was used on Cielo and only one application was run at a time to ensure predictable node allocation.

As can be seen from the results in Table 4.3, Cray's XPMEM does not show any benefit for Charon solution time. This was surprising since MPI microbenchmarks do show bandwidth and latency improvements when using XPMEM. It is possible that Cray's proprietary MPI is lacking the specific SMP-optimized MPI collective that Charon needs. Further investigation would be required to confirm this theory. Another possibility is that on Cielo, something other than intra-node core-to-core communication is the primary bottleneck for Charon.

Results for Charon overall runtime are improved with XPMEM optimizations, as shown in Table 4.4. Overall runtime includes job startup and shutdown overhead as well as I/O overhead. It is possible that something about using XPMEM accelerates these operations. On Cielo, use of XPMEM is the default so Cray has likely not spent much time optimizing for when XPMEM is turned off (the baseline case in the tables).

Table 4.3: XPMEM/Linux on Cielo, Charon Solution Time

Nodes	Cores	Baseline solution time (seconds)	With XPMEM solution time (seconds)	**Improvement**
128	2048	181.6	181.7	**0%**
512	8192	248.8	248.8	**0%**

Table 4.4: XPMEM/Linux on Cielo, Charon Overall Elapsed Time

Nodes	Cores	Baseline overall time (seconds)	With XPMEM overall time (seconds)	**Improvement**
128	2048	245.6	232.0	**5.5%**
512	8192	343.7	331.5	**3.5%**

4.4 Conclusion

The impact of SMARTMAP has been demonstrated by benchmarking the Charon semiconductor device simulation application running on Red Storm with and without SMARTMAP optimizations. Results show that SMARTMAP provides a 4.5% improvement to solution time and a 7.5% improvement to overall elapsed time when running Charon on 2,048 quad-core compute nodes (8,192 cores) of Red Storm. Results on Cielo using Cray's XPMEM optimization, which provides functionality similar to SMARTMAP, are less encouraging. XPMEM provided 0% benefit to Charon solution time and 3.5% improvement to overall elapsed time on 512 Cielo compute nodes (8192 cores). SMARTMAP-like techniques such as XPMEM should in theory provide increased benefit as the number of cores per node is increased, since they reduce memory bandwidth, so the XPMEM results were surprising. A possible next step is to run SMARTMAP/Catamount on Cielo to see if a larger performance benefit is observed.

Chapter 5

Smart Allocation Algorithms

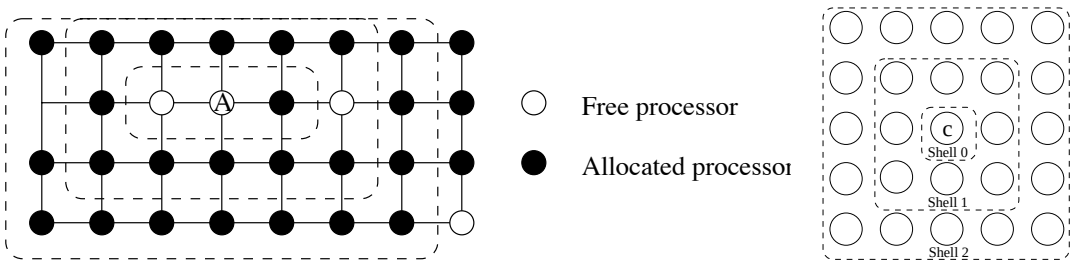
Abstract: Enhancements to processor allocation that enable improved job throughput are described here. Experiments are described that demonstrate improvement in stream running time using actual job mix samples from Red Storm whose running times were scaled down to fit our test windows on Red Storm and Cielo. While improvement was demonstrated on both Red Storm and Cielo, the magnitude of the improvement seems to increase with running time. The magnitude of the improvement increased by tenfold when the average job running time of the sample increased by less than twenty percent. This is consistent with our previous result of over twenty percent improvement on a stream of jobs that took several hours to run on a smaller machine.

5.1 Background and Motivation

Previous experiments have shown that allocating nearby processors to each job can improve throughput on a range of architectures [13, 95, 86, 80, 138]. The quality of an allocation can have a significant effect on job running time; previous work has shown that hand-placing a pair of high-communication jobs into a high-contention configuration can roughly double their running times [80]. The placement of job tasks has also been shown to speed up an actual application by up to 1.64 times [62]. Furthermore, with the exponential growth in the number of cores on chips, high-quality processor allocation of cores will be necessary for good performance on mesh-connected chips (e.g. [10]) as well.

Several papers suggest that minimizing the average number of communication hops is an appropriate metric for job placement [76, 86]. This metric was experimentally shown to correlate with running times by Leung et al. [80]. Krumke et al. [76] considered a generalization of this problem on arbitrary topologies for several measures of "locality", motivated by allocation on the CM5. They prove it is NP-hard to approximate average pairwise distance in general. We survey approximation algorithms in Subsection 5.1.1. In previous experiments, Walker et al. [137] used the simplest of these approximation algorithms, MC1x1 (see Figure 5.1(b)), to benchmark some of the heuristic algorithms surveyed in Subsection 5.1.2.

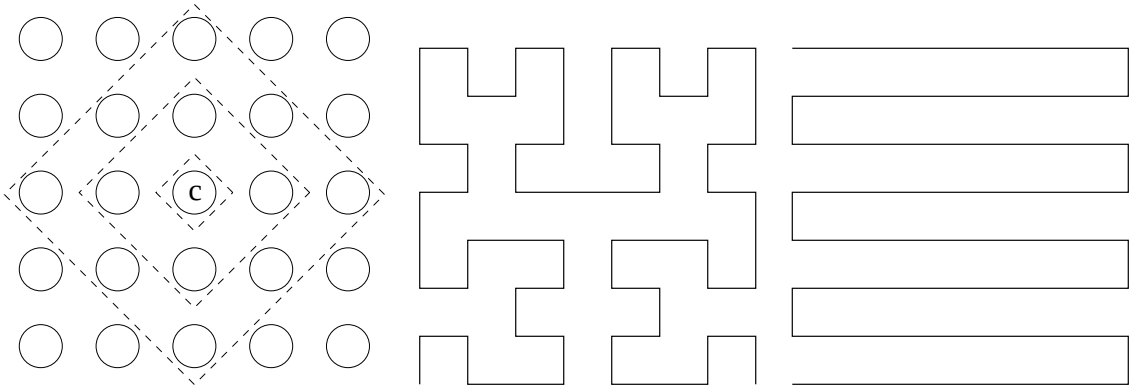
For square meshes, Walker et al. [137] find that a 1D curve-based strategy using a Hilbert curve (see Figure 5.2(b)) can give allocations of comparable quality to a fully 2D algorithm,



(a) Illustration of MC: Shells around processor A for a 3x1 request.

(b) Shells generated by MC1x1 around processor c.

Figure 5.1: MC and MC1x1.



(a) Gen-Alg's shells centered on "c".

(b) Hilbert curve.

(c) "Snake" curve.

Figure 5.2: Gen-Alg, Hilbert curve, and "snake" curve.

MC1x1. However, most meshes are not square or cubic. For non-cubic meshes, Walker et al. [137] find that a 1D curve-based strategy using a "snake" curve (see Figure 5.2(c)) that goes along the mesh's shorter dimensions first can give allocations of comparable quality to a fully 3D algorithm, MC1x1. Furthermore, these algorithms are much faster than MC1x1, which takes more than 200 times as long in some cases [137]. Based on how Cray XE6 systems are assembled, Albing et al. [4] were able to develop a Hilbert-like curve for the Cray Application Level Placement Scheduler (ALPS).

5.1.1 Approximation Algorithms

A natural algorithm for high-quality mesh allocation is MC1x1 by Bender et al. [14]. MC1x1 was developed as a variant of the algorithm MC (see Figure 5.1(a)), introduced by Mache et al. [87]. The work of Krumke et al. [76] implies that MC1x1 is a $(4 - 4/k)$-approximation for average pairwise distance on k processors.

Krumke et al. [76] proposed another member of this family called Gen-Alg (see Figure 5.2(a)). Krumke et al. [76] were explicitly trying to minimize the sum of L_1 distances

between processors and they show that Gen-Alg is a $(2-2/k)$-approximation for this problem, meaning it always finds an allocation within a factor of $(2-2/k)$ of the optimal, where k is the job size.

Also in the same family of algorithms as MC1x1 is MM, proposed by Bender et al. [14]. Bender et al. [14] show that MM is a $2-1/(2d)$-approximation algorithm in a d-dimensional mesh, making it a 7/4-approximation in 2D and an 11/6-approximation in 3D. Bender et al. [14] also give an arbitrarily-good approximation algorithm (PTAS) to find allocations that minimize the sum of pairwise distances.

5.1.2 Heuristic Algorithms

Another family of allocation algorithms is based on a linear order of processors. The first of these is Paging, proposed by Lo et al. [84], which uses a linear order to maintain a sorted list of free processors. When an allocation is needed, Paging assigns it a prefix of this list. Lo et al. [84] proposed several linear orders to use with the Paging algorithm, including the row-major, a "snake" curve that traverses alternate rows in opposite directions, and "shuffled" versions of these.

An ordering-based strategy was independently proposed by Leung et al. [80], who introduced a couple of refinements. They ordered processors using a space-filling curve such as the recursively-generated Hilbert curve [67]. These curves are recursively defined and are known to preserve several measures of "locality" [61, 94]. Although the Hilbert curve is in 2D, it has generalizations to higher dimensions [3]. Later work expanded on this by considering another curve [29] and non-square meshes [137].

They also proposed the use of strategies adapted from bin packing to select processors from the list [80]. They found that both change of ordering and the use of bin-packing heuristics gave improvements, with the curve giving the main improvement [80]. Weisser et al. [138] and Albing et al. [4] adopted the curve portion of the algorithm on BigBen and in ALPS, respectively.

As mentioned above, Walker et al. [137] find that a 1D curve-based strategy using a Hilbert curve and bin-packing heuristics can give high quality allocations for square meshes. They also find that a 1D curve-based strategy using bin-packing heuristics and a"snake" curve that goes along the mesh's shorter dimensions first can give high quality allocations for non-cubic meshes.

5.2 Experiments

To measure the performance impact of these algorithms, the three run streams described in Section 2.2 were used. Throughputs were compared by running each of these streams twice

on Red Storm or Cielo, once with the default algorithm and once with the best heuristic algorithm for the machine. The best heuristic algorithms were determined by the configuration of the machine and the simulation and smaller scale experimental results reported in Walker et al [137].

For Red Storm, a Cray XT4 that is not cubic and not running ALPS, the 1D curve-based strategy using a "snake" curve that goes along the mesh's shorter dimensions first is the best heuristic algorithm. For Cielo, a Cray XE6 running ALPS, the Hilbert-like curve for ALPS is the best heuristic algorithm. Also the algorithm on Red Storm uses strategies adapted from bin packing, while the algorithm on Cielo does not. (The strategies were not transferred from Red Storm in time to meet security requirements for Cielo.) Note that due to the research in this area, the default algorithms for both Red Storm and Cielo are far superior to the default algorithm for the original Cray XT3 [138].

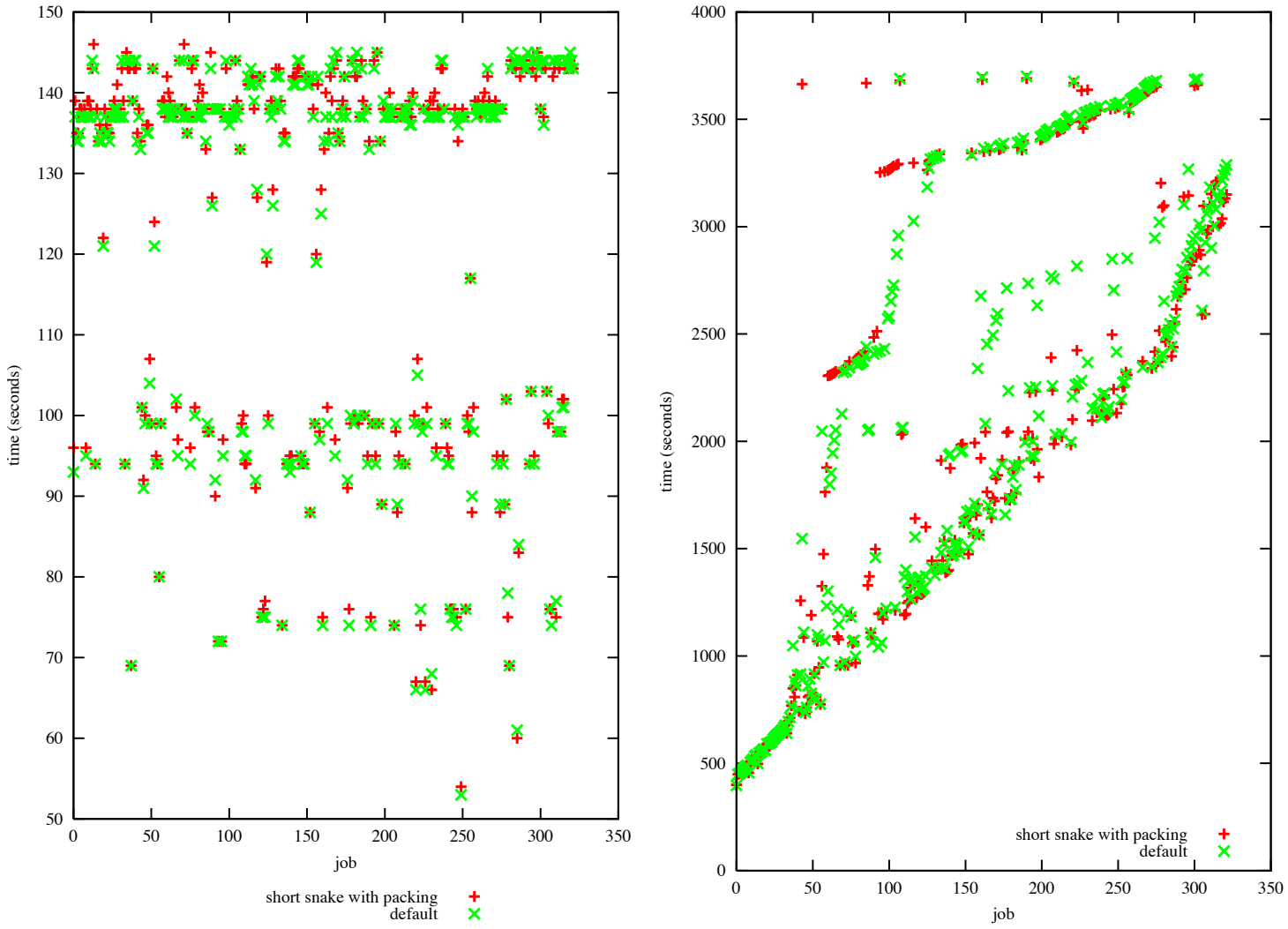
5.2.1 Details

The details of the test are similar to those in Section 2.2.1. The creation, submission, and runs in the two modes differed only in one way, the allocation algorithm used. The shock physics code, CTH was used for all of the jobs. Since we are interested in the changes in run times between the two modes, we could not request a Yod enforced time limit of the run time. Instead, we reduced the number of iterations in CTH. Using CTH in this manner allowed us to scale a two-week window of actual Red Storm usage down to about one hour of Red Storm or Cielo usage. Since CTH has highly optimized communications, this reduction in work for each job will further reduce the run time differences between the two modes and make for a very challenging test of the best heuristic algorithms.

Table 2.1 shows the start date and number of jobs for each two week window. The start dates are progressively later in time and the number of jobs are progressively fewer from windows one to three. Since Red Storm was fully utilized during each of the two week windows, the jobs in the later windows ran longer than the jobs in the earlier windows on average. A consequence of this would be greater run time difference between the two modes in later windows. This is at least partially born out in the results described in the next section.

5.2.2 Results

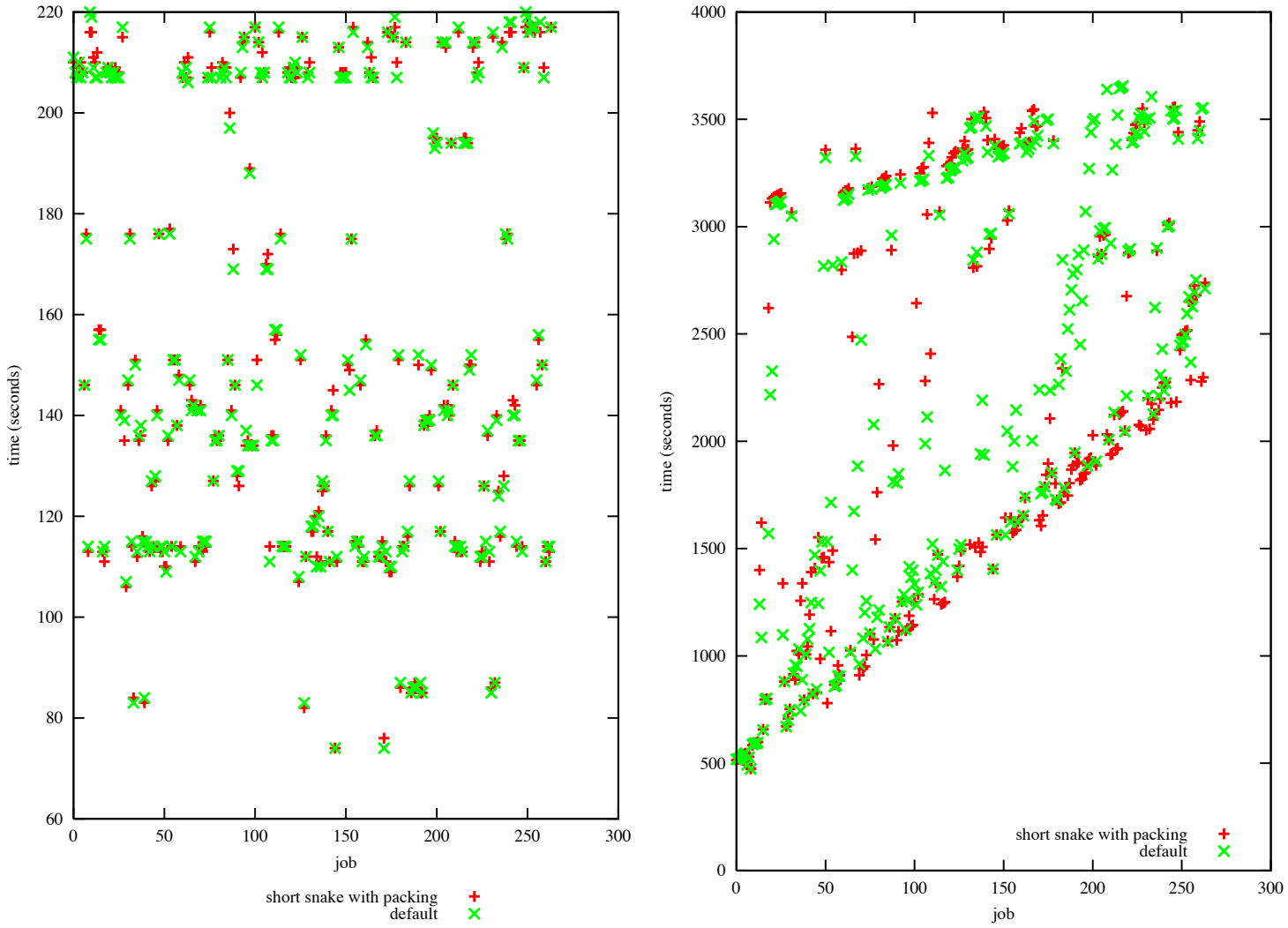
The results for smart allocation are shown in Figures 5.3, 5.4, 5.6, and 5.7 which plot the run and completion times for the runs of the job streams derived from windows one, two, and three on Red Storm or Cielo.



(a) run times

(b) completion times

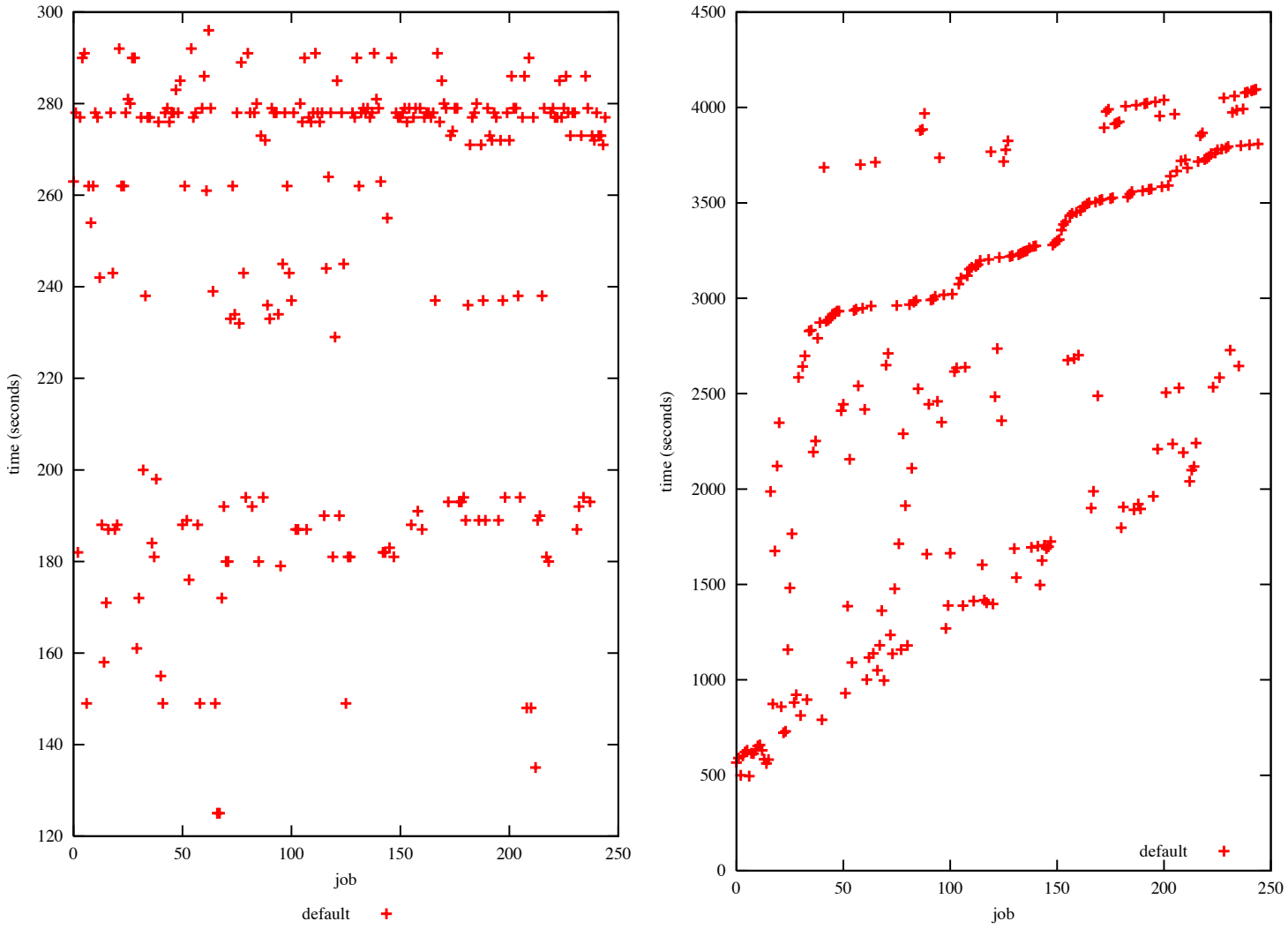
Figure 5.3: The run and completion times for the runs of the job stream derived from window one on Red Storm.



(a) run times

(b) completion times

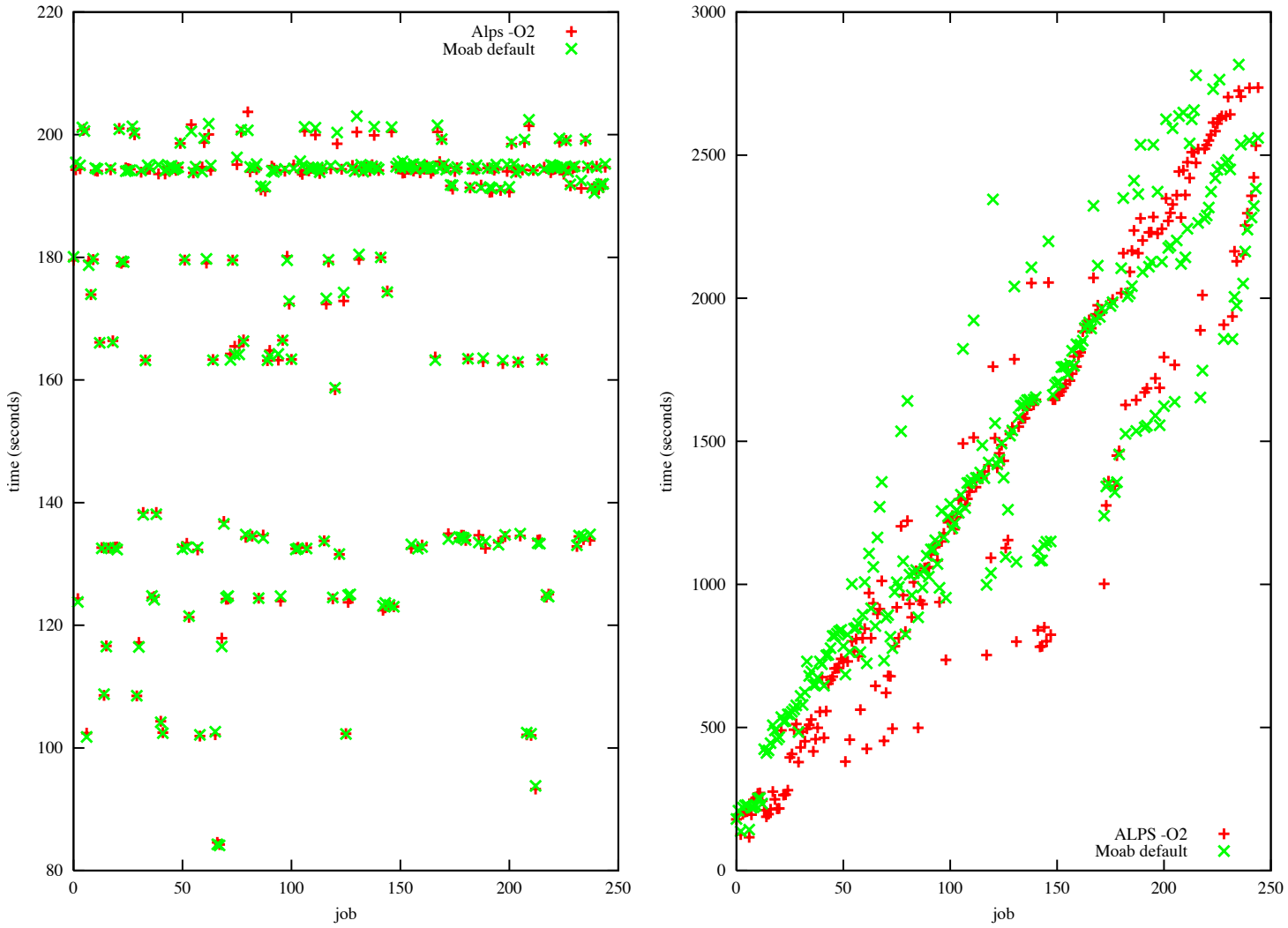
Figure 5.4: The run and completion times for the runs of the job stream derived from window two on Red Storm.



(a) run times

(b) completion times

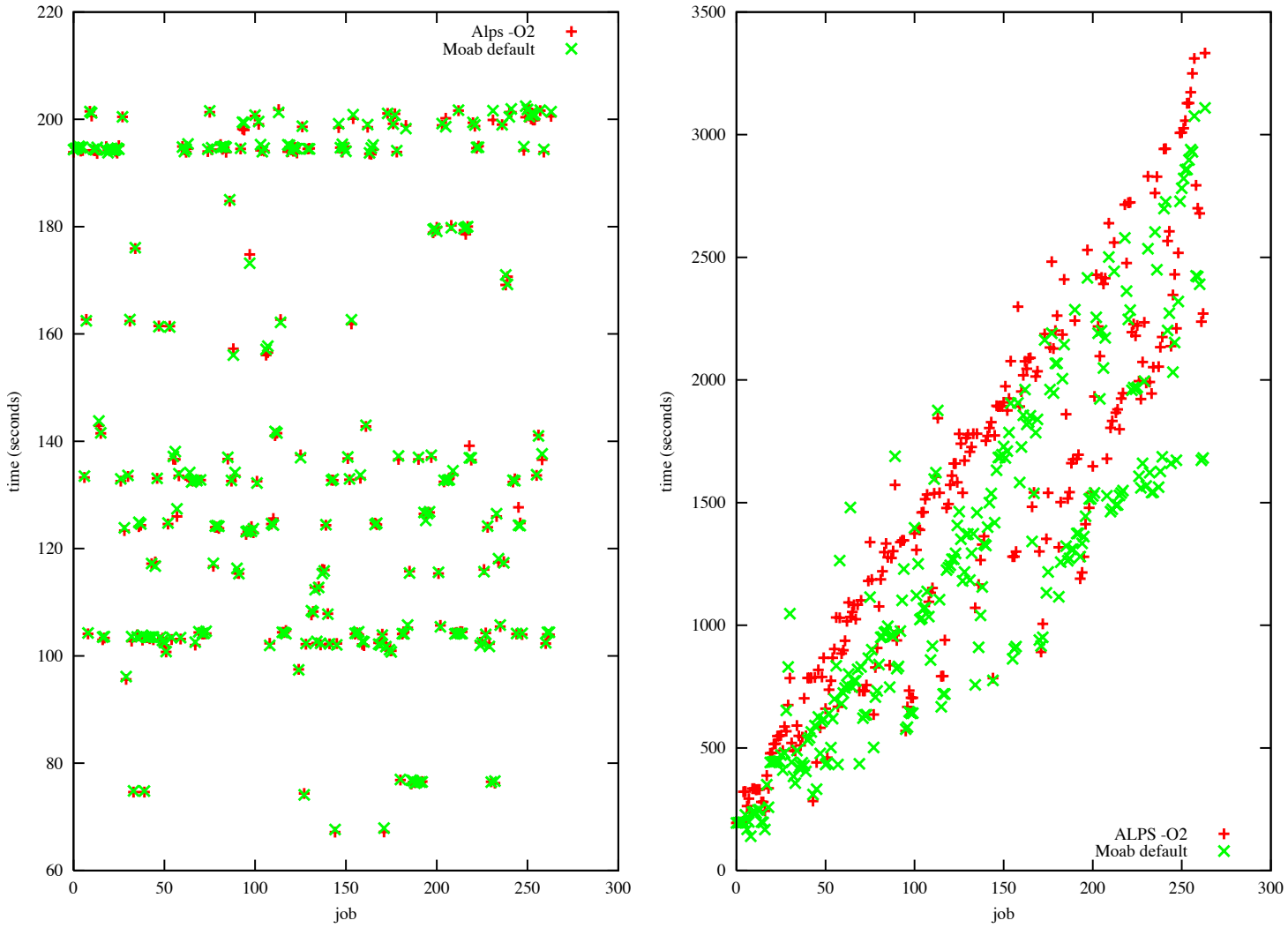
Figure 5.5: The run and completion times for the run of the job stream derived from window three on Red Storm.



(a) run times

(b) completion times

Figure 5.6: The run and completion times for the runs of the job stream derived from window three on Cielo.



(a) run times

(b) completion times

Figure 5.7: The run and completion times for the runs of the job stream derived from window two on Cielo.

5.2.2.1 Red Storm

Figure 5.3 shows the run and completion times for the runs of the job stream derive from window one on Red Storm. The completion time for a job is determined by both when it is scheduled and how it is allocated. Since both the schedule and allocation differed between the two runs, we compared the run time for each job and the job stream as a whole. Seventy-one jobs took less time, ninety-six jobs took about the same amount of time, and 155 jobs took more time with the best heuristic algorithm. Overall, it takes approximately 0.33% less time for the stream to run with the best heuristic algorithm. As noted above, the research in this area have improved the default algorithm on Red Storm, and the use of CTH and the reduction in work for each job would reduce the run time differences between the two modes.

Figure 5.4 shows the run and completion times for the runs of the job stream derive from window two on Red Storm. Ninety jobs took less time, seventy-eight jobs took about the same amount of time, and ninety-six jobs took more time with the best heuristic algorithm. Overall, it takes approximately three percent less time for the stream to run with the best heuristic algorithm. Note that window two with fewer jobs takes about the same time to run as window one. Therefore the jobs in window two must take on average more time to run. This difference could account for the greater separation between the two modes for window two.

Figure 5.5 shows that window three has fewer jobs than window two and took longer to run in the default mode. Unfortunately the experiments on Red Storm were not without some minor glitches, and we ran out of time before we could run window three with the best heuristic algorithm. So we ran the Cielo experiments in the reverse order.

5.2.2.2 Cielo

Figure 5.6 shows the run and completion times for the runs of the job stream derive from window three on Cielo. 161 jobs took less time and eighty-four jobs took more time with the best heuristic algorithm. Overall, it takes approximately three percent less time for the stream to run with the best heuristic algorithm. This is the same percentage difference for window two on Red Storm. Note that this job stream took less time to run on Cielo than it did on Red Storm. The job stream derived from a Red Storm window appears not to have posed a similar challenge to Cielo, a couple of generations beyond Red Storm.

Figure 5.7 shows the run and completion times for the runs of the job stream derive from window two on Cielo. 173 jobs took less time, and ninety-one jobs took more time with the heuristic algorithm. Overall, it takes approximately six percent more time for the stream to run with the heuristic algorithm. Note that the run times for the jobs in windows two and three seem to be fairly consistent on Cielo. This could explain the relationship between the times for windows two and three being reversed on Cielo, i.e. the time for window three was less than the time for window two. However, it is not clear why the performance of the best

heuristic algorithm flipped between the two windows.

We were able to run only two windows for both the default and best heuristic algorithm in the time allotted on Cielo. Therefore, the only overlaps between the runs on Red Storm and the runs on Cielo were for window two and part of window three. Note that the jobs were scheduled differently on Red Storm and Cielo. The scheduler on Red Storm selects the largest job possible, while the scheduler on Cielo does not. It is not clear whether this is a consequential difference.

5.3 Conclusions

Due to the research in this area, the default algorithms for both Red Storm and Cielo are far superior to the default algorithms for the original Cray XT3 [138]. Additionally, the use of CTH with its optimized communications and the reduction in its run time to fit into our test window made for a very challenging test for our best heuristic algorithms. Despite these very challenging conditions, the best heuristic algorithms were able to show improvements over the default algorithms on Red Storm and Cielo. This was especially true for longer running jobs. For both Red Storm and Cielo, the longer running jobs in the later windows yielded better results for the best heuristic algorithms. For example, both the 322 jobs in window one and the 264 jobs in window two took about an hour to run on Red Storm, but the percentage improvement of the best heuristic algorithm increased by a factor of ten from window one to window two. This is not inconsistent with our previous result of over twenty percent improvement on a stream of jobs that took several hours to run on a smaller machine [80]. The advantage of the best heuristic algorithm seems to increase for longer running jobs.

Chapter 6

Enhancements to Red Storm and Catamount to Increase Power Efficiency During Application Execution

Abstract: In the scientific High Performance Computing sector, run-time performance has been the primary, if not sole, metric for measuring success. Recently, however, the importance of power as a metric has been recognized and new multi-dimensioned metrics are being considered such as; FLOPs per Watt and energy or cost to solution. We have addressed this changing priority by demonstrating that power efficiency can be achieved with little to no impact on performance. Our experiments have shown that increased power efficiency can be gained by exploiting existing bottle-necks in scientific applications. In a series of experiments, we characterize the effect of both CPU frequency and network bandwidth tuning on power usage and demonstrate energy savings of up to 39% with little to no impact on run-time performance. These results on existing platforms indicate next generation large-scale platforms should not only approach CPU frequency scaling differently, but could also benefit from the capability to tune other platform components, such as the network, to achieve greater energy efficiency. Our results also indicate that there are significant opportunities for this work to impact how future algorithms are implemented to maximize both performance and energy efficiency.

6.1 Introduction and Motivation

Power has increasingly been identified as the *tall pole* in the path to Exascale. Applications executing on next generation systems will not only have to meet performance goals but accomplish these goals within an environment that will possibly diverge greatly from current platforms. It is clear that one of our future challenges will be increasing energy efficiency while preserving application performance. Our motivation for this work is to demonstrate improvements in energy efficiency using current ASC scientific applications on existing ASC platforms. Our experiments will not only reveal where energy efficiency can be increased but what software and hardware platform characteristics will have the most impact, if available,

on future ASC platforms.

In response to this challenge we apply a, thus far, unique ability to measure current draw and voltage, in situ, to analyze the trade-offs between performance and energy efficiency of production ASC scientific applications run at large scale (thousands of nodes) while manipulating CPU frequency and network bandwidth.

The results of our experiments clearly indicate that opportunities exist to save energy by tuning platform components without sacrificing application performance. Our goal is to reduce energy consumption of scientific applications run at very large scale while minimizing the impact on run-time performance (wall-clock execution time).

Evaluating acceptable trade-offs between energy efficiency and run-time performance is, of course, somewhat subjective. Our work indicates that the parameters of these trade-offs are application dependent. With annual power costs on track to meet or exceed acquisition costs of next generation large scale platforms, our traditional prioritization of performance above all will be forced to change. It is likely that more emphasis will be placed on energy efficiency metrics like FLOPS/Watt, Energy Delay Product or energy to solution. Regardless, performance remains a critical parameter of our evaluation.

6.2 CPU Frequency Tuning

Typical approaches to CPU frequency scaling employed by operating systems, such as Linux, while efficient for single server or laptop implementations, have proven to be detrimental when used at scale causing the equivalent of operating system jitter[113]. For this reason, it is common practice at most HPC sites that deploy medium to large scale clusters to disable frequency scaling. It is clear to us that techniques designed for laptop energy efficiency are not directly applicable to large scale HPC platforms. It is particularly important that any approach taken has a system-level, rather than node-level, view of these issues.

To accomplish our goals, we made a small number of targeted modifications to the Catamount light-weight kernel. Typically, light-weight kernels are small and accommodate low level changes more easily than general purpose operating systems, such as Linux. Our changes to Catamount involved leveraging the Dynamic Voltage and Frequency Scaling (DVFS) capabilities available on the processor (CPU). We made additions to Catamount that allowed a user space process to request a CPU frequency change. Currently, our method of frequency scaling is limited to frequencies defined in the P-state table, although most processors support frequency stepping in 100MHz increments.

Power on CMOS chips has three primary contributing factors; dynamic power, static power and leakage current. Equation 6.1 depicts each contributing factor.

$$P = dynamic_P + static_P + leakage_I$$
$$P = ACV^2f + tAVI_{short}f + VI_{leak} \quad (6.1)$$

Dynamic power currently dominates the equation, but leakage current is quickly becoming a significant contributor. We will focus on the contribution of dynamic power. Notice, there is a quadratic relationship between voltage (V) and power (P). Consequently, the largest impact on power can be obtained by lowering input voltage which is what our frequency changes ultimately accomplish.

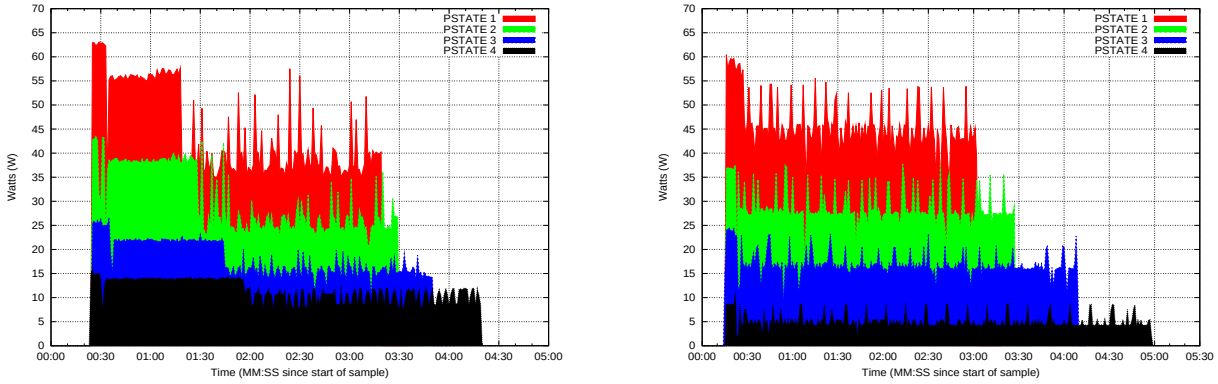
In our CPU frequency experiments we employ *static* frequency scaling. Static frequency scaling is defined, for our purposes, as changing the processor frequency to a specified state and keeping that frequency state constant over the duration of the application execution. The applications used in our experiments were selected based on their importance to the three DOE National Nuclear Security Administration (NNSA) nuclear weapons laboratories (Sandia, Los Alamos and Lawrence Livermore). The test applications include; SAGE, CTH, AMG2006, xNOBEL, UMT, Charon and LAMMPS. We tested each application at a range of frequencies (P-states) and measured the energy used over the duration of the application execution.

Decreasing CPU frequency, in general, will slow active computation. If applications were solely gated by computation this approach would be entirely detrimental. However, applications exhibit a range of characteristics. In this experiment, we altered CPU frequency and measured the impact on CPU energy and run-time (other platform parameters are left unchanged). Note, for all experiments we contrast both run-time and CPU energy to the baseline runs conducted at P-states 0 or 1 (depending on the platform used) and report the contrast as percent difference. For the baseline runs we record the execution time in seconds (s) and the energy used in Joules (J).

Our CPU frequency experiments focused on the effect that CPU frequency modifications had on CPU energy alone. CPU energy is the single largest contributor to total node energy used as observed in [52]. On our test platforms, CPU energy ranges from 44-57% of total node power. For this reason we feel measuring CPU energy in isolation is important. Further, the results of this experiment have the potential to be more widely applicable. Later, we take a broader look at total system energy in our network bandwidth tuning experiment. Total energy considers the contribution of additional node components and is somewhat more platform specific than our CPU energy analysis. We feel both approaches have utility and provide interesting insights.

6.2.1 Results: CPU Frequency Tuning

Table 6.1 lists the results of our CPU frequency scaling experiments. We obtained results for AMG2006 at P-states 1-4 at a scale of 6K cores. At P-state 2 we observed an increase in



(a) AMG

(b) LAMMPS

Figure 6.1: Application Energy Signatures of AMG2006 and LAMMPS run at P-states 1-4

run-time of 7.47% accompanied by an energy savings of 32.0%. Note, we used the longest of three run-times in each case for our final measurements. AMG2006 had a short run-time and we found the shortest run-time in P-state 2 was actually faster than the longest run-time in P-state 1. For this reason we are confident in stating that AMG2006 could benefit from a reduction in frequency to, at a minimum, P-state 2. The trade-off at P-state 3 is not as clear. The run-time impact increases more than the energy is reduced at P-states 3 and 4. We do note, while P-state 4 exhibits a significant hit in run-time, we measured the largest savings in energy recorded in our experiments. Depending on policies and/or priorities AMG2006 might be able to take advantage of any of the available P-states to produce significant savings in energy.

LAMMPS (tested at 16K cores), in contrast to AMG2006, does not display a clear win when run at lower frequencies. Our results for P-state 2 show a 16.3% increase in run-time and a 22.9% decrease in energy. The results for P-states 3 and 4 demonstrate a very significant hit in run-time. While it is our opinion that increases in run-time of this magnitude are not acceptable, LAMMPS does, however, show a larger percent savings in energy versus run-time impact at P-states 2, 3 and 4. This may be acceptable in some circumstances where energy consumption is the primary consideration or policy decisions enforce energy limitations.

Figures 6.1(a) and 6.1(b) graphically depict each of the four executions of AMG2006 and LAMMPS at P-states 1-4. The shaded area under each curve represents the energy used over the duration of the application. Figure 6.1(a) clearly depicts the positive run-time vs. energy trade-off for AMG2006 indicated in Table 6.1, especially between P-states 1 and 2. In contrast, more dramatic increases in run-time can be seen in figure 6.1(b) for LAMMPS. While AMG2006 showed a favorable trade-off between run-time and energy when run at lower frequencies there might be even more benefit to obtain. Notice the compute intensive phase of AMG2006 early in the application execution. If we were to maintain high CPU frequency during this phase but transition to lower frequency for the remainder of the application it is likely that the energy savings would be even greater. While LAMMPS did not show a clear

win when lowering CPU frequency we observed very regular compute and communication phases throughout the entire application execution (indicated by peaks and valleys in the graph). We have work planned to further examine these application characteristics, and potentially leveraged them using a more dynamic approach, to obtain power savings even in applications that do not present a clear choice when we statically modify the CPU frequency.

We obtained results for SAGE at two different scales (4K and 16K cores). In both cases, a small increase in run-time (0.402% at larger scale and 3.86% at smaller scale) is observed accompanied by a very significant reduction in energy (39.5% at large scale and 38.9% at small scale). We were able to obtain results for a 4k core run of SAGE at P-state 3. The impact on run-time remains low while additional energy savings were recorded. Based on our observations, it is possible that SAGE run at 16k cores would have also demonstrated a favorable trade-off at P-state 3.

CTH was executed at P-states 0, 2 and 3 at a scale of 16K cores. Similar to LAMMPS, there is no clear win with CTH when the CPU frequency is lowered. Also, like LAMMPS, CTH has very regular compute and communication phases. LAMMPS and CTH will likely both be targets of our future work in dynamic CPU frequency scaling at large scale.

We obtained results for xNOBEL at 6K cores at P-states 0, 2 and 3. Our results indicate that xNOBEL, like AMG2006, is a good candidate for CPU frequency reduction. Having the ability to tune CPU frequency at large scale for this application is a clear win.

UMT and Charon behaved in a very similar manner. Since UMT was run at a much larger scale than Charon (16K cores vs. 4K cores) we feel the results obtained for UMT are more meaningful and more accurately represent what we could expect at large scale. Charon may act differently when run at larger scale, but these results indicate that both UMT and Charon are sensitive to CPU frequency changes. It is possible that further analysis will reveal opportunities to dynamically scale frequency during the execution of these applications.

The CPU is only one component that affects application performance. In the following section we will experiment with tuning network bandwidth and observe the trade-offs between performance vs. total system energy.

Table 6.1: Experiment 1: CPU Frequency Scaling: Run-time and CPU Energy %Difference vs. Baseline

		Baseline Frequency		P-2 - 1.7 GHz %Diff		P-3 - 1.4 GHz %Diff		P-4 - 1.1 GHz %Diff	
	Nodes/Cores	Run-time (s)	Energy (J)	Run-time	Energy	Run-time	Energy	Run-time	Energy
HPL	6000/24000	1571	4.49×10^{8}	21.1	(26.4)				
Pallas	1024/1024	6816	1.72×10^{8}	2.30	(43.6)				
AMG2006	1536/6144	174	9.49×10^{6}	7.47	(32.0)	18.4	(57.1)	39.1	(78.0)
LAMMPS	4096/16384	172	2.79×10^{7}	16.3	(22.9)	36.0	(48.4)	69.8	(72.2)
SAGE	4096/16384	249	4.85×10^{7}	0.402	(39.5)				
(weak)	1024/4096	337	1.51×10^{7}	3.86	(38.9)	7.72	(49.9)		
CTH	4096/16384	1753	3.60×10^{8}	14.4	(28.2)	29.0	(38.9)		
xNOBEL	1536/6144	542	4.96×10^{7}	6.09	(35.5)	11.8	(50.3)		
UMT	4096/16384	1831	3.48×10^{8}	18.0	(26.5)				
Charon	1024/4096	879	4.47×10^{7}	19.1	(27.8)				

Table 6.2: Experiment 2: Network Bandwidth: Run-time and Total Energy %Difference vs. Baseline

		Baseline Bandwidth (BW)		1/2 BW %Diff		$1/4^{th}$ BW %Diff		$1/8^{th}$ BW %Diff	
	Nodes/Cores	Run-time (s)	Energy (J)	Run-time	Energy	Run-time	Energy	Run-time	Energy
SAGE_strong	2048/4096	337	5.79×10^{7}	(0.593)	(15.3)	8.90	(15.5)	20.2	(11.4)
SAGE_weak	2048/4096	328	5.64×10^{7}	0.609	(14.3)	8.23	(15.8)	22.6	(9.63)
CTH	2048/4096	1519	2.58×10^{8}	9.81	(7.09)	30.2	1.04	40.4	3.50
AMG2006	2048/4096	859	1.45×10^{7}	(0.815)	(15.8)	(0.116)	(22.7)	0.931	(25.9)
xNOBEL	1536/3072	533	7.01×10^{7}	(0.938)	(15.4)	(0.375)	(22.2)	(0.375)	(25.9)
UMT	512/1024	838	3.57×10^{7}	0.357	(14.7)	1.07	(21.7)	6.32	(21.8)
Charon	1024/2048	1162	9.96×10^{7}	1.55	(13.7)	2.15	(20.8)	2.67	(24.5)

6.3 Network Bandwidth Tuning

Our goal in this experiment was to determine the affect on run-time performance and energy of production scientific applications run at very large scale while tuning the network bandwidth of an otherwise balanced platform. To accomplish network bandwidth scaling we employed two different tunable characteristics of the Cray XT architecture. First, we tuned the Seastar NIC to reduce the *interconnect* bandwidth in stages to 1/2 and $1/4^{th}$ of full bandwidth. Next, we used the ability to tune the node *injection* bandwidth, effectively reducing the network bandwidth to $1/8^{th}$. This allowed for the most complete stepwise reduction in overall network bandwidth we were able to achieve using this architecture.

Modifying the network interconnect bandwidth on the Cray XT requires a fairly simple change to the router configuration file, consulted during the routing process of the boot sequence. A full system reboot is required for every alteration of the interconnect bandwidth. Typically, all four rails of the Seastar are configured to be enabled. Alternatively, the number of enabled rails can be reduced by specifying a bitmask in the system's router configuration file (e.g., 1111 for four rails, 0011 for two rails). In our experiments, we configured the interconnect bandwidth of the Seastar to effectively tune the network bandwidth to full, 1/2 and $1/4^{th}$.

Since the interconnect bandwidth on the XT architecture is far greater than the injection bandwidth of an individual node, the interconnect bandwidth had to be reduced to 1/2 before it produced a measurable effect. Multiple nodes may route through an individual Seastar depending on communication patterns and node placement relative to logical network topology. For this reason, we limited our experiments to one application executing at a time. This allowed for the nearest estimation of the impact of network bandwidth tuning on an individual application. Running other applications concurrently would be an interesting experiment but would greatly complicate analysis and was beyond the scope of this experiment but is planned for future investigations.

Tuning the node injection bandwidth, to further reduce the network bandwidth, requires a small modification to the Cray XT bootstrap source code. Cray provided us access to this source code under a proprietary license. The portion of the code that required modification (*coldstart*) serves an equivalent purpose to the BIOS on a personal computer or server. Our experiments were conducted in phases beginning with a baseline full bandwidth run for each application followed by subsequent executions at each reduced bandwidth.

For each phase we collected power samples (current draw and voltage). The scale used for each application, number of nodes and cores, is listed in Table 6.2. No operating system modifications were necessary for either the interconnect or injection bandwidth experiments.

6.3.1 Results: Network Bandwidth Tuning

Total energy in Table 6.2 includes the measured energy from the CPU, a measured energy from the Seastar and an estimated energy from the memory subsystem. Node energy is calculated by totaling the energy used by all nodes measured for each experiment divided by the number of nodes measured to produce the average energy used by each node (E_{cpu}). Current draw of the Seastar, measured from the VRM supporting the entire mezzanine, is constant since the SerDes do not throttle up and down based on network traffic or demand. There are four Seastars in a mezzanine, therefore, we multiply the current reading by the input voltage and divide by four to produce the baseline Seastar energy value ($E_{network}$). For 1/2, $1/4^{th}$ and $1/8^{th}$ network bandwidth calculations we assume a linear reduction in power. As stated, we use an estimated per node memory energy value which remains constant for each experiment (E_{memory}). Finally, the separate individual node CPU, network and memory values are summed then multiplied by the number of nodes participating in each experiment to determine the total energy. The calculation is as follows (where E = Energy):

$$(E_{cpu} + E_{network} + E_{memory}) \times \text{number of nodes} = \textbf{Total Energy} \tag{6.2}$$

In our calculations we used 25 W for the full network bandwidth value, 12.5 W for 1/2, 6.25 W for $1/4^{th}$ and 3.125 W for $1/8^{th}$ network bandwidth. We used 20 watts for the memory value in all calculations primarily to avoid the network energy having a disproportional affect on the total energy calculation.

Addressing each application in table order (see Table 6.2) we see SAGE displays similar characteristics for both input problems tested. Reducing the network bandwidth by 1/2 has little affect on the run-time while a significant savings in energy is experienced. The impact on run-time is larger when the network bandwidth is reduced to $1/4^{th}$ with little additional energy savings. At $1/8^{th}$ network bandwidth SAGE, for both input problems, experiences significant impacts on run-time accompanied by smaller energy savings. Based on this data, reducing network bandwidth by 1/2, if we could reduce the corresponding energy consumption of the network by half, would be advantageous. Considering the run-time energy trade-off we would likely not choose to reduce the network bandwidth beyond 1/2.

CTH was affected more by changes in the network bandwidth than any other application we tested. Even at 1/2 bandwidth, CTH experiences a greater percent increase in run-time (9.81%) than is saved by reducing network energy (7.09% decrease in total energy). At $1/4^{th}$ bandwidth, CTH experiences a very large increase in run-time (30.2%) accompanied by an actual increase in energy used of 1.04%. Clearly, reducing network bandwidth further is highly detrimental to both run-time and energy as can be seen from the $1/8^{th}$ network bandwidth results. Even at this moderately large scale CTH requires a high performance network to execute efficiently.

AMG2006 and xNOBEL, in contrast with CTH, are insensitive to the network band-

width changes in terms of run-time, but demonstrate large energy savings opportunities. Reductions down to $1/8^{th}$ network bandwidth cause virtually no impact in run-time for both AMG2006 and xNOBEL while a 25.9% savings in energy can be achieved for both. We do note the savings in energy seems to be flattening by the time we reduce network bandwidth to $1/8^{th}$. While further reductions in network bandwidth may or may not increase run-time, there is likely little additional energy savings available.

UMT produced similar results to AMG2006 and xNOBEL when the network bandwidth was reduced up to $1/4^{th}$, little to no impact in run-time accompanied by a large energy savings. At $1/8^{th}$ network bandwidth we see different characteristics. UMT experiences a much higher impact on run-time at $1/8^{th}$ network bandwidth (6.32%) than at $1/4^{th}$ (1.07%) with virtually no additional energy savings (21.7% at $1/4^{th}$ and 21.8% at $1/8^{th}$). We seem to have found the limit of network bandwidth tuning that should be applied to UMT at least at this scale. We should note that UMT was run at a smaller scale relative to the other applications. It is possible that at larger scale our results would differ.

Charon showed small, but increasing, impact on run-time as we reduced network bandwidth. At this scale it is clear that we could reduce the network bandwidth down to $1/4^{th}$ with probably an acceptable impact in run-time (increase of 2.15%) accompanied by a very significant savings in energy (decrease of 20.8%). Moving from $1/4^{th}$ to $1/8^{th}$ network bandwidth shows indications that the energy savings is flattening but results are not conclusive. Experiments with Charon at larger scale are also warranted.

Overall, we observed significant evidence that a tunable network would be beneficial for most of the applications tested. In all cases but CTH, virtually no impact to run-time would be experienced by tuning the network bandwidth to 1/2. The result would be significant energy savings with little to no performance impact. In the case of AMG2006, xNOBEL and UMT the network bandwidth could be reduced to $1/4^{th}$ full bandwidth with little run-time impact, allowing for even larger energy savings. Our observations indicate that a tunable network would be beneficial but they also indicate a high performance network is critical for some applications. The ability to tune the network, similar to how frequency is tunable on a CPU, would be an important characteristic on next generation exascale platforms.

It should be stressed that our data is representative of a single application running at a time. One of the reasons the interconnect bandwidth of the Seastar was designed to be greater than the injection bandwidth of a single node is that the network is a shared resource on the Cray XT architecture, shared by all simultaneously running applications. Often many hops are required for a messages to travel from source to destination, and poor node mappings result in individual network links carrying messages for multiple applications. Having a greater interconnect bandwidth is essential for handling this increased load. Thus, the ability to tune network performance could not be exploited without considering the possible impact on other applications running on the platform, at least for network topologies like meshes and 3D-toruses. Network topologies with fewer hops on average could benefit more easily from a tunable network since less consideration would be necessary regarding the impact on other applications co-existing on the platform.

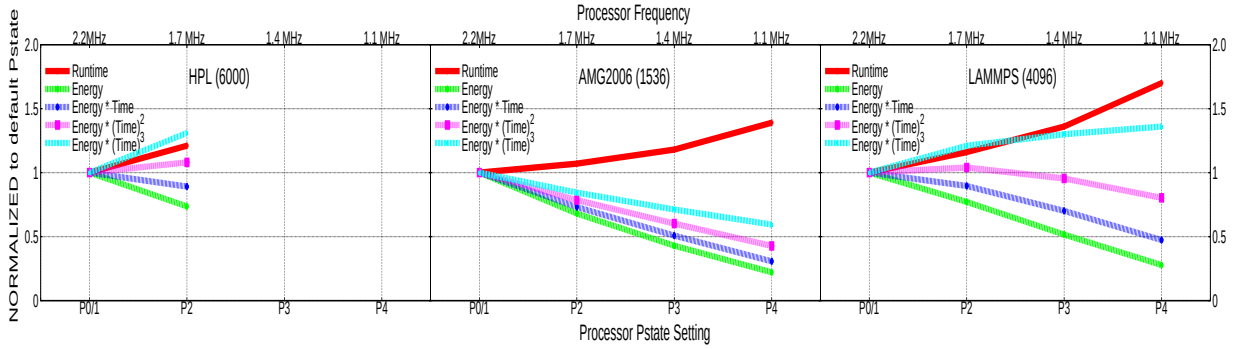


Figure 6.2: Normalized Energy, Run-time and $(E * T^w)$ where $(w) = 1, 2, or 3$

6.4 Energy Delay Product

Calculating the **E**nergy **D**elay **P**roduct (EDP) [69],[28] is one way of producing a single one-dimensional metric to evaluate both the energy and performance impact. In figure 6.2 we have graphed the energy and runtime normalized to baseline and three separate EDP curves using the equation $E * T^w$ where $w = 1, 2$ *or* 3 for HPL, AMG2006 and LAMMPS. The graphs shown are based on the same data used for Table 6.1 from experiment #1. Data from both experiments #1 and #2 have been analyzed and while too extensive for this report results can be found in SAND2011-5702 in their entirety. The HPL graph demonstrates the progressive affect increasing the exponent of the time factor has on the metric. This graph reinforces why we would likely not use an unweighted EDP calculation for HPC applications. The EDP squared and EDP cubed plots more accurately represent our previous analysis of the affect of lowering frequency for HPL. The graphs representing AMG and LAMMPS parallel our previous analysis nicely. Even the EDP cubed curve indicates it is beneficial to lower the CPU frequency while running AMG. In the case of LAMMPS if we based our analysis on EDP squared we might determine there is benefit if we run at P-state 4. If we evaluate based on the EDP cubed curve all P-states are detrimental. Generally, no significantly different insights were discovered using EDP for analysis but they did serve as reinforcement of our initial conclusions.

6.5 Conclusions

Applications like AMG2006, and xNOBEL were fairly insensitive to CPU frequency reductions. Both AMG2006 and xNOBEL were also tolerant of network bandwidth reductions. It is possible that if we tune both the CPU frequency and network bandwidth, at the same time, we could realize even greater energy savings. As an example, if we use the per node energy value for xNOBEL when executed at P-state 2 in our total energy calculation for the $1/8^{th}$ network bandwidth experiment, xNOBEL would experience a total of 56.4% in energy savings with a 6% impact in runtime.

We specifically selected xNOBEL in our previous example since it was the least impacted by any tuning we applied. CTH, however, was significantly affected by CPU frequency adjustments. If we did choose to reduce CPU frequency, could we reduce the network bandwidth without further impact on run-time but save additional energy making the run-time impact more palatable? Possibly, if memory or some other component is the bottleneck, but we cannot definitively state this would be the result. A better approach for applications like CTH would be to apply dynamic frequency scaling but coordinated with application compute and communication cycles.

Our experiments indicate each application has a *sweet spot* based on its computation and communication requirements. Additionally, we have observed energy savings and run-time are impacted, often significantly, by scale. We feel that the trade-offs are platform and application specific – when one bottleneck is removed, another will appear, and the order that the bottlenecks appear will depend on the platform. Conducting these experiments on a platform like the Cray XT is valuable since it is a well-balanced in its default configuration. As a result of our experiments, we have concluded that components on future HPC platforms (exascale and beyond) should be as tunable as possible under software control so that end-users or system software can optimize the energy/performance trade-off.

Chapter 7

Reducing Effective I/O Costs with Application-Level Data Services

Abstract:

Data Services are applications that run on compute or service nodes of HPC platforms that provide a staging or processing capability on behalf of another running application. A traditional example of a system-level service is a file-system server that processes requests on an I/O node of a parallel file system like Lustre or Panasas. In contrast, an application-level service is owned and managed by the user. These services can provide a data-processing capability that cooperates with a large-scale scientific application to significantly reduce the I/O overheads seen by applications on HPC systems. This chapter describes the Network Scalable Service Interface (Nessie), a framework for developing and deploying data services on existing HPC system, and presents performance results from two example data services: a simple data-transfer service, and a production-level data-staging service for a commonly used high-level I/O library.

7.1 Background and Motivation

For decades there has been a growing mismatch in the rate scientific applications generate data and the rate parallel file systems can consume that data. There are a number of compounding reasons for this mismatch. First, technology for parallel file systems for large-scale capability class systems has remained fairly stagnant for the last decade and is still largely based on parallel disks and a sequential POSIX model. Second the scale in number of processors, system memory, and fidelity of the application puts an increasing burden on the file system by increasing the size of resulting data sets from application simulations. Finally, the primary means of resilience which is application-directed checkpoint, is incredibly I/O intensive and consumes a high percentage of I/O bandwidth at large scales [101]

For high-performance computing (HPC) applications, it is often easy to identify distinct phases in the application's runtime where computation, communication, or I/O dominate the current activity. For example, a finite-difference code three distinct phases for each timestep calculation. The first phase is a period of intense computation where the code calculates a

resulting value for a particular data point based on a function applied to its nearest neighbors. After the compute-intensive phase, there is a period of intense communication where processor boundary values are exchanged with other processors in the parallel application. Finally, at the end of the communication-intensive phase for some of the timestep calculations, there is an I/O-intensive phase where the application writes a (perhaps large) portion of its memory to the file system as a "restart" file allowing the application to make progress in the event of an application or system failure.

Of the three phases, I/O is often the largest performance bottleneck for the application. While there are known techniques for overlapping communication and computation (e.g., double buffering and asynchronous message passing), the I/O cost is difficult to hide. Using traditional POSIX interfaces, the majority of the application is idle while data streams from memory to the remote storage devices. Efforts to make the I/O phase asynchronous require large local memory reserves and have often led to unexpected interference effects that further reduce application performance [1].

One approach to both reduce the burden on the file system and improve "effective" I/O rates for the application is to employ the use of "data services" [1, 85, 97, 100, 139]. Simply put, a data service is a separate (possibly parallel) application that performs operations on behalf of an actively running scientific application. This type of processing was first demonstrated for HPC computing as part of the Salvo seismic imaging application in mid 1990s [106]. Salvo used an "I/O Partition" to perform preprocessing (FFTs) on data between the storage systems and the application. As HPC systems continued to evolve, the need to supplement parallel file systems with similar approaches became evident. Now there are entire research communities exploring better ways to use similar processing techniques for data staging and in-situ analysis.

A data service architecture uses remote direct-memory access (RDMA) to move data from memory to memory between the application and the service(s). Figure 7.1 illustrates the organization of an application using data services. On current capability-class HPC systems, services execute on compute nodes or service nodes and provide the application the ability to "offload" operations that present scalability challenges for the scientific code. One commonly used example for data services is data staging, or caching data between the application and the storage system [100, 101, 118]. Section 7.4 describes such a service. Other examples include proxies for database operations [105] and in-situ data analysis [57, 85, 139].

This chapter provides a description of our data-service framework the Network Scalable Service Interfaice (Nessie), as well as examples and performance results of data services currently in use or in development at Sandia. The data-transfer service, described in Section 7.3, is the canonical example on how to develop a data service using Nessie, and the PnetCDF service from Section 7.4 is an example of link-time replacement I/O library that performs data-staging for bursty I/O operations. Finally, we briefly discuss other options for data services, including an in-development service for real-time analysis of data from the CTH shock physics application.

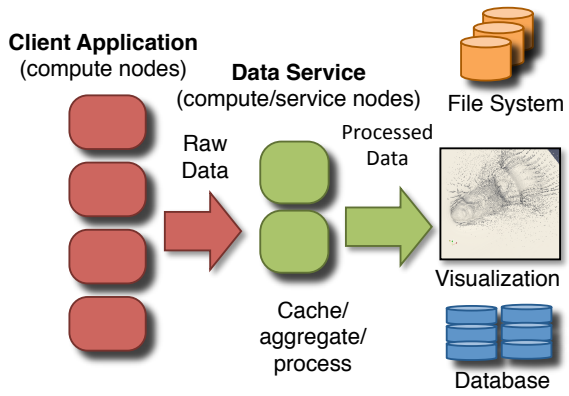


Figure 7.1: A data service uses additional compute resources to perform operations on behalf of an HPC application.

7.2 Nessie

The NEtwork Scalable Service Interface, or Nessie, is a framework for developing parallel client-server data services for large-scale HPC systems [85, 104].

Nessie was originally developed out of necessity for the Lightweight File Systems (LWFS) project [103], a joint effort between researchers at Sandia National Laboratories and the University of New Mexico. The LWFS project followed the same philosophy of "simplicity enables scalability", the foundation of earlier work on lightweight operating system kernels at Sandia [119]. The LWFS approach was to provide a core set of fundamental capabilities for security, data movement, and storage and afford extensibility through the development of additional services. For example, systems that require data consistency and persistence might create services for transactional semantics and naming to satisfy these requirements. The Nessie framework was designed to be the vehicle to enable the rapid development of such services.

Because Nessie was originally designed for I/O systems, it includes a number of features that address scalability, efficient data movement, and support for heterogenous architectures. Features of particular note include 1) using asynchronous methods for most of the interface to prevent client blocking while the service processes a request; 2) using a server-directed approach to efficiently manage network bandwidth between the client and servers; 3) using separate channels for control and data traffic; and 4) using XDR encoding for the control messages (i.e., requests and results) to support heterogenous systems of compute and service nodes.

A Nessie service consists of one or more processes that execute as a serial or parallel job on the compute nodes or service nodes of an HPC system. We have demonstrated Nessie services on the Cray XT3 at Sandia National Laboratories, the Cray XT4/5 systems at ORNL, and a large InfiniBand cluster at SNL. The Nessie RPC layer has direct support of Cray's SeaStar interconnect [26], through the Portals API [27]; Cray's Gemini interconnect [6]; and InfiniBand [9].

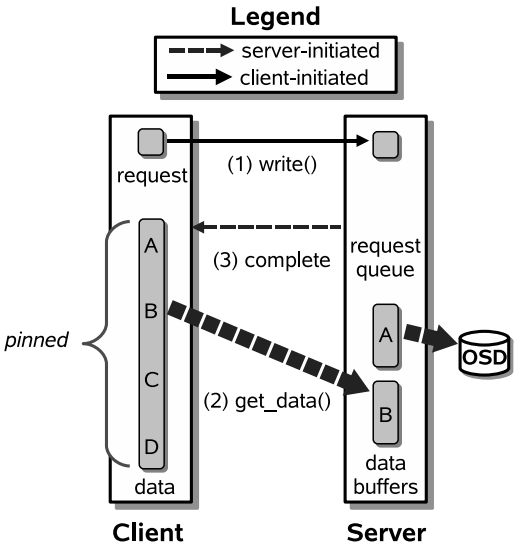


Figure 7.2: Network protocol for a Nessie storage server executing a write request. The initial request tells the server the operation and the location of the client buffers. The server fetches the data through RDMA get commands until it has satisfied the request. After completing the data transfers, the server sends a small "result" object back to the client indicating success or failure.

The Nessie API follows a remote procedure call (RPC) model, where the client (i.e., the scientific application) tells the server(s) to execute a function on its behalf. Nessie relies on client and server stub functions to encode/decode (i.e., marshal) procedure call parameters to/from a machine-independent format. This approach is portable because it allows access to services on heterogeneous systems, but it is not efficient for I/O requests that contain raw buffers that do not need encoding. It also employs a 'push' model for data transport that puts tremendous stress on servers when the requests are large and unexpected, as is the case for most I/O requests.

To address the issue of efficient transport for bulk data, Nessie uses separate communication channels for control and data messages. In this model, a control message is typically small. It identifies the operation to perform, where to get arguments, the structure of the arguments, and so forth. In contrast, a data message is typically large and consists of "raw" bytes that, in most cases, do not need to be encoded/decoded by the server. For example, Figure 7.2 shows the transport protocol for an I/O server executing a write request.

The Nessie client uses the RPC-like interface to push control messages to the servers, but the Nessie server uses a different, one-sided API to push or pull data to/from the client. This protocol allows interactions with heterogeneous servers and benefits from allowing the server to control the transport of bulk data [75, 130]. The server can thus manage large volumes of requests with minimal resource requirements. Furthermore, since servers are expected to be a critical bottleneck in the system (recall the high proportion of compute nodes to I/O nodes in MPPs), a server directed approach affords the server optimizing request processing for efficient use of underlying network and storage devices – for example, re-ordering requests

to a storage device [75].

7.3 A Simple Data-Transfer Service

The data-transfer service is included in the "examples/xfer-service/" directory of the Trios package. This example demonstrates how to construct a simple client and server that transfer an array of 16-byte data structures from a parallel application to a set of servers. The code serves three purposes: it is the primary example for how to develop a data service, it is used to test correctness of the Nessie APIs, and we use it to evaluate network performance of the Nessie protocols.

Creating the transfer-service requires the following three steps:

1. Define the functions and their arguments.
2. Implement the client stubs.
3. Implement the server.

7.3.1 Defining the Service API

To properly evaluate the correctness of Nessie, we created procedures to transfer data to/from a remote server using both the control channel (through the function arguments or the result structure) and the data channel (using the RDMA put/get commands). We defined client and server stubs for the following procedures:

`xfer_write_encode` Transfer an array of data structures to the server through the procedure arguments, forcing the client to encode the array before sending and the server to decode the array when receiving. We use this method to evaluate the performance of the encoding/decoding the arguments. For large arrays, this method also tests our two-phase transfer protocol in which the client pushes a small header of arguments and lets the server pull the remaining arguments on demand.

`xfer_write_rdma` Transfer an array of data structures to the server using the data channel. This procedure passes the length of the array in the arguments. The server then "pulls" the unencoded data from the client using the `nssi_get` function. This method evaluates the RDMA transfer performance for the `nssi_get_data` function.

`xfer_read_encode` Transfer an array of data structures to the client using the control channel. This method tells the server to send the data array to the client through the result data structure, forcing the server to encode the array before sending and the client to decode the array when receiving. This procedure evaluates the performance

```
/* Data structure to transfer */
struct data_t {
    int int_val; /* 4 bytes */
    float float_val; /* 4 bytes */
    double double_val; /* 8 bytes */
};

/* Array of data structures */
typedef data_t data_array_t<>;

/* Arguments for xfer_write_encode */
struct xfer_write_encode_args {
    data_array_t array;
};

/* Arguments for xfer_write_rdma */
struct xfer_write_rdma_args {
    int len;
};

...
```

Figure 7.3: Portion of the XDR file used for a data-transfer service.

of the encoding/decoding the arguments. For large arrays, this method also tests our two-phase transfer protocol for the result structure in which the server pushes a small header of the result and lets the client pull the remaining result on demand (at the `nssi_wait` function).

`xfer_read_rdma` Transfer an array of data structures to the client using the data channel. This procedure passes the length of the array in the arguments. The server then "puts" the unencoded data into the client memory using the `nssi_put_data` function. This method evaluates the RDMA transfer performance for the `nssi_put_data` function.

Since the service needs to encode and decode remote procedure arguments, the service-developer has to define these data structures in an XDR file. Figure 7.3 shows a portion of the XDR file used for the data-transfer example. XDR data structures definitions are very similar to C data structure definitions. During build time, a macro called "`TriosProcessXDR`" converts the xdr file into a header and source file that call the XDR library to encode the defined data structures. `TriosProcessXDR` executes the UNIX tool "rpcgen" the remote procedure call protocol compiler to generate the source and header files.

7.3.2 Implementing the client stubs

The client stubs provide the interface between the client application and the remote service. The stubs do nothing more than initialize the RPC arguments, and call the `nssi_call_rpc`

```
int xfer_write_rdma(
  const nssi_service *svc,
  const data_array_t *arr,
  nssi_request *req)
{
  xfer_write_rdma_args args;
  int nbytes;

  /* the only arg is size of array */
  args.len = arr->data_array_t_len;

  /* the RDMA buffer */
  const data_t *buf=array->data_array_t_val;

  /* size of the RDMA buffer */
  nbytes = args.len*sizeof(data_t);

  /* call the remote methods */
  nssi_call_rpc(svc, XFER_WRITE_RDMA_OP,
    &args, (char *)buf, nbytes,
    NULL, req);
}
```

Figure 7.4: Client stub for the `xfer_write_rdma` method of the transfer service.

method. For RDMA operations, the client also has to provide pointers to the appropriate data buffers so the RDMA operations know where to put or get the data for the tranfer operation.

Figure 7.4 shows the client stub for the `xfer_write_rdma` method. Since the `nssi_call_rpc` method is asynchronous. The client checks for completion of the operation by calling the `nssi_wait` method with the `nssi_request` as an argument.

7.3.3 Implementing the server

The server consists of some initialization code along with the server-side API stubs for any expected requests. Each server-side stub has the form described in Figure 7.5. The API includes a request identifier, a peer identifier for the caller, decoded arguments for the method, and RDMA addresses for the data and result. The RDMA addresses allow the server stub to write to or read from the memory on the client. In the case of the `xfer_write_rdma_srvr`, the stub has to pull the data from the client using the `data_addr` parameter and send a result (success or failure) back to the client using the `res_addr` parameter.

For complete details on how to create the transfer service code, refer to the online documentation or the source code in the trios/examples directory.

```
int xfer_write_rdma_srvr(
        const unsigned long request_id,
        const NNTI_peer_t *caller,
        const xfer_pull_args *args,
        const NNTI_buffer_t *data_addr,
        const NNTI_buffer_t *res_addr)
{
  const int len = args->len;
  int nbytes = len*sizeof(data_t);

  /* allocate space for the buffer */
  data_t *buf = (data_t *)malloc(nbytes);

  /* fetch the data from the client */
  nssi_get_data(caller,buf,nbytes,
       data_addr);

  /* send the result to the client */
  rc = nssi_send_result(caller,request_id,
         NSSI_OK, NULL, res_addr);

  /* free buffer */
  free(buf);
}
```

Figure 7.5: Server stub for the `xfer_write_rdma` method of the transfer service.

7.3.4 Performance of the transfer service

As mentioned earlier in the text, the transfer service is also a tool for evaluating the correctness and performance of the network protocols. Here we present performance results from three different HPC platforms: the Red Storm system at Sandia [30], a Cray XT3 that uses the Seastar interconnect [26] interfaced through the Portals API [27]; RedSky, a cluster of Oracle/Sun Blade Servers on a an InfiniBand network; and the Cielo supercomputer, a Cray XE6 system that uses the new Gemini interconnect [6].

Figure 7.6 shows a comparison of using the `xfer-write-rdma` and `xfer-write-encode` methods to transfer an array of `data_t` data structures from Figure 7.3. The objective is to evaluate the overhead of the XDR encoding scheme. The `xfer_write_rdma` method has very little encoding overhead–just the cost of encoding the request. These results clearly demonstrate the value of having separate control and data channels for bulk data, and while it is possible to transfer all data through the control channel, it is clearly not an efficient way to implement a bulk data-transfer operation.

The Gemini port of Nessie, developed for the Cray XE6 Cielo system, is our latest port and still requires quite a bit of tuning to achieve reasonable performance. To demonstrate the efficiency of a well-tuned implementation, Figures 7.7(a) and 7.7(b) show `xfer-write-rdma`

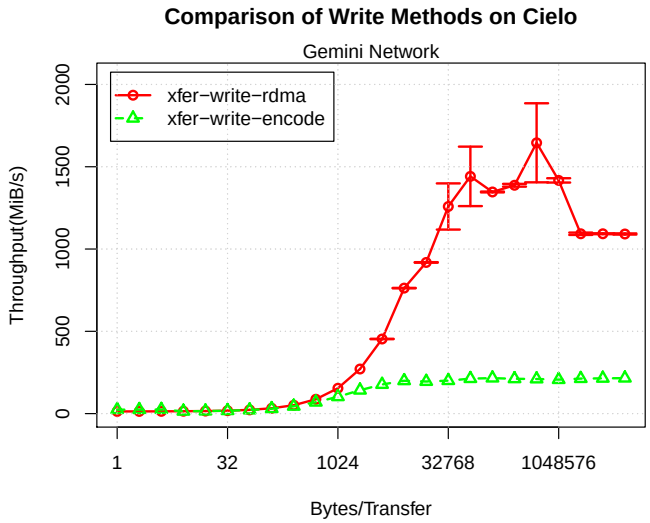


Figure 7.6: Comparison of `xfer-write-encode` and `xfer-write-rdma` on the Cray XE6 platorm using the Gemini network transport.

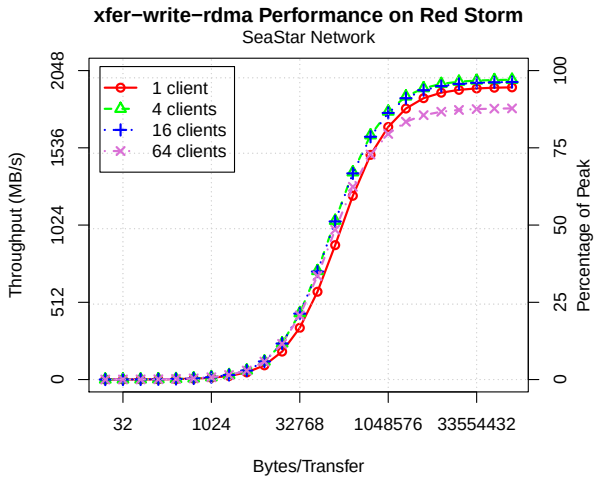
performance for the SeaStar (Portals) and InfiniBand interconnects as the number of clients per server ranges from 1–64. The Portals implementation achieves near peak performance with only slight interference effects when using 64 clients. The InfiniBand port performs at near 75% of peak for large transfers.

7.4 PnetCDF staging service

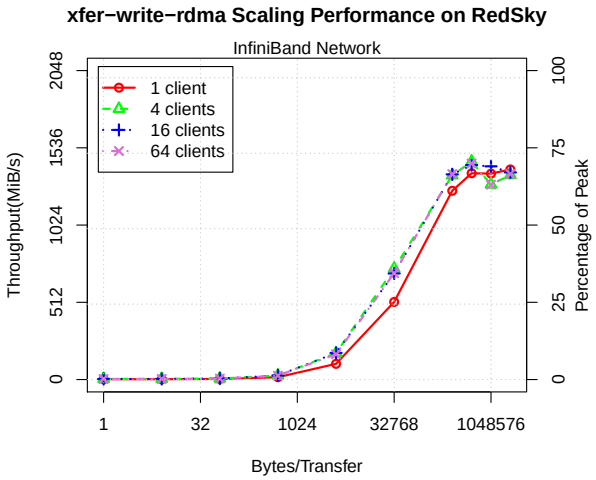
Demonstrating the performance and functionality advantages Nessie provides, the NetCDF/PnetCDF link-time replacement library offers a transparent way to use a staging area with hosted data services without disturbing the application source code and not impacting the ultimate data storage format. At a simple level, the library is inserted into the I/O path affording redirecting the NetCDF API calls into the staging area for further processing prior to calling the native NetCDF APIs for the ultimate movement of data to storage. This structure is illustrated in Figure 7.7.

At a minimum, this architecture affords reducing the number of processes participating in collective coordination operations enhancing scalability [85]. Overall, it affords changing or processing the data prior to writing to storage without impacting the application source code.

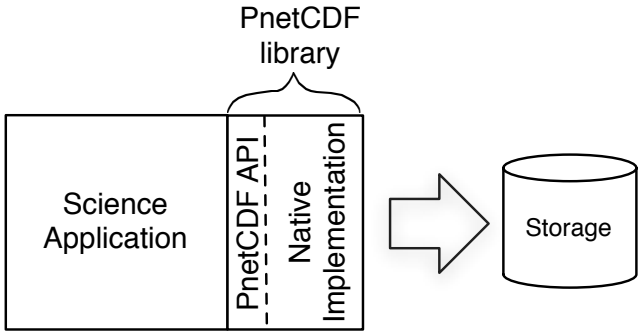
The staging functionality can be hosted over any number of processes and nodes as memory and processing capabilities demand. The initial results shown in the Parallel Data Storage Workshop 2011 paper uses a single staging node, but with 12 staing processes on that node. Those processes are capable of coordinating among themselves in order to manipulate



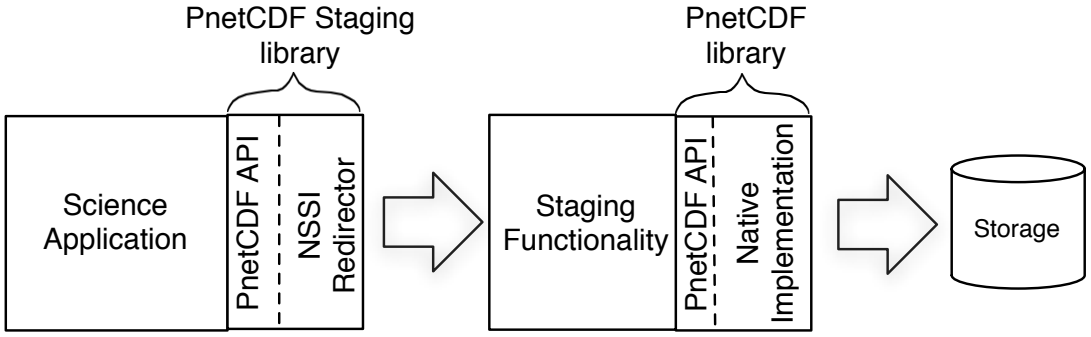
(a) RedStorm



(b) RedSky



(c) Native PnetCDF



(d) PnetCDF Staging

Figure 7.7: System Architecture

the data. Currently there are five data processing modes for the data staging area:

1. *direct* - immediately use the PnetCDF library to execute the request synchronously with the file system
2. *caching independent* - caches the write calls in the staging area until either no more buffer space is available or the file close call is made. At that time, the data is written using an independent IO mode rather than collective IO. This avoids both coordination among the staging processes and any additional data rearrangement prior to movement to storage.
3. *aggregate independent* - similar to caching independent except that the data is aggregated into larger, contiguous chunks as much as possible within all of the server processes on a single compute node prior to writing to storage. That is, to optimize the data rearrangement performance, the movement is restricted to stay within the same node avoiding any network communication overhead.
4. *caching collective* - works like the *chaching indpendent* mode, except that it attempts to use as many collective I/O calls as possible to write the data to storage. If the data payloads are not evenly distributed across all of the staging processes, a number of collective calls corresponding to the number of smallest number of data payloads in any staging process followed by a series of independent calls to complete writing the data.
5. *aggregate collective* - operates as a blend of the *caching collective* in that it tries to use as many collective I/O calls as possible to write the data, but uses the aggregation data pre-processing steps to reduce the number of data packets written.

Unlike many aschronous staging approaches, the PnetCDF staging service ultimately performs synchronously. The call to the file close function blocks until the data has been flushed to storage.

Using the staging service at run time is a 4 step process. First, the staging area is launched generating a list of contact strings. Each string contains the information necessary to reach a single staging process. The client (science application) can choose which client process communicates with which staging service process. Second, these strings are processed to generate a standard XML-based format making client processing simpler and environment variables are set exposing the contact file filename in a standard way. Third, the science application is launched. Finally, as part of the PnetCDF initialization, the re-implementation of the PnetCDF reads the environment variable to determine the connection information file filename, reads the file, and broadcasts the connection information to all of the client processes. These processes select one of the server processes with which to communicate based on a load-balancing calculation.

The current functionality of increasing the performance of PnetCDF collective operations is just a first step. The current architecture offer the ability to have any parallel or serial

processing engine installed in the staging area application. The scaling of this application is independent of scaling of the science application. This decoupling of concerns simplifies programming of the integrated workflow of the simulation generating raw data and the analysis routines distilling the data into the desired processed form.

Ultimately, this technique of reimplementing the API for accessing staging offers a way to enhance the functionality of online scientific data processing without requiring changing the application source code. As in the case of the PnetCDF service, these analysis or other data processing routines can be inserted as part of the I/O path with the data ultimately hitting the storage in the format prescribed by the original API.

7.4.1 PnetCDF staging service performance analysis

Evaluating the performance of the service is performed in two parts. First, an examination of IOR [70] performance is evaluated followed by an I/O kernel for Sandia's S3D [35] combustion code.

7.4.1.1 IOR Performance

To evaluate the potential of PnetCDF staging, we measured the performance of our PnetCDF staging library when used by the IOR benchmark code. IOR (Interleave-or-random) [70] is a highly configurable benchmark code from LLNL that IOR is often used to find the peak measurable throughput of an I/O system. In this case, IOR provides a tool for evaluating the impact of offloading the management overhead of the netCDF and PnetCDF libraries onto staging nodes.

Figure 7.8 shows measured throughput of three different experiments: writing a single shared file using PnetCDF directly, writing a file-per-process using standard netCDF3, and writing a single shared file using the PnetCDF staging service. In every experiment, each client wrote 25% of its compute-node memory, so we allocated one staging node for each four compute nodes to provide enough memory in the staging area to handle an I/O "dump".

Results on Thunderbird show terrible performance for both the PnetCDF and netCDF file-per-process case when using the library directly. The PnetCDF experiments maxed out at 217 MiB/s and reached the peak almost immediately. The PnetCDF shared file did not do much better, achieving a peak throughput of 3.3 GiB/s after only 10s of clients. The PnetCDF staging service, however, achieved an "effective" I/O rate of 28 GiB/s to a single shared file. This is the rate observed by the application as the time to transfer the data from the application to the set of staging nodes. The staging nodes still have to write the data to storage, but for applications with "bursty" IO patterns, staging is very effective.

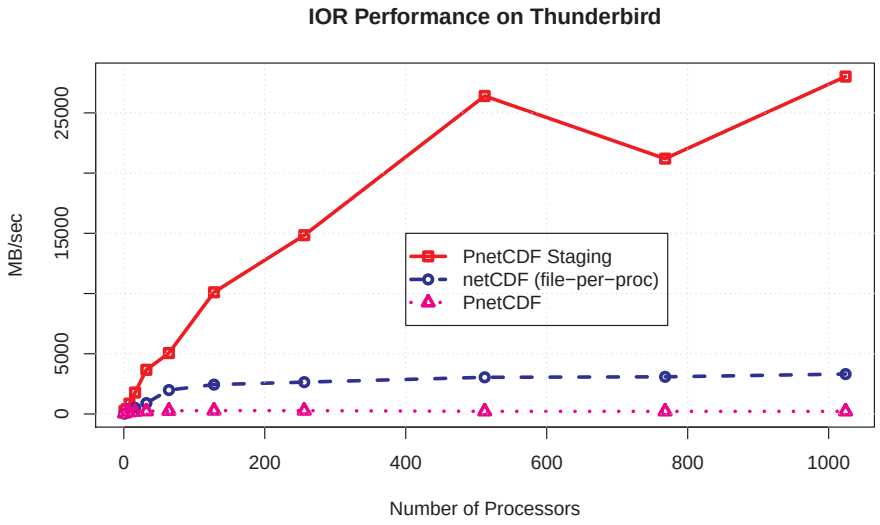


Figure 7.8: Measured throughput of the IOR benchmark code on Thunderbird

7.4.1.2 S3D Performance

In the final set of experiments, we evaluate the performance of the PnetCDF staging library when used by Sandia's S3D simulation code [35], a flow solver for performing direct numerical simulation of turbulent combustion.

All experiments take place on the JaguarPF system at Oak Ridge National Laboratories. JaguarPF is a Cray XT5 with 18,688 compute nodes in addition to dedicated login and service nodes. Each compute node has dual hex-core AMD Opteron 2435 processors running at 2.6GHz, 16 GB RAM, and a SeaStar 2+ router. The PnetCDF version is 1.2.0 and uses the default Cray MPT MPI implementation. The file system, called Spider, is a Lustre 1.6 system with 672 object storage targets and a total of 5 PB of disk space. It has a demonstrated maximum bandwidth of 120 GB/sec. We configured the file system to stripe using the default 1 MB stripe size across 160 storage targets for each file for all tests.

In our test configuration, we use ten, 32 cubes (32×32×32) of doubles per process across a shared, global space. The data size is 2.7 GB per 1024 processes. We write the whole dataset at a single time and measure the time from the file open through the file close. We use five tests for each process count and show the best performance for each size. In this set of tests, we use a single node for staging. To maximize the parallel bandwidth to the storage system, one staging process per core is used (12 staging processes). Additional testing with a single staging process did not show significant performance differences. The client processes are split as evenly as possible across the staging processes in an attempt to balance the load.

Figure 7.9 shows the results of S3D using the PnetCDF library directly with the four different configurations of our PnetCDF staging library described in Section 7.4. In all cases measured, the base PnetCDF performance was no better than any other technique

at any process count. The biggest difference between the base performance and one of the techniques is for 1024 processes using the caching independent mode at only 32% as much time spent performing IO. The direct technique starts at about 50% less time spent and steadily increases until it reached parity at 7168 processes. Both cache independent and aggregate independent advantages steadily decrease as the scale increases, but still have a 20% advantage at 8192 processes.

In spite of there only being 12 staging processes with a total gross of 16 GB of RAM, the performance improvement is still significant. The lesser performance of the direct writing method is not very surprising. By making the broadly distributed calls synchronous through just 12 processes, the calling application must wait for the staging area to complete the write call before the next process will attempt to write. The advantage shown for smaller scales shows the disadvantage of the communication to rearrange the data compared to just writing the data. Ultimately, the advantage is overwhelmed by the number of requests being performed synchronously through the limited resources.

The advantage of the caching and aggregating over the direct and base techniques shows that by queueing all of the requests and letting them execute without interruption and delay of returning back to the compute area offers a non-trivial advantage over the synchronous approach. Somewhat surprisingly, the aggregation approach that reduces the number of IO calls via data aggregation did not yield performance advantages over just caching the requests. This suggests that for the configuration of the Spider file system at least, reducing the number of concurrent clients to the IO system is the advantageous approach. Additional efforts to reduce the number of IO calls do not yield benefits.

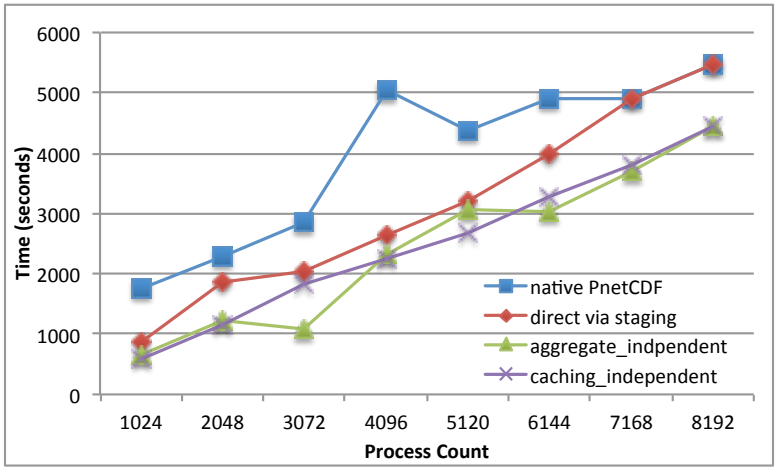


Figure 7.9: Writing performance on JaguarPF one staging node (12 processes)

7.5 Availability of Data Services Software

All of the non export-controlled software developed by the data-services project is available as part of the Trilinos software repository [65] under the Trilinos I/O Support (Trios) capability

area. Trios is a newly formed capability area to facilitate the co-design and development of open-source I/O tools and libraries developed for the ASC/CSSE program. Trilinos provides a fantastic development and testing environment and also provides a great tool for rapid deployment of R&D codes into production applications.

7.6 Summary and Future Work

This chapter describes R&D at Sandia National Laboratories to use integrated data services to reduce the effective I/O cost for scientific applications. The Nessie framework provides an RPC abstraction that allows for the rapid development of experimental data services to explore new new I/O techniques and programming models. We are using this framework to explore improved access interfaces to staging areas, standard approaches integrate 'in flight' data processing between the science application and storage, and we are exploring how to leverage the data services model with advanced accelerators such as GPGPUs and FPGAs.

While we expect data services to be common on next-generation systems, it is important to note that data-services also provide practical solutions for applications in the current generation of HPC systems. The results presented in this paper demonstrate the advantage of data-services on systems with three different interconnects: the Seastar for Cray XT systems, the Gemini interconnect for Cray XE systems, and InfiniBand for large capacity clusters. In addition, we plan to modify the Exodus interface to use our PnetCDF staging service, providing an immediate impact to the Sierra applications, as well as any other applications using Sandia's Exodus I/O library.

For next-generation systems, there are many improvements that could better support data services in a production environment. For example, runtime support for dynamic allocation and reconfiguration of nodes would allow for on-demand data-processing services that effectively utilize system resources. Control over placement algorithms within an allocation would also allow data-services to better utilize networks to avoid contention with application communication and file-system communication. The projected promulgation of NVRAM could also improve use of data-staging services, especially services primarily used to manage large bursts of data. In short, nearly every proposed Exascale architecture would benefit from, and perhaps require, a data-services architecture. Research prospects in this area will continue to grow throughout the next decade.

Chapter 8

In situ and In transit Visualization and Analysis

Abstract: Traditional analysis workflow prescribes storing simulation results to disk and later retrieving them to perform analysis and visualization for final results. This workflow becomes increasingly difficult due to data movement at petascale and beyond. In situ [49] and In transit [98] are both techniques that by-pass the traditional workflow and manage the data movement problem. In situ refers to linking the visualization tools directly into the simulation code to run in the same memory. In transit uses data staging, as described in the previous chapter, and executes the analysis and visualization in a separate staging job concurrently with the main computation job. In the following chapter, we describe experiments and results using these coupling techniques.

8.1 Background

Scientific simulation on parallel supercomputers is traditionally performed in four sequential steps: meshing, partitioning, solver, and visualization. Not all of these components are actually run on the supercomputer. In particular, the meshing and visualization usually happen on smaller but more interactive computing resources. However, the previous decade has seen a growth in both the need and ability to perform scalable parallel analysis, and this gives motivation for coupling the solver and visualization.

The concept of running a visualization while the simulation is running is not new. It is mentioned in the 1987 National Science Foundation Visualization in Scientific Computing workshop report [90], which is often attributed to launching the field of scientific visualization. It is only recently, however, as we move into petascale that the idea has started gaining significant traction [2, 71]. Recent studies show that cost of specialized hardward for visualization at petascale is prohibitive [36] and that time spent writing data to and reading data from disk storage is beginning to dominate the time spent in both simulation and visualization [111, 112, 122].

There are a variety of analyses and visual representations available and appropriateness of each depends highly on the type of problem being analyzed. Thus, it is important that

a framework for coupling visualization into a simulation is both flexible and expandable. In the following, we show methods using the ParaView Coprocessing Library directly coupled with the simulation and indirectly coupled using Nessie.

8.1.1 In situ Implementation

The ParaView Coprocessing Library is a C++ library with an externally facing API to C, FORTRAN and Python that allows in situ access to a set of the algorithms available in the ParaView visualization application. It is built on top of the Visualization Toolkit which makes available a large number of algorithms including I/O, rendering, and processing algorithms such as isosurface extraction, slicing, and fragment detection. Although it is possible to construct pipelines entirely in C++, the ParaView control structure allows pipelines configured through Python scripts. Results shown here were all made using the Python scripts to construct the pipelines.

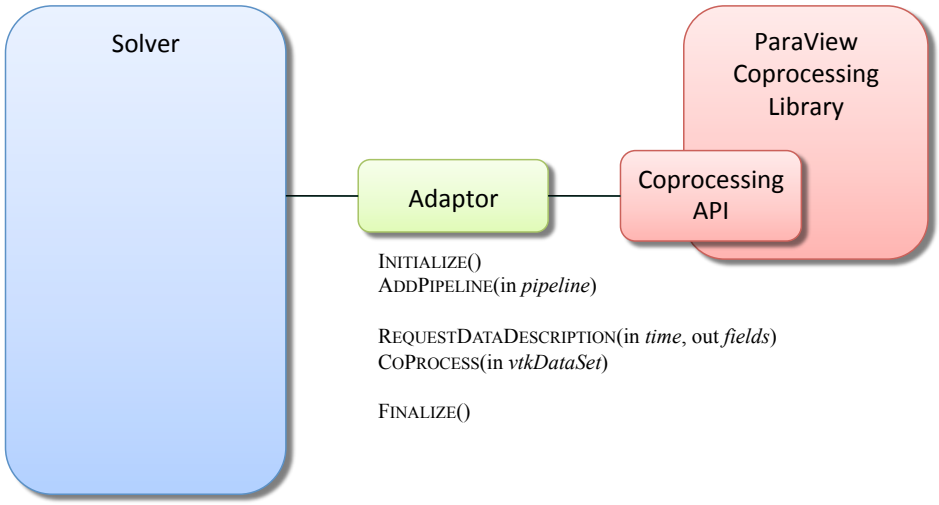


Figure 8.1: The ParaView Coprocessing Library generalizes to many possible simulations, by means of adaptors. These are small pieces of code that translate data structures in the simulation's memory into data structures the library can process natively. In many cases, this can be handled via a shallow copy of array pointers, but in cases where that is not possible, it must perform a deep copy of the data.

Since the coprocessing library will extend a variety of existing simulation codes, we cannot expect its API to easily and efficiently process internal structures in all possible codes. The solution used is to rely on adaptors, Figure 8.1, which are small pieces of code written for each new linked simulation, to translate data structures between the simulation's code and the coprocessing library's VTK-based architecture. An adaptor is responsible for two categories of input information: simulation data (simulation time, time step, grid, and fields) and temporal data, i.e., when the visualization and coprocessing pipeline should execute. To do this effectively, the coprocessing library requires that the simulation code invoke the adaptor at regular intervals.

The simulation we used for this work was CTH [72]. It is an Eulerian shock physics code that uses an adaptive mesh refinement (AMR) data model. We developed an adaptor to convert the CTH AMR model to ParaView's AMR model with minimal copying. We also modified CTH to make calls into the coprocessing library API.

8.1.2 In transit Implementation

The NEtwork Scalable Service Interface (Nessie) described in chapter 7 is the framework used to develop the in transit analysis capabilities used here. In transit visualization occurs on the data service nodes instead of the computation nodes as it is done in situ. We link the ParaView Coprocessing library against the Nessie server and process as data is transferred in from the computational client. Nessie transfers from client to server all the API calls that were modified in the original CTH. These calls are then passed to the in situ adaptor linked in on the Nessie Server, so that the adaptor implementation can stay the same whether linked directly or linked through the Nessie implementation. The only change apparent to the in situ processing is the number of cores available.

8.2 Performance on Cielo

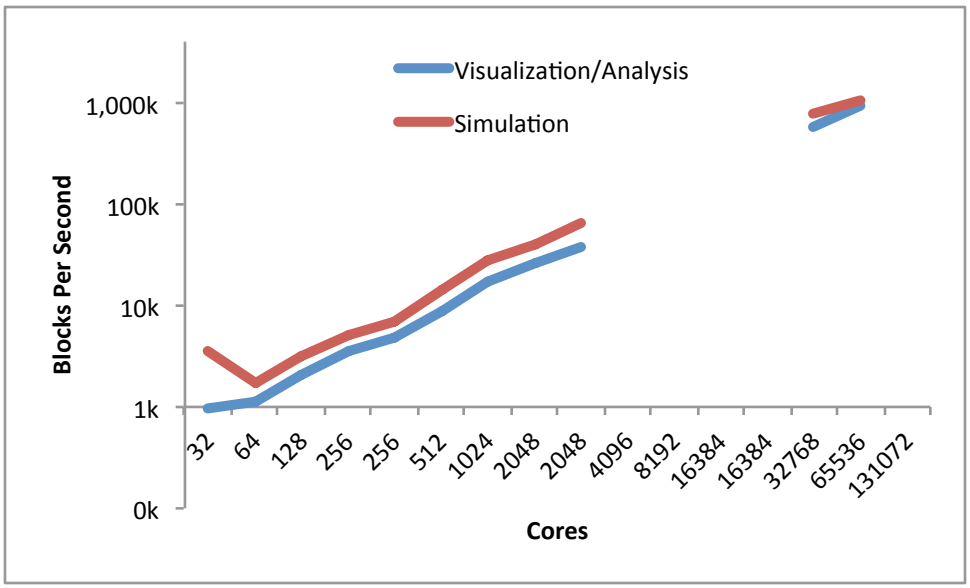


Figure 8.2: Block processing rate of simulation and visualization at growing scales on Cielo. Note that 256 and 16384 core counts are repeated twice. This is an overlap where the block count was increased in order to continue scaling effectively. Gaps are locations where memory errors are preventing execution.

To begin measuring performance, we tested the direct coupling of the ParaView library using a canonical pipeline that performs an isocontour using the well known marching cubes

algorithm. This pipeline performs very little communication itself and exposes any communication overhead associated with framework itself, especially those used in setting up this pipeline. The results of the scaling are shown in Figure 8.2. This plot shows an increase in the block processing rate as we increase the number of processors. This rate is similar for both the simulation part and the ParaView part. It is worth noting that these costs are for computing both parts every simulation step. It is usually not necessary to visualize every step of the simulation and the ParaView processing could be run less frequently to decrease overall runtime.

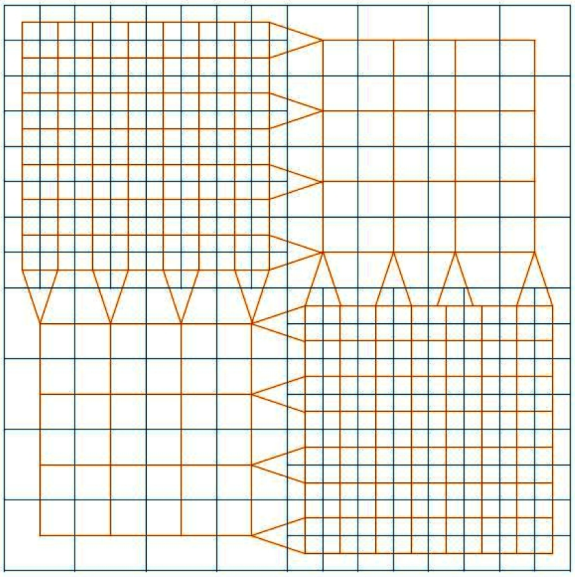


Figure 8.3: Generating a contour on an AMR mesh requires special processing at the boundary between two different resolutions. Degenerate triangles must be created to close the seams between the otherwise irregularly shaped boundary, as shown in this figure.

While the framework itself scales to the entirety Cielo, there may be pipeline analysis algorithms which run fine for post processing, but do not scale well enough to run efficiently at 16,000 cores and above. One particular algorithm is the AMR contouring algorithm. Unlike the isocontouring algorithm used previously, the AMR algorithm does not assume there is a regular distribution on the data. It must generate special triangles at regions with different spatial distributions (i.e., different resolutions as a result of the AMR process), as shown in Figure 8.3. Therefore it must communicate between these neighborhoods. Currently this algorithm works by redistributing neighborhood information via all-to-all communication at the beginning of every pipeline execution. Its scaling performance is shown in Figure 8.4.

Although the AMR algorithm is dominated by the all-to-all neighbor communication, the remainder of the computation scales effectively. We are currently investigating a way to transfer information about the communicating neighborhoods used by the simulation. This should allow us to shortcut the all-to-all communication by taking advantage of the same neighborhoods the simulation is already tracking. This information is available in memory while the simulation is running, but would be too expensive to spend time writing out to a

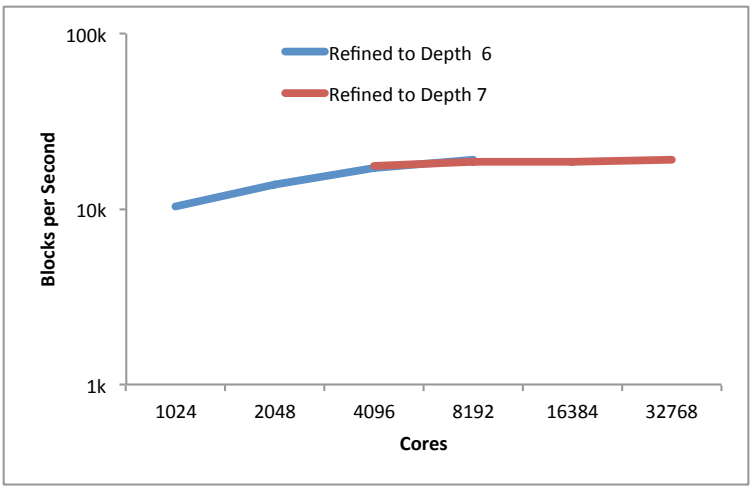


Figure 8.4: Block processing rate of AMR contouring at growing scales on Cielo. The two different lines represent different AMR sizes. Large depth means more blocks are in the simulation.

file and not necessary for post-processing anyway.

In general as we broaden use of the in situ capability using more of the algorithms available in VTK and ParaView, we expect to find a similar pattern of scaling performance. Some of these algorithms will require very little communication and will scale very well. Some may be rewritten to take advantage of information available in simulation memory which is not normally written out to disk. Finally, some of these may never scale, because they require too many communications not already performed by the simulation. Streamline visualization may be an example of such an algorithm.

In this last case where the algorithm doesn't scale, we would use in transit visualization. This is also a practical solution for algorithms which could be rewritten, but have not yet been, as is the case AMR Contour algorithm above. Figure 8.5 shows the timing, running on Redsky, of the AMR Contour relative to the simulation itself. It runs fast enough to keep up with the simulation on up to 128 cores, but beyond that starts to run increasingly slower and would become a bottleneck to the running simulation. It is beyond this point that we switch to in transit operation, allowing the simulation to continue to operate at full speed, while offloading the processing to the service nodes and allow the AMR contour to run as quickly as it can without impacting the simulation.

8.3 Conclusion

Using these methods of coupling the analysis and visualization in with the simulation while it is running and by-passing the need to write everything out to disk, we are able to optimize the standard HPC workflow to shrink the time from initial meshing to final results. The ParaView Coprocessing Library scales effectively to run at full scale on Cielo, and in the

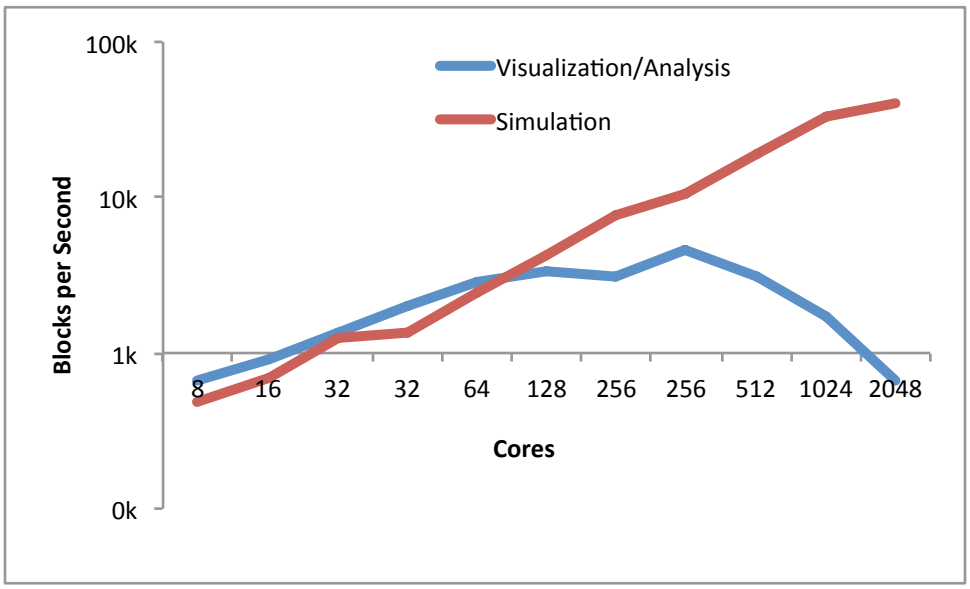


Figure 8.5: Running on Redsky, the block processing rate of AMR pipeline at growing scales compared with the simulation processing rate.

case of pipeline algorithms which do not or are unable to scale to that size, we can fall back to a lower scale by using the in transit coupling method. In both cases, we reap the savings of using memory and network instead of disk as the medium of interchange. A detailed comparison of the in-situ and in-transit approaches is the focus of an FY13 ASC Level II milestone titled, "Data Co-processing for Extreme Scale Analysis." We will have more thorough examination of of the two approaches, as well as performance analysis for a number of different application use cases in that report.

Chapter 9

Dynamic Shared Libraries on Cielo

Abstract: Popularity of dynamically linked executables continues to grow within the scientific computing community. The latest copy of a shared library can be accessed when the application is run. By contrast, statically linked applications use the version of the library that was available when the application was built. Secondly, a shared library, as the name asserts, is shared across all application images running on a node. This can relieve memory pressure, since the same text is accessed by all. Lastly, shared libraries are often memory mapped from disk to further minimize the memory usage. Only the portions of the library that are being accessed need to reside in memory. Although attractive from a flexibility and resource reduction perspective, dynamically linked executables continue to suffer performance issues on massively parallel processing (MPP) systems. This is particularly true for systems such as Cielo, where each node does not have a disk drive on which to store the libraries. Instead, a custom file system configuration was installed on Cielo. This chapter describes the results of two experiments that analyze its performance. Results showed it superior to the other available file systems. However, when the Charon application was run in static and then dynamic link configurations, run time was increased by 34% on 8192 cores. The chapter concludes with possible areas for further investigation, including a possible Linux kernel bug.

9.1 Motivation for the Implementation

Historically, applications running in capability mode are built (i.e. linked) statically. Increasingly, application code groups are asking for the flexibility provided by dynamic shared objects. Dynamically linked executables have been supported on desktop system for decades. More recently, they have been successfully used on cluster computers of considerable scale (a few thousand cores). When specifying the Cielo system, the procurement team elected to require support for dynamically linked executables on compute nodes. However, full scale testing was done with static binaries.

Following the delivery of the system, the Cielo bring-up team began assessing the Cray (and other) options for providing support for dynamic libraries. We considered both hardware and software alternatives to achieve as much scalability as possible when using a dynamically linked executable. One important aspect was that in addition to system shared

objects, the user community wished to provide their own shared libraries and other dynamic objects, such as python modules.

As implied by the adjective, *dynamic*, requests for shared objects are asynchronous. Since they are unpredictable, they can introduce noise to a bulk synchronous application. Consistent quality of service for file access cannot be assured. Secondly, since all of Cielo's jobs are SPMD (single program, multiple data), the I/O node serving the shared object file can be flooded trying to satisfy a request from every single process in the job. With over 100,000 cores, that could be a very large number of requests.

We elected to remain with the Linux software solution for linking and loading dynamic shared objects. This implementation is familiar to the Cielo user community and is supported by all third party libraries. It is not trivial and a custom re-implementation was a larger effort/committment than we could make. On the down side, the Linux implementation is not tunable for parallel applications at all. We focused primarily on the hardware implementation to maximize parallel access to the shared objects.

9.2 Configuration and Implementation of the Solution

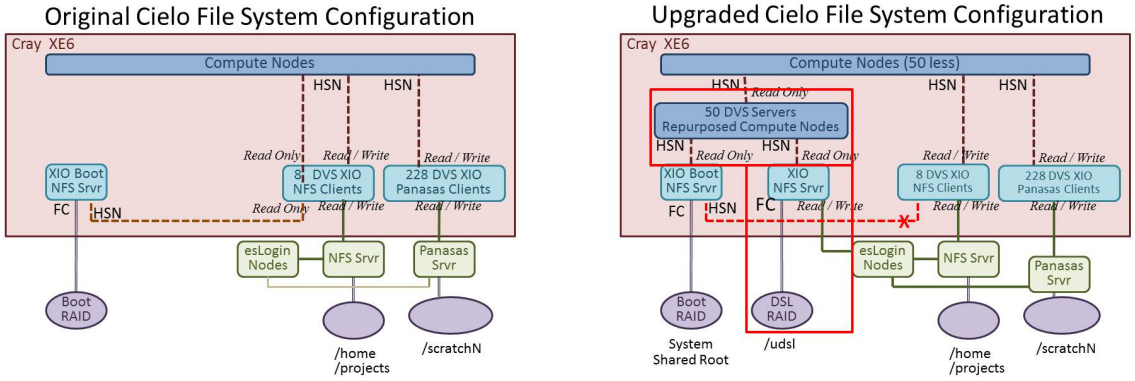
Cray proposed Cielo use Cray's DVS (Data Virtualization Service) to distribute the system-provided shared libraries to compute nodes. DVS makes use of the read-only nature of shared libraries to enable parallel access from compute nodes. Multiple DVS servers can be configured to serve some unique subset of compute nodes. The first compute node to ask for a shared library will ask its pre-defined DVS server. This DVS server will then ask for the data from the single NFS server hosting the disk with the library/file. Any subsequent requests to that pre-defined DVS server can hopefully be satisfied locally in its cache and there won't be a need to re-request from the NFS server. Other DVS servers, serving the remaining compute nodes, will need to go back to the NFS server. In this case, the NFS server should have the file contents cached, eliminating the need to do the actual disk I/O. While caches are constrained by memory size, this design can significantly reduce the number of accesses to the physical file. It distributes the requests between the DVS servers and the one NFS server.

There were several options for where to place user provided shared libraries. The options and their pros and cons are discussed fully in [73]. While no solution was ideal, we elected to mimic the solution used by Cray for system shared libraries that is described above. The file system serving the user-built shared libraries to the compute nodes is mounted read-only. It is served for read and write access from the *login* nodes where user compilations are performed.

Figure 9.1 shows the before and after file system configuration. Changes are outlined in red. To achieve scalability, 50 nodes were repurposed to serve the almost 9000 compute nodes. This solution means that only 50 nodes are actually requesting the file data from the nodes serving the system and user shared objects. The hardware spreads out the I/O

load to a hierarchy of caches. There is a file cache on the compute node for the processess running the same binary on the other cores of the node. There are caches on the 50 DVS servers, each of which is supporting 1/50 of the compute nodes. Lastly, there is a cache on the NFS server where the file is actually located.

Figure 9.1: Fifty DVS servers were added to serve both system and user dynamic shared libraries to compute nodes.



9.3 Evaluating the Impact of the New Configuration

Two experiments were run to assess the impact of the implemented configuration. The first experiment used the Pynamic benchmark [79] to determine if the new user shared library (/udsl) file system would meet the needs of the user community. The second test used the Charon application to collect data points on the impact of a dynamically linked executable versus a static executable.

9.3.1 Benchmark Results

The Pynamic benchmark was developed to simulate the dynamic loading activity of a particularly demanding ASC application. The benchmark does no computation. It just continuously loads python modules. At the end of the execution, it calls a few MPI collectives to gather statistics of the run and load times. We used the default Pynamic configuration for the tests. Our goal was to understand if a dedicated, read-only, file system was necessary to achieve some level of scalability. We did not expect full-system capability-class jobs to run with dynamic libraries, but we were hopeful dynamically built executables could run at a reasonable scale without performance loss. Figure 9.2 shows a summary of the results. A detailed discussion of the setup, configuration and results is provided in [73]. While users might have preferred to keep their shared objects with their other data in their personal

(home) directory or on the parallel file system, the performance does not scale. The custom file system was validated as a necessary implementation for this ASC application.

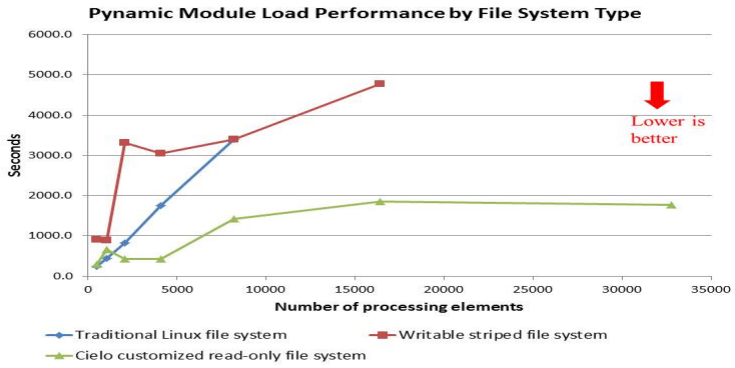


Figure 9.2: Pynamic timing results gave acceptable results only with the special purpose file system.

9.3.2 Application Results

The results in subsection 9.3.1 were encouraging. The implemented configuration was optimal as compared to the two other possible locations for user shared libraries. Since the system shared libraries were configured to use the same 50 DVS nodes as a staging area, we anticipated similar scalability. To validate this assertion, a second experiment was run on Cielo using V4.6 of the Charon application. A batch job was submitted with two executions of Charon. By using a single job we could be assured that the two executions would use the identical compute nodes. This eliminated any variability that could be introduced by different layouts in the mesh. The first execution used a statically linked binary. The second used a dynamically linked binary. The 2D test problems were identical for both executations.

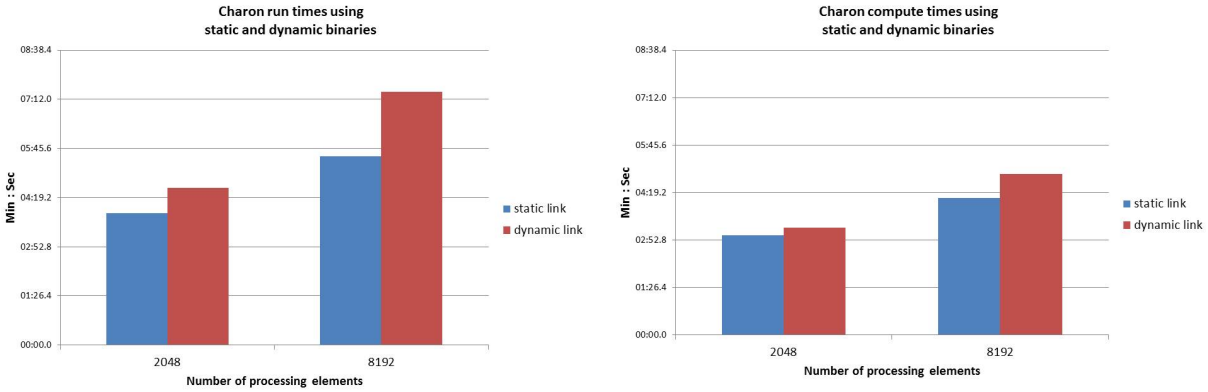
There were no user supplied dynamic libraries. Only the system libraries were dynamic. Based on the loadmap produced during the dynamic build, there were 23 shared object files loaded and 7 functions in the code that caused shared objects to be loaded. They are __pgCC_throw, _mp_penter, __mth_i_cexp, __mth_i_csqrt, __mth_i_cdsqrt, ftn_str_copy, and __fmth_i_exp_gh.

We ran two sizes: 2048 and 8192 cores (processing elements). The test results were run on a quiet system to eliminate any question of variability due to network or I/O activity from other jobs. The results are shown in Figure 9.3a. They report a 19% and 34% increase in run times for the 2048 and 8192 core runs, respectively. Some of this increase may be due to an increased load/initialization phase. Longer runs might amoratize any additional, perhaps fixed, start-up cost. However, a disturbing additional data point is the Charon-calculated compute time. This time, which is not believed to include any application I/O,

also increased. Figure 9.3b illustrates this increase. It is a 7% increase in computation time for the 2048 core run and an 18% increase for the 8192 core run.

This data is very disconcerting. We *speculate* that there is an additional delay caused by the indirect address needed for every function call. We also do not know how frequently the shared objects are being called during the computation phase of Charon. Additional study is required.

Figure 9.3: Based on a small number of data points, dynamic shared libraries negatively impact application run time.



9.4 Conclusion

The significant increase in Charon run time when using a dynamically linked executable was disappointing. Based on the results reported in subsection 9.3.1 for the Pynamic benchmark, we can assume the run time would have been even worse with a different file system configuration. As a result of this testing, we may have identified a potential bug in the Cray software. We have determined that not all cores on the same node are served fairly. Some cores wait a significant amount of time before they have access to the shared object. This unfairness would introduce noise in the application. If this Bug 771251 is resolved with a fix, we can hope performance will improve. Otherwise, a more dramatic change to the loading of shared libraries may likely be required for future systems.

Chapter 10

Process Replication for Reliability

Abstract: As high-end computing machines continue to grow in size, issues such as fault tolerance and reliability limit application scalability. Current techniques to ensure progress across faults, like checkpoint-restart, are increasingly problematic at these scales due to excessive overheads predicted to more than double an application's time to solution. Replicated computing techniques, particularly state machine replication, long used in distributed and mission critical systems, have been suggested as an alternative to checkpoint-restart. In this chapter, we evaluate the viability of using state machine replication as the primary fault tolerance mechanism for upcoming exascale systems. We use a combination of modeling, empirical analysis, and simulation to study the costs and benefits of this approach in comparison to checkpoint/restart on a wide range of system parameters. These results, which cover different failure distributions, hardware mean time between failures, and I/O bandwidths, show that state machine replication is a potentially useful technique for meeting the fault tolerance demands of HPC applications on future exascale platforms.

10.1 Introduction

Process replication, generally referred to as *state machine replication* [126], is a well-known technique for tolerating faults in systems that target high-availability. In this approach, a process's state is replicated such that if the process fails, its replica is available or can be generated to assume the original process's role without disturbing the other application processes. Process replication can be costly in terms of space if replicas have dedicated resources or time if replicas are co-located with other primary processes. However, process replication can dramatically increase an application's mean time to interrupt (MTTI). Additionally, variants of this technique can detect or correct faults that do not crash a process but instead cause it to yield incorrect results [32].

Primarily due to its high costs, process replication has been examined only in a limited manner for high performance computing (HPC) systems [132]. Instead, HPC applications have relied primarily on rollback recovery techniques [46], particularly coordinated checkpoint/restart, where application state is periodically written to stable storage (checkpointed), and when failures occur, this state is used to recover the application to a previously known-good state. However, future exascale systems are expected to present a much more

challenging fault tolerance environment to applications than current systems [17]. Additionally, recent studies conclude that for these systems, high failure rates coupled with high checkpoint/restart overheads will render current rollback-recovery approaches infeasible. For example, several independent studies have concluded that potential exascale systems could spend more than 50% of their time reading and writing checkpoints [47, 102, 128].

In this chapter, we examine the viability of the process replication paradigm as a *primary* exascale fault tolerance mechanism, with checkpoint/restart providing *secondary* fault tolerance when necessary. Our goal is to understand the advantages and limitations of this approach for extreme scale computing systems. We focus on exascale MPI applications: redundant copies of MPI processes provide failover capabilities, which allow these applications to *transparently* run through most errors without the need for rollback. Checkpoint/restart augments our process replication scheme in cases where process replication is insufficient, for example, if all replicas of a process crash simultaneously or become inconsistent due to faults corrupting the machine state.

To summarize, we present a study of redundant computing for exascale applications to address the scalability concerns of disk-oriented coordinated checkpoint/restart techniques and the inability of checkpoint/restart methods to tolerate undetected hardware errors and non-crash failures (Section 10.2). Our results show that redundant computing should be considered as a viable approach for exascale systems because:

- Even at system scales less than those projected for exascale systems, our *model-based analysis* shows that process replication's hardware overheads are less than those of traditional checkpoint/restart (Section 10.5);

- Based on a full implementation MPI process replication that has been evaluated on more than 4000 nodes of a Cray XT-3/4 system, our *empirical analysis* shows that process replication overheads are minimal for real HPC applications (Section 10.6); and,

- Additional *simulation-based analysis* that includes both software and hardware overheads shows that process replication is a viable alternative to traditional checkpoint/restart on systems with more than 20,000 sockets (Section 10.7), depending on system checkpoint I/O bandwidth and per-socket mean time between failures (MTBF).

10.2 Background

10.2.1 Disk-based Coordinated Checkpoint/Restart

10.2.1.1 Current State of Practice

Disk-based coordinated checkpoint/restart has been the dominant fault tolerance mechanism in high performance computing systems for at least the last 30 years. In current large distributed memory HPC systems, this approach generally works as follows:

1. Applications periodically quiesce all activity at a global synchronization point, for example a barrier;
2. After synchronization, all nodes send some fraction of application and system state, generally comprising most of system memory, over the network to dedicated I/O nodes;
3. These I/O nodes store received checkpoint information data to stable storage, currently hard disk-based storage;
4. In the event of application crash, the stored checkpoint can be used to restart the application at a prior known-good state.

The dominance of coordinated checkpoint-restart as the *primary* HPC fault tolerance technique rests on a number of key assumptions that have thus far remained true:

1. Application state can be saved and restored much more quickly than a system's mean time to interrupt (MTTI);
2. The hardware and upkeep (e.g. power) costs of supporting frequent checkpointing are a modest portion (currently perhaps 10-20%) of the system's overall cost; and
3. System faults that do not crash (fail-stop) the system are very rare.

Heroic work in both I/O systems and hardware error detection has largely kept these assumptions valid through the present day, allowing large parallel applications to scale to over a petaflop of sustained performance in the face of occasional fault-induced system crashes.

10.2.1.2 Scaling of Coordinated Disk-based Checkpoint/Restart

There are a number of excellent studies investigating the efficiency of disk-based checkpoint/restart for large scale systems, including past petascale systems [47, 107, 46] and upcoming exascale systems [128, 7, 17]. These projections invalidate essentially *all* of the

assumptions on which traditional checkpoint/restart depend. For example, these studies suggest exascale MTTIs ranging from 3-37 minutes, checkpoint times for *low memory* systems taking up to five hours, and significant non-crash system failures, for example undetected DRAM errors.

We begin by assuming a 15-minute checkpoint time, a modest improvement over the approximate 20 minute checkpoint time that supercomputers both a decade ago (ASCI Red [89]) and today (BlueGene and Jaguar [31]) achieve. We also assume a one hour system MTTI, again a generous assumption given recent studies. Daly's model [39] estimates that such a system should checkpoint once every 27 minutes and would achieve only 44% system utilization. Scaling the I/O system to achieve a utilization greater than 80% would require checkpoint times of approximately one minute. Assuming that the I/O system supporting that 15 minute checkpoint took only 10% of the system's original budget and I/O throughput scaled up perfectly, a simple Amdahl's law calculation shows that an I/O supporting such checkpoint speeds would comprise 63% of the total cost of the system!

Checkpoint-restart is also problematic when dealing with non-crash failures such as so-called "soft errors". In particular, checkpoint-restart *preserves* the impact of failures that corrupt application state. Addressing this would require application developers to either restart an application from scratch or analyze the contents of their checkpoints looking for one prior to when the fault that corrupted application state occurred.

10.2.2 State Machine Replication

Redundant computation, process replication, and state machine replication have long histories and have been used extensively in both distributed [58, 34] and mission critical systems [91, 12, 109, 126] as a technique to improve fault tolerance. State machine replication, the focus of this paper, maintains one or more replicas of each node and assumes every node computes deterministically in response to a given external input, for example a message being received. It then uses an ordering protocol to guarantee all replicas see the same inputs in the same order, and additional communication to detect and recover from failures.

State machine replication offers a different set of trade-offs compared to rollback recovery techniques such as checkpoint/restart. In particular, it completely masks a large percentage of system faults, preventing them from causing application failures *without the need for rollback*. Some forms of state machine replication can also be used to detect and recover from a wider range of failures than checkpoint/restart, potentially including Byzantine failures [32]. Unlike checkpoint/restart, however, state machine replication is not sufficient by itself to recover from all node crash failures; faults that crash all of a node's replicas will cause a computation to fail.

This approach has previously been dismissed in HPC as being too expensive for the meager benefits that are seen at present machine scale. For the reasons described above in Section 10.2.1, however, several authors have recently suggested using this technique in

HPC systems [128, 140, 48]. In the remainder of this paper, we examine the suitability of a specific type of state machine replication in HPC systems.

10.3 Replication for Message Passing HPC Applications

10.3.1 Overview

State machine replication is conceptually straightforward in message passing-oriented HPC applications. In this approach, each replica is created on independent hardware for every processor rank in the original application of which failure cannot easily be tolerated. Note that we do not require *all* ranks to be replicated—in master/slave-style computations where the master can recover from the loss of slaves, only the master might be replicated.

The replication system then guarantees that every replica receives the same messages in the same order and that a copy of each message from one rank is sent to each replica in the destination rank. In addition, the replication system must detect replica failures, repair failed nodes when possible, and restart failed nodes from active replicas. The replication system may also periodically check that replicated ranks have the same state.

Checkpoint/restart recovery is still required in this approach for recovery from faults that fail *all* replicas of a particular process rank. It is also used to recover from situations where replica state becomes inconsistent, for example due to silent (undetected) failures.

10.3.2 Costs and Benefits

This approach requires significantly increased computational resources—at least double the hardware for replicated ranks. In cases where only portions of an application must be replicated, these requirements are potentially modest. For many HPC applications (e.g. traditional stencil calculations), however, this approach *doubles* the required hardware—$2N$ nodes are required to fully replicate a job that would otherwise run (perhaps much more slowly) on N nodes. In addition, there are runtime overheads for maintaining replica consistency.

With these costs, however, come significant advantages:

- **Dramatically increased system MTTI.** This approach dramatically reduces the number of faults visible to applications. Specifically, the application only sees faults that crash (or otherwise fail) *all* replicas of a particular rank.
- **Significantly reduced I/O requirements.** Increased system MTTI reduces the

speed at which checkpoints must be written to storage to allow applications to effectively utilize the system. A smaller fraction of the system cost and power budget must as a result be spent on the I/O system.

- **Detection of "soft errors".** By comparing the state of multiple replicas (e.g. using memory checksums) prior to writing a checkpoint, replication can detect if application state has been corrupted and trigger restart from a previous checkpoint.

- **Increased system flexibility.** The extra nodes used for redundant computation when running the largest jobs can be used for providing extra system *capacity* when running multiple smaller jobs for which fault tolerance is less of a concern. A system that uses N nodes and an expensive I/O system to reach exascale can only run 100 10PF jobs at a time, for example. A system that uses $2N$ nodes and a less expensive I/O system to reach exascale, however, can potentially run 200 10PF jobs at a time.

10.4 Evaluating Replication in Exascale Systems

The advantages described in Section 10.3 provide a compelling reason to examine the viability of state machine replication for extreme-scale HPC systems. Without quantifiable performance benefits compared to other approaches, however, state machine replication will not be viable for use in exascale systems. The remainder of this paper therefore examines the performance costs and benefits of state machine replication.

10.4.1 Comparison Approach

Our primary performance evaluation criteria is: at what node counts, if any, state machine replication provides quantitative performance *advantages* over past approaches particularly in terms of system utilization, *after* accounting for the overheads of state machine replication. If, for example, state machine replication achieves 46% utilization at a given system socket count and another technique only achieves 40% system utilization, we regard state machine replication as superior at that point.

We use traditional checkpoint/restart fault tolerance as the baseline technique against which to compare because its performance characteristics are well-understood. We hope that comparing against a well-understood baseline will facilitate future comparisons against other proposed exascale fault tolerance techniques as their costs and benefits at scale are more fully quantified. A brief qualitative comparison with several such techniques, however, is provided in [55].

10.4.2 Assumptions

Because we are comparing a new technique on projected hardware systems, our comparisons make a number of assumptions that are important to make explicit:

1. Full dual hardware redundancy for all applications, resulting in a maximum possible efficiency for state machine replication of 50%.
2. The MPI library is the only potential source of non-determinism in the application.
3. Machines suffer only crash failures, not more general failures from which checkpoint/restart may not be able to recover. While this replication approach can handle a more general fault model, the numbers in this paper do not include the checks required to handle a more general fault model.
4. Based on past study results [127], system MTTI decreases linearly with increased system socket count.

10.5 Model-based Analysis

We first examine the performance benefits of state machine replication compared to its fundamental redundant hardware costs. For this initial comparison, we assume every MPI process is replicated, and make very simple assumptions about system characteristics, particularly that there is no software overhead for maintaining replica consistency, that the system can checkpoint in a fixed amount of time regardless of scale, and that all failures follow a simple exponential distribution. These assumptions will be successively relaxed in the following sections.

When two nodes are used to represent the same MPI rank, the failure of one node in a pair does not interrupt the application. Only when both nodes fail does the application need to restart. The frequency of that occurring is much lower than the occurrence of a single node fault and can be characterized using the well-known *birthday problem*.

One version of the birthday problem asks how many people on average need to be brought together until there are enough to have a greater than 50% chance that two of them share the same birth month and day. If we equate days in a year with nodes and let the number of people represent the faults occurring, we can use the birthday problem to calculate how many faults can occur until both nodes in a pair are damaged and cause an application interrupt.

Equation 10.1, from [88, 68], shows how to calculate this version of the birthday problem.

$$F(n) = 1 + \sum_{k=1}^{n} \frac{n!}{(n-k)! \cdot n^k} \approx \sqrt{\frac{\pi n}{2}} + \frac{2}{3} \tag{10.1}$$

Essentially, replicas act like a filter between the system and an application, and the birthday problem helps us estimate how many faults can be absorbed before the application is interrupted.

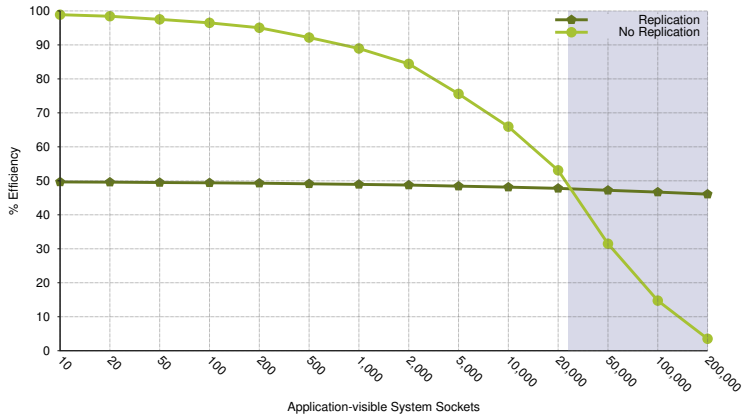


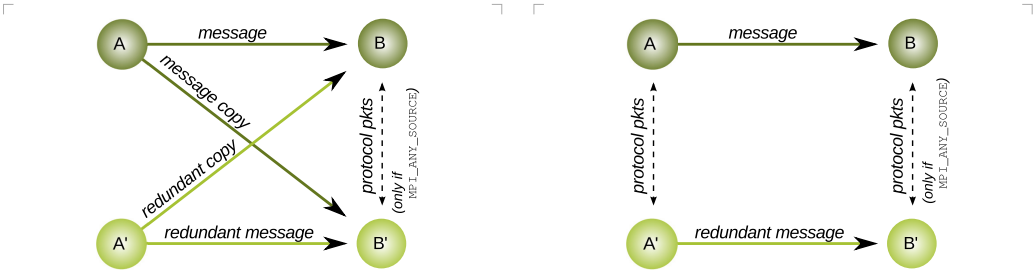
Figure 10.1: Modeled application efficiency with and without state machine replication for a 168-hour application, 5-year per-socket MTBF, and 15 minute checkpoint times. Shaded region corresponds to possible socket counts for an exascale class machine [17].

Figure 10.1 estimates the resulting application efficiency with optimal checkpoint intervals for both state machine replication and using only traditional checkpoint/restart. MTTI was computed directly from the birthday problem approximation in Equation 10.1, while the resulting efficiency is computing using Daly's higher-order checkpoint/restart model and optimal checkpoint interval [39]. These calculations assume a 43800 hour (5 year) per-socket MTBF based on past studies [127, 63], 15 minute checkpoint times as discussed in Section 10.2.1, and a 168 hour application solve time.

These results show the dramatic increase in system MTTI that state machine replication provides, allowing it to maintain efficiency close to 50% as system socket count increases dramatically towards the 200,000 heavyweight sockets suggested for exascale systems [17]. In contrast, the efficiency of a checkpointing-only approach drops precipitously as system scales approach those of upcoming exascale systems.

10.6 Runtime Overhead of Replication

While the previous sections demonstrate that state machine replication is viable at exascale in terms of the basic hardware costs, they do not evaluate the runtime overhead of the necessary consistency management protocols. Transparently supporting state machine replication for MPI applications on HPC systems requires maintaining sequential consistency between replicas. It also requires protocols for detecting and repairing failures. As mentioned in Section 10.2, these are potentially expensive in communication-intensive HPC systems as every replica must see messages arrive in the same order.



(a) Mirror Protocol (b) Parallel Protocol

Figure 10.2: Basic replicated communication strategies for two different *r*MPI message consistency protocols. Additional protocol exchanges are needed in special cases such as MPI_ANY_SOURCE.

To study this overhead, we designed and implemented *r*MPI, a portable user-level MPI library that provides redundant computation transparently to deterministic MPI applications. *r*MPI is implemented on top of an existing MPI implementation using MPI profiling hooks. In the remainder of this section, we outline the basic design and implementation of *r*MPI and measure the runtime overhead of this implementation for several real applications on a large scale Cray XT-3/4 system. A complete description of *r*MPI, including low-level protocol and implementation details is available elsewhere [53, 22].

10.6.1 *r*MPI Design

The basic idea for the *r*MPI library is simple: replicate each MPI rank in an application and let the replicas continue when an original rank fails. To ensure consistent replica state, *r*MPI implements protocols that ensure identical message ordering between replicas. Unlike more general state machine replication protocols [126, 32], these protocols are specific to the needs of MPI in an attempt to reduce runtime overheads. In addition, *r*MPI uses the underlying Reliability, Availability and Serviceability (RAS) system to detect node failures, and implements simple recovery protocols based on the consistency protocol used.

10.6.1.1 Basic Consistency Protocols

*r*MPI implements two different consistency protocols, named *mirror* and *parallel* , to ensure that every replica receives a copy of every message and to order message reception at replicas. Both protocols take special care when dealing with MPI operations that could potentially result in different message orders or MPI results being seen at different replicas. Note that collective operations in *r*MPI call the point-to-point operations internal to *r*MPI.

Figure 10.2(a) shows the basic organization of how the mirror protocol ensure that all replicas see the same messages. In this figure, A and B represent distinct MPI ranks and A' and B' are A's and B's replicas respectively. In this protocol, each sender transmits duplicate

messages to each of the destinations. Similarly, receivers must post multiple receives for the duplicate messages, but only require one of those messages to arrive in order to progress. This eases recovery after a failure, but doubles network bandwidth requirements.

The parallel protocol, in comparison, is shown in Figure 10.2(b). In this approach, each replica has a *single* corresponding replica for each other rank with which it communicates in non-failure scenarios. In the case of failure, one of the remaining replicas of a rank takes over sending and receiving for the failed node. This failure detection requires frequent message-based interaction with the reliability system on current systems. As a result, the parallel protocol will initiate approximately double the number of messages for each send operation. These extra messages contain MPI envelope information and are small. Therefore, the parallel protocol reduces network bandwidth requirements for an increased number of short messages.

10.6.1.2 MPI Consistency Requirements

*r*MPI assumes that only MPI operations can result in non-deterministic behavior, and there are a few specific MPI operations that can result in application-visible non-deterministic results. For example, *r*MPI must address non-blocking operations, wildcard (e.g. `MPI_ANY_SOURCE` and `MPI_ANY_TAG`) receives, and operations such as `MPI_Wtime`. As a first step, both *r*MPI protocols use the notion of a leader node for each replicated MPI rank, while non-leader nodes are referred to as replicas or redundant nodes. When a leader drops out of a computation, the protocol chooses a new replica from among those remaining for a rank to take over as leader. *r*MPI uses one high order bit in the tag to distinguish messages from leader and replica nodes.

For the remainder of consistency protocol discussions, we focus on the mirror protocol implementation; the parallel protocol implementation is generally similar and described in more depth elsewhere [53]. For blocking non-wildcard receives, one of the the most common forms of MPI communication, *r*MPI posts a receive for *both* senders A and A' into the buffer provided by the user. Since the data in the two arriving messages is identical, there is no danger of corrupting the user buffer. If multiple messages from the replica set A arrive with the same tag, *r*MPI must make sure that the first active and first redundant message arrive in the first buffer, and the second active and second redundant in the second buffer. *r*MPI achieves this by using one high-order tag bit, setting it on all outgoing redundant messages and setting the same bit for all receives of redundant messages.

Due to MPI message-passing semantics and the possibility of wildcard source receives, this basic approach is not completely sufficient. To handle `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, *r*MPI relies on explicit communication between the leader of each rank and other replicas. Essentially, *r*MPI allows only one actual wildcard receive to be posted at any time on a node, and then only on the leader. When a wildcard receive is matched, the leader then sends the MPI envelope information to replica nodes which then post for the actual message needed. The situation is more complicated for non-blocking wildcard receives, test, and wait

operations, requiring a queue of outstanding wildcard receives, but the basic approach is similar.

Finally, *r*MPI must guarantee that operations such as `MPI_Wtime()` return the same value on active and redundant nodes, as some applications make decisions based on the amount of time elapsed. For these situations, the leader node sends its computed value to the redundant node. As an option, *r*MPI can synchronize the `MPI_Wtime()` clocks across the nodes [77].

10.6.1.3 Failure Detection

*r*MPI's failure detection requirements are relatively modest, and it assumes the underlying supercomputer RAS system can provide much of this functionality. Both mirror and parallel protocols require that messages from failed nodes will be consumed and do not deadlock the network or cause other resources, such as status in the underlying MPI implementation, to be consumed. Furthermore, failing nodes must not corrupt state on other nodes. I.e., corrupted or truncated messages in flight must be discarded. Most systems already do this using CRC or other mechanisms to detect corrupt messages. The RAS system is responsible for ensuring that messages are not continually retransmitted from and to failed nodes .

For the parallel protocol we expect that there is a method to learn whether a given node is available or has failed. On our test systems, we emulate a RAS system at the user-level. This consists of a table which *r*MPI consults, and the RAS system updates, when a node's status changes. It could also be an event mechanism that informs *r*MPI whenever the RAS system detects a failed node.

10.6.2 Evaluation

10.6.2.1 Methodology

From the discussion in the previous sections it should be clear that *r*MPI will add overhead and lengthen the execution time of an application. To measure this overhead we ran multiple tests with applications on the Cray Red Storm system at Sandia National Laboratories compiled with both *r*MPI and the original unmodified Cray MPI library. Red Storm is a XT-3/4 series machine consisting of over 13,000 nodes, with each compute node containing a 2.2 GHz quad-core AMD Opteron processor and 8 GB of main memory.

To ensure leader and replica are on separate physical nodes, and to avoid memory bandwidth bottlenecks on the nodes themselves, we only used one core on each of the CPU sockets. Note, this memory bandwidth bottleneck can be removed from *r*MPI by having the NIC duplicate messages rather than the host CPU at the cost of the libraries portability.

We used four applications tested on up to 2,048 application-visible nodes (4096 total nodes in the case of replication): CTH [44], SAGE [74], LAMMPS [115, 124], and HPCCG [125].

These application represent a range of computational techniques, are frequently run at very large scales, and are key simulation workloads for the US DOD and DOE. These four applications represent both different communication characteristics and compute-to-communication ratios. Therefore, the overhead of *r*MPI affects them in different ways.

Because a given node allocation may impact the performance of an application, we ran our tests in three different modes. The first mode, called *forward*, assigns rank $n/2$ as a redundant node to rank 0, rank $n/2+1$ to rank 1, and so on resulting in a mapping like this: ABCD|A'B'C'D'. *Reverse* mode is ABCD|D'C'B'A', and *shuffle* mode is a random shuffle (Fisher/Yates) such as ABCD|C'B'D'A'.

10.6.2.2 LAMMPS

Figure 10.3 shows the performance impact of *r*MPI with both the mirror and parallel protocol. The impact of each redundancy protocol is less than 5%, independent of the nodes used, while the baseline overhead for each is negligible.

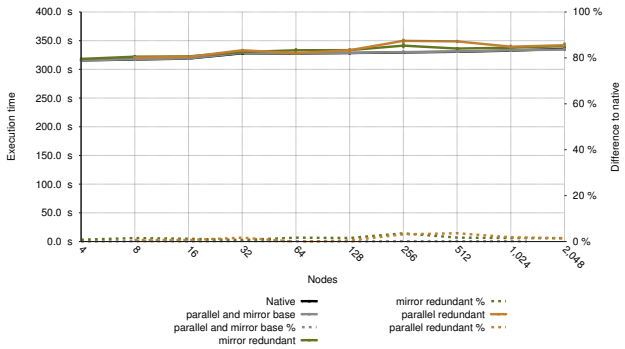


Figure 10.3: LAMMPS *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant is equivalent.

10.6.2.3 SAGE

Figure 10.4 shows the *r*MPI performance for SAGE. Similar to LAMMPS, the baseline performance degradation is negligible. Also similar to LAMMPS, the parallel protocol performance remains nearly constant and performance decrease is negligible in the tested node range; with performance overhead generally less than 5%. In contrast, full redundancy for the mirror protocol loses about 10% performance over native, with performance increasing with scale. We attribute the performance degradation for SAGE to the factor of two increase of large network messages sent by SAGE and the limited available network bandwidth.

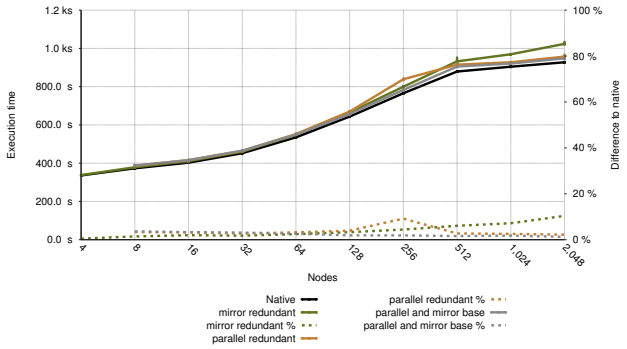


Figure 10.4: SAGE *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant is equivalent.

10.6.2.4 CTH

In Figure 10.5 we see the impact of our consistency protocols for CTH at scale. Again, baseline for both mirror and parallel shows little performance difference. For CTH, mirror has the greatest impact on performance with full redundancy. This impact, which is nearly 20% at the largest scale, is due to CTH's known sensitivity to network bandwidth [110] (the greatest of each of the applications tested) and the increased bandwidth requirements of the mirror protocol. Interestingly, the parallel protocol version of CTH runs slightly *faster* then the native versions (around 5-8%) for forward, reverse, and shuffle replica node mappings. Though further testing is needed, current performance analysis results suggest this decrease in application runtime is due to parallel reducing the number of unexpected messages received.

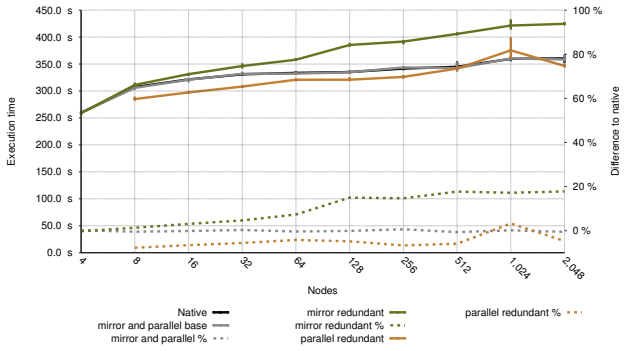
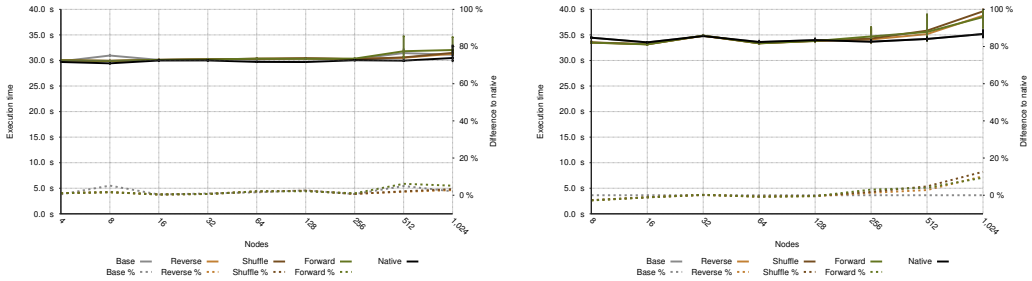


Figure 10.5: CTH *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant is equivalent.

10.6.2.5 HPCCG

Figure 10.6 shows the performance impact of *r*MPI on the HPCCG mini-application. In contrast to the other results presented in this section we present the mirror and parallel results separately. Though the results presented in Figure 10.6(a) and Figure 10.6(b) represent the same computational problem, the native results of each vary due to different node allocations between the two plots. Allocation issues aside, we see that mirror has very little impact. Parallel on the other hand shows a significant impact at higher node counts, with slowdowns of around 10% at 1,024 nodes. Also, in contrast to all the other applications tested, impact from the parallel protocol is greater than that of mirror. This is because unlike other applications, HPCCG stresses the system's message rate and parallel's synchronization messages are causing it to reach the maximum messaging rate of a node.



(a) Mirror protocol (b) Parallel protocol

Figure 10.6: HPCCG *r*MPI performance comparison. Varying performance for native and baseline between mirror and parallel protocols is due to different node allocations.

10.6.3 Analysis and Summary

Our results evaluating the runtime overhead of state machine replication show that the runtime costs of implementing state machine replication for a wide range of production HPC applications at significant scale is minimal. In particular, for each application, either the parallel or mirror protocol provides almost negligible performance impact. Examining the best protocol for each application, SAGE has the highest net overhead, 2.2% at 2048 application-visible nodes, using the best protocol.

To project these overheads to future systems, we take a basic curve-fitting approach. We expect communication overheads to be sublinear because replication's communication overheads are proportional to the application's communication demands, and scalable application must keep communication overheads sublinear with increasing node counts. This analysis shows that a logarithmic curve, shown in Equation 10.2, can be used to characterize the overhead for the worst-case application, SAGE.

$$g(S) = \frac{1}{10} \log S + 3.67 \tag{10.2}$$

This curve would result in a 4.9% additional overhead on a projected exascale system with 200,000 sockets.

Note, however, that this analysis assumes the absence of hardware bottlenecks such as observed for the mirror protocol for HPCCG. More sophisticated techniques, for example ones based upon the simulation of exascale networking hardware, would be necessary to account for such bottlenecks. Based on our experience so far, however, we expect that either the mirror or parallel protocol could avoid such bottlenecks.

10.7 Simulation-Based Analysis

10.7.1 Overview

In this section, we use a simulation-based approach to verify, integrate, and expand the results from the previous sections into a more complete analysis of the costs and benefits of state machine replication for HPC systems. This approach allows us to examine real failure distributions derived from studies of failures of real HPC systems in addition to the exponential distributions assumed in analytical models such as those of the Daly model or the birthday problem. We also use it to examine additional machine parameters and their impact on the viability of state machine replication, particularly variations in available I/O system bandwidth and failure rates of components.

In the remainder of this section, all results assume software runtime overheads as shown in Equation 10.2; efficiency results also include a factor of two reduction for replication because of the required redundant hardware. Unless otherwise stated, we also continue to assume checkpoint and restart times of 15 minutes as in previous sections.

10.7.2 Combined Hardware and Software Overheads

As a first study, we reexamine state machine replication under exponential failure distributions with a 5 year per-socket MTTI as in Sections 10.2 and 10.5, but this time including projected software runtime overheads from Section 10.6. As can be seen in Figure 10.7, these results are similar to those of Figure 10.1, with the break-even point for state machine replication shifted to a somewhat higher socket count due to the additional software runtime overheads. Despite this, state machine replication still outperforms traditional checkpoint/restart at socket counts currently projected for use in exascale systems.

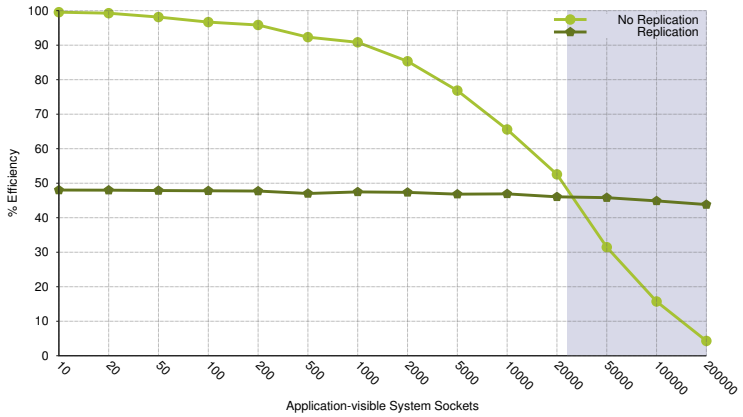


Figure 10.7: Simulated application efficiency with and without replication including *r*MPI run time overheads. Shaded region corresponds to possible socket counts for an exascale class machine [17].

10.7.3 Scaling at Different Failure Rates

While the 5 year per-socket MTBFs used above are based on well-known studies of large-scale systems, the challenges of exascale systems make changes to these reliability statistics likely. For example, more reliable nodes could be deployed to address fault tolerance concerns, or power conservation, miniaturization, or cost concerns could lead to a *reduced* per-socket MTBF. Because of this, we also examined the viability of state machine replication over a range of per-socket MTBFs.

This evaluation focuses on determining the *break-even point* in number of system sockets for state machine replication compared to traditional checkpoint/restart. This is the number of sockets above which state machine replication is more efficient than traditional checkpoint/restart even accounting for replication's software and hardware overheads. At socket counts greater than or MTBFs less than this break-even point, replication is preferable; at socket counts less than this or MTBFs above it, traditional checkpoint/restart is preferable.

Figure 10.8 shows these results for per-socket MTBFs up to 100 years; socket counts and per-socket MTBF commonly discussed for exascale systems (socket counts above 25,000 and MTBFs between 4 and 50 [17]) are shaded; the shaded area above and to the left of the break-even curve represents the portion of the exascale design space in which state machine replication is beneficial.

These results show that state machine replication is viable for a large range of socket MTBFs and node counts in the exascale design space, but not the entire space. In particular, state machine replication performs worse that traditional checkpoint/restart for low socket-count systems with MTBFs greater than about 10 years. For socket MTBF above 50 years, state machine replication is outperformed by traditional checkpoint/restart at all expected socket counts.

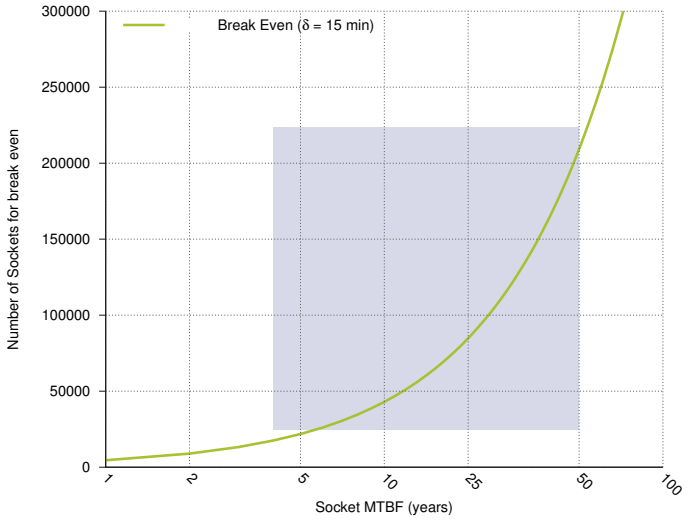


Figure 10.8: Simulated replication break-even point assuming a constant checkpoint time (δ) of 15 minutes. Shaded region corresponds to possible socket counts and MTBFs for an exascale class machine [17]. Areas of the shaded region where replication uses less resources are above the curve. Areas below the curve are where traditional checkpoint/restart uses lower resources.

10.7.4 Scaling at Different Checkpoint I/O Rates

We also examined the viability of replication at a wide range of checkpoint I/O rates. Because checkpoint I/O is an area of active study, including work on a wide range of hardware and software techniques to improve its performance for exascale systems (as in [55]), understanding the potential impact of this research on exascale fault tolerance approaches is critical.

For this analysis, we used recent modeling work which extends Daly's checkpoint modeling work to account for how variations in checkpoint system throughput impact checkpoint times and system utilization [102]. We assume each socket in the system has 16 GB of memory associated with it, and again examine the break-even point for replication over checkpoint/restart at a range of checkpoint I/O bandwidths and socket MTBFs. We choose an aggressive range of such bandwidths ranging from 500 GB/sec to 30 TB/sec to fully understand the impact of dramatic increases in I/O rates on the viability of replication.

Figure 10.9 shows the results of this analysis. Replication outperforms checkpointing for the vast majority of the exascale design space at checkpoint I/O bandwidths of 1 TB/sec or less. However, beginning at I/O bandwidths of approximately 5 TB/sec, checkpoint/restart becomes competitive for a substantial fraction of the design space, particularly systems with high per-socket MTBFs and low numbers of sockets. At checkpoint bandwidths of 30 TB/sec or higher, several orders of magnitude faster than current I/O systems, checkpoint/restart is preferable across a large majority of the design space.

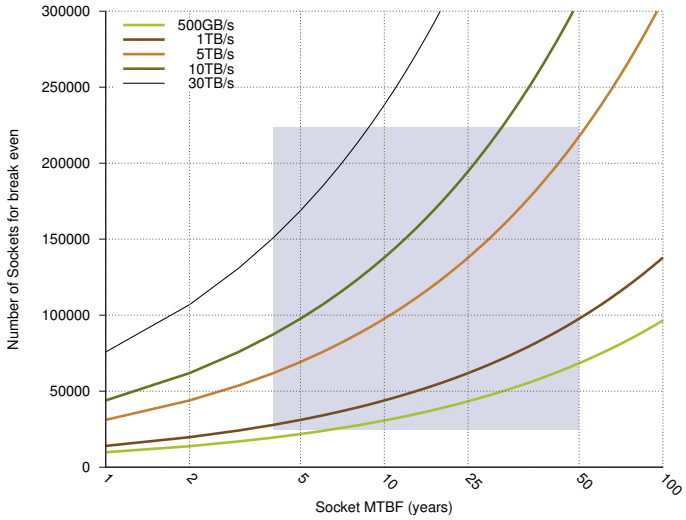


Figure 10.9: "Break even" points for replication for various checkpoint bandwidth rates. The shaded region corresponds to possible socket counts and socket MTBFs for exascale class machines [17]. State machine replication is a viable approach for most checkpoint bandwidths, but with a checkpoint bandwidth greater than 30TB/sec, replication is inappropriate for the majority of the exascale design space. Areas of the shaded region where replication uses less resources are above the curve. Areas below the curve are where traditional checkpoint/restart uses lower resources.

10.7.5 Non-Exponential Failure Distributions

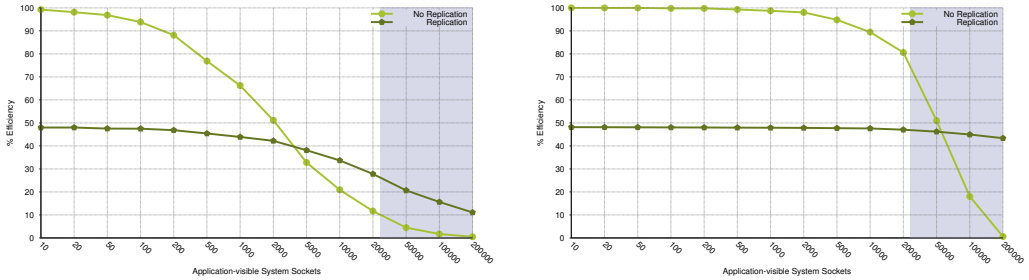
Finally, we also examine the viability of replication with more realistic failure distributions. For failure information, we use numbers from a recent study of failures on two BlueGene supercomputer systems, a 16,384 node system at Rennesseler Polytechnic Institute (RPI) and a 4,096 node system at École Polytechnique Fédérale de Lausanne (EPFL) [63].

This study shows that failures in these systems are best described by a Weibull distribution with MTBFs of 6.6 hours (11.7 years/socket) and 8.4 hours (3.9 years/socket), and shape (β) values of 0.156 and 0.469, respectively. These β values ($\beta < 1.0$) describe distributions that decrease in probability over time; in HPC systems, this indicates that failures are more likely to happen at the start of a system's lifetime or an application run and reduce in frequency as the system runs.

To examine the impact of these failure distributions, we build on the results of the previous subsection and examine how the efficiency of replication and checkpoint/restart change under Weibull failures assuming a fixed 1 TB/sec checkpoint bandwidth and 16 GB of memory per socket. Note that the systems from which these distributions were measured experienced a significant number of I/O system failures, and it is unclear how these failures should be properly scaled up to larger systems. As a result, we use the failure data from the larger of the two systems (the RPI system), and focus on how Weibull distributions change the efficiency of replication and checkpoint/restart approach as opposed to the specific

efficiency crossover point.

Figure 10.10 presents impact of these failure distributions on both a replication-based approach and a purely checkpoint-based approach. These results show that Weibull fail-



(a) Weibull $\beta = 0.156$, socket MTBF = 12 years (b) Exponential, socket MTBF = 12 years

Figure 10.10: Comparison of simulated application efficiency with and without replication, including *r*MPI run time overheads, using a Weibull and Exponential fault distribution. In both these figures we assume a checkpoint bandwidth of 1TB/sec. The shape parameter (β) 0.156 and MTBF corresponds to the RPI BlueGene system [63]. Shaded region corresponds to possible socket counts for an exascale class machine [17].

ures experienced by real-world systems result in a much more challenging fault tolerance environment, reducing the effectiveness of both replication and traditional checkpointing approaches. However, replication is less severely impacted than traditional checkpointing, again pointing to the potential viability of a replication-based fault tolerance approach for exascale systems.

10.8 Conclusions

In this chapter, we evaluated the suitability of replication, an approach well-studied in other fields, as the primary fault tolerance methods for upcoming exascale high performance computing systems. A combination of modeling, empirical evaluation, and simulation were used to study the various costs and benefits of state machine replication over a wide range of potential system parameters. This included both the hardware and software costs of state machine replication for MPI applications, and covered different failure distributions, system mean time to interrupt ranges, and I/O speeds.

Our results show that a state machine replication approach to exascale resilience outperforms traditional checkpoint/restart approaches over a wide range of the exascale system design space, though not the entire design space. In particular, state machine replication is a particularly viable technique for the large socket counts and limited I/O bandwidths frequently anticipated at exascale. However, replication-based approaches are less relevant for designs that have per-socket MTBFs of 50 years or more, less than 50,000 sockets, and checkpoint bandwidths of 30 terabytes per second.

Outside of its performance benefits, using replication as the primary exascale fault tolerance method provides a number of other advantages. First among these is that it can be used to detect and aid in the recovery from faults that corrupt system state instead of crashing the system, sometimes referred to under the banner of silent errors. Checkpoint-based approaches, on the other hand, potentially preserve such errors. In addition, the extra hardware nodes needed to support replication-based approaches can also be used to increase the capacity of exascale systems when it runs more but smaller (e.g. 1-10 petaflop-scale) jobs.

Chapter 11

Enabling Dynamic Resource-Aware Computing

Abstract: Improved application performance and resource utilization can be achieved through better matching of application tasks to resource capabilities. Potential use-cases range from better initial application placement, based on application communication profiles and current network utilization, to dynamic application reconfiguration based upon contention for resources or upon detection of failing or degrading resources. Such modes of operation, which we call *Resource-Aware Computing*, rely on capabilities for 1) run time assessment of application needs 2) run time assessment of resource state 3) evaluation and feedback of application-to-resource mapping fit and 4) dynamic remapping of application processes to resources. Monitoring and assessment capabilities must have minimal contention with application needs, scale to large numbers of processes, and perform on timescales that can benefit applications during their run-times. We have designed and developed lightweight, scalable, interacting capabilities in these areas and demonstrated them on Cielo, proving the feasibility of and providing an infrastructure for Dynamic Resource-Aware Computing.

11.1 Introduction

The amount of work, communication, and memory access of each process in a distributed parallel application can vary significantly across processes during the course of a long lived application. Additionally resources allocated to processes over the lifetime of an application run may not be homogeneous (e.g. may be shared with system processes, affected by other applications or processes, etc.). Thus the ability to perform run-time assessment of resource efficacy for the processes using them and the process workload is key to efficient and effective dynamic run-time management of resources, process to resource mappings, and process work allocation. The necessary monitoring, assessment, and dynamic re-allocation must not adversely impact application performance and should, either through better understanding and more informed initial allocations or via run-time dynamic modification, yield better application performance (e.g. shorter run times, less power consumption, more efficient use of platform resources, etc.).

Currently, system resources are assigned to applications at launch time and are fixed for

the duration of a run. While information from fine grained performance profiling tools such as Open|SpeedShop [108] and CrayPat [37] can be used to intelligently map processes to resources, these types of profiling tools are typically highly intrusive (from an application perspective), with high over-head (e.g. 5% to 15% for Open|SpeedShop [108]) and low scalability, and may not account for the actual workload on the machine at run time. While dynamic load balancing is well-studied (see, for example [33] and references therein), in production implementation, it is not necessarily tied to capabilities for providing detailed run-time data about resource state; for example, the applications targeted in this work in practice rely on no or limited information about resource state (e.g., balancing based on number of elements or upon run-time).

In contrast, we have developed tools for resource/application monitoring, run-time analysis, and dynamic run-time application feedback. Our scalable tools provide run-time resource utilization, resource-state, and coarse grained application profiling information with low ($< 0.01\%$) overhead. The intent is that this information be used to inform and/or trigger appropriate and available system responses, such as better initial application placement based on application communication profiles determined via profiling, dynamic task placement based on resource availability, and dynamic application reconfiguration triggered by discovery of contention for resources or upon detection of failing or degrading resources.

In this work, we have developed a dynamic evaluation and feedback mechanism for the Zoltan partitioner, triggering dynamic repartitioning of Sierra applications on Cielo, with no required changes to the application code. This work demonstrates the feasibility of providing an infrastructure for Dynamic Resource-Aware Computing. It addresses the need called out by an NNSA Workshop on Exascale Computing Technologies tools working group [129] to "enable run-time application optimization in the face of changing application needs and platform state" and targets the identified development of capabilities for "scalable collection and analysis of performance data", "ability to feed analysis results back to the application and/or system software", and "analysis results to be used by applications for run-time optimization".

We describe our tools and present preliminary experimental results in this chapter[1].

11.2 Infrastructure Components

In this section we describe the components we have developed for enabling Resource-Aware Computing, with respect to architecture and overhead. The various components' relationships to each other are shown in Figure 11.1.

[1]More details of this work can be found in [20].

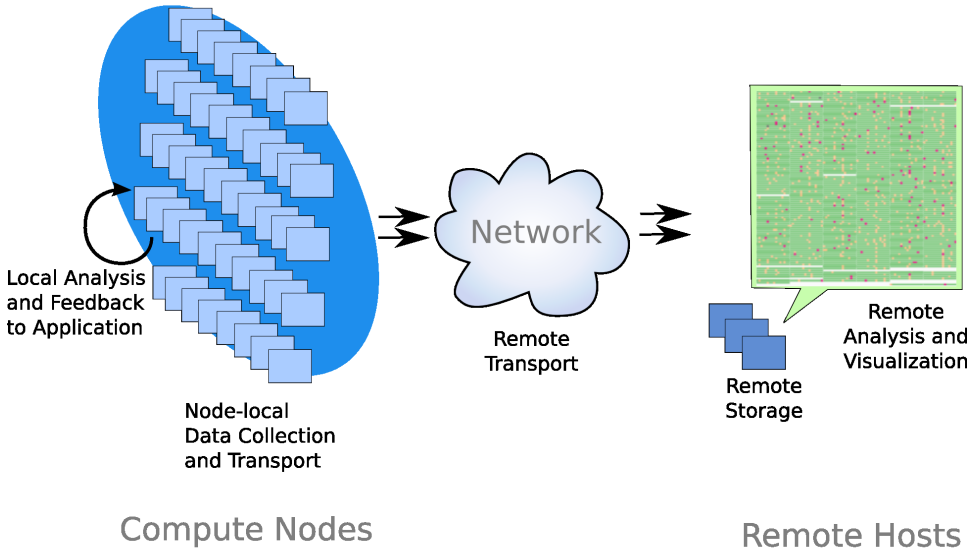


Figure 11.1: Integration of capabilities for Low Impact Monitoring, Transport and Storage, Analysis, and Feedback to enable Resource-Aware Computing

11.2.1 Low Impact Monitoring

In order to minimize the impact of monitoring on running applications we have created a lightweight on-node data collection and transport infrastructure (called lightweight distributed metric service, or LDMS) with the following characteristics: 1) both RDMA and socket based transports are available if supported by the platform, 2) kernel modules interact with the linux scheduler to only collect data during what would otherwise be idle cycles, 3) no data processing is performed on compute nodes, 4) collection rate is controlled externally and may be dynamically modified, and 5) facility for maintaining a node local pool of remote peer data in order to minimize remote peer data access latency. Overhead for utilizing our monitoring infrastructure with various user mode collection components with respect to both CPU and memory footprint is shown in Figure 11.2.

11.2.1.1 Data collection

LDMS allows for zero CPU overhead when gathering data from compute nodes through the RDMA transport (if RDMA is supported by the platform), however, there is still overhead associated with gathering the data at the sampled node. Data of interest is kept in many different places on a Linux system. Some data is maintained by user-mode programs, for example, Zoltan, or MPI rank and job data; other data is kept in the kernel, for example scheduler and VM data; and still other data is kept by hardware itself such as hardware performance counters.

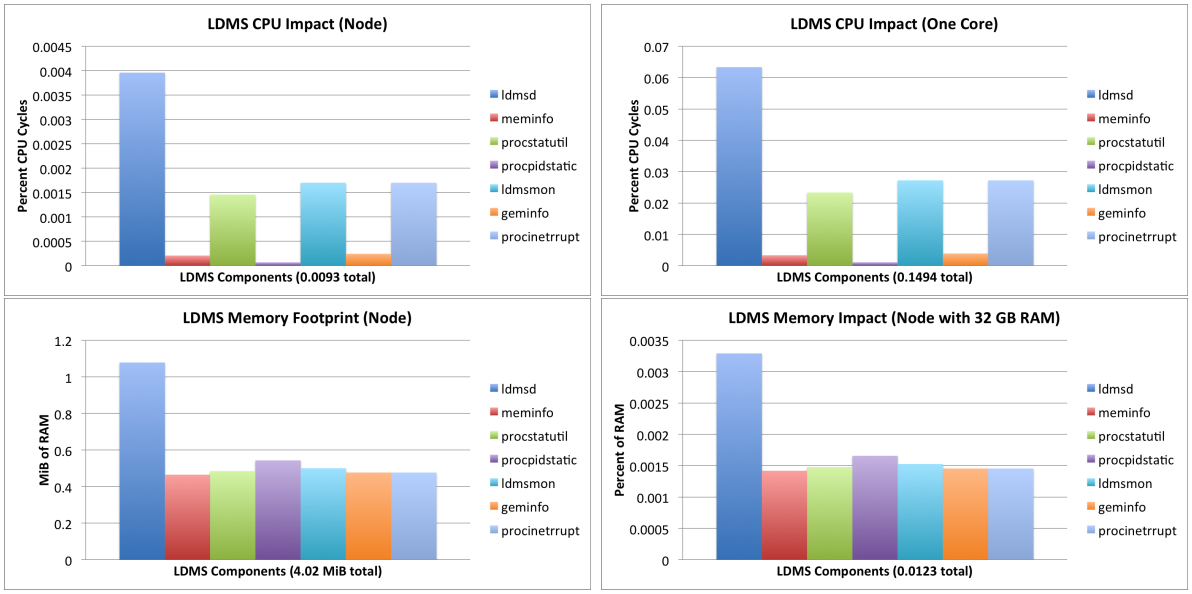


Figure 11.2: Monitoring infrastructure overhead with respect to CPU and memory footprint. Impact on CPU resources with respect to the total available on this 16 core per node system are shown in the upper left while that for running on a dedicated core (using corespec (Section 11.3.3.1) is shown in the upper right. Memory footprint in MiB is shown in the lower left while that as a fraction of the 32GB available is shown in the lower right. Note that these figures show overhead on a per component basis and that the total is shown in parentheses in the X-axis label.

In order to be used, data has to be gathered into a Metric Set where it can be fetched by the sampling node. For user-mode data, this can be trivially accomplished by either keeping the data natively as a LDMS Metric Set, or by gathering the data of interest and writing it to an LDMS Metric Set. For kernel data, there is the problem of accessing the data itself. Some data is accessible trivially by kernel modules because the containing data structure is exported, e.g., `vmstat` data. Other data, however, is "hidden" by virtue of being declared `static` in C and can only be accessed through a user-mode `/proc` filesystem interface; e.g. `schedstat`. Finally hardware performance counter data can be accessed with the kernel perfctr API. In all cases, however, there is the problem of introducing scheduler jitter when sampling. For kernel data gathering, this can be avoided by using a kernel work-queue and only gathering data during I/O wait or otherwise idle cycles. In user-mode, this can be accomplished by scheduling the sampler thread with very low priority.

11.2.1.2 Node-local Transport

Node local transport is performed via Unix domain socket on the node. This provides a low latency path for an application to acquire both local data and/or data from a pool of designated remote peer nodes. The difference between this and the remote data transport is that node-local data cannot utilize the RDMA transport and hence potentially utilizes

more CPU resources. However, node-local transport is expected to be utilized relatively infrequently (e.g. when an application is requesting data in order to evaluate a load balancing response) and hence will introduce insignificant overhead and/or jitter to the application as a whole.

11.2.2 Remote Transport and Storage

Remote transport of data is desirable for several reasons: 1) analysis of aggregations of data to understand statistical distributions, anomalous situations, and overall headroom for improvement, 2) storage for comparison against data from other application runs, and 3) enabling low latency acquisition of a statistically meaningful set of peer node data for use in load balancing and scheduling decisions.

11.2.2.1 Remote transport

In order to minimize the impact of remote data transport on application processes running on a node LDMS supports remote distributed memory access (RDMA) where hardware and software permit. Thus a remote entity may access another node's data as often as desired with no impact on running applications past an initial RDMA registration which may stay active over the uptime of a node. Remote transport of data is performed using a pull model in which the owning entity, whether it is the node on which the data is being collected or an intermediate transport node, is either completely passive (as in the case when using RDMA) or only sends a segment of data when it is requested (in the case where socket based transport is being used). Metadata and data generation numbers are used to ensure that information being stored is both new and correct. In order to support socket based transport of data across asymmetric network boundaries we support two socket setup methods for accessing remote data. In order to describe these we define the "top" of a data stream to be the node on which the data is collected and originates and the "bottom" to be the endpoint from which the request for data is made. In this context "upstream" denotes closer to the data source, or top, and "downstream" denotes closer to the data requester, or bottom. If a downstream LDMSD (LDMS daemon) entity is allowed to initiate a socket connection with his upstream neighbor LDMSD entity it is called "active" and does so directly. If, however, security profiles are such that a connection between two such entities can only be initiated from upstream to downstream, then we utilize a "passive" LDMSD entity on the downstream node which listens for a connection request from its upstream neighbor which we refer to as a "bridge" entity. Once the socket connection has been established the passive LDMSD entity makes its requests for data from its upstream neighbor over this connection in the same way an active LDMSD entity does. Note that these are persistent connections due to the desire to minimize overhead i.e., we don't want the overhead of negotiation for every data request as these are expected to occur on sub- to few-second intervals.

11.2.2.2 Remote Storage

Constraints on storage on the remote host(s) include not only the ability to ingest data at run-time rates, but also to represent that data in a way suitable for analysis. We have developed a database backend, consisting of multiple, non-replicated databases. Analyses are performed on the databases in parallel, with the results aggregated to a common front end. Details of the storage implementation are beyond the scope of this work, but can be found in [19]. In the exploratory phase of this work, we ingested samples of 1012 variables per node from 556 compute nodes at a rate of 5 samples per second into 4 distributed databases. We expect that, after analysis, the necessary number of collected variables can be reduced such that an application run on the full Cielo system could be handled at the necessary rate by the same 4 databases. For example 122 variables per node collected on 10 second intervals across the whole Cielo system would require the same data ingestion rate as the experimental work presented here.

11.2.3 Analysis

Our data analysis serves several functions: 1) provides understanding of how an application as a whole is utilizing the resources allocated to it in order to enable optimal matching of application needs to resource requests on subsequent but similar application runs, 2) enable dynamic resource to load mapping over the course of an application run, and 3) comparison of resource utilization/contention across processes, resources including nodes, and application runs to understand runtime variability and how utilization of data, analysis, and response has impacted both system resource utilization and application run time. We describe both run-time and post processing use of the data in the following subsections. Additionally we briefly describe the analysis role of our visualization component.

11.2.3.1 Post-processing

Before trying to dynamically adjust how an application is utilizing available resources it needs to be determined if any adjustment is needed (i.e., is there a problem?) and, if so, what measurements should be taken into account, how each should be weighted, and what functional aggregation should be performed. After these things are determined and implemented, application runs incorporating dynamic adjustment based on the implementation should be compared against the baseline case to determine if there was improvement and if there might be room for more. Post-processing analysis of system state profiles related to application runs is used to address these use cases.

11.2.3.2 Run-time

Run-time analysis using our framework can be done in a node-local manner taking into account information about both the local node and a peer node pool or remotely at the data storage nodes as both locations have access to the same information. The tradeoffs for these two modes of run-time analysis are latency, processing impact on application processes, and compute node memory footprint. While performing run-time analysis remotely minimizes processing impact on application processes and compute node memory footprint it also presents a high latency path for response. Thus for low-latency response we perform small computations on node, using the on-node data collection and local transport API to access data. In this work our run-time analysis was done in the context of dynamic load partitioning. Utilizing information gained in the post-processing analysis (Section 11.2.3.1), we determined a subset of variables to use in determining load imbalance and functional forms for evaluating relative balance with respect to those variables. We then wrote small functions that accessed only the necessary data to calculate the imbalance measures. Those resultant relative measures of load imbalance were then used in the run time repartitioning algorithms (Section 11.2.4.2).

11.2.3.3 Visualization

Our visualization tool is useful for imparting qualitative understanding of balance and/or contention in a running system. It enables a user to directly view raw measurables, such as CPU utilization, memory utilization, interrupts, context switches, etc., as a color-mapped value on a component. Simultaneously viewing these over all components being utilized by an application (including the operating system) provides a qualitative evaluation of how far out of balance a particular measurable is across all components and how that unfolds over time. We also provide methods for rendering the time history of a small subset of values and for statistical computations over ranges of time and components. This tool can be used for both run-time and post data capture viewing of such information. Figure 11.3 is an example of using this tool for visualizing where system processes are running.

11.2.4 Application Feedback Methods

Our goal is that monitoring and analysis information be used to inform and/or trigger appropriate available system responses. Analysis or response mechanisms can call the node local transport API for acquisition of both node local and peer node pool metrics of interest or use the remote analysis results. In this work we utilized both static and dynamic forms of feedback described in this section. Note that in this work all compute node and core allocations are static for the duration of an application run and that in the context of this work dynamic feedback results in process load re-allocation.

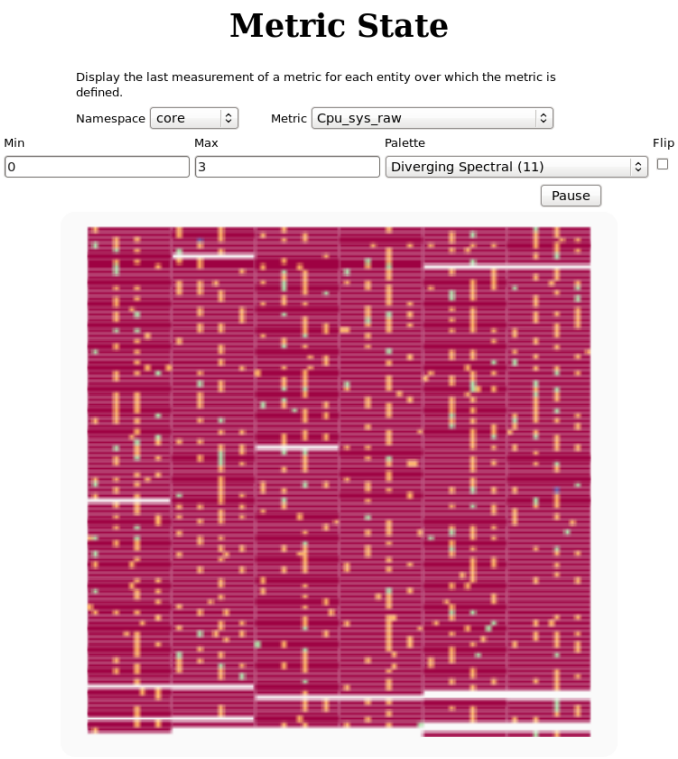


Figure 11.3: Screenshot of visualization tool showing system processes across the 576 nodes of Cielo Del Sur (Cray XE6) with no application running. Each horizontal row represents data values for each of 16 cores on each of 6 nodes (separated by fine bands). Note that white stripes represent nodes for which data was not being collected (e.g. service nodes or out of service). The color scheme highlights system process utilization with red being zero ticks and blue being three.

11.2.4.1 Static feedback mechanisms

We define static feedback as *the use of knowledge gained through analysis of resource utilization/contention characteristics from one application run to drive how resources are allocated in subsequent application runs*. In this work we identified two potential mechanisms for invoking static feedback: 1) Cray's "core specialization" utility to bind system processes to a particular core and 2) specification at job submission time how many and which cores to use on a per node basis. The first mechanism enables the user to ensure that there is no contention between system processes and application processes while the second enables the user to adjust how many processes share resources such as cache, main memory, network bandwidth, etc. In this work we only utilized the first of these mechanisms because from the data we were able to collect we could not infer whether or not contention for resources shared by cores was occurring.

11.2.4.2 Dynamic feedback mechanisms

We define dynamic feedback as *the use of knowledge gained through analysis of resource utilization/contention characteristics from both current and previous application runs to drive run-time resource allocation (including work to process allocation) for the current application run.* In order to enable dynamic workload to resource mapping, we targeted applications with facilities for redistribution of their workload during their runtime. In this work we chose applications in the Sierra [45] suite, which periodically call the Zoltan partitioner [40] to determine a new partitioning and which also have the facility to relocate data accordingly.

Zoltan contains a number of algorithms which assign objects to partitions, taking into account specified *object weights* and *partition sizes*. Examples of the former include number of elements or particles. In this work, we sought to define partition sizes, reflective of the relative ability of a resource to handle the workload assigned to it. We enhanced Zoltan to, at the time of a partitioning calculation, make the appropriate calls to obtain variable data via our on-node data collection API and calculate the partition sizes via the defined functional form. These partition sizes were then used in the partitioning calculation as described in Section 11.2.3.2. The data movement to adjust the workload in response to the determined partitioning is handled by Sierra itself.

11.3 Applying the Infrastructure to Enable Resource-Aware Computing

In this section we describe results of applying our infrastructure and tools as an integrated capability to targeted application runs on several Cray XE6 systems including LANL's Cielo. Here we describe characteristics of two applications we used and how our feedback mechanisms influenced dynamic run-time partitioning of their workload to process mapping.

11.3.1 Applications characteristics and experimental setup

In this work we targeted two codes: Aria [99] and Fuego [43, 42]. Aria is a thermal code and thus the per-processor computational load is dependent on the number of elements in the computation and less tied to the physical dynamics of the computation. As a result, significant dynamic load imbalance is not expected, however the Aria example code has been shown to scale well to large numbers of processors. Fuego is a particle code whose per-processor computational load changes throughout the computation as particles move, are injected, split, and/or removed. Repartitioning throughout the runtime thus redistributes the mapping of particles to processors and rebalances the load. Rebalancing capabilities for such codes may potentially have significant benefit, however the particular Fuego example utilized here is limited in its scalability without significant additional tuning/investigation which was beyond the scope of this work.

The applications were run 1) on Cielo, with node-local analysis and feedback via Zoltan, and remote transport over DISCOM, an ASC wide-area high speed network, with remote analysis and storage on the ASC TLCC Whitney cluster at Sandia CA and 2) on a Cray XE platform at Sandia NM, again with node-local analysis and feedback via Zoltan, with remote transport to desktop machines at Sandia CA for analysis and storage. The Aria example was run with a process count of up to 10,112 while the Fuego example only scaled to the 64 processes represented in this work.

11.3.2 Methodology used to search for appropriate indicators

Because we specifically targeted load imbalance and contention we were looking for data that indicated evidence of wide variation of related metric values over processes of an application. This suggested that we target things with a large statistical variance. A problem with just computing mean and higher order moments on this type of data is that even for a single process there is large variance on most dynamic metrics (CPU utilization, non-voluntary context switches, interrupts, etc.) over our sampling intervals of 5 to 10 seconds. Thus we summed these values, on a per processor basis, over time windows commensurate with the applications rebalancing interval, rather than our sampling interval, and examined the distribution of these sums over all processes.

11.3.3 Aria measures of interest, feedback, and results

Due to the aforementioned static nature of Aria's per process workload we first looked for indication of contention for resources. Analysis of several runs of our Aria example indicated that there were indeed metrics for which there was a wide distribution across processes, low variance for each particular process, and a high degree of correlation between system process core affinity and application processes with large numbers of these metrics (non-voluntary context switches, interrupts, idle CPU, etc.). The ones which directly implied contention were non-voluntary context switches and interrupts. Since these would adversely impact the time hardware resources are dedicated to the Aria application, we targeted these for use in the repartitioning calculation. (Note that the magnitude of counts of these are vastly different and the impact of any occurrence on the application run time and memory eviction was beyond the scope of this work.) This discovery drove the nature of the feedback and run-time analyses utilized in conjunction with this application and described below.

11.3.3.1 Static feedback with Aria

In this case we utilized Cray's "core specialization" (aka "corespec") feature [38] which enables the user to specify that no system processes will be run on cores dedicated to application processes. This comes at the expense of having to give up a core per node (the default of core 15 was used in this case, but any core can be specified for this use) to host system

processes. We ran the same number of processes (8310) over 12% more nodes to accommodate the loss of one core per node. The results show that non-voluntary context switches for application processes drop to zero and, though we expected slightly worse run-times given more inter-node communication, we obtained similar run-times presumably due to the higher cost of more inter-node communication being offset by better per-node performance. While in this case we utilized all cores available for application use per node, indication of process contention for shared cache, main memory, or network resources would be indicators that sparser use of cores, to reduce the sharing of resources for which contention has been seen, would be in order.

11.3.3.2 Dynamic feedback with Aria

We utilized the Zoltan partitioner to implement dynamic feedback to both codes. As described in Section 11.2.4.2, Zoltan assigns objects to partitions, taking into account specified object weights and partition sizes. In practice, in the targeted codes, the partition size is uniform. In contrast, in this work, we sought to define partition sizes, reflective of the relative ability of a resource to handle the workload assigned to it.

From an analysis perspective the measures of imbalance had to additionally be expressed in terms commensurate with the object weights. In this work we selected the number of *non-voluntary context switches* and the number of *rescheduling interrupts* to be used as measures of contention in an expression for relative imbalance. We not only needed to determine a functional form in which a greater number of non-voluntary context switches would result in a smaller partition size but also one *scaled appropriately* to still achieve a reasonable relative distribution across all the partitions.

Using functional forms for partition sizes based on our post-run profiling, we ran our applications, with Zoltan repartitioning the workload per processor based on the data values exhibited at run time.

We first utilized the following function of two contention related metrics (non-voluntary context switches and rescheduling interrupts) to drive partitioning decisions in Aria for a 10,112 processor run on Cielo: for measures of contention c1 and c2 $partitionsize = (c1 + c2) * (100)/max(c1 + c2)$ with a floor of 1 and a ceiling of 100. The measures of contention are evaluated from values after the previous rebalancing step to the current rebalancing calculation. At run time, non-voluntary context switches and interrupts occurred preferentially on cores 0 and 9 across all nodes. Our run-time feedback thus resulted in smaller partitions on those cores, however our functional form resulted in too great a range of partition sizes to derive benefit in this particular run. Additional tuning of the parameters of the functional form would be required in order to reduce the range appropriately.

Likewise we utilized the following function of CPU related metrics to drive partitioning decisions as a result of imbalance in CPU utilization by an 8310 processor Aria run on Sandia's Cielo Del Sur (CDS): $partitionsize = [(usertime+systemtime+idletime)/(usertime+systemtime)]$. The times are measured in ticks between after the previous rebalancing step to

the current rebalancing calculation. Partitioning with inclusion of the partition size slightly broadened the distribution of partition sizes beyond that based on object weights alone. Again, the degree of application benefit is dependent upon tuning of the functional form used for feedback. However, for this function, when the partitioning became too broad, resulting in increased imbalance, the partitioning feedback criteria adjusted the distribution in the correct direction in later iterations as seen in Figure 11.4.

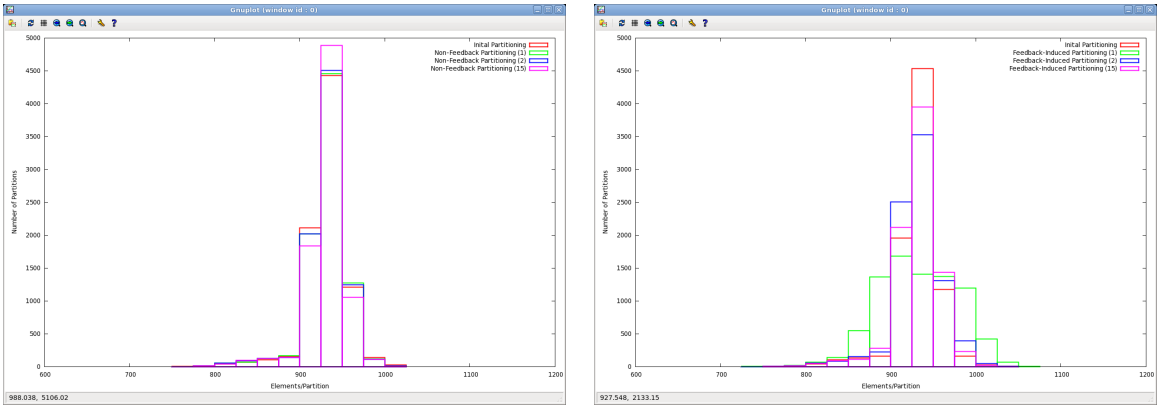


Figure 11.4: Partition Distributions for selected timesteps for an 8310 processor run of an Aria example on CDS. Processors exhibiting larger ratio of idle cycles to total cycles utilized since the last partitioning are assigned a larger partition size. Uniform partition sizes (left) results in tighter distributions than those without feedback. In the feedback case, when too broad a partitioning occurs early (right, green), the partitioning feedback criteria will adjust the distribution (right, blue and magenta).

These two cases illustrate that run time evaluation and feedback are feasible at the targeted scale. The performance benefit of this work to an application is contingent upon reasonable determination of the variables and the functional form for contention, imbalance, and resulting partition sizes. Such tuning is necessarily an iterative process.

11.3.4 Fuego measures of interest, feedback, and results

As with the Aria application described above, for our Fuego example we examined process to process metric differences across multiple runs, examining CPU resource utilization, context switches, and interrupts in particular. Again we saw the contention due to context switches and interrupts but further saw a wider distribution of workload across processes as evidenced by the distribution of idle CPU cycles across processors. This distribution was also seen to vary over time due to the dynamic nature of the Fuego simulation application as described in Section 11.3.1.

11.3.4.1 Dynamic feedback with Fuego

We again used feedback to Zoltan to rebalance a 64 processor Fuego example run. In this case the object weights were the particle numbers and, for partition size, we used the imbalance function of Section 11.3.3.2. As particles move in time the particle-to-partition mapping changes and the distribution is spread out, thus triggering a rebalancing. Figure 11.5 shows the resultant particles per processor distribution at various stages in the problem evolution with and without feedback.

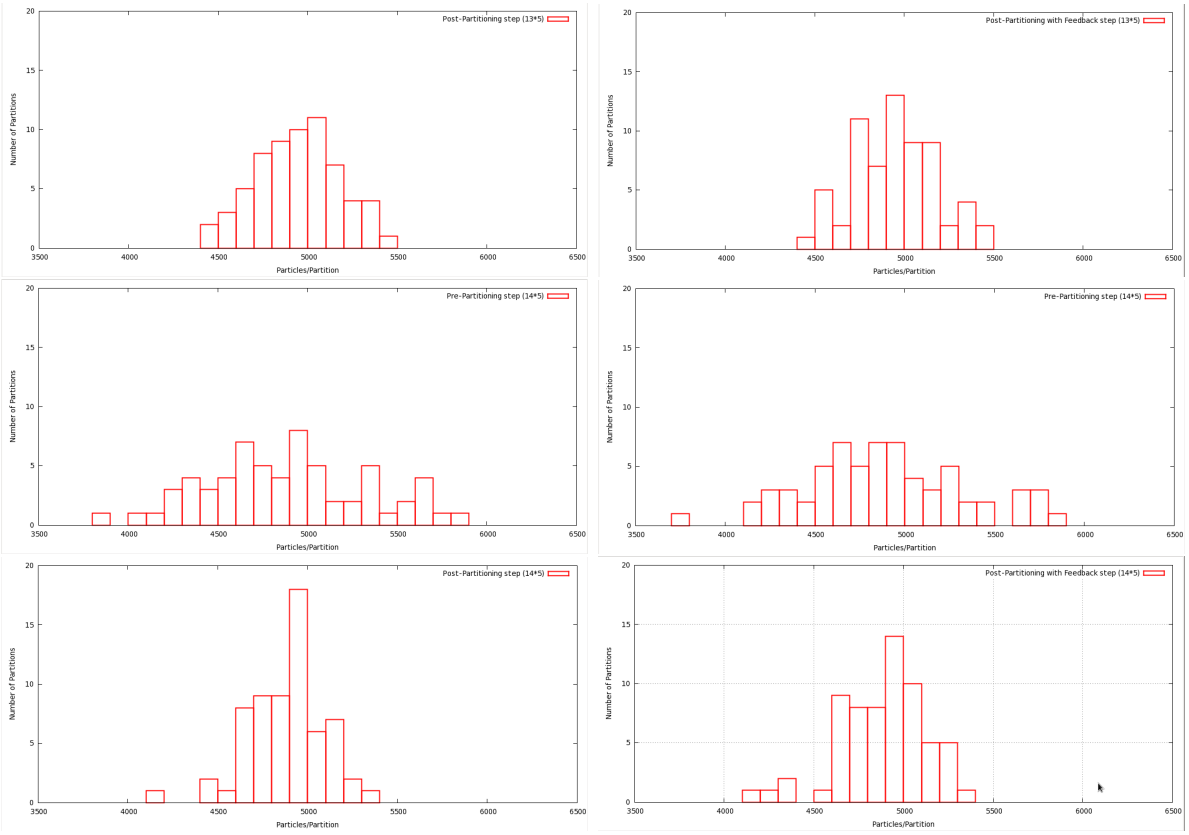


Figure 11.5: Particle Distributions in the Partitioning Evolution in a Fuego problem (a) after a rebalance (top) (b) after a few timesteps before the next rebalance (middle) (c) after the next rebalance (bottom) (d) without feedback (left column) (e) with feedback (right column). Rebalancing in general seeks to provide an even distribution but is not in practice completely uniform (top and bottom). As the problem evolves and particles move the distribution widens triggering the need for rebalancing (middle).

Figure 11.6 shows total fractional utilization of available CPU cycles for each core used in the computation. This figures shows that 1) that even without feedback (green), when particle number is used as object weight for partitioning, there is still imbalance of computational cycles across processes during the run and 2) in this case, even in the absence of significant imbalance and without fine tuning there is a resultant 1% improvement in computational cycles dedicated to the application for each of the processors involved. There is a corresponding 1% decrease in run-time for the entire application. Again, detailed determination

of the functional form for the repartitioning calculation was beyond the scope of this work.

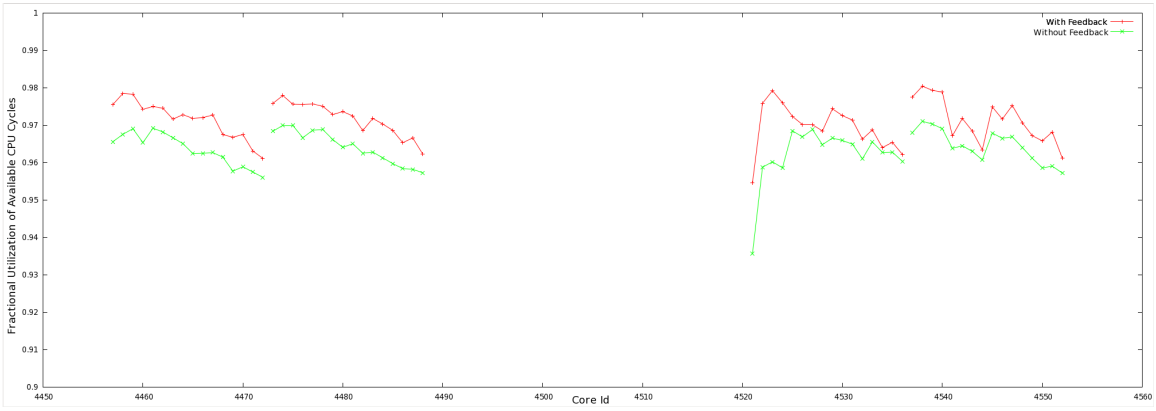


Figure 11.6: Simple demonstration of resource evaluation and feedback shows improvement in computational cycles across all processors involved. Greater fractional utilization of CPU cycles is achieved with feedback than without (green) Note: Y axis is the fractional utilization of available CPU cycles and goes from 0.9 to 1.0. X axis is a core unique identifier.

11.4 Conclusion

We have developed tools to enable new capabilities for Dynamic Resource-Aware Computing that require no changes to existing application codes. These tools provide resource/application monitoring, run-time analysis, and dynamic run-time application feedback and satisfy the constraints of low resource impact, scalability, and responsiveness.

In this work we demonstrated a system to not only enable understanding of how applications are utilizing platform resources, but to also enable run-time modification of their use. Fine tuning of the rebalancing was beyond the scope of this work. These monitoring, analysis, and feedback mechanisms can be used to evaluate any system measurables (we are currently investigating memory and communication subsystems) and inform and/or trigger any system responses available.

Additional potential use cases include better initial application placement based on application communication profiles determined via profiling, which would leverage work in Chapter 5, and dynamic application reconfiguration triggered by discovery of contention with other applications for resources or upon detection of failing or degrading resources, which would build upon our previous research work in triggering virtual machine migration response [18] as part of Sandia's ASC Resilience efforts and which could leverage work in Chapter 12.

Finally, better understanding of application needs and the appropriateness of resource fit that can be obtained via this work can also be used to drive better design for future platforms.

Chapter 12

A Scalable Virtualization Environment for Exascale

Abstract: Hardware-accelerated virtualization support has become ubiquitous on modern computing platforms, driven primarily by the requirements of the data center and cloud computing industries. This technology also has compelling use-cases for exascale computing, such as using guest operating systems to deploy specialized compute node functionality on-the-fly (e.g., for visualization or in-transit data analysis), providing backwards-compatibility to existing MPI applications running on exascale systems via "legacy" virtual machine environments, and leveraging the full-system view of the virtualization layer to facilitate the mapping of abstract machine models to the available physical hardware. As a step towards realizing these use-cases and others, we have been developing a scalable virtual machine monitor that augments the traditional fixed software stack provided by vendors. We have deployed this system on Red Storm and evaluated the performance of the CTH and Sage applications running inside of a virtual machine on up to 4096 compute nodes. The results show that the overhead induced by our virtualization layer is low, typically less than 5%, proving the feasibility of running communication intensive, tightly coupled applications in a virtual machine environment.

12.1 Introduction

Current supercomputers provide a fixed software environment that is typically limited in its functionality for performance and scalability reasons. Examples include the Red Storm system at Sandia and the BlueGene/L system at LLNL, which both use custom lightweight kernel (LWK) operating systems that provide only the functionality needed to support scalable parallel applications. If an application requires more functionality than is available, it must be reworked to eliminate the dependencies or not be able to make use of the system.

Together with collaborators at Northwestern University and the University of New Mexico, we have been working to address this issue by developing a virtual machine monitor (VMM) that can be embedded in a traditional LWK operating system. Specifically, we have embedded the Palacios VMM developed at Northwestern inside of Sandia's Kitten LWK (Figure 12.1). The VMM capability enables users to dynamically replace the normal fixed

software environment with an environment of their choosing, without rebooting the system. Users with applications that require more functionality than what the LWK provides can use the VMM to "boot" a more full-featured "guest" operating system underneath (nested within) the native LWK. The application can then be launched inside of the guest environment and make use of the additional functionality it provides. From the LWK's perspective, the guest OS looks much the same as a native LWK process or thread.

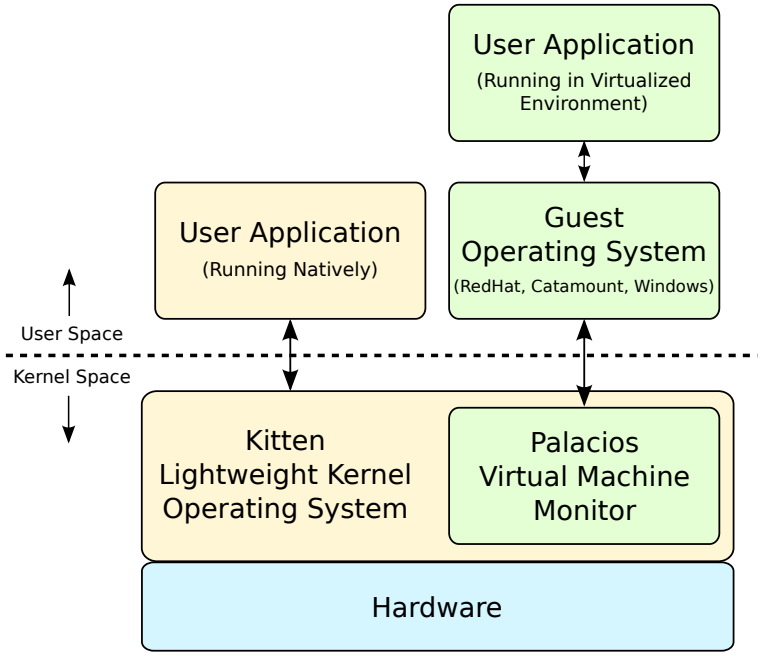


Figure 12.1: Block diagram of Kitten LWK running a user application natively alongside a user application running inside a virtualized environment.

We believe this combination provides a good balance of performance and flexibility. Applications that require the highest level of performance and scalability can continue to use the native LWK environment as they do today without penalty. When the native LWK environment does not provide enough functionality, if for example a compute node needs to have specialized system software available for visualization or in-transit data analysis, a guest operating system can be launched on-demand to provide the missing functionality. The added flexibility provided by the VMM mitigates one of the most frequently cited weaknesses of the LWK model – limited functionality.

The VMM capability enables other interesting use cases as well. System administrators could potentially test a new operating system release without having to take the target system out of production. Likewise, system software developers could perform large-scale testing without having to request dedicated system time. Security researchers could boot up many thousands of different commodity guest operating systems and, in effect, simulate the Internet. In the context of exascale computing, the VMM layer could be used to provide a "legacy" virtual machine environment to support unmodified MPI applications designed

for prior-generation supercomputers. The extra layer of indirection provided by the VMM layer makes it a powerful tool for mapping abstract machine models, such as a "legacy" distributed memory massively parallel machine model, to the available underlying physical hardware, which may be substantially different than what is exposed to software.

12.2 Application Results

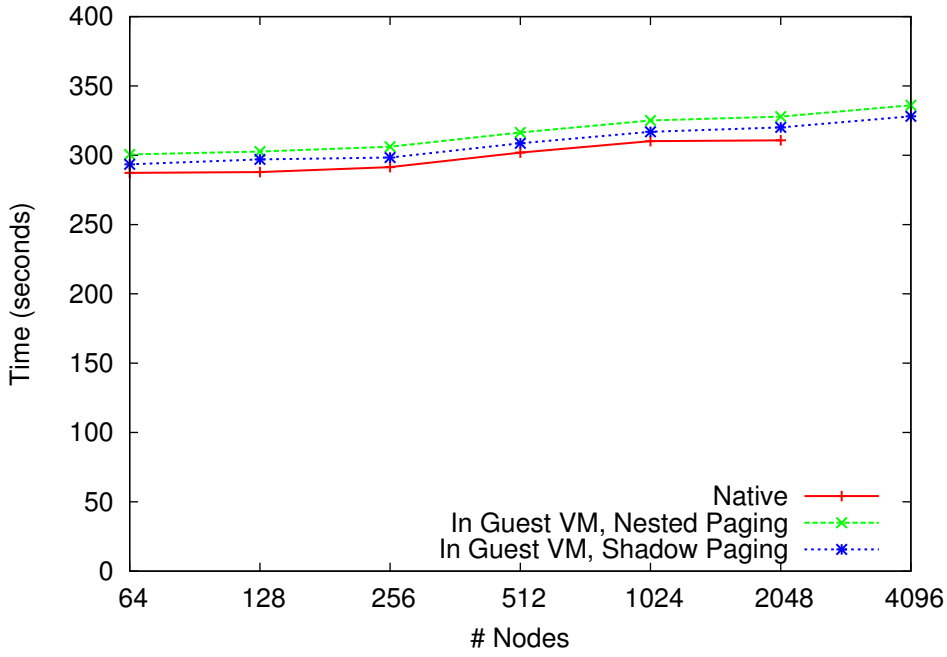
The price of the added flexibility afforded by the VMM is increased overhead in the virtualized environment compared to native execution. The extra level of indirection in the VMM and virtualization hardware necessarily adds some overhead. In order to evaluate this overhead we performed experiments on 2048 nodes of Red Storm with two real applications – the CTH [44] hydrocode and Sage [74] adaptive grid Eulerian hydrocode – comparing performance executing natively to performance executing within the virtualized environment provided by Kitten and Palacios. To the best of our knowledge, these are the only virtualization experiments performed on a peta-flop class architecture – Cray XT – and are by far the largest scale experiments performed to-date. Previous studies have used a maximum of 128 nodes and have only considered benchmarks, not real applications.

Figure 12.2 compares the performance of CTH and Sage executing natively on Red Storm, in the Catamount LWK, to their performance executing within Catamount running as a guest operating system on top of Kitten and Palacios. The results show that the virtualized environment adds less than 5% overhead in all cases, even at the highest scales tested. This demonstrates for the first time the feasibility of running communication intensive, tightly coupled applications such as these within a modern virtualized parallel environment.

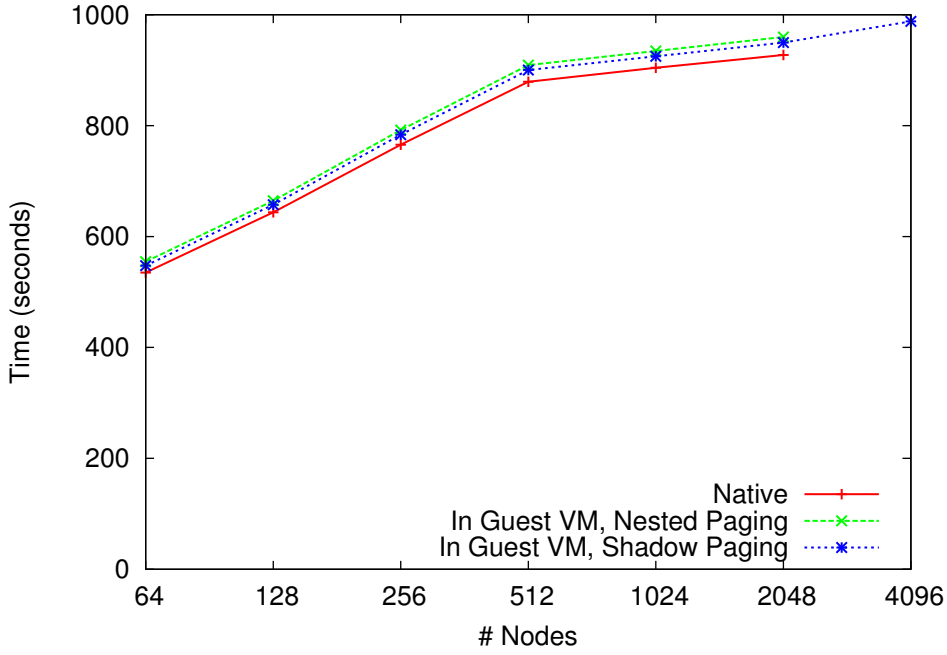
Additionally, we evaluated two memory management schemes for the virtualized environment. With shadow paging, the Palacios VMM is responsible for memory management (memory management is performed in software). With nested paging, virtualization hardware in the host processor performs memory management. The results indicate, counter intuitively, that software memory management is slightly faster than hardware memory management for these HPC workloads. This was found to be due to extra overheads in the processor's translation lookaside buffer (TLB) when using nested paging.

12.3 Conclusion

The end goal of this research is to provide users with a more flexible supercomputer environment without requiring the most scalable applications – those that work best in a LWK environment – to sacrifice performance. Given the promising results on Red Storm, we hope to deploy this technology in future capability supercomputer platforms. For more information about Kitten and Palacios, see [78].



(a) CTH



(b) Sage

Figure 12.2: Comparison of CTH (top) and Sage (bottom) applications running natively vs. in a virtualized environment with shadow (software) paging memory management vs. in a virtualized environment with nested (hardware) paging memory management. The results show that the virtualized environment introduces less than 5% overhead in all cases.

Chapter 13

Goofy File System for High-Bandwidth Checkpoints

Abstract: The Goofy File System, a file system for exascale under active development, is designed to allow the storage software to maximize quality of service through increased flexibility and local decision-making. By allowing the storage system to manage a range of storage targets that have varying speeds and capacities, the system can increase the speed and surety of storage to the application. We instrument CTH to use a group of RAM-based GoofyFS storage servers allocated within the job as a high-performance storage tier to accept checkpoints, allowing computation to potentially continue asynchronously of checkpoint migration to slower, more permanent storage. The result is a 10-60x speedup in constructing and moving checkpoint data from the compute nodes.

13.1 Introduction and Motivation

GoofyFS is a peer-to-peer-inspired file system for exascale under active development. It is part of a multi-year effort among Sandia National Laboratories, the University of Alabama at Birmingham, Clemson University, and Argonne National Laboratory. The motivation behind GoofyFS is that, as HPC systems (and their storage systems) grow, increasing component counts will imply a growth in the number of device failures. The storage system will be in some sort of failure mode at all times, requiring it to adapt dynamically to its own health. Peer-to-peer systems are well known for their ability to provide resilience by handling client turnover [131], and provide a ripe area of exploration to improve large scale storage systems.

Resilience and performance concerns influence the design of GoofyFS in a number of ways. First, data should be automatically migrated and replicated through the system to maintain data integrity. For example, if data is replicated among three servers, and one fails, data may be replicated immediately without central coordination to another server or set of servers, ensuring that at least three copies remain. Second, data may be relocated or replicated for performance reasons, including to and from a set of compute nodes that hold data in RAM as the fastest tier in a caching model. Finally, clients may write data to any server that provides the client with high quality of service quantified by bandwidth, reliability, I/O operations per second, latency, and so on.

Many DOE applications checkpoint frequently to defend against job failure. This is often a synchronous operation to disk-based storage, requiring all nodes to stop computing until the checkpoint has been placed in its ultimate location. Today's parallel file systems are not able to accept data at the speed compute nodes can produce it, so the application must spend significant time waiting for the checkpoint to complete. To improve performance, the storage system could recruit compute nodes to provide an intermediate fast store that can migrate checkpoints back to disk asynchronously, improving checkpoint performance for the job.

For this report, we demonstrate running GoofyFS storage servers on compute nodes as a first tier of fast storage. We deploy the GoofyFS nodes alongside a compute job running CTH, with CTH writing checkpoint data to GoofyFS. This experiment demonstrates the promise of GoofyFS: As development continues, this first tier will transfer the checkpoint data to slower, disk-based storage asynchronously as necessary, allowing bursty write patterns to experience higher performance than the underlying disk-based storage can provide.

13.2 Related Work

The tiered storage approach can be seen as a blend of several functionalities offered by other projects. These projects augment the underlying storage system to improve apparent performance or address shortcomings.

IOFSL [5] is an I/O forwarding layer that decreases the number of clients a file system must handle simultaneously. This is done by dedicating nodes to the task of accepting file system requests and performing them on behalf of the compute nodes. IOFSL does not provide asynchronous checkpoint transfer to the parallel file system, but operates synchronously.

SCR [92] is a library that provides a range of checkpoint strategies to applications. This includes having each node within a job checkpoint its data to the memory of a neighboring node, among other arrangements. SCR includes a notion of tiered checkpoints where only some checkpoints are stored to the parallel file system, while others remain resident in other storage tiers.

The PLFS-enabled burst buffer [16] is another piece of software that has similar goals. The burst buffers are special-purpose servers distributed throughout an HPC platform, e.g. one per compute rack. Each burst buffer server is filled with enough flash memory to accept a small number of checkpoints from nearby compute nodes. The data is asynchronously drained from the burst buffers to the parallel file system. PLFS [15] is used to maintain a consistent view of data regardless of its location.

13.3 The GoofyFS Storage Server and Client

The GoofyFS storage server provides an object-based storage API. Each object has a 128-bit identifier and a 64-bit address space. The storage server supports sparse objects and conflict resolution of objects via update identifiers. The API is RDMA-based, as the network layer (Single-Sided Messaging, or SSM [51]) is heavily influenced by the Portals 4.0 specification [120]. For this experiment, SSM used an MPI-based transport layer, but others are under active development.

No generally useful POSIX-like client exists for GoofyFS. While a FUSE-based client has been demonstrated as a proof-of-concept, it is not meant for use by applications. Instead, a simple custom client was crafted for CTH that stores checkpoint data with a file per object, with a one megabyte local buffer to ensure that larger messages are sent to the storage server as needed. Because CTH's checkpoints have predictable names based on the iteration and rank of the process, a job ID and rank can be used to derive a unique object ID. A more friendly POSIX client for HPC is under development.

13.4 Instrumenting CTH

To give CTH access to GoofyFS servers in the Cielo application environment, we launched extra processes within the CTH allocation specifically dedicated to GoofyFS. Where CTH calls `MPI_Init`, we inserted a call to our own initialization routine that created a global communicator, splitting the nodes between a CTH group and a GoofyFS group. The GoofyFS processes were instructed to enter an event loop to satisfy storage requests, while the CTH nodes were released to participate in the compute job normally. A number of operations CTH performed over and within `MPI_COMM_WORLD` had to be changed to use the new communicator.

Other changes were less invasive. To have CTH use GoofyFS's storage protocol, we leveraged syncio, an alternative I/O infrastructure already present in CTH. Syncio optionally coordinates I/O from nodes so that storage servers in certain previous platforms would not be overloaded from many simultaneous I/O requests from CTH. Syncio's API includes read, write, open, and close, creating a straightforward mapping of common I/O tasks. As Cielo does not require syncio functionality, we created simple replacements of those routines that would interact with the GoofyFS storage servers directly. A small number of further changes had to be made to routines that were not written to use syncio.

We evenly distributed nodes running GoofyFS processes throughout the CTH job. Specifically, we treat the nodes in the job in groups of five, where each group k contains the process IDs between $k \times 80$ and $(k+1) \times 80 - 1$, inclusive. The lowest 16 ranks, which are located in the first node of the group, run GoofyFS server instances. The highest 64 ranks, which comprise the other four nodes of the group, run CTH processes.

The CTH processes are directly mapped to the GoofyFS processes in the first node of

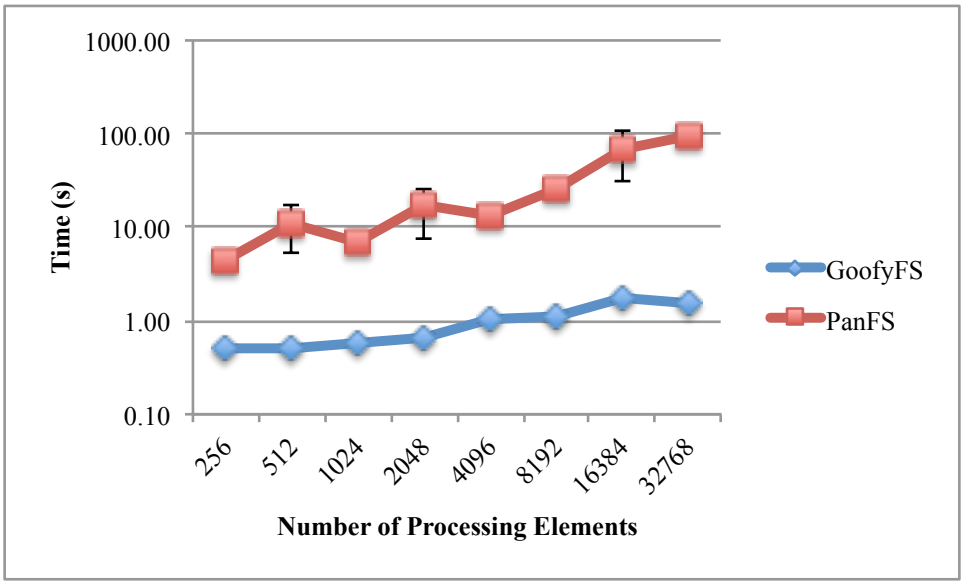


Figure 13.1: Average checkpoint time for GoofyFS versus PanFS. All points include error bars indicating a 99% confidence interval, but may not be visible for data points that had little variance. Both axes use log values.

its group. For example, if a GoofyFS process in node n has rank $n \times 16 + i$, it will service the CTH processes found in ranks $(n + 1) \times 16 + i$, $(n + 2) \times 16 + i$, $(n + 3) \times 16 + i$, and $(n + 4) \times 16 + i$.

13.5 Application Impact

All tests were conducted on Cielo, a Cray XE6 that is used as a capability platform at Sandia National Laboratories, Lawrence Livermore National Laboratory, and Los Alamos National Laboratory. The job was run with the shaped charge example problem at scales varying from 256 to 32,768 cores. Baseline runs were completed with CTH writing checkpoints directly to the 10PB Panasas storage, which uses PanFS as the file system. The GoofyFS runs require extra cores for storage servers, so each GoofyFS run included 25% more nodes than the CTH job required. For example, the 256 core job had 320 cores requested in the allocation, with 64 cores used exclusively as GoofyFS storage servers. All tests were run through at least four checkpoints, and we recorded the average of the first four checkpoints. The time interval between checkpoints was specified to be small, but varied between thirty seconds to two minutes among different batches of runs.

Figure 13.1 gives the results of this experiment. Checkpoint performance was increased by at least 10x, with larger jobs demonstrating up to a 60x performance increase. One significant feature of Figure 13.1 is the variance experienced by each storage platform. These benchmarks were run while the machine was in general use, so it is likely that there was occasional significant contention for PanFS I/O services. The GoofyFS partition, because it

is dedicated to a single job, displayed almost no variance.

While these jobs used a rather large 25% increase in allocation of nodes per job, this was a decision made to ensure that each node had enough space available for checkpoint storage. CTH compresses its checkpoints, causing some processes to write much more data than others. With a good mechanism for load balancing, and implementation of data migration capability, it may be possible to exploit a much smaller fraction of servers with good performance.

13.6 Conclusion

This demonstration of early GoofyFS functionality shows a significant benefit for a real I/O workload, checkpointing, in a real application, CTH. By running GoofyFS storage servers within a job as RAM-only stores, CTH was able to store checkpoints 10-60x faster than storing to PanFS, allowing the job to continue computing sooner. While this prototype did not include automatic data migration, the checkpoint was available to be pushed or pulled to disk-based storage as needed after the compute nodes continued computing. Future developments include the ability to dynamically spawn GoofyFS nodes to absorb checkpoints, expanding this mechanism to other fast tiers of storage like flash memory, and sharing of dynamic GoofyFS nodes between multiple jobs as needed.

Chapter 14

Exascale Simulation - Enabling Flexible Collective Communication Offload with Triggered Operations

Abstract: In addition to software techniques, it is also possible to provide improved capabilities in the system architecture to allow more efficient application execution, without requiring code changes. One area where this is possible is in the network interface controller (NIC) architecture. Specifically, it is possible to offload collective communications to the NIC, providing both lower latency and higher tolerance to system noise, which will both lead to better application performance and scaling. We used the Structural Simulation Toolkit (SST) to simulate the performance of the `MPI_Allreduce()` collective using semantic building blocks of the Portals 4 communications API. The simulations (run up to 32,768 nodes in size) show that offloading the collective to the NIC provides for a nearly 40% reduction in latency, and more importantly, provides for a much higher degree of tolerance to varying types of system noise (simulated using varying durations and frequencies of noise). These benefits are seen across a wide range of network performance parameters.

14.1 Introduction

Collective communications are widely used in MPI applications, because applications frequently perform global computations using data from all of the nodes. When the result must be distributed to all nodes (e.g. `MPI_Allreduce()`), synchronization is introduced that can expose the application to OS interference[114] (also called OS noise or OS jitter). As system sizes increase, collective communications inherently take longer due to more computation steps, communication steps, and time of flight across the physically larger machine. Unfortunately, applications become even more sensitive to scalability challenges as the system size increases. Improvements in the performance of collectives will clearly be needed for future systems.

One approach to accelerating collectives is the use of offload onto the network interface controller (NIC). In addition to accelerating the collective operation itself, offloading can reduce the impact of OS interference during collective execution. While offload has many

attractive aspects, current implementations show some drawbacks as well. Network adapters are typically not easy to program to experiment with new collective algorithms. In many cases, the programmable resources that are available have significantly lower performance than the host processor[93, 136]. Rather than adding fixed-function hardware to implement a single collective algorithm, it is desirable to expose a semantic building block that can implement multiple collective algorithms for multiple upper-level libraries, including MPI. Given all of the potential factors used in determining an optimal collective algorithm, such as process arrival[50], the ability to implement the algorithm on the host while offloading a collective can be advantageous.

We recently incorporated the concept of triggered operations, including triggered remote writes and triggered remote atomic operations, into the Portals 4[121] network programming interface. Triggered operations allow an application to setup a communication operation that will be issued in the future by the NIC when a trigger condition is met. Triggered operations require a new type of event to provide efficient tracking of message completion to use as a trigger condition. We introduce counting events that can be used in combination with a threshold to trigger operations. As soon as an application initiates a collective communication, the MPI library on each node can set up all message transactions that will be required for the entire collective operation and then simply wait for completion of all of the message requests. This approach allows the application to provide the algorithm and higher-level functions, while the NIC offloads the message processing and arithmetic operations. The flexibility of host-based algorithms and the host-based development cycle are maintained with the performance advantages of collective offload.

One commonly used collective operation is `MPI_Allreduce()`[136, 93]. `MPI_Allreduce()` is frequently used for a small number of data items (often only one) with a simple arithmetic operation (often minimum or maximum). This paper describes the triggered operations needed to implement `MPI_Allreduce()` and how `MPI_Allreduce()` is implemented using those operations. A performance analysis is used to explain how the use of triggered operations can impact collective algorithm design. In addition, simulations are used to present the performance impact of collective offload using triggered operations, including how the tolerance to OS noise of the collective operation is impacted by offload.

14.2 Triggered Operations in Portals 4

Portals 4[121] added two semantic building blocks to support the efficient offload of collectives. Counting events enable lightweight tracking of operation completion. Triggered operations leverage counting events to defer the start of an operation until a threshold is reached. Together, these constructs allow software on the host to chain new outgoing operations as responses to incoming operations.

The semantics of counting events are simple: when an operation completes, a counter is incremented in application memory. Counting events can be associated with any operation.

Triggered operations leverage counting events to allow the application to create dependency chains through several operations. The standard form of a triggered operation is to add a counting event handle and threshold value to the standard arguments for the corresponding Portals call. A triggered operation can be passed from the application to the network interface when the triggered operation call occurs, but the network interface does not retrieve the data from memory and send the message until the associated counting event reaches the specified threshold.

14.3 Evaluation Methodology

The Structural Simulation Toolkit (SST) v2.0 was used to simulate both a host-based and an offloaded version of two algorithms that implement `MPI_Allreduce()`. A torus network of up to 32K nodes ($32 \times 32 \times 32$) was simulated. Simulations were run with and without simulated OS interference to determine the success of offloaded implementations in eliminating noise.

14.3.1 Algorithms for Allreduce

Many applications use `MPI_Allreduce()` to perform a global operation on floating-point inputs. Unfortunately, floating-point numbers are not associative. As such, there are two algorithms that are prevalently used for `MPI_Allreduce()` to insure that all participants receive the same result: a tree-based algorithm and a recursive doubling algorithm.

The classic allreduce implementation is a reduction followed by a broadcast, using a tree-based communication topology. We used a binomial tree, in which nodes toward the top of the tree do more work than nodes toward the bottom of the tree. The binomial tree proved more efficient in both host-based and triggered implementations than a strict binary tree.

The recursive doubling allreduce algorithm [116] uses a butterfly communication pattern and redundant calculations to reduce the number of communication steps compared to a tree algorithm. The recursive doubling algorithm is an optimization for small input vectors, where the cost of communication is latency dominated and outweighs the cost of redundant calculations. Whereas the binomial tree algorithm has $2log(N)$ communication steps with each step involving a smaller number of processors, the recursive doubling algorithm performs $log(N)$ steps, with each step involving every node.

14.3.1.1 Implementation with Triggered Operations

Figure 14.1 illustrates the functions used in the pseudo-code examples. Only the subset of the standard Portals arguments that are most relevant are shown in Figure 14.1. As shown in the figure, data movement operations take a source memory descriptor (a handle to a local memory region). The "Portal Table Index" is used like a protocol switch — conceptually

```
PtlTriggeredAtomic(
    source_md_h, //source of data on local node
    destination, //target node ID
    pt_index, //Portal table index (target buffer)
    op, dtype, //operation and datatype
    trigger_ct_h,//counting event to trigger on
    threshold); //trigger threshold

PtlTriggeredPut(
    source_md_h, //source of data on local node
    destination, //target node ID
    pt_index, //Portal table index (target buffer)
    trigger_ct_h,//counting event to trigger on
    threshold); //trigger threshold

PtlAtomic(
    source_md_h, //source of data on local node
    destination, //target node ID
    pt_index, //Portal table index (target buffer)
    op, dtype); //operation and datatype

PtlTriggeredCTInc(
    modify_ct_h, //counting event to increment
    increment, //increment value
    trigger_ct_h, //counting event to trigger on
    threshold); //trigger threshold

PtlCTWait(
    wait_ct_h, //counting event to wait on
    threshold);//wait until this threshold
```

Figure 14.1: Function definitions for Portals pseudo-code

similar to a TCP/IP port — in Portals and is used to segregate phases of an algorithm. Atomic operations take an operation and datatype argument, which includes such things as double precision floating-point. Finally, triggered operations require arguments for the counter to be monitored and the threshold at which it triggers.

Figure 14.2 shows implementations of the tree-based and recursive doubling allreduce algorithms using triggered operations. In the tree-based algorithm, Leaf nodes push contributions directly to their parent node and wait for an answer to arrive. Other nodes atomically add their local contribution to a temporary buffer, and when data is received from all children, send the aggregated contribution to their parent node. In the recursive doubling algorithm, each pair of participants at level A add their contributions in bounce buffer A, and a triggered operation atomically adds that answer to bounce buffer $A + 1$ for the next iteration. After $log(N)$ iterations, each node has the final result of the allreduce.

14.3.2 Structural Simulation Toolkit v2.0

The Structural Simulation Toolkit v2.0 (SST) [133] provides a component-based simulation environment designed for simulating large-scale HPC environments. The simulator is designed to be flexible enough to simultaneously provide both cycle-accurate and event-based simulation environments. The simulations presented utilize a cycle-accurate router network model, based on the Cray SeaStar router, combined with a high-level, event-driven network interface and host processor model.

14.3.3 Simulated Architecture

The network interface architecture that was simulated is shown in Figure 14.3. Portions of Portals processing is offloaded to the network interface, including the processing of incoming

```
// compute parent node and list of child nodes
num_children = 0
for (i = 1 ; i <= num_nodes ; i *= radix) {
    parent = (id / (radix * i)) * (radix * i);
    if (parent != id) break;
    for (j = 1 ; j < radix ; ++j)
        children[num_children++] = id + i + j;
}
// push data up tree
if (num_children == 0) {
    // leaf node: send data up tree
    PtlAtomic(user_md_h, parent, 0, op, dtype);
} else {
    PtlAtomic(user_md_h, id, 0, op, dtype);
    if (0 == my_id)
        // setup trigger to move data to right place,
        // then send down tree
        PtlTriggeredPut(up_tree_md_h, my_id, 1,
                        up_tree_ct_h, num_children);
    else
        // setup trigger to move data up the tree when
        // have enough updates
        PtlTriggeredAtomic(up_tree_md_h, parent, 0,
                           op, dtype,
                           up_tree_ct_h, num_children);
    // reset counter
    PtlTriggeredCTInc(up_tree_ct_h, -num_children,
                      up_tree_ct_h, num_children);
}
// push data back down tree
for (i = 0 ; i < num_children ; ++i)
    PtlTriggeredPut(user_md_h, children[num_children],
                    1, user_ct_h, 1);
// wait for local data buffer to be updated
PtlCTWait(user_ct_h, 1);
```

Tree

```
// push local data to local and remote buffer
PtlAtomic(user_md_h, id, 0, op, dtype);
PtlAtomic(user_md_h, id ^ 0x1, op, dtype);
// setup butterfly communication pattern
for (i = 1, level = 0x1 ; level < num_nodes ;
     level <<= 1, ++i) {
    remote = id ^ level;
    // when data arrives in buffer A - 1, copy into both
    // local buffer A and current peer's buffer A
    PtlTriggeredAtomic(level_md_hs[i - 1], id, i, op,
                       dtype, level_ct_hs[i - 1], 2);
    PtlTriggeredAtomic(level_md_hs[i - 1], remote, i, op,
                       dtype, level_ct_hs[i - 1], 2);
    // reset counter for buffer A
    PtlTriggeredCTInc(level_ct_hs[i - 1], -2,
                      level_ct_hs[i - 1], 2);
}
// copy the answer into user buffer
PtlTriggeredPut(level_md_hs[levels - 1], id, 1,
                level_ct_hs[levels - 1], 2);
// reset counter for final buffer
PtlTriggeredCTInc(level_ct_hs[levels - 1], -2,
                  level_ct_hs[levels - 1], 2);
// wait for completion
PtlCTWait(user_ct_h, 1);
```

Recursive Doubling

Figure 14.2: Pseudo-code for allreduce algorithms based on Portals 4 triggered operations.

floating-point atomic operations. Triggered operations are enqueued on the NIC, where they await their triggering threshold. As new messages complete, the counting events are incremented on the NIC and these increments cause pending triggered operations to issue into the network. The key to the performance of triggered operations is the ability to service floating-point operations with low latency and to issue pending triggered operations at extremely high rate when their triggering threshold is reached. Since this set of capabilities does not exist on modern network interfaces, we simulate an architecture with an embedded floating-point unit and a NIC-side unit to manage the queuing and issuing of triggered operations.

14.3.4 Simulation Parameter Definitions

We used a range of simulation parameters to represent the relevant space for near-term networks. Specifically, "back-to-back" message latencies were simulated at 1 μs, and 1.5 μs,

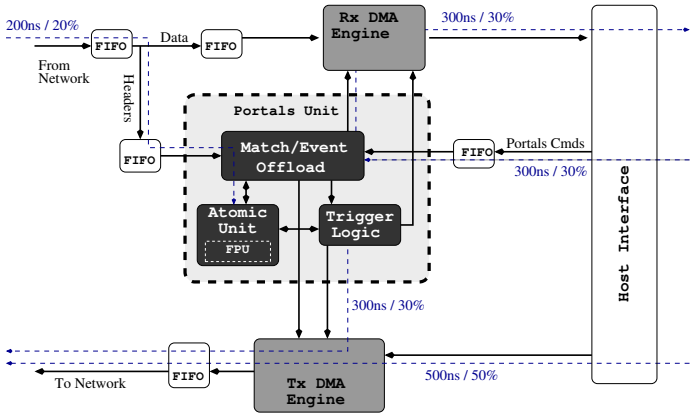


Figure 14.3: High level NIC architecture annotated with key portals simulation timings. Timings are shown as a percentage of back-to-back latency along with absolute values for 1000 ns latency.

not including router time, to cover the range from current network latencies to the most aggressive latency targets. The latency is divided between transmit side and receive side with 50% going to each. Unidirectional message rates of 2.5, 5, and 10 million messages were simulated representing realistic message rates today[11] to the largest practical message rate for a single core[1]. We assume that the message rate is bound by overhead (software time), and place this overhead on both the transmit and receive sides. The various latencies used by the simulation model are shown as annotations of the high level NIC architecture in Figure 14.3.

Setup time for the collective operation — time needed by MPI before communication starts to setup the algorithm — is set at 200 ns. In addition, two other assumptions about the system do not appear directly in the simulation, but do affect the choice of internal simulation parameters. The cache line size is 64 bytes, and the cache miss penalty is 100 ns. The parameters used in the simulation are summarized in Table 14.1.

In addition to "no noise" scenarios, we evaluate three noise signatures to assess the impact of offload on noise sensitivity. Previous research into the effects of OS noise suggest two forms of "long noise at infrequent intervals" [54]. Our work adds a third signature to assess the finer-grained uncertainty in the system. In the baseline model, the node is perfectly deterministic with no variability in timing. Modern CPUs are not perfectly deterministic when the processor interacts with I/O, so we add a very fine-grained noise signature to explore the potential impact of such variability.

[1]Assuming modern network interface architecture, where each command requires an `sfence()` memory flush operation.

Table 14.1: Summary of Simulation Parameters

Property	Range
Message Latency	1000 ns, 1500 ns
Message Rate	2.5 Mmsgs/s, 5 Mmsgs/s, 10 Mmsgs/s
Overhead	$\frac{1}{MsgRate}$
NIC Message Rate	62.5 Mmsgs/s
Router Latency	50 ns
Setup Time	200 ns
Cache Line	64 Bytes
Cache Miss Latency	100 ns
Noise Signatures	25 μ, 2.5 ms @ 10Hz

14.4 Results

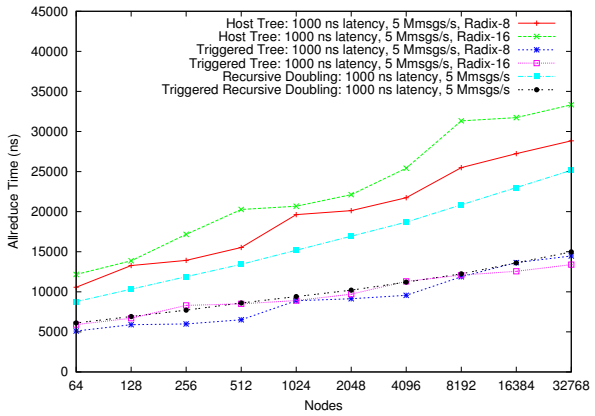
Results from the simulation of several configurations with no noise plus one configuration with three noise profiles are shown here. For each configuration, the recursive doubling algorithm for both the host and triggered operations is compared to four tree configurations: host-based at the optimal host radix, host-based at the optimal triggered operation radix, triggered operation-based at the optimal host radix, and triggered operation-based at the optimal triggered operation radix. While only one configuration is presented for the three noise profiles, all of the configurations exhibit similar behavior under noise.

14.4.1 Impact of Triggered Operations

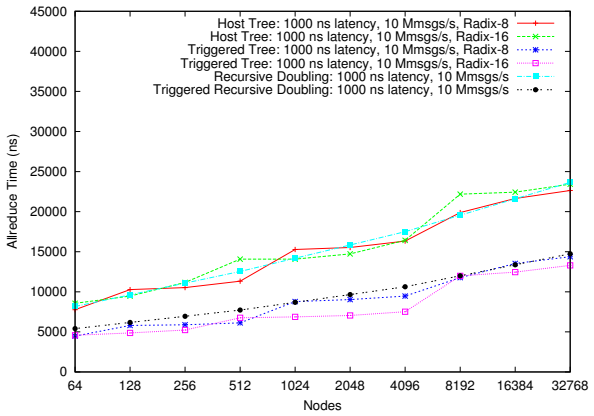
Figures 14.4 and 14.5 show two additional interesting impacts of the relationship between latency and message rate. First, they clearly illustrate that the ratio of message rate to latency is the driver that determines whether a tree-based algorithm or a recursive doubling algorithm is best. Second, the rate of posting messages is a much bigger contributor to the performance of the recursive doubling algorithm over triggered operations than it is on the host. This is both because the effective latency of the triggered operation is substantially lower and because the collective over triggered operations has one extra message per step in the algorithm (the triggered operation to self).

14.4.2 Impact of Offload on Noise Sensitivity

Figure 14.6 compares the various algorithms using a latency of 1000 ns and a message rate of 10 million messages per second at the optimal radix for each implementation under two noise profiles. Each profile corresponds to 2.5% interference from noise, but the profiles represent different combinations of duration and frequency[54]. The most striking result in

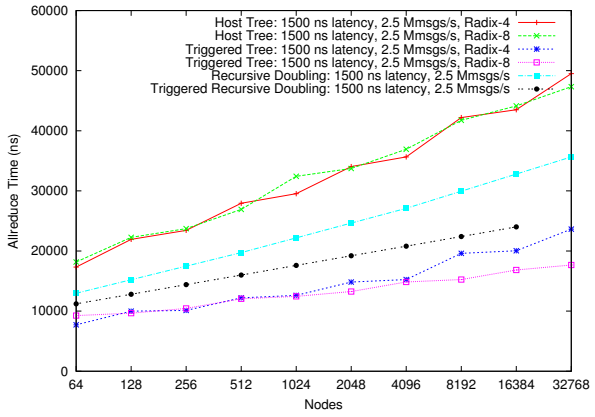


5 Million Messages per second

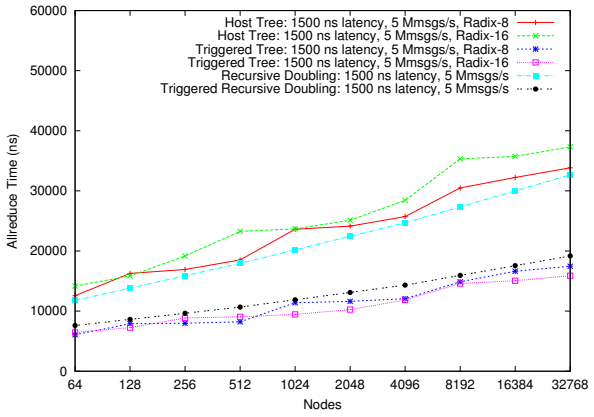


10 Million Messages per second

Figure 14.4: Allreduce time for 1000 ns message latency. Simulation results show that triggered operations provide approximately 30% better allreduce latency over host based implementations.

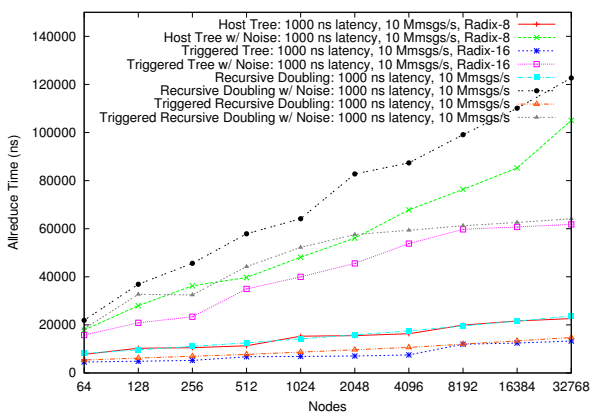


2.5 Million Messages per second

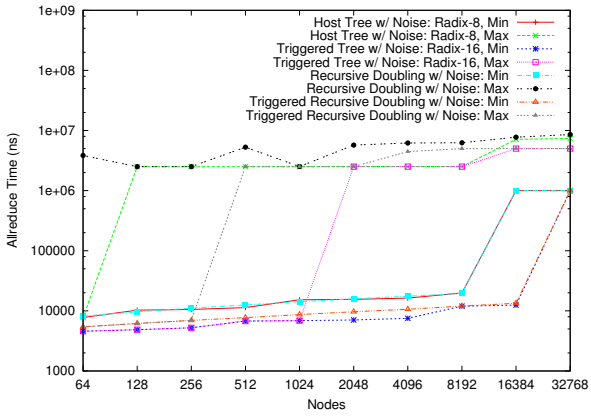


5 Million Messages per second

Figure 14.5: Allreduce time for 1500 ns message latency. Simulation results show that triggered opertions provide greater than 40% better allreduce latency compared to host based implementations.



25 μs @ 1kHz (note log scale on Y-axis)



2.5 ms @ 10Hz (min, max)

Figure 14.6: Allreduce performance under varying noise signatures. Simulation results show that offloaded triggered operations are much less sensitive to various types of noise than comparable host based algorithms.

these graphs is that the recursive doubling algorithm is highly susceptible to interference from noise. While the shortest noise duration has almost no impact on the tree algorithms, the recursive doubling algorithm is noticeably slower. As the length of the noise duration increases to 25 μs, the impact on the recursive doubling algorithm becomes dramatic. These impacts arise from the fact that the recursive doubling algorithm involves every node in every step of the algorithm, which makes the time for each step dependent on every node.

Triggered operations improve the noise resistance of the algorithms, but they do not eliminate the noise impacts. Noise resistance is improved by reducing the involvement of the host processor in the communication as well as shortening the total time for the collective. This does not eliminate the noise impact, since the host spends a substantial amount of its time posting messages to be transmitted. There are two particularly notable points. First, with a 25 μs noise duration, the triggered operation-based algorithms appear to reach a plateau in their noise response, while the host-based time continues to rise. Second, with a noise interval of 2.5 ms, the host-based collectives first encounter noise impact at system sizes 16× smaller than the triggered operations. The minimum and maximum times are graphed to show that at large scale, *every* host collective is impacted by noise and some receive three noise interrupts. In contrast, even at the largest scale, some triggered collectives are not interrupted by noise and none receive more than two noise interrupts. It is possible that adding an ability to aggregate triggered operation requests would decrease the time they spend posting messages sufficiently to further reduce noise sensitivity.

14.5 Conclusions

This section presents the triggered operations that were added to the Portals 4.0 API. Algorithms using those triggered operations were illustrated and implemented in simulation. The results highlight how changes in underlying assumptions can impact the choice of algorithm. Specifically, the increased operation rate that can be achieved using offload can increase the radix of a tree algorithm and make it more competitive with recursive doubling algorithms. The impact of noise on the collective algorithms is also striking. Recursive doubling is clearly more sensitive to noise, with the noise time dominating the collective time in many cases. Triggered operations are able to reduce, but not eliminate, the sensitivity of collective operations to OS interference.

Chapter 15

Application Scaling

Two application codes, representative of a broad set of application configuration strategies, provided us with a means for identifying some performance-limiting characteristics that we expect to be issues as machine and applications scale to exascale.

CTH is implemented using a Bulk Synchronous Parallel (BSP[134]) programming model for regular, nearest neighbor communication. Charon solves a sparse linear system of equations, sending many relatively small messages across an unstructured mesh.

As the number of processors increases to the scale of interest in this report, we find that applications can expect to encounter performance issues related to the computing environment, system software, and algorithms. Careful profiling of runtime performance will be needed to identify the source of the issue, in strong combination with knowledge of system software and application source code.

15.1 Boundary Exchange with Nearest Neighbors

CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories [66]. CTH has models for multi-phase, elastic viscoplastic, porous and explosive materials, using second-order accurate numerical methods to reduce dispersion and dissipation and produce accurate, efficient results. Written using the Fortran programming language, parallelism is enabled by MPI.

Computation is characterized by regular memory accesses, and is fairly cache-friendly, with operations focusing on two dimensional planes. Inter-process communication aggregates internal-boundary data for all variables into user managed message buffers, subsequently sent to up to six nearest neighbors.

We begin the analysis with a case study of an important simulation at problem scales previously unavailable. Experiences from this work then informs a deeper dive on performance issues using a miniapp from the Mantevo suite, and then an additional study considering a hybrid MPI + OpenMP implementation.

15.1.1 A Case Study at High Processor Counts

Within the context of a CCC project, titled "Critical Asset Protection Security: Lightweight, Blast Resistant Structure Development", we examined the performance of a 32K core CTH simulation. The simulation helps designers at Sandia understand the response of structures under severe blast loading conditions so that the robustness of these structures may be improved. As shown in Figure 15.1, a structure composed of sheet metal is loaded by an

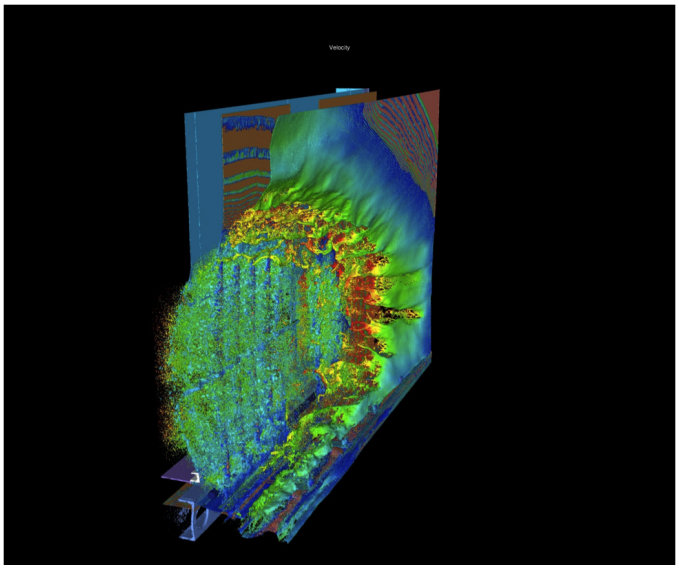


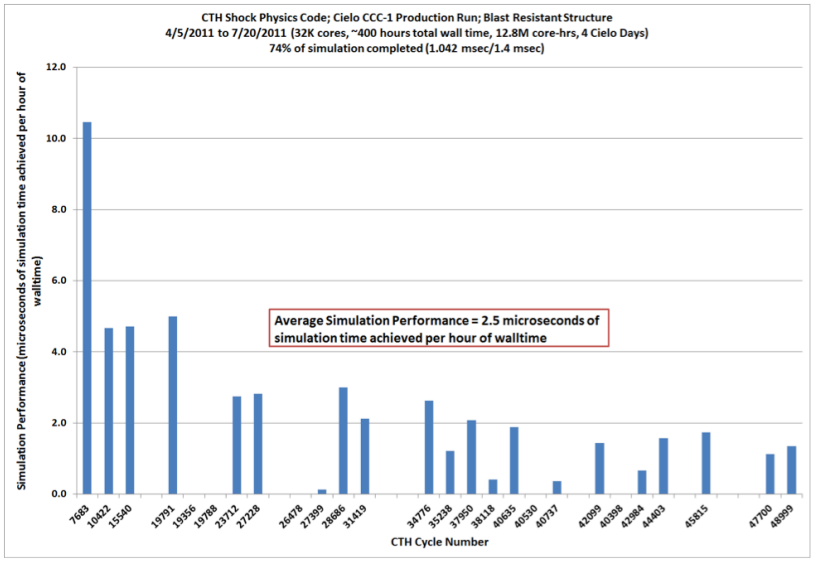
Figure 15.1: CTH: Multilayered thin-walled structure torn apart by an explosive blast

explosive blast. The plot gives the velocities of various components of the structure as they are torn apart. Accurately resolving these thin parts required a very fine mesh resolution which can presently be achieved using only very large scale computing platforms.

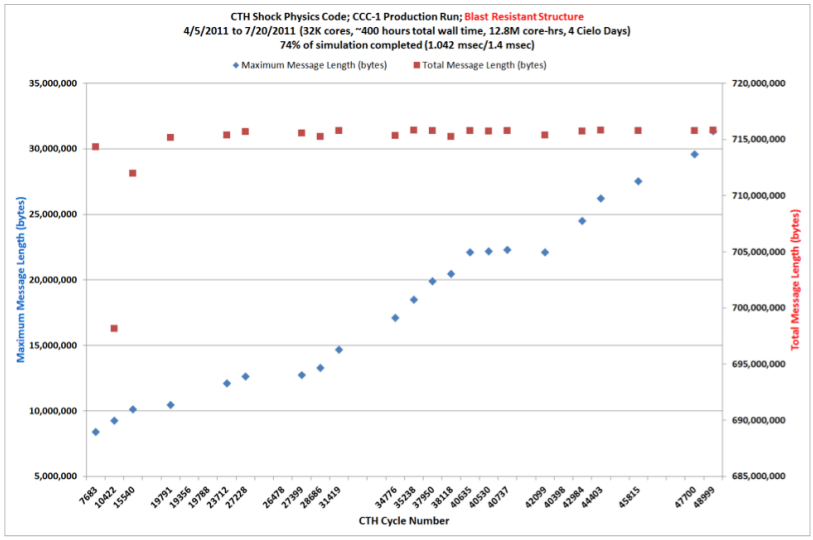
Figure 15.2(a) plots the "simulation performance" from a 400 hour run on the mesh (simulation performance is defined as "microseconds of simulation time achieved per hour of walltime"). The average simulation performance over the 400 hours was 2.5 microseconds of simulation time achieved per hour of walltime. This problem is configured to use AMR to refine on specified indicators with defined criteria, so runtime performance is expected to decrease as the mesh refines to higher levels of resolution and the calculation advances.

We collected basic default CTH messaging information over the course of the 400 hour simulation to investigate CTH inter-process communication behavior. This is a production version of CTH and was not instrumented for more in-depth metrics. Figure 15.2(b) plots a sampling of Maximum Message Length (bytes) and Total Message Length (bytes) on a per cycle basis. The Maximum Message Length (bytes) per cycle gradually increases over the course of the simulation from approximately 8 MBytes to approximately 30 MBytes. Total Message Length (bytes) per cycle, however, remains relatively constant at 715 MBytes.

The simulation performance illustrated that improvements in code performance (algo-



(a) Runtime: microseconds of simulation time achieved per hour of walltime



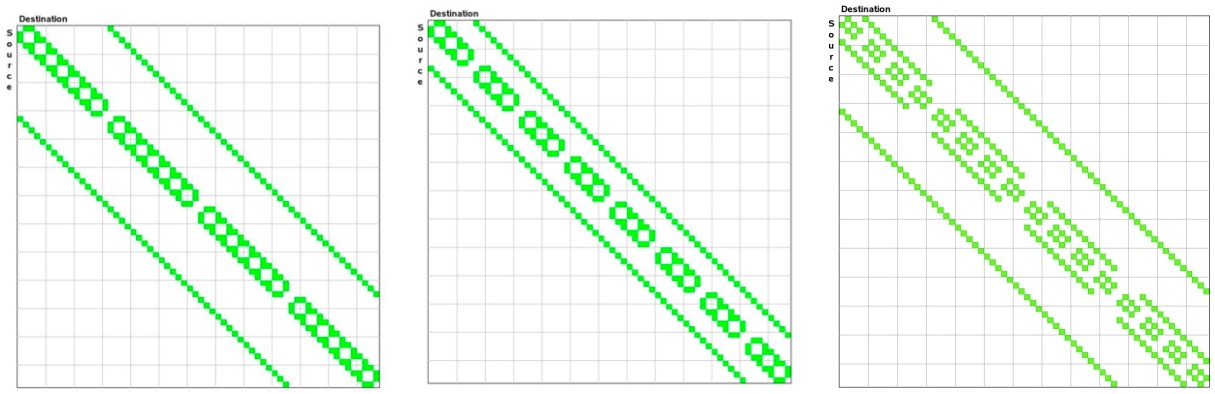
(b) Message traffic

Figure 15.2: CTH performance by time steps (cycles) at 32k processor cores

rithms, inter-process communication/messaging, etc.) and platform performance (on-node, interconnects, file system, etc.) will be necessary to achieve better simulation performance, especially at very large core counts.

15.1.2 A deeper dive

MiniGhost, a mini-app from the Mantevo miniapps project[1], provides a tractable means for studying the scaling characteristics of the CTH interprocess communication requirements. The communication patterns for two distinct CTH problems and miniGhost are shown in Figure 15.3. The x-axis represents the destination process, the y-axis represents the source



(a) CTH shaped charge (b) CTH mesoscale (c) miniGhost

Figure 15.3: CTH and miniGhost ommunication patterns.

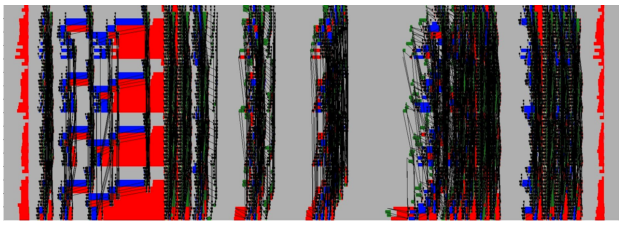
process. That is, the processor in row i is sending to the processor in row j. Color represents relative volume of data, with red higher than green.

We examined two common problems modeled by CTH. The *meso-scale impact in a confined space* problem is computationally well-balanced across the parallel processes. This problem involves 11 materials, inducing the boundary exchange of 75 variables. The *shaped charge* problem involves four materials, inducing the boundary exchange of 40 variables. (This problem was used in the acceptance testing for Cielo[41].)

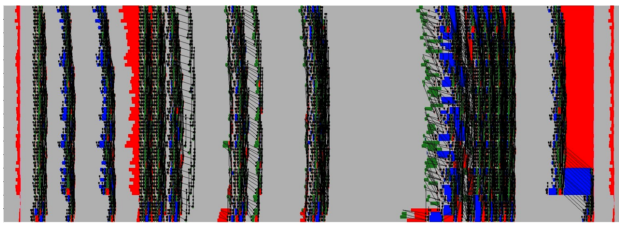
For the shaped charge problem, each time step CTH makes 90 calls to MPI collective functionality (significant, but about seven times fewer than Charon), 19 calls to exchange boundary data (two dimensional "faces"), and three calls to propagate data across faces (in the x, y, and z directions). Collective communication is typically a reduction (`MPI_Allreduce`) of a single variables, though some are of fairly large sizes. Each boundary exchange aggregates data from 40 three dimensional arrays, representing 40 variables. Message buffers are constructed from faces, approximately one third of which are contiguous, one third of which are stride y, and one third of which are stride $x \times y$.

[1] `http://software.sandia.gov/mantevo`

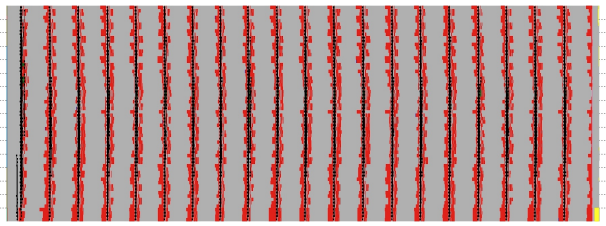
The space-time runtime profiles[2] for the two CTH problems and miniGhost, shown in Figure 15.4, illustrate the very large message aggregation scheme in the BSP model, which induces a strong separation of computation and communication. Given this runtime profile,



(a) CTH shaped charge



(b) CTH mesoscale



(c) miniGhost

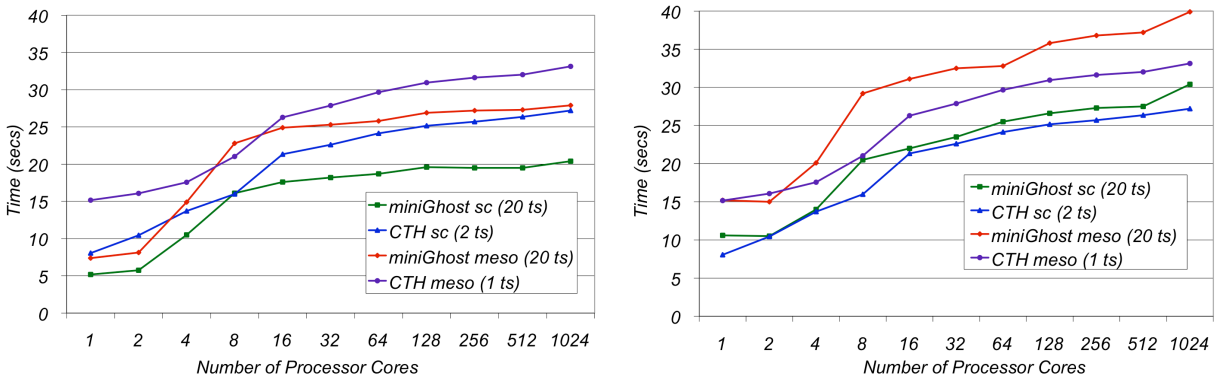
Figure 15.4: CTH and miniGhost space-time runtime profiles

CTH performance will be most strongly impacted by the exchange of very large messages between nearest neighbors, preceded by the accumulation of that data into message buffers and succeeded by the unpacking of the messages into the appropriate arrays.

We compared performance of miniGhost with CTH on a Cray XT5, the most recent evolutionary ancestor of Cielo. Results are shown in Figure 15.5. Run in weak scaling mode on up to 1,024 processor cores (this XT5 is a dual-socket AMD Opteron Istanbul hex-core node based machine with SeaStar interconnect, details in [135]), miniGhost tracks CTH performance reasonably well. Combined with the above runtime profiles, we are confident that miniGhost may be used as a proxy for CTH in the sense of the nearest neighbor and collective communication behavior.

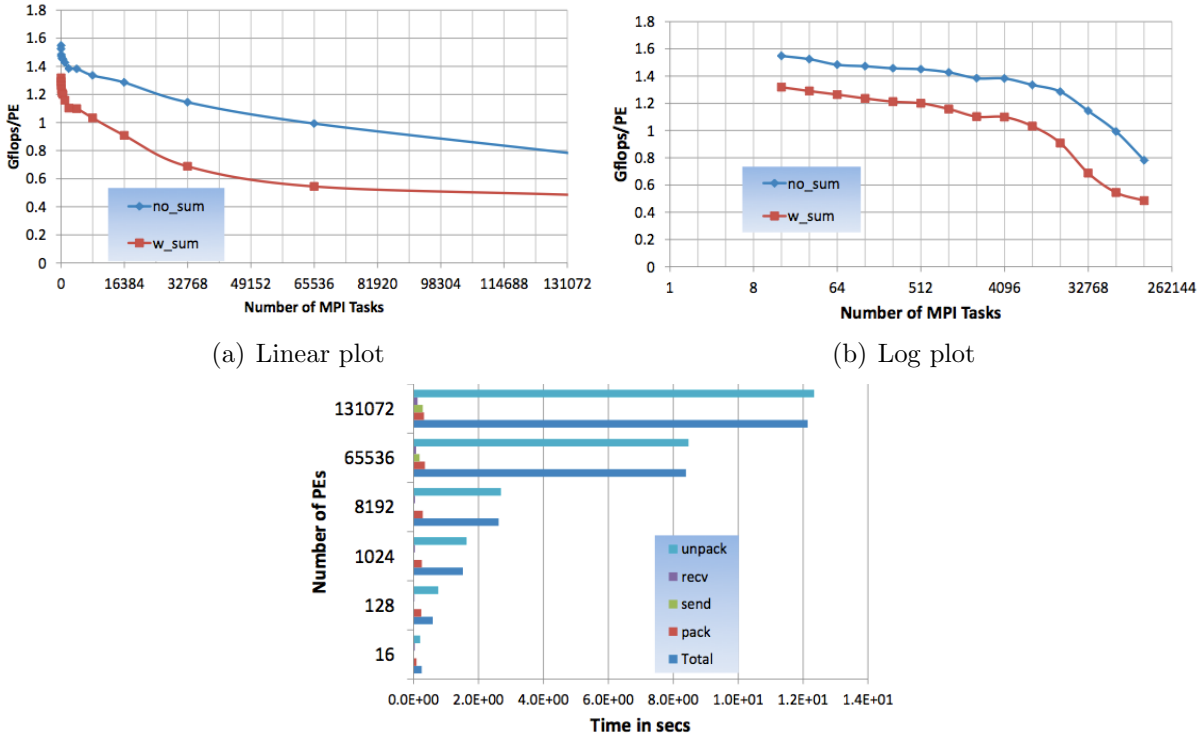
MiniGhost was run, in weak scaling mode, on Cielo, on 2^i processor cores, for $i = 1, \ldots, 18$, so the largest run was on 131,072 cores. Performance is shown in Figure 15.6. Configured to mimic the shaped charge problem, and applying a 7-point and 27-point dif-

[2]Generated using CrayPAT and visualized using Apprentice2.



(a) (b) with reductions

Figure 15.5: CTH and miniGhost performance comparison on a Cray XT5.



(a) Linear plot (b) Log plot

(c) Communication requirements

Figure 15.6: miniGhost weak scaling on Cielo

In the top two graphs, the blue line represents performance of the nearest neighbor only configuration; the red line includes the grid summary (`MPI_Allreduce` *case.*

ference stencil on the three dimensional domain, we see very nice scaling behavior up to 2,048 cores. Then at 4,096 cores, the collective communication (`MPI_Allreduce(sum)`) performance begins to degrade, and at very high scales becomes a significant problem. Perhaps even more troubling, nearest neighbor communication performance begins degrading at 16,384 cores, developing into a significant problem. Inexplicably, performance improves for all tests at the highest core count.

The decrease in parallel efficiency for the no-sum case is related to the increasing load imbalance due to message transfer time differences. Load imbalance grows steadily with increasing core counts. The tools register this as increase in `MPI_Waitany` time in the miniGhost routine (`MG_UNPACK`) that moves the aggregated incoming data to the individual faces of the 40 three dimensional data arrays. When including a global summation across each data array immediately following the stencil computation on it, an additional overhead impacting parallel efficiency can be traced to `MPI_Allreduce(sum)` SYNC time. This is again a direct consequence of the above mentioned differences in message time between the slowest pair and the fastest pair in the communication exchanges.

Some additional observations:

- Small improvements in GFLOPs/PE were observed for 8k, 16k, 32k, and 64k processor core runs with resource manager MOAB allocation of required number nodes as opposed to running under the 8192 node allocation. This suggests that node topology has an effect, but the scaling trend seen in the plots is still very valid.
- This is probably the first time runs were conducted by us with CrayPat at this large scales. CrayPat has bugs that we need to share with Cray, but gave useful data.
- Future work includes analyzing the MPI task to node mapping.

15.1.3 A Hybrid MPI+OpenMP exploration

An Cielo node is configured as two Magny-Cours oct-core processors. Further, each processor is configured into two NUMA regions (called NUMA nodes), each containing four cores. Illustrated in Figure 15.7, trends show that this sort of node memory hierarchy will probably continue as computers increase the number of processor cores. Results of this work are shown in Figure 15.8. The above work induced us to consider an on-node OpenMP implementation. Placing one MPI rank on each NUMA node, where OpenMP then spawned four threads, one per processor, performance exceeded that of the MPI everywhere model. Once the OpenMP region spanned more than one NUMA node, performance degraded such that it was worse than the MPI everywhere mode. While this does not definitively prove that a full CTH MPI+OpenMP implementation will result in a similar performance improvement, it can help guide further explorations. Additionally, this work illustrates the importance of understanding the intra-node memory hierarchy, a characteristic that is expected to continue and increase on future architectures.

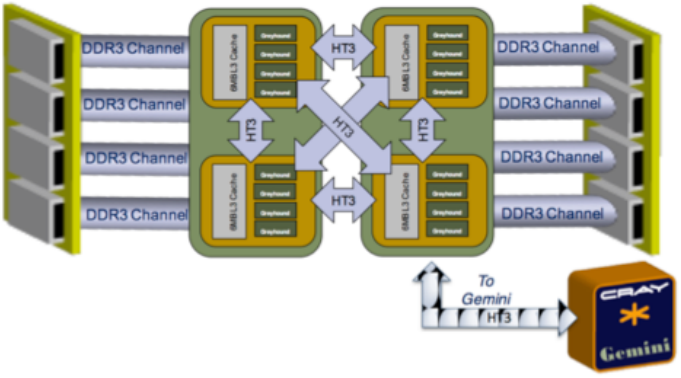


Figure 15.7: The XE6 compute node architecture. Images courtesy of Cray, Inc.

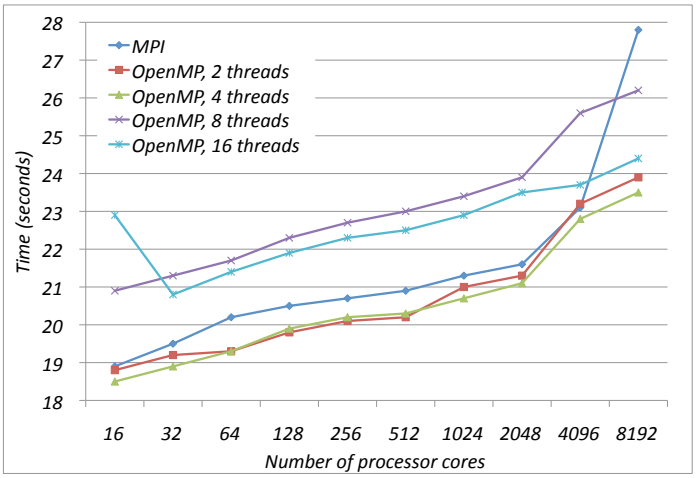


Figure 15.8: miniGhost configured for MPI and OpenMP.

15.2 Implicit Finite Element solver

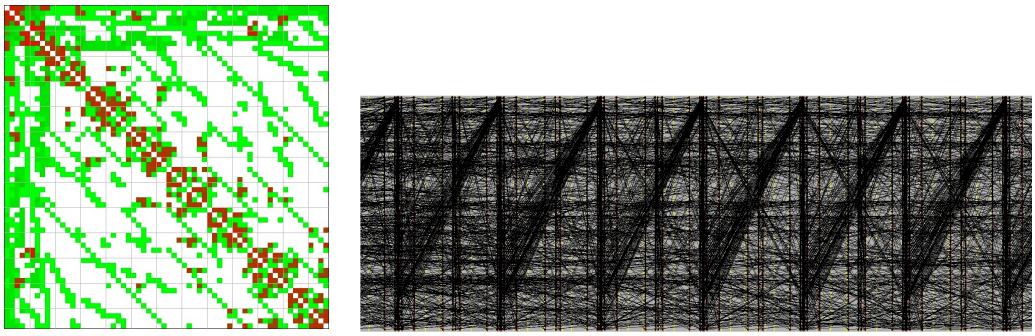
Charon is a semiconductor device simulation computer program[3] developed at Sandia National Laboratories. It is a transport reaction code used to simulate the performance of semiconductor devices under irradiation. Using a finite element method (FEM), it solves a coupled system of nonlinear partial differential equations (PDEs) in three dimensions describing the drift-diffusion model,

Finite element discretization of these equations in space on an unstructured mesh produces a sparse, strongly coupled nonlinear system. These equations are solved using a method based on a Jacobian-free Newton-Krylov approach, resulting in a large sparse linear systems.

The linear systems are solved either using TFQMR [56] (typically when the system is reasonably well-conditioned) or GMRes [123], without restart, when the system is poorly conditioned. A multigrid preconditioner [59] significantly improves scaling and performance [82]. This is based on a Schwarz method, with some form of local incomplete factorization applied.

The code, written using C++, is configured for parallel computation using the MPI-everywhere model. The linear systems are solved using functionality from the Trilinos library [64], with preconditioning from the ML package [60].

Figure 15.9(a) illustrates Charon's inter-process communication pattern, with the x-axis



(a) Point-to-point communication pattern.

(b) Runtime trace profiles, for 1 Newton iteration

Figure 15.9: Charon run time profiles, illustrated using 64 processor cores

representing the destination process and the y-axis representing the source process. Color represents volume of data, with red higher than green. Figure 15.9(b) illustrates Charon's runtime profile, with the horizontal axis representing time, the vertical axis representing individual cores. Black represents point-to-point communication, gray is computation, green is send, blue is receive, red is synchronization, and pink is reduce.

[3] `http://charleston.sandia.gov/Charon/`

File system issues prevented running Charon at large scale on Cielo. However, issues identified at scale on the IBM BlueGene/P platform (named Dawn) at Lawrence Livermore National Laboratory revealed an issue at large scales that is also expected to be an issue on other platforms, including Cielo.

Figure 15.10 demonstrates the weak scaling of Charon on up to 64k (65,536) cores and

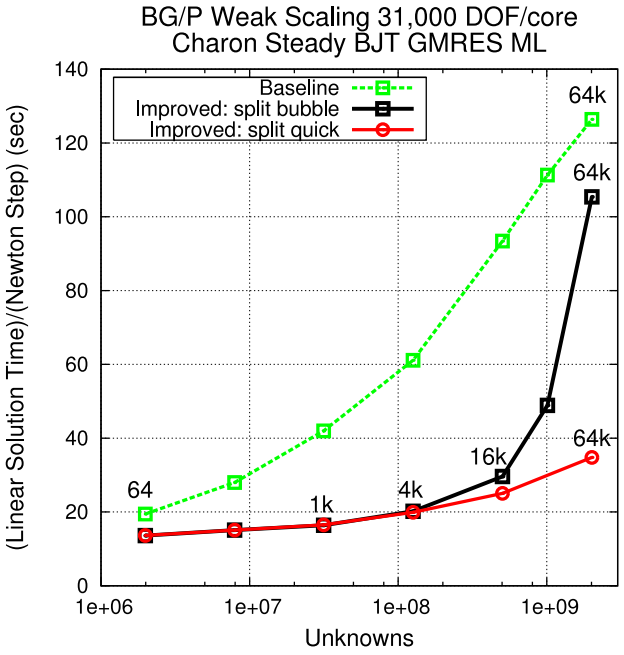


Figure 15.10: Charon on Dawn

two billion degrees of freedom (DOF) on Dawn for a steady-state simulation of a transistor [81]. The vertical axis plots the linear solution time per Newton step; this calculation typically takes 7-8 Newton steps to achieve convergence. The green line (labelled "baseline") denotes the multigrid method we were typically using when we were running problems in the 1k-4k core range. However, once we had the opportunity to scale the calculations up to 64k cores, the scaling was not as good as we would have hoped, which led us to evaluate alternative multigrid approaches. The black line (labelled "improved: split bubble") denotes a promising alternative approach. It scaled well up to 16k cores, but then scaled extremely poorly beyond that, even worse than the baseline approach. We were ready to give up on this initially promising alternative approach, when further examination determined that the problem was not with the multigrid algorithm, but with the MPI implementation. The alternative approach relied on `MPI_Comm_split`, but the MPI implementation employed a bubble sort, which scales as the number of MPI processes squared. After the MPI developers replaced the bubble sort by quicksort (which scales as $O(p * logp)$ where p is the number of MPI processes), the linear solution time scaled as the red curve (labelled "improved: split quick"). At 64k cores, one can see that the choice of bubble sort compared with quicksort in `MPI_Comm_split` increased the *total* solve time by a factor of three. And with quicksort, the alternative approach scaled much better than the baseline approach. This

demonstrates that a poor algorithmic choice in system libraries such as MPI can really destroy the performance of algorithms in an application code. Typically MPI developers work independently of application code developers, but this example demonstrations that more interaction could be beneficial. Other MPI implementations such as Cray MPI were based on the MPI implementation which used the bubble sort for `MPI_Comm_split`. So the change to quicksort significantly improved the scaling of `MPI_Comm_split` on the Cielo Cray XE6 machine.

15.3 Conclusions and Summary

Executing application codes at significantly increased scale can result in performance degradation. The root cause(s) can be many and varied: hardware, system software, and algorithmic. Collective communication is expected to present a natural scaling issue due to the known algorithms for supporting this functionality, and we do see this. What is somewhat unexpected is that the nearest neighbor communication exacerbates this as the number of processors increases, due to the increasing probability of load imbalances induced by issues other than the computation.

We are further investigating the use of the on-node OpenMP threading model, which for some relatively small test cases we find may be useful for improving on-node performance. At the full application source code complexity the level of effort required to achieve this relatively modest performance may be prohibitive.

Chapter 16

Conclusion

The purpose of this report is to prove that the team has completed milestone 4467–Demonstration of a Legacy Application's Path to Exascale. As a review, we provide the milestone text below.

> Cielo is expected to be the last capability system on which existing ASC codes can run without significant modifications. This assertion will be tested to **determine where the breaking point is for an existing highly scalable application.** The goal is to **stretch the performance boundaries of the application by applying recent CSSE RD** in areas such as resilience, power, I/O, visualization services, SMARTMAP, lightweight LWKs, virtualization, simulation, and feedback loops. Dedicated system time reservations and/or CCC allocations will be used to **quantify the impact of system-level changes to extend the life and performance of the ASC code base.** Finally, a **simulation of anticipated exascale-class hardware will be performed using SST** to supplement the calculations.

The bolded text represents the actions to be completed in order to complete the milestone itself. We will address each phrase individually.

Determine where the breaking point is for an existing highly scalable application: Chapter 15 presented the CSSE work that sought to identify the breaking point in two ASC legacy applications–Charon and CTH. Their mini-app versions were also employed to complete the task. There is no single breaking point as more than one issue was found with the two codes. The results were that applications can expect to encounter performance issues related to the computing environment, system software, and algorithms. Careful profiling of runtime performance will be needed to identify the source of an issue, in strong combination with knowledge of system software and application source code.

Stretch the performance boundaries of the application by applying recent CSSE R&D: Chapters 2 through 14 all presented CSSE R&D work that stretched the performance boundaries of applications. All areas enumerated in the milestone description were covered. Resilience milestone documentation is in chapter 10, power in chapter 6, I/O in chapters 7 and 13, visualization services in chapter 8, SMARTMAP in chapter 4, lightweight kernels in chapters 2, 3 and 12, virtualization is also in chapter 12, simulation in chapter 14, and finally, feedback loops in chapter 11.

Simulation of anticipated exascale-class hardware will be performed using SST: Chapter 14 describes the SST simulation that was done on anticipated exascale hardware.

Quantify the impact of system-level changes to extend the life and performance of the ASC code base: All of the chapters quantified the impact based on metrics appropriate to the area. It is not possible to provide a single number that represents the net improvment demonstrated by this milestone work. The measures included reduced runtime, improved job throughput, improved user productivity, and reduced power consumption. See Table 16.1 which lists the preformance improvement area(s) addressed by each area. The unit of measure is different in most cases and cannot be summed. In most cases, the CSSE work has benefit for future systems. The work provides lessons learned on what worked well and what worked adequately. As typically happens, all efforts offer opportunities for enhancement and/or to explore new questions prompted by the work. Lastly, there is data that was collected strickly for future co-design work and design making as we continue on our path to exascale.

Table 16.1: Summary of Contributions by Performance Improvement Area

R&D Contribution	Improvement Area
Red Storm Catamount Enhancements to Fully Utilize Additional Cores per Node	Improved Job Throughput
Fast_where: A Utility to Reduce Debugging time on Red Storm	Increased User Productivity
SMARTMAP Optimization to Reduce Core-to-Core Communication Overhead	Reduced Runtime
Smart Allocation Algorithms	Reduced Runtime
Enhancements to Red Storm and Catamount to Increase Power Efficiency During Application Execution	Reduced power consumption; Data for Co-Design
Reducing Effective I/O Costs with Application-Level Data Services	Reduced Runtime
In situ and In transit Visualization and Analysis	Improved Job Throughput; Increased User Productivity; Data for Co-Design
Dynamic Shared Libraries on Cielo	Increased User Productivity
Process Replication for Reliability	Improved Job Throughput; Data for Co-Design
Enabling Dynamic Resource Aware Computing	Reduced Runtime; Data for Co-Design
A Scalable Virtualization Environment for Exascale	Increased User Productivity; Data for Co-Design
Goofy File System for High-Bandwidth Checkpoints	Reduced Runtime
Exascale Simulation - Enabling Flexible Collective Communication Offload with Triggered Operations	Reduced runtime; Data for Co-Design
Application Scaling	Data for Co-Design

References

[1] Mohammad H. Abbasi. *Data Services: Bringing I/O Processing to Petascale*. PhD thesis, Georgia Institute of Technology, May 2011.

[2] Sean Ahern, Arie Shoshani, Kwan-Liu Ma, et al. Scientific discovery at the exascale. Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization, February 2011.

[3] Jochen Alber and Rolf Niedermeier. On multi-dimensional Hilbert indexings. *Theory of Computing Systems*, 1998.

[4] Carl Albing, Norm Troullier, Stephen Whalen, Ryan Olson, Howard Pritchard, Joe Glenski, and Hugo Mills. Topology, bandwidth and performance: A new approach in linear orderings for application placement in a 3d torus. In *Proceedings of the Cray User Group Annual Technical Conference*, 2011.

[5] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable I/O forwarding framework for high-performance computing systems. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, September 2009.

[6] R. Alverson, D. Roweth, and L. Kaplan. The Gemini system interconnect. In *Proceedings of the 18th Annual Symposium on High Performance Interconnects (HOTI)*. IEEE Computer Society Press, August 2010.

[7] Saman Amarasinghe and et al. Exascale software study: Software challenges in extreme scale systems. `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf`, September 2009.

[8] Dorion C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Stack trace analysis for large scale applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Long Beach, California, March 2007.

[9] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2, October 2004.

[10] James Balfour and William J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, 2006.

[11] Brian W. Barrett and K. Scott Hemmert. An application based MPI message throughput benchmark. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing*, September 2009.

[12] Joel F. Bartlett. A nonstop kernel. In *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles*, pages 22–29, 1981.

[13] Sandra Johnson Baylor, Caroline Benveniste, and Yarsun Hsu. Performance evaluation of a massively parallel I/O subsystem. In *Input/Output in Parallel and Distributed Computer Systems*. Kluwer Academic, 1996.

[14] Michael A. Bender, David P. Bunde, Erik D. Demaine, Sándor P. Fekete, Vitus J. Leung, Henk Meijer, and Cynthia A. Phillips. Communication-aware processor allocation for supercomputers: Finding point sets of small average distance. *Algorithmica*, 2008.

[15] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A checkpoint filesystem for parallel applications. In *SC '09 Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 21:1–21:12, New York, NY, USA, November 2009. ACM.

[16] John Bent and Gary Grider. U.S. Department of Energy best practices workshop on file systems archives: Usability at Los Alamos National Lab. In *Proceedings of the 5th DOE Workshop on HPC Best Practices: File Systems and Archives*, September 2011.

[17] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Peter Kogge, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems. `http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf`, September 2008.

[18] J. Brandt, F. Chen, V. De Sapio, A. Gentile, J. Mayo, P. Pebay, D. Roe, D. Thompson, and M. Wong. Combining Virtualization, Resource Characterization, and Resource Management to Enable Efficient High Performance Compute Platforms Through Intelligent Dynamic Resource Allocation. In *24th IEEE International Parallel and Distributed Processing Symposium, 6th Workshop of System Management Techniques, Processes, and Services*, 2010.

[19] J. Brandt, B. Debusschere, A. Gentile, J. Mayo, P. Pebay, D. Thompson, and M. Wong. OVIS-2: A Robust Distributed Architecture for Scalable RAS. In *22nd IEEE International Parallel and Distributed Processing Symposium, 4th Workshop of System Management Techniques, Processes, and Services*, 2008.

[20] J. Brandt, A. Gentile, D. Thompson, and T. Tucker. Develop Feedback System for Intelligent Dynamic Resource Allocation to Improve Application Performance. Technical Report SAND2011-6301, Sandia National Laboratories, 2011.

[21] Ron Brightwell. A prototype implementation of MPI for SMARTMAP. In *Proceedings of the 15th European PVM/MPI Users' Group Conference*, September 2008.

[22] Ron Brightwell, Kurt B. Ferreira, and Rolf Riesen. Transparent redundant computing with mpi. In Rainer Keller, Edgar Gabriel, Michael M. Resch, and Jack Dongarra, editors, *EuroMPI*, volume 6305 of *Lecture Notes in Computer Science*, pages 208–218. Springer, 2010.

[23] Ron Brightwell and Kevin Pedretti. Optimizing multi-core mpi collectives with SMARTMAP. In *Third International Workshop on Advanced Distributed and Parallel Network Applications*, September 2009.

[24] Ron Brightwell and Kevin Pedretti. An intra-node implementation of openshmem using virtual address space mapping. In *Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS '11, October 2011.

[25] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. Smartmap: operating system support for efficient data sharing among processes on a multi-core processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, November 2008.

[26] Ron Brightwell, Kevin Pedretti, Keith Underwood, and Trammell Hudson. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, 2006.

[27] Ron Brightwell, Rolf Riesen, Bill Lawry, and Arther B. Maccabe. Portals 3.0: protocol building blocks for low overhead communication. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, April 2002.

[28] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P.W. Cook. Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *Micro, IEEE*, 2000.

[29] David P. Bunde, Vitus J. Leung, and Jens Mache. Communication patterns and allocation strategies. In *Proceedings of the 3rd International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, 2004.

[30] William J. Camp and James L. Tomkins. The red storm computer architecture and its implementation. In *The Conference on High-Speed Computing: LANL/LLNL/SNL*, Salishan Lodge, Glenedon Beach, Oregon, April 2003.

[31] Franck Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *IJHPCA*, 23(3):212–226, 2009.

[32] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 2002.

[33] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and Riesen L. Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations. In *21st IEEE International Parallel and Distributed Processing Symposium*, 2010.

[34] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: fault containment for shared-memory multiprocessors. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 12–25, New York, NY, USA, 1995. ACM.

[35] J H Chen, A Choudhary, B de Supinski, M DeVries, E R Hawkes, S Klasky, W K Liao, K L Ma, J Mellor-Crummey, N Podhorszki, R Sankaran, S Shende, and C S Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(1):31pp, 2009.

[36] Hank Childs. Architectural challenges and solutions for petascale postprocessing. *Journal of Physics: Conference Series*, 78(012012), 2007. DOI=10.1088/1742-6596/78/1/012012.

[37] Using Cray Performance Analysis Tools. Technical Report CrayDoc S-2376-52, Cray, Inc.

[38] Workload Management and Application Placement for the Cray Linux Environment. Technical Report CrayDoc S-2496-31, Cray, Inc.

[39] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[40] K. Devine, E. Boman, R. Heaphy, B. Henrickson, and C. Vaughan. Zoltan Data Management Services for Parallel Dynamic Applications. In *Computing in Science and Engineering*, volume 4, pages 90–97, 2002.

[41] D. Doerfler, Mahesh Rajan, Cindy Nuss, Cornell Wright, and Thomas Spelce. Application-Driven Acceptance of Cielo, an XE6 Petascale Capability Platform. In *Proc. 53rd Cray User Group Meeting*, 2011.

[42] S. Domino, G. Wagner, A. Luketa-Hanlin, A. Black, and J. Sutherland. Verification for Multi-Mechanics Applications. In *48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (AIAA Paper 2007-1933)*, 2007.

[43] S. P. Domino, C. D. Moen, S. P. Burns, and G. H. Evans. SIERRA/Fuego A Multi-Mechanics Fire Environment Simulation Tool. In *41st AIAA Aerospace Sciences Meeting (AIAA Paper 2003-0149)*, 2003.

[44] Jr. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In *Proceedings of the 19th International Symposium on Shock Waves*, pages 377–382, July 1993.

[45] H. C. Edwards. Sierra Framework Version 3: Core Services Theory and Design. Technical Report SAND2002-3616, Sandia National Laboratories, 2002.

[46] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[47] E.N. Elnozahy and J.S. Plank. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *Dependable and Secure Computing, IEEE Transactions on*, 1(2):97–108, April 2004.

[48] Christian Engelmann, Hong H. Ong, and Stephen L. Scott. The case for modular redundancy in large-scale high performance computing systems. In *Proceedings of the 8^{th} IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009*, pages 189–194, Innsbruck, Austria, February 16-18, 2009. ACTA Press, Calgary, AB, Canada.

[49] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C. Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth E. Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, 2011.

[50] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. A study of process arrival patterns for MPI collective operations. *International Journal of Parallel Programming*, 36(6):543–570, December 2008.

[51] Matthew S. Farmer, Anthony Skjellum, Matthew L. Curry, and H. Lee Ward. The design and implementation of SSM: A single-sided message passing library based on Portals 4.0. Technical Report UABCIS-TR-2012-01242012, Department of Computer and Information Sciences, University of Alabama at Birmingham, 115A Campbell Hall, 1300 University Blvd, Birmingham, Alabama 35294-1170, to appear.

[52] Xizhou Feng, Rong Ge, and Kirk W. Cameron. Power and Energy Profiling on Scientific Applications on Distributed Systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium IPDPS*, 2005.

[53] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James H. Laros III, Kevin Pedretti, and Ron Brightwell. rMPI: Increasing fault resiliency in a message-passing environment. Technical Report SAND2011-2488, Sandia National Laboratories, 2011.

[54] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[55] Kurt B. Ferreira, Jon Stearley, James H. Laros III, Ron Oldfield, Kevin T. Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In Scott Lathrop, Jim Costa, and William Kramer, editors, *SC*, page 44. ACM, 2011.

[56] Roland W. Freund. A transpose-free quasi-minimum residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Comp.*, 14(2):470–482, 1993.

[57] Jing Fu, Ning Liu, O. Sahni, K.E. Jansen, M.S. Shephard, and C.D. Carothers. Scalable parallel I/O alternatives for massively parallel partitioned solver systems. In *International Parallel and Distributed Processing Symposium, Workshops and PhD Forum*, Atlanta, GA, April 2010.

[58] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[59] Michael W. Gee, Chris M. Siefert, Jonathan J. Hu, Ray S. Tuminaro, and Marzio G. Sala. ML 5.0 Smoothed Aggregation User's Guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[60] M.W. Gee, C.M. Siefert, J.J. Hu, R.S. Tuminaro, and M.G. Sala. ML 5.0 Smoothed Aggregation User's Guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[61] Craig Gotsman and M. Lindenbaum. On the metric properties of discrete space-filling curves. *IEEE Transactions on Image Processing*, 1996.

[62] Francois Gygi, Erik W. Draeger, Martin Schulz, Bronis R. de Supinski, John A. Gunnels, Vernon Austel, James C. Sexton, Franz Franchetti, Stefan Kral, Christoph W. Ueberhuber, and Juergen Lorenz. Large-scale electronic structure calculations of high-Z metals on the Blue-Gene/L platform. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006.

[63] Thomas J. Hacker, Fabian Romero, and Christopher D. Carothers. An analysis of clustered failures on large supercomputing systems. *J. Parallel Distrib. Comput.*, 69:652–665, July 2009.

[64] M. A. Heroux, R. A. Bartlett, V. E. Howle, R.J. Hoekstra, J. J. Hu, T.G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R. S. Tuminaro, J.M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31:397–423, September 2005.

[65] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An overview of the trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, 2005.

[66] E.S. Hertel, Jr., R.L. Bell, M.G. Elrick, A.V. Farnsworth, G.I. Kerley, J.M. McGlaun, S.V. Petney, S.A. Silling, P.A. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings, 19th International Symposium on Shock Waves*, 1993.

[67] David Hilbert. Über die stetige Abbildiung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 1891.

[68] Lars Holst. The general birthday problem. In *Random Graphs 93: Proceedings of the sixth international seminar on Random graphs and probabilistic methods in combinatorics and computer science*, pages 201–208, New York, NY, USA, 1995. John Wiley & Sons, Inc.

[69] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, 1994.

[70] IOR interleaved or random HPC benchmark. http://sourceforge.net/projects/ior-sio/.

[71] Chris Johnson, Rob Ross, et al. Visualization and knowledge discovery. Report from the DOE/ASCR Workshop on Visual Analysis and Data Exploration at Extreme Scale, October 2007.

[72] E. S. Hertel Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In R. Brun and L.D. Dumitrescu, editors, *Proceedings of the 19th International Symposium on Shock Physics*, volume 1, pages 377–382, Marseille, France, July 1993.

[73] Suzanne M. Kelly, Ruth Klundt, and James H. Laros III. Shared libraries on a capability class computer. In *Cray User Group Annual Technical Conference*, May 2011.

[74] D. J. Kerbyson, H. J. Alme, Adolfy Hoisie, Fabrizio Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 37–48, 2001.

[75] David Kotz. Disk-directed I/O for MIMD multiprocessors. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 35, pages 513–535. IEEE Computer Society Press and John Wiley & Sons, 2001.

[76] S. O. Krumke, M. V. Marathe, H. Noltemeier, V. Radhakrishnan, S. S. Ravi, and D. J. Rosenkrantz. Compact location problems. *Theoretical Computer Science*, 1997.

[77] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[78] John R. Lange, Kevin Pedretti, Peter Dinda, Patrick G. Bridges, Chang Bae, Philip Soltero, and Alexander Merritt. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on virtual execution environments*, VEE '11, 2011.

[79] G. L. Lee, D. H. Ahn, B. R. de Supinski, J. Gyllenhaal, and P. Miller. Pynamic: the python dynamic benchmark. In *Proceedings of the IEEE 10th International Symposium on Workload Characterization*, pages 101–106, September 2007.

[80] Vitus J. Leung, Esther M. Arkin, Michael A. Bender, David Bunde, Jeanette Johnston, Alok Lal, Joseph S. B. Mitchell, Cynthia Phillips, and Steven S. Seiden. Processor allocation on Cplant: Achieving general processor locallity using one-dimensional allocation strategies. In *Proceedings of the 4th IEEE International Conference on Cluster Computing*, 2002.

[81] P. T. Lin. Improving multigrid performance for unstructured mesh drift-diffusion simulations on 147,000 cores. *submitted to the International Journal for Numerical Methods in Engineering*, 2011.

[82] Paul T. Lin and John N. Shadid. Towards Large-Scale Multi-Socket, Multicore Parallel Simulations: Performance of an MPI-only Semiconductor Device Simulator. *Journal of Computational Physics*, 229(19), 2010.

[83] Paul T. Lin, John N. Shadid, Marzio Sala, Raymond S. Tuminaro, Gary L. Hennigan, and Robert J. Hoekstra. Performance of a parallel algebraic multilevel preconditioner for stabilized finite element semiconductor device modeling. *Journal of Computational Physics*, 228(17), 2009.

[84] Virginia Lo, Kurt J. Windisch, Wanqian Liu, and Bill Nitzberg. Non-contiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 1997.

[85] Jay Lofstead, Ron Oldfield, Todd Kordenbrock, and Charles Reiss. Extending scalability of collective I/O through nessie and staging. In *Proceedings of the 6th Parallel Data Storage Workshop*, November 2011.

[86] Jens Mache and Virginia Lo. The effects of dispersal on message-passing contention in processor allocation strategies. In *Proceedings of the 3rd Joint Conference on Information Sciences, Sessions on Parallel and Distributed Processing*, 1997.

[87] Jens Mache, Virginia Lo, and Kurt Windisch. Minimizing message-passing contention in fragmentation-free processor allocation. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, 1997.

[88] Frank H. Mathis. A generalized birthday problem. *SIAM Review*, 33(2):265–270, 1991.

[89] Timothy G. Mattson and Greg Henry. An overview of the Intel TFLOPS supercomputer. *Intel Technology Journal*, Q1:12, 1998.

[90] Bruce H. McCormick, Thomas A. DeFanti, and Maxine D. Brown, editors. *Visualization in Scientific Computing (special issue of Computer Graphics)*, volume 21. ACM, 1987.

[91] Dennis McEvoy. The architecture of tandem's nonstop system. In *ACM '81: Proceedings of the ACM '81 conference*, page 245, New York, NY, USA, 1981. ACM.

[92] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[93] Adam Moody, Juan Fernandez, Fabrizio Petrini, and Dhabaleswar K. Panda. Scalable NIC-based reduction on large-scale clusters. In *Proceedings of the ACM/IEEE SC2003 Conference*, November 2003.

[94] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 2001.

[95] Sherry Moore and Lionel M. Ni. The effects of network contention on processor allocation strategies. In *Proceedings of the 10th International Parallel Processing Symposium*, 1996.

[96] José E. Moreira, Michael Brutman, José Castaños, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a highly-scalable operating system: The blue gene/l story. In *Proceedings of the ACM / IEEE Supercomputing SC'2006 conference*, 2006.

[97] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Joudain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, Michael E. Papka, and Scott Klasky. Examples of in transit visualization. In *Proceedings of the PDAC 2011 : 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities*, November 2011. Submitted.

[98] Kenneth Moreland, Ron Oldfield, Pat Marion, Sabastien Jourdain, Norbert Podhorski, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, Michael E. Papka, and Scott Klasky. Examples of in transit visualization. In *Proceedings of the 2nd International Workshop on Petascale Data Analytics*, 2011.

[99] P. Notz, S. Subia, M. Hopkins, H. Moffat, and D. Noble. Aria 1.5 User Manual. Technical Report SAND2007-2734, Sandia National Laboratories, 2007.

[100] Ron A. Oldfield. Lightweight storage and overlay networks for fault tolerance. Technical Report SAND2010-0040, Sandia National Laboratories, Albuquerque, NM, January 2010.

[101] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Rolf Riesen, Maria Ruiz Varela, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, San Diego, CA, September 2007.

[102] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, Washington, DC, USA, 2007. IEEE Computer Society.

[103] Ron A. Oldfield, Arthur B. Maccabe, Sarala Arunagiri, Todd Kordenbrock, Rolf Riesen, Lee Ward, and Patrick Widener. Lightweight I/O for scientific applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006.

[104] Ron A. Oldfield, Patrick Widener, Arthur B. Maccabe, Lee Ward, and Todd Kordenbrock. Efficient data-movement for lightweight I/O. In *Proceedings of the 2006 International Workshop on High Performance I/O Techniques and Deployment of Very Large Scale I/O Systems*, Barcelona, Spain, September 2006.

[105] Ron A. Oldfield, Andrew Wilson, George Davidson, and Craig Ulmer. Access to external resources using service-node proxies. In *Proceedings of the Cray User Group Meeting*, Atlanta, GA, May 2009.

[106] Ron A. Oldfield, David E. Womble, and Curtis C. Ober. Efficient parallel I/O in seismic imaging. *The International Journal of High Performance Computing Applications*, 12(3):333–344, Fall 1998.

[107] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*, page 299.2, 2005.

[108] Open|SpeedShop User's Guide. Technical Report 2.0.1 Release, Open|SpeedShop, 2011.

[109] Hewlett Packard. HP NonStop computing. `http://h20338.www2.hp.com/NonStopComputing/cache/76385-0-0-0-121.html`.

[110] Kevin T. Pedretti, Courtenay Vaughan, Karl Scott Hemmert, and Brian Barrett. Application sensitivity to link and injection bandwidth on a Cray XT4 system. In *Proceedings of the 2005 Cray User Group Annual Technical Conference*, Helsinki, Finland, May 2008.

[111] Tom Peterka, Hongfeng Yu, Robert Ross, and Kwan-Liu Ma. Parallel volume rendering on the IBM Blue Gene/P. In *Proceedings of Eurographics Parallel Graphics and Visualization Symposium 2008*, 2008.

[112] Tom Peterka, Hongfeng Yu, Robert Ross, Kwan-Liu Ma, and Rob Latham. End-to-end study of parallel volume rendering on the IBM Blue Gene/P. In *Proceedings of ICPP '09*, pages 566–573, September 2009. DOI=10.1109/ICPP.2009.27.

[113] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of SC*, 2003.

[114] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Identifying and eliminating the performance variability on the ASCI Q machine. In *Proceedings of the 2003 Conference on High Performance Networking and Computing*, November 2003.

[115] Steve J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J Comp Phys*, 117(1):1–19, 1995.

[116] Rolf Rabenseifner. Optimization of collective reduction operations. In *Computational Science - ICCS 2004*, 2004.

[117] Mahesh Rajan, Courtenay T. Vaughan, Doug W. Doerfler, Richard F. Barrett, Paul T. Lin, Kevin T. Pedretti, and K. Scott Hemmert. Application-drivern analysis of two generations of capability computing platforms: The transition to multicore processors. *Concurrency and Computation: Practice and Experience*, to appear, to appear.

[118] Charles Reiss, Gerald Lofstead, and Ron Oldfield. Implementation and evaluation of a staging proxy for checkpoint I/O. Technical report, Sandia National Laboratories, Albuquerque, NM, August 2008.

[119] Rolf Riesen, Ron Brightwell, Patrick Bridges, Trammell Hudson, Arthur Maccabe, Patrick Widener, and Kurt Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, August 2008.

[120] Rolf Riesen, Ron Brightwell, Kevin Pedretti, Brian Barrett, Keith Underwood, Arthur B. Maccabe, and Trammell Hudson. The Portals 4.0 message passing interface. Technical Report SAND2008-2639, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, April 2008.

[121] Rolf E. Riesen, Kevin T. Pedretti, Ron Brightwell, Brian W. Barrett, Keith D. Underwood, Trammell B. Hudson, and Arthur B. Maccabe. The Portals 4.0 message passing interface. Technical Report SAND2008-2639, Sandia National Laboratories, April 2008.

[122] R B Ross, T Peterka, H-W Shen, Y Hong, K-L Ma, H Yu, and K Moreland. Visualization and parallel I/O at extreme scale. *Journal of Physics: Conference Series*, 125(012099), 2008. DOI=10.1088/1742-6596/125/1/012099.

[123] Yousef Saad and Martin H. Schultz. GMRes: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[124] Sandia National Laboratory. LAMMPS molecular dynamics simulator. `http://lammps.sandia.gov`, Apr. 10 2010.

[125] Sandia National Laboratory. Mantevo project home page. `https://software.sandia.gov/mantevo`, Apr. 10 2010.

[126] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[127] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, June 2006.

[128] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022, 2007.

[129] M. Schulz, A. Arowojolu, J. Blanchard, J. Brandt., S. Futral, J. Mellor-Crummey, B. Miller, D. Montoya, M. Rajan, K. Roche, and Zosel. M. Tools and tool support for the exascale era from the nnsa workshop of exascale computing technologies. LLNL-TR-472494, 2011.

[130] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings ofSupercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[131] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.

[132] K. Uhlemann, C. Engelmann, and S.L. Scott. JOSHUA: Symmetric active/active replication for highly available hpc job and resource management. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[133] Keith D. Underwood, Michael Levenhagen, and Arun Rodrigues. Simulating Red Storm: Challenges and successes in building a system simulation. In *21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, March 2007.

[134] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33:103–111, August 1990.

[135] C.T. Vaughan, M. Rajan, R.F. Barrett, D.W. Doerfler, and K.T. Pedretti. Investigating the Impact of the Cielo Cray XE6 Architecture on Scientific Application Codes. In *Workshop on Large Scale Parallel Processing, at the IEEE International Parallel & Distributed Processing Symposium (IPDPS) Meeting*, 2011. SAND 2010-8925C.

[136] Adam Wagner, Darius Buntinas, Ron Brightwell, and Dhabaleswar K. Panda. Application-bypass reduction for large-scale clusters. In *Proceedings 2003 IEEE Conference on Cluster Computing*, December 2003.

[137] Peter Walker, David P. Bunde, and Vitus J. Leung. Faster high-quality processor allocation. In *Proceedings of the 11th LCI International Conference on High-Performance Clustered Computing*, 2010.

[138] Deborah Weisser, Nick Nystrom, Chad Vizino, Shawn T. Brown, and John Urbanic. Optimizing job placement on the Cray XT3. In *Proceedings of the Cray User Group Annual Technical Conference*, 2006.

[139] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Scott Klasky, Qing Liu, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. PreDatA - preparatory data analytics on Peta-Scale machines. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1–12, April 2010.

[140] Ziming Zheng and Zhiling Lan. Reliability-aware scalability models for high performance computing,. In *Cluster'09: Proceedings of the IEEE conference on Cluster Computing*, 2009.

DISTRIBUTION:

	MS	,
		,
1	MS 0899	RIM-Reports Management, 9532 (electronic copy)

Sandia National Laboratories