Coping at the User-Level with Resource Limitations in the Cray Message Passing Toolkit MPI at Scale: How Not to Spend Your Summer Vacation

Richard T. Mills, Forrest M. Hoffman, Patrick H. Worley, and Kalyan S. Perumalla
Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA

Art Mirin
Lawrence Livermore National Laboratory, Livermore, CA 94551-0808

Glenn E. Hammond
Pacific Northwest National Laboratory, Richland, WA 99352

Barry F. Smith
Argonne National Laboratory, Argonne, IL 60439

ABSTRACT: *As the number of processor cores available in Cray XT series computers has rapidly grown, users have increasingly encountered instances where an MPI code that has previously worked for years unexpectedly fails at high core counts ("at scale") due to resource limitations being exceeded within the MPI implementation. Here, we examine several examples drawn from user experiences and discuss strategies for working around these difficulties at the user level.*

KEYWORDS: Message Passing Toolkit, MPT, Portals, Message Passing Interface, MPI, MPICH

1 Introduction

The number of processing elements being deployed in the Cray XT series of computers has grown at a prodigious pace: The Red Storm XT3 machine at Sandia National Laboratories debuted at number 10 in the TOP500 list in June 2005 with 5 000 processor cores; three and a half years later, the Jaguar XT5 machine at Oak Ridge National Laboratory occupied the number 2 spot in the list with 150 152, more than thirty times that of Red Storm's debut count! With this rapid increase, many users have encountered situations in which MPI codes that have previously worked very reliably fail at large core counts—with puzzling error messages from the Cray Message Passing Toolkit (MPT)—due to message passing resources being exhausted.

Many of the aforementioned problems can be satisfactorily addressed by setting appropriate environment variables (see [1]) to increase resource limits or specify different algorithms for the MPI implementation to use. There are, however, several reasons that it may be preferable to instead rely upon flow-control mechanisms implemented in user code. For instance, problems with resource limitation might be encountered only in portions of an application code in which little execution time is spent. In such cases, using environment variable settings that significantly reduce the amount of memory available to the application or that have deleterious effects on performance are likely not a worthwhile trade to get the piece of problem code to run. In other cases, it may not be possible to increase the resources devoted to the message-passing library in a way that scales to high processor counts while maintaining adequate memory and performance. Finally, a user-level approach may be worthwhile because it does away with the need for much trial-and-error to determine adequate environment variable settings are

core counts are increased.

In our interactions with other Cray XT series users, we have found that many users have implemented user-level flow control schemes in their application codes. Their various solutions appear to have been arrived at independently, and we are aware of no references to which Cray XT users can refer for ideas on how to address flow-control problems. This paper aims to partially fill this void by presenting several case studies illustrating some ways in which application developers have coped with message passing resource limitations on the Cray XT series machines.

2 Resource Limitations in the Cray Message Passing Toolkit Implementation of MPI

Cray provides an implementation of MPI as part of the Cray Message Passing Toolkit (MPT) [2]. The implementation in the current version of MPT (3.1.x) is based on MPICH2 from Argonne National Laboratory and supports two abstract device interfaces (ADI3): the Portals interface for communication between nodes and the SMP ADI3 for on-node communication.

Portals [3] is a low-level software interface for inter-node communication. Its original design was focused on performing MPI communication, although other high-level communication interfaces can be built on top of it. Like many message passing libraries, Portals uses a so-called *eager* protocol for sending short messages: short messages are sent with the assumption that the receiving process has the resources to store the message, and the receiver is responsible for buffering the message if the matching receive has not yet been posted. If the matching receive has been posted, the data are placed into that receive's buffer; otherwise, the data are placed in the *unexpected buffer* and two entries will be generated in the *unexpected event queue* that tracks incoming unexpected messages: a *put start* event that is generated as the data begin to arrive, and a *put end* event that is generated when the data have been received and are ready to be used. Exhaustion of the *unexpected buffer* and exceeding the maximum length of the *unexpected event queue* are two of the most common resource limitation problems encountered when scaling up application codes. The most obvious way to address these problems is to increase the buffer size and maximum queue length by increasing the `MPICH_UNEX_BUFFER_SIZE` and `MPICH_PTL_UNEX_EVENTS` environment variables, respectively. Doing so decreases available memory, however, and it may not be practical (or even possible, if sends are posted much more rapidly than receives) to increase these sufficiently. The number of unexpected messages can also be decreased by lowering the value of the `MPICH_MAX_SHORT_MESSAGE_SIZE`, which specifies the maximum size of messages to be classified as "short" and sent using the eager protocol: longer messages utilize a different protocol and do not utilize the *unexpected buffer* and *unexpected event queue*. If some combination of these approaches does not work, a flow control mechanism can be enabled by setting the environment variable `MPICH_PTL_SEND_CREDITS` to '-1'. This mechanism should prevent the *unexpected event queue* from being exhausted in any situation, possibly with a negative impact on performance.

There is another Portals event queue that may also become problematic as the MPI process count grows: the *other events queue*, which handles all other MPI-related Portals events. MPI-2 remote memory access (RMA) requests, sending of data (*send end* and *reply end* events), and pre-posted receives all generate events in this queue. Ironically, this means that restructuring an application to pre-post receives to avoid failures due to too many unexpected messages may in fact result in failures due to too many *other* events being generated! The size of the *other events queue* can be increased by setting the `MPICH_PTL_OTHER_EVENTS` environment variable, but there is no flow control mechanism that can be enabled to limit the number of *other* events.

Since MPT version 3.0, the SMP abstract device has been available for handling intra-node communication. Although problems with resource limitations at scale usually show up in the Portals device, sometimes application codes exceed the maximum number of internal MPI message headers, and the limit has had to be increased via the environment variable `MPICH_MSGS_PER_PROC`. This should be much less of a problem with MPT 3.1.x, which dynamically allocates space for more message headers in quantities of `MPICH_MSGS_PER_PROC`.

3 Application Case Studies

In this section we describe examples detailing how MPT resource limitations ave been addressed in five disparate application codes.

3.1 Parallel k-means Cluster Analysis

Cluster analysis is a statistical method for grouping like objects together based on their multivariate characteristics. Cluster analysis is often applied to geophysical data analysis, and Hargrove and Hoffman have applied cluster analysis to a variety of environmental science domains, including ecological regionalization; environmental monitoring network design; analysis of satellite-, airborne-, and ground-based remote sensing, and climate model-model and model-measurement intercomparison [4]. For these applications, Hoffman and Hargrove have developed a scalable, parallel k-means clustering algorithm for use on high performance computing platforms that can be applied as a data mining tool to analyze and compare very large data sets of high dimensionality, such as very long or high frequency/resolution time series measurements or model output. Originally developed and implemented on a 128-node Beowulf-style parallel computer, called the Stone SouperComputer and constructed from surplus commodity desktop PCs [5], the high-performance parallel clustering algorithm [6] scales to at least thousands of processors. More recently, algorithm improvements that significantly reduce the time-to-solution by exploiting the triangle inequality have been implemented into the parallel clustering code [7].

The clustering method consists of two parts: initial centroid determination and iterative assignment of points to centroids until convergence is reached. In the iterative portion of the clustering algorithm, each point is assigned to the cluster centroid to which it is closest, by simple Euclidean distance, in the data space formed by using each characteristic as an axis in this n-dimensional space. After all the points are assigned to a cluster, new positions are calculated for each centroid as the mean value along every axis of the points assigned to that centroid. This procedure of assigning points to centroids and recomputing the centroid locations repeats until the number of points that change cluster assignment drops below a convergence threshold. Once this threshold is met, the final cluster assignments and centroid locations are saved. This algorithm is implemented with a traditional master/slave parallel architecture using the Message Passing Interface (MPI). The algorithm is nearly perfectly parallelizable and produces the same result whether run in serial or in parallel. In each iteration, the master process distributes the current centroids to the slave processes, assigns blocks of points to slaves for classification (*i.e.*, assignment to nearest centroid), collects those classifications, and recomputes new centroid locations based on cluster membership. This procedure repeats until convergence is attained. The block size is specified by the user and can be set to optimize code performance on various parallel systems with different numbers of processors, amounts of memory, and I/O characteristics.

In porting and benchmarking the cluster analysis code on a Cray XT4, it was discovered that the program crashed with MPI/Portals errors at certain process counts. In particular for the AmeriFlux IIIA data set with $k = 8000$ clusters, the code performs very well using 1 025 cores (1 024 slaves plus 1 master), it crashes at 2 049 cores, and runs to completion at 4 097 cores, albeit with longer run time than with 1 025 cores because of increased communication and an insufficient quantity of work. Specifically, with Cray MPT 3.0.3, the program aborted with the following error:

```
[128] MPICH has run out of unexpected
buffer space.
Try increasing the value of env
var MPICH_UNEX_BUFFER_SIZE (cur
value is 62914560),
and/or reducing the size of
MPICH_MAX_SHORT_MSG_SIZE (cur
value is 128000).
aborting job:
out of unexpected buffer space
```

This error appeared to occur in the *MPI_Allgatherv()* call that collects from all the slave processes the 8000×8000 element array of sorted centroid distances used in the accelerated version of the algorithm. In an attempt to get the code to run, the `MPI_UNEX_BUFFER_SIZE` environment variable was set to twice the default (shown in the error message above), but the program still aborted with the same error, listing the larger current value for `MPI_UNEX_BUFFER_SIZE`. Next, `MPI_UNEX_BUFFER_SIZE` was set to four times the default, and this time the program ran to normal completion in 28 min. With Cray MPT

3.1.0, the same problem runs in 28 min when MPI_UNEX_BUFFER_SIZE is set to four times the default value, but curiously, it runs in 14 min when the environment variable is not set and the default value is used.

Additional development was performed on the clustering code to pre-post receives on the master process to improve communications performance, based on Cray recommendations. When the master process assigns blocks of data to slave processes, the master first pre-posts a receive (using *MPI_Irecv()*) that is satisfied when the slave process completes processing the assigned data block. On a vanilla Linux cluster using only a few processes, runtime was improved by a few percent; however, when tested on the XT4 with the same problem described above using 2 049 processes, the program crashed with the following error:

```
[0] :  (/tmp/ulib/mpt/nightly/3.1/
112008/mpich2/src/mpid/cray/src/adi/
ptldev.c:2854) PtlMEMDPost()
failed :  PTL_NO_SPACE
aborting job:
PtlMEMDPost() failed
```

Apparently, the 2 048 pre-posted receives exceeded some Portals resource limit. A space limitation was unexpected because only a single long integer is to be received from each slave process. This Portals error was eliminated by disabling the registration of receive requests in Portals by setting the MPICH_PTL_MATCH_OFF environment variable. However, doing so results in ∼15% longer runtime than the previous code on the same problem. In this case, pre-posting receives requires disabling a communications feature on the XT4 and has a deleterious effect on performance.

3.2 Subsurface Flow and Reactive Transport (PFLOTRAN)

PFLOTRAN [8; 9; 10; 11] is a parallel code for simulation of multiphase flow and multicomponent reactive (geochemical) transport in porous media. The code is composed of different modules for flow and transport, with the option of running the modules in coupled or decoupled mode. PFLOTRAN is capable of simulating fluid flow though porous media with the following fluid phases: air, water, supercritical CO_2. PFLOTRAN-generated fluid flow velocities or fluxes are utilized by the transport module to compute solute transport. Within PFLOTRAN, transport and reaction are fully coupled. PFLOTRAN's problem domain is discretized spatially using an integrated finite volume approach, with fully-implicit backward-Euler time differencing. The use of fully-implicit time stepping leads to large, sparse systems of algebraic equations that must be solved at each time step, and the great majority of the execution time is spent in such solves. PFLOTRAN is built on top of the PETSc framework [12; 13; 14] and employs numerous features from PETSc, including nonlinear and linear solvers, sparse matrix data structures (both blocked and non-blocked matrices), vectors, constructs for managing parallel communications for structured mesh problems, and binary I/O.

PFLOTRAN's parallel paradigm is based on domain decomposition: each MPI process is assigned a subdomain of the problem and a parallel solve is implemented over all processors. Message passing (3D "halo exchange") is required at the subdomain boundaries with adjacent MPI processes to fill ghost points in order to compute flux terms. Domain decomposition preconditioners are usually employed inside of a global inexact Newton-Krylov solver. Within the Krylov solver, numerous *MPI_Allreduce()* operations are required to compute vector inner products and norms, and communication is highly latency-bound. In addition to these "computation phase" communications mentioned above, the other significant source of communication occurs during I/O. The output files used for visualization and analysis are written using Parallel HDF5 (which employs MPI-IO). PFLOTRAN also generates checkpoint files using the binary PetscViewer format.

Despite the significant amount of communication performed in PFLOTRAN, we have actually found that most phases of the code are fairly robust in terms of MPT resource limits and have required little adjustment of the default settings. The exception has been the generation of the binary PetscViewer checkpoint files. Here, we detail our experiences in investigating and correcting this problem.

PFLOTRAN checkpoint files are relatively lightweight and essentially consist only of a few scalar values and a handful of PETSc vectors, which are written via calls to PETSc's *VecView()* function. The default behavior of VecView is to send all entries of the vector through process 0 for writing to disk: All processes with nonzero rank post a send to

process 0, and process 0 loops through the receives, writing the portion of the vector to disk before processing the next receive. For jobs of any significant size, this of course causes problems with the *unexpected buffer* and *unexpected event queue*, and for very large jobs the size of these must be increased to impractical size for the *VecView()* to complete.

A very simple attempt we tried was changing the *MPI_Send()* to the synchronous version, *MPI_Ssend()*. Our hope was that this not incur any buffering by the MPI implementation, as the MPI standard offers the following advice to implementors: "Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation." This failed to fix our problem, however.

Our first real attempt to address this problem was to add a backend to *VecView()* that utilized collective MPI-IO writes. This would address the unscalable performance of the default *VecView()* behavior (though this was not a real problem in PFLOTRAN, since comparatively little time is spent writing checkpoints) and would also prevent resource exhaustion due to so many simultaneous sends being posted to process 0. Unfortunately, although this did allow the code to run at somewhat higher core counts, we quickly encountered a new failure mode in which Portals reported "`PtlMEMDPost() failed : PTL_NO_SPACE`". We were unable to determine why this message appeared.

As our MPI-IO backend did not work, we implemented a simple flow-control scheme that proved successful. The processes of nonzero rank are divided into groups of `flow_control_group_size` processes. All processes participate in a series of broadcasts, in which process 0 sends out a minimum rank, increased in increments of `flow_control_group_size`, that a process must possess before it is permissible for it to send its message to process 0. Although the numerous broadcasts mean a deal of wasteful communication, our performance tests indicated that the flow-control scheme makes no discernible difference in execution time, likely because the performance is entirely limited by the rate at which process 0 can write to disk. We later implemented a more elegant scheme in which we divide the nonzero rank processes into disjoint sets of size `flow_control_group_size`. An MPI sub-communicator is then formed from the union of each disjoint set with process 0. At the initiation of the *VecView()* call, each process issues an *MPI_Barrier()* on its sub-communicator. Process 0 joins each barrier only when it is ready to process the receives for those processes. This scheme eliminates the numerous broadcasts of the first scheme, though we have found that it makes no practical difference for the message sizes typical in PFLOTRAN.

We note that the `PTL_NO_SPACE` error that we encountered when using our MPI-IO backend have since been eliminated. When attempting to use this backend with the default environment variable settings in MPT 3.1.x, we encounter a somewhat different error: "`PtlMDBind failed with error : PTL_NO_SPACE`". This problem proves easily fixed by setting `MPICH_MPIIO_CB_ALIGN` to 1, which enables new algorithms that align collective buffering file domains on Lustre boundaries. The MPI-IO version of *VecView()* now scales to large processor counts and displays appreciable speedup as well.

3.3 Community Atmosphere Model (CAM)

The Community Atmosphere Model (CAM) [15] is an atmospheric general circulation model (AGCM) that has been developed at the National Center for Atmospheric Research (NCAR), with contributions from external National Science Foundation (NSF), Department of Energy (DOE), and National Aeronautics and Space Administration (NASA) funded researchers. CAM is also the atmospheric component of the Community Climate System Model [16; 17]. CAM is characterized by two computational phases: the dynamics, which advances the evolutionary equations for the atmospheric flow, and the physics, which approximates subgrid phenomena such as precipitation processes, clouds, long- and short-wave radiation, and turbulent mixing. Control moves between the dynamics and the physics during each model simulation timestep. The approach to parallelization in CAM is domain decomposition, where each subdomain is assigned to a single MPI [18] process; when available, OpenMP [19] is used for additional parallelization. The dynamics and physics each use separate decompositions, and in typical usage the dynamics itself uses two different decompositions. These decompositions need not be the same size, enabling, for example, the use of more active processes in the physics than in the dynamics. These decompositions are linked by "transposes" of the associated distributed arrays at each model timestep, and are implemented with

MPI collectives or point-to-point commands. The decomposition strategy also requires halo-update-like communications between logically neighboring processes when calculating the wind velocities and in the advection of other quantities. A number of distributed sums are also calculated during each timestep. Model input requires reading input files by a subset of "reader" processes, followed by a scatter to the rest of the processes. Model output requires a gather to a subset of "writer" processes before the data are written to the parallel file system. Model input is most prevalent during the model initialization phase, but tables of data are read periodically during a model run. Output frequency is a runtime parameter, but typically occurs at least once per simulation month.

MPI communication patterns in CAM are static, but include both global and local communication operators, and both bandwidth (large message) and latency-sensitive communications. Performance portability requirements of this community code have led to the support of a large number of MPI communication protocols and options [20; 21; 22]. Adapting CAM to the current idiosyncrasies of MPI communication on the Cray XT systems has been relatively straightforward. Identifying the appropriate options has been more of a challenge.

In general, the default communication protocol has been to pre-post all receive requests, issue all (non-blocking) send requests, then wait for the receive requests to be satisfied. At scale, this has the potential of overwhelming any given process with messages for which it has not yet posted receive requests. This can cause failures if the system cannot allocate sufficient system buffer space to handle all of the requests, and will degrade performance in any case with all of the additional buffer copying. When there are different numbers of active processes in different phases of the code, the likelihood of this situation to occur increases significantly. For example, we noticed anomalously large communication times in MPI-only experiments on the XT4 and XT5 when transposing between two dynamics decompositions in which one decomposition was three times smaller. In this instance, the runtime for the model was 50% to 100% *slower* than when using one-third as many processes but using decompositions with the same size. Apparently the early arrival of messages from the otherwise idle two-thirds of the processes at the one-third active processes was causing the performance anomaly. Similar performance problems have been observed in both gather and scatter operations (using both MPI collective calls and point-to-point implementations), resulting in runs terminating with error messages indicating that, for example, MPI has "run out of unexpected buffer space" or that an event was "dropped." The first error message occurred on an XT5 during a gather associated with writing a restart file. This particular run used 4-way OpenMP parallelism and 256 MPI processes. The second error message occurred on an XT5 during a series of scatters as part of the initialization for an MPI-only run on 3 328 processors.

Setting appropriate MPI environment variables to larger values does eliminate the errors in the previous two examples. However, this is a fragile approach because a sufficient value is a function of the process count and problem size, and is difficult to predict. In some circumstances we have not been able to set the environment variable large enough, because it exhausts the available memory.

One option to address these issues is to use flow control in the form of handshaking messages. After each non-blocking receive is posted, a "zero-byte" message is sent to the source process. Upon receipt of this signal, the source process can send the message. This eliminates all unexpected messages of size greater than zero. There is still a potential problem in pre-posting more non-blocking receive requests than are supported on a given system (with any given MPI environment variable settings). There may also be a performance impact from having a large number posted, if only in the cost of matching receive requests with the incoming messages. There is also a potential problem of overwhelming a given process with the handshaking messages. To address these issues, another option is to limit the maximum number of outstanding send and receive requests.

The handshaking and message request limit protocols allow us to address problems within the point-to-point implementation of a single logical collective operation. Problems can also arise from communication demands of a series of collective requests, for example the series of scatter requests described above, even though any single collective request may not cause a problem. Moreover, invoking a handshaking protocol in a scatter may replace one problem with another. We have found that replacing the non-blocking send requests with blocking sends in the point-to-point implementations can slow down the rate at which message requests are generated, slow enough to avoid problems arising from multiple

collective calls.

The second approach, using *MPI_Send()* instead of *MPI_Isend()*, eliminated the failure in the I/O-related scatters. The first approach, handshaking and message request limits, eliminated the failure in I/O-related gathers and eliminated the performance problem when transposing between two different-size decompositions. In certain situations this approach also improved performance significantly (up to a factor of 5) in I/O-related gathers that had not been failing. Using *MPI_Alltoallv()* also decreased the performance anomaly, but the point-to-point implementation with handshaking is still approximately twice as fast for transposes between the two decompositions, resulting in a 10% improvement in model runtime in typical cases as compared to the implementation using the MPI collective. However, in our experience *MPI_Alltoallv()* is faster than the point-to-point implementation when all processes are sending and receiving from all other processes.

3.4 XGC1

XGC1 is a five-dimensional (three-dimensional real space and two-dimensional velocity space) particle-in-cell code used to study turbulent transport in magnetic confinement fusion plasmas. XGC1 is able to model the plasma and neutral species in the edge region of a tokamak, allowing the first full-function gyrokinetic simulation of whole device tokamak plasma in diverted geometry. XGC1 has been developed by S. Ku and C. S. Chang within the Department of Energy SciDAC project "Center for Plasma Edge Simulation" (CPES), with contributions from the CPES project team [23].

Each timestep of an XGC1 execution includes, minimally, the following.

1. Calculate charge density on underlying grid (from particles);
2. Solve gyrokinetic Poisson equation on grid;
3. Calculate electric field and time derivatives used in integration of particle equations of motion;
4. (periodically) Calculate and output diagnostic quantities;
5. Calculate new particle positions and velocities.

This is a simplified view of the algorithm in that these steps are used within a Runge-Kutta or predictor-corrector time integration method. Depending on the experiment configuration, particles are ions, electrons, or both. Experiments can also include collisions and other physical processes important to full device simulations.

Parallelization of XGC1 is based on decompositions of both the spatial grid and of the particles. In particular, the assignment of particles to processes is based on a decomposition of the spatial domain. Minimally, both the spatial grid and the particles decompositions utilize a one-dimensional decomposition in the toroidal direction of the toroidal geometry.

All but one of the above steps (step 3) requires MPI communication in the parallel implementation. The charge density calculation step requires MPI communication between processes assigned adjacent parts of the spatial grid in neighboring poloidal planes (neighboring slices in the one-dimensional toroidal decomposition), as well as two distributed reductions. The Poisson problem is solved using (numerical and parallel) algorithms supplied by the PETSc library [12; 13; 14]. MPI communication in the diagnostic routines is limited to gathers and reductions. The calculation of the new particle positions requires reassigning some of the particles to different processes.

A recent modification to XGC1 introduced an option to decompose particles using a two-dimensional spatial decomposition. The initial implementation of the particle reassignment required in step 5 was as follows. For each process i,

1. determine the number of particles being sent to i (*MPI_Allreduce()* call);
2. send all particles that need to go off process (Pack send buffers and call *MPI_Isend()*);
3. rearrange particles to remove holes in data structures introduced by particles that have moved off process;
4. receive particles sent from other processes (call *MPI_Recv()* and copy into local data structures, repeating until the required number of particles has been received).

For small numbers of processes (no more than 8 000), this communication algorithm is very efficient. Unfortunately, for larger process counts it fails on the Cray XT4 and Cray XT5. Since the receives are not posted before the sends, and due to the work

between the sends and the receives, many of these sends are going into MPI system buffer space on the destination processes. Eventually there is no more space available to receive the messages and the program fails. Note that due to the nature of the simulations, the number of processes sending particles to any given process is typically not large, but the number of particles received can be large.

Initial alternative implementations, for example, introducing flow control into the original algorithm, eliminated runtime failure. However, they were significantly more expensive than the original algorithm at process counts for which both worked. After examining a number of alternatives, the following was determined to be robust and as efficient as the original algorithm.

1. pack send buffers with particles that need to go off processor;
2. determine number of processes sending to i (*MPI_Allreduce()* call);
3. determine which processes are sending to i, and how many particles each is sending (*MPI_Irecv()* and *MPI_Send()* calls);
4. post receive requests (*MPI_Irecv()* calls);
5. post send requests (*MPI_Isend()* calls);
6. receive particles and copy into local data structures

with options for handshaking messages and limiting the number of outstanding MPI requests in steps 4, 5, and 6. The default is to use both handshaking and to limit the number of requests. With this logic, XGC1 has been run successfully with over 20 000 processes, and runs on 150 000 cores will be attempted in the near future. Based on our experiences with the Community Atmosphere Model, we also introduced flow control options into the gather algorithms used in the diagnostic routines, and have set these as the default.

3.5 Parallel Discrete Event Simulation (PDES)

PDES is a class of simulations representing a wide range of emerging, large-scale applications, such as packet-level Internet simulations, agent-based social behavioral models, epidemiological disease spread models, vehicular traffic models, and air traffic control simulations, to name a few.

Here, we document our difficulties in attempting to port and scale a unique PDES engine, μsik, to the NCCS Jaguar machine (Cray XT5 with approx. 150 000 processor cores). μsik has been previously demonstrated to scale to 32 768 processor cores on the Blue Gene/L platform. Our interest was to begin porting μsik to the full scale of 150 000 processor cores of the Cray XT5, planned as a series of phased improvements to the μsik software, algorithms and application benchmarking. However, the port encountered unforeseen difficulties in the first phase itself, many of which could be traced to the MPI communication subsystem of the Cray XT5.

We document the details of some of these difficulties, along with our current (sub-optimal) ad-hoc solutions (much additional optimization beyond what is reported here is clearly possible).

We will first briefly review the PDES execution style, followed by its ramifications on MPI-based communication patterns, the nature of some of our problems on our Cray XT5 port, and our ad-hoc solutions.

3.5.1 Modeling Paradigm and Parallel Execution Style

Parallel discrete event simulation (PDES) is a modeling and execution paradigm that is in direct contrast to time-stepped simulations [24]. In parallel time-stepped simulations, simulation time is globally advanced in fixed increments, with all processors executing in synchrony with respect to the time-step of their state updates. In contrast, in PDES, simulation time can be an arbitrary real value (typically, a double precision floating point value), with state updates "scheduled" to be executed in arbitrary time instants in the future on the real-valued simulation-time axis. Each processor can potentially be updating its system state at simulation time instants that are later or earlier than the simulation time of other processors. Processors exchange data using "timestamped" events scheduled by one processor to another. Events, which are payload (data) tagged with a timestamp value, are thus scheduled for arbitrary times in the future. To ensure global causality, events are constrained to be executed at every processor in non-decreasing timestamp order.

The schedulability of updates, the timestamp-ordered event execution, and the staggered processing of events across processors, all together trans-

late into a unique parallel execution style of PDES. The difference in styles ends up exercising the communication subsystems of the parallel platform in ways and combinations not typically encountered by traditional time-stepped simulations. Processors are typically highly staggered in simulation time. Due to such staggered execution, even small levels of local jitter due to communication overheads can accumulate globally. Message sizes are typically small (in the range of 64 to 512 bytes each), but the number of messages and the frequency of sends can be very high (10^2–10^4 of inter-processor messages per second). Non-blocking messaging is used heavily, to permit the processors to overlap computation and communication. All these messaging characteristics serve to stress the communication subsystems, amplifying even the slightest inefficiencies and overheads.

3.5.2 Application Interface and the μsik Engine

In PDES, one or more units, called logical processes (LPs), are mapped to each processor core. The LPs are abstractions that correspond to modeling units that interact asynchronously with each other by scheduling events to each other in simulation time future. For example, each LP represents either an agent in an agent-based simulation (exchanging timestamped events representing social behavioral influences), or a host in an Internet simulation (exchanging timestamped events representing TCP/IP packet exchanges), or an intersection in a transportation network simulation (exchanging timestamped events representing vehicular arrivals and departures).

Due to the unique communication and synchronization properties, PDES "engines" are used to implement and reuse a wide variety of important functionalities and concepts such as logical processes, timestamped events, global timestamp-ordered event processing at every processor, dynamic concurrency enhancement, and so on.

μsik is a scalable PDES engine, under development for over half a decade, providing many interface features needed in a large variety of applications. μsik's scalability has been gradually increased from 10^2 to 10^4 processors; more recently (2006–07), we were able to scale μsik to a large Blue Gene/L installation, demonstrating scalability to 32 768 cores. The software is written in standard C/C++, with support for multiple communication subsystems, including MPI. Additional detail can be found in [25; 26].

3.5.3 Computation and Synchronization Granularities

Many PDES applications involve very fine-grained event computation. The granularity can be as low as 2–10 μs in applications such as Internet simulations and vehicular traffic simulations. In other words, the execution of model-level code computation only consumes 2–10 μs inside each event. Notionally, PDES execution requires synchronization (safety against timestamp order violation) after processing every event, and/or scheduling one or more events to another logical process (potentially off-processor). In practice, this strict notion of synchronization gets somewhat relaxed by using concepts such as "look-ahead."

For the purposes of this article, the PDES execution of each processor can be expressed as the regular expression: $((E[S+]) * [Y])*$, where E is an event execution, $[S+]$ is an (optional) scheduling of one or more events, and $[Y]$ is a (dynamically determined) participation in synchronization with other processor-cores, and $*$ represents zero or more repetitions.

3.5.4 MPI-based Implementation

Many PDES applications are unstructured in nature. Due to the unstructured state spaces, inter-state dependencies, and staggered time evolution, no processor has knowledge of to which processor it might send a message and from which processor it should expect to receive. As a result, it is infeasible to post receives before sends. However, non-blocking communication is a necessity to deal with dynamic load imbalances. With these considerations, the non-blocking, buffered-send *MPI_Bsend()* is used to send messages, and the non-blocking *MPI_Iprobe()* is used to poll for incoming messages, and *MPI_Recv()* is used to retrieve probed messages.

While an event is being processed, that event could generate new events destined for other processors. Such events are immediately sent using *MPI_Bsend()*. Since many events could be sent in this manner before the MPI subsystem can dispatch them, MPI is given user buffer space at initialization via *MPI_Buffer_attach()*. The space allocated for this is in tens of megabytes to a couple of hundred megabytes (per core), which has been found to

be more than adequate on the Blue Gene/L platform on up to 32 768 cores.

Plain MPI is used, with one MPI rank per core. No multithreading (OpenMP) is used.

3.5.5 PDES Benchmark: PHOLD

The PHOLD benchmark application is a parameterized abstract model, designed as a generalized core of parallel discrete event simulations, modeling multiple entities that interact via time stamped events/messages.

Good performance of the PDES engine on the PHOLD benchmark is a necessary condition for good performance of PDES applications.

The benchmark is designed to capture the essence of simulation dynamics to exercise and experiment with computational performance of event-based model execution in parallel. Most interacting-entity simulations map well to this model.

In PHOLD, the interactions are abstracted by each unit selecting a target unit at random for interaction in the future. Each interaction is realized as a timestamped event sent from one LP to another.

In our experiments, we use weak scaling, with 10 LPs per core. An event population of 1 000 events per core, which implies that, at any given moment, there are an average of 1 000 events active per core, all with exponentially distributed timestamps into the future. Every LP, when it processes an event (in timestamp order), sends an event with a randomly chosen timestamp increment into the future to a random LP. The event is scheduled in the timed future with a period drawn from an exponential distribution with mean 1.0, plus a minimum increment of 0.1 (*i.e.*, a "look ahead" of 0.1). A percentage ρ of events is biased to be sent to a local core LP, while the others are sent to randomly selected remote LPs. Typical values for ρ are 80–99%, giving reasonably high locality, and low inter-processor communication.

3.5.6 Runtime Issues and Ad-hoc Workarounds

The PHOLD benchmark was initially executed with $\rho = 90$, to match the inter-processor communication percentage used in our earlier runs on the Blue Gene/L. As expected, everything worked flawlessly until core counts of 32 768. However, execution on the next doubling of cores resulted in fatal runtime errors (segmentation faults). The source of the problem was hard to detect, partly because of the difficulty of discovering the MPI error message (about the overflow of the unexpected event queue and the need to increase the queue size using the `MPICH_PTL_UNEX_EVENTS` environment variable) lost in the standard output of the simulation. Exhaustion of network resources resulted in generation of this error.

Our understanding of the reason for the segmentation fault is that the (apparent) transient loss of some events seemed to violate the FIFO message ordering assumptions made in the code, which made the engine enter an invalid software state (invalid event buffer space). Notwithstanding the reason behind the runtime error, it was clear that the cores needed to be throttled, using some kind of flow control mechanism. Alternatively, the amount of communication needed to be reduced.

We first experimented with the easier alternative, namely, reducing the inter-processor communication by one order of magnitude, increasing ρ from 90 to 99 percent. The problem persisted—this is because of the random selection of the destination for the messages, spanning the entire range of MPI ranks, which makes the destination list highly dynamic when only a small number of "active connections" per processor are established and maintained by the Cray's network system. This active connection table was easily being flushed and refilled with newer connections established on demand to newer destinations at runtime, resulting in significant latencies. While the application managed to terminate without errors, the performance degraded by one to two orders of magnitude (with hundreds of microseconds per event, as opposed to an average of 10–20 μs per event on 32 768 cores). When the inter-processor communication was decreased by an additional one order of magnitude (increasing ρ to 99.9%), the average event cost decreased to 400–600 μs per event. This confirmed that the inter-processor communication was the contributing factor to the total cost.

In order to get decent performance on the desired value range of inter-processor communication, we implemented a simple flow control algorithm at the user level. For every sender-receiver pair of cores exchanging events, a user-level (MPI message-based) acknowledgement is sent by the receiver once every k events, and the sender stalls to receive the acknowledgement message before transmitting the next $(k+1)^{th}$ message. This scheme ensures that the network subsystem is burdened by at most k

messages for any given (i, j) ordered pair of MPI ranks. This flow control scheme improved the performance dramatically, bringing the amortized event cost down to around 100 μs on 100 000 cores. Another modification that led to this improvement was the use of message bundling such that *MPI_Bsend()* was invoked only outside of an event computation (instead of during event computation), making it somewhat less vulnerable to the delays in *MPI_Bsend()* processing. Nevertheless, the event performance of 100 μs is far from the expected levels of 10–20 μs for the low inter-processor communication percentage of $\rho = 99.9$.

The selection of k is ad-hoc in nature, and is chosen by trial-and-error empirically. Larger k is good for concurrency, but is more prone to network errors. Smaller k ensures normal termination, but at significant loss of runtime efficiency. We are currently investigating additional solutions to this issue, including some of the environment variables for flow control in Cray MPT. An additional, drastic alternative is to move one level down in the network stack and use the Portals interface, which is the communication layer on which Cray XT5's MPT is based.

For PDES virtual time synchronization, μsik employs user-level messaging-based asynchronous reductions based on butterfly (and other) communication patterns to compute the global minimum of event timestamps at all processors (this global value, called global virtual time [27], is needed for proper execution order, and to detect termination). Suspecting that our implementation is a potential source of runtime cost, we attempted to use *MPI_Allreduce()* to compute the global minimum of event timestamps. To our amazement, the *MPI_Allreduce()*-based solution resulted in poorer performance, instead of improved performance. The performance differential reached close to one order of magnitude on the largest core counts of 147 576 cores.

Much remains to be investigated with respect to performance problems, most of which have been traceable to the MPI subsystem when our PDES runs were executed on core counts larger than 32 768.

4 Conclusions

In this paper, we have examined several case studies detailing modifications made to user codes in order to prevent exceeding resource limits within the Cray Message Passing Toolkit (MPT) implementation of MPI. Although it is often possible to remedy such problems by using appropriate settings of MPT environment variables, user-level solutions are sometimes preferable: they can eliminate the need for trial and error in determining environment variable settings when scaling to higher processor counts, they can avoid use of MPT settings that might be deleterious to the performance of other portions of the code, and they can preserve more physical memory for use by the application code. We hope that the examples presented here can provide guidance for other users who need to implement user-level solutions in their codes.

Acknowledgments

The authors wish to thank Jeff Larkin of Cray, Inc., for helpful conversations about issues encountered in MPT, and Dr. Vinod Tipparaju of ORNL for his insight and suggestions for flow control in the PDES application, and for putting the author of this code (Perumalla) in contact with the other authors of this paper.

This research was partially sponsored by the Climate and Environmental Sciences Division (CESD) of the Office of Biological and Environmental Research (BER) and the Computational Science Research and Partnerships (SciDAC) Division of the Office of Advanced Scientific Computing Research (ASCR) within the U.S. Department of Energy's Office of Science (SC). Portions of the PDES work reported here were supported under the DHS SERRI program. This research used resources of the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL), which is managed by UT-Battelle, LLC, for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Lawrence Livermore National Laboratory (LLNL) is managed by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy under Contract No. DE-AC52-07NA27344. Pacific Northwest National Laboratory is managed for the U.S. Department of Energy by Battelle Memorial Institute under Contract No. DE-AC06-76RL01830. Argonne National Laboratory is managed by UChicago Argonne, LLC, for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.

About the Authors

Richard Tran Mills is a computational scientist in the Computational Earth Sciences Group of the Computer Science & Mathematics Division at ORNL. After dropping out of high school, he earned a B.A. in Geology and Geophysics from the University of Tennessee, Knoxville, and a Ph.D. in Computer Science from the College of William and Mary. His research interests include parallel and high-performance computing, computational science and scientific computing, software for the iterative solution of sparse algebraic systems of equations, computational geoscience applications, geospatiotemporal data mining, and execution context-aware scientific software. He can be reached at ORNL, MS 6015, Oak Ridge, TN 37831, E-Mail: `rmills@ornl.gov`.

Forrest M. Hoffman is a computational scientist in the Computational Earth Sciences Group of the Computer Science & Mathematics Division at ORNL. His research focuses primarily on global climate, ecological, and terrestrial biogeochemical modeling and biogeophysical data mining. He can be reached at ORNL, MS 6016, Oak Ridge, TN 37831, E-Mail: `forrest@climatemodeling.org`.

Patrick H. Worley is a senior R&D staff member in the Computer Science and Mathematics Division of Oak Ridge National Laboratory. His research interests include parallel algorithm design and implementation (especially as applied to simulation models used in climate and fusion energy research) and the performance evaluation of parallel applications and computer systems. Worley has a Ph.D. in computer science from Stanford University. He is a member of the Association for Computing Machinery and the Society for Industrial and Applied Mathematics. E-mail: `worleyph@ornl.gov`.

Kalyan S. Perumalla is a senior research staff member in the Modeling & Simulation Group of the Computational Sciences & Engineering Division at ORNL, and holds an Adjunct Professor appointment at the Georgia Institute of Technology (Georgia Tech). His areas of interest include high performance computing, and parallel simulation/optimization applications. He can be reached at ORNL, MS 6085, Oak Ridge, TN 37831, E-Mail: `perumallaks@ornl.gov`.

Art Mirin is a computational physicist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research interests include scientific computing, high-performance computing, numerical hydrodynamics, and global climate modeling. He can be reached at LLNL, Box 808, L-561, Livermore, CA 94551, E-Mail: `mirin1@llnl.gov`.

Glenn Hammond is a computational hydrologist in the Hydrology Technical Group at Pacific Northwest National Laboratory. His research interests include high-performance computing and subsurface reactive transport. He can be reached via E-Mail at `ghammond@pnl.gov`.

Barry Smith is a Senior Computational Mathematician in the Mathematics and Computer Science Division at Argonne National Laboratory and is the leader of the PETSc project. He can be reached via E-Mail at `bsmith@mcs.anl.gov`.

References

[1] Geir Johansen. Managing cray XT MPI runtime environment variables to optimize and scale applications. In *Proceedings of the Cray User Group 2008 Meeting (CUG2008)*, 2008.

[2] Howard Pritchard, Doug Gilmore, Monika ten Bruggencate, David Knaak, and Mark Pagel. Message passing toolkit (MPT) software on XT3. In *Proceedings of the Cray User Group 2006 Meeting (CUG2006)*, 2006.

[3] Ron Brightwell, Bill Lawry, Arthur B. MacCabe, and Rolf Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 268, Washington, DC, USA, 2002. IEEE Computer Society.

[4] William W. Hargrove and Forrest M. Hoffman. Potential of multivariate quantitative methods for delineation and visualization of ecoregions. *Environmental Management*, 34(5):s39–s60, 2004. doi:10.1007/s00267-003-1084-0.

[5] William W. Hargrove, Forrest M. Hoffman, and Thomas Sterling. The Do-It-Yourself Supercomputer. *Scientific American*, 265(2):72–79, August 2001.

[6] Forrest M. Hoffman and William W. Hargrove. Multivariate geographic clustering using a Beowulf-style parallel computer. In Hamid R. Arabnia, editor, *Proceedings of the*

International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99), volume III, pages 1292–1298, Las Vegas, Nevada, June 1999. CSREA Press.

[7] Forrest M. Hoffman, William W. Hargrove, Richard T. Mills, Salil Mahajan, David J. Erickson, and Robert J. Oglesby. Multivariate Spatio-Temporal Clustering (MSTC) as a data mining tool for environmental applications. In Miquel Sànchez-Marrè, Javier Béjar, Joaquim Comas, Andrea E. Rizzoli, and Giorgio Guariso, editors, *Proceedings of the iEMSs Fourth Biennial Meeting: International Congress on Environmental Modelling and Software (iEMSs 2008)*, Barcelona, Catalonia, Spain, July 2008.

[8] Richard T. Mills, Chuan Lu, Peter C. Lichtner, and Glenn E. Hammond. Simulating subsurface flow and transport on ultrascale computers using PFLOTRAN. In David Keyes, editor, *SciDAC 2007 Scientific Discovery through Advanced Computing*, volume 78 of *Journal of Physics: Conference Series*, page 012051, Boston, Massachusetts, 2007. IOP Publishing.

[9] Glenn E. Hammond, Peter C. Lichtner, and Chuan Lu. Subsurface multiphase flow and multicomponent reactive transpo rt modeling using high-performance computing. In David Keyes, editor, *SciDAC 2007 Scientific Discovery through Advanced Computing*, volume 78 of *Journal of Physics: Conference Series*, page 012025, Boston, Massachusetts, 2007. IOP Publishing.

[10] Chuan Lu and Peter C. Lichtner. High resolution numerical investigation on the effect of conve ctive istability on long term co_2 storage in saline aqui fers. In David Keyes, editor, *SciDAC 2007 Scientific Discovery through Advanced Computing*, volume 78 of *Journal of Physics: Conference Series*, page 012042, Boston, Massachusetts, 2007. IOP Publishing.

[11] Glenn E. Hammond, Peter C. Lichtner, , Richard T. Mills, and Chuan Lu. Towards petascale computing in geosciences: application to the hanford 300 area. In David Keyes, editor, *SciDAC 2008 Scientific Discovery through Advanced Computing*, volume 125 of *Journal of Physics: Conference Series*, page 012051, Seattle, Washington, 2008. IOP Publishing.

[12] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[13] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[14] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[15] W. D. Collins, P. J. Rasch, B. A. Boville, J. J. Hack, J. R. McCaa, D. L. Williamson, B. P. Briegleb, C. M. Bitz, S.-J. Lin, and M. Zhang. The Formulation and Atmospheric Simulation of the Community Atmosphere Model: CAM3. *Journal of Climate*, 19(11):2144–2161, June 2006.

[16] Community Climate System Model. http://www.ccsm.ucar.edu/.

[17] W. D. Collins, C. M. Bitz, M. L. Blackmon, G. B. Bonan, C. S. Bretherton, J. A. Carton, P. Chang, S. C. Doney, J. H. Hack, T. B. Henderson, J. T. Kiehl, W. G. Large, D. S. McKenna, B. D. Santer, and R. D. Smith. The Community Climate System Model Version 3 (CCSM3). *J. Climate*, 19(11):2122–2143, 2006.

[18] W. Gropp, M. Snir, B. Nitzberg, and E. Lusk. *MPI: The Complete Reference*. MIT Press, Boston, 1998. second edition.

[19] L. Dagum and R. Menon. OpenMP: an industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January/March 1998.

[20] A.A. Mirin and W. B. Sawyer. A scalable implemenation of a finite-volume dynamical core in the Community Atmosphere Model. *International Journal of High Performance Computing Applications*, 19(3):203–212, Fall 2005.

[21] W.M. Putman, S. J. Lin, and B. Shen. Cross-platform performance of a portable communication module and the NASA finite volume general circulation model. *International Journal of High Performance Computing Applications*, 19(3):213–224, Fall 2005.

[22] P. H. Worley and J. B. Drake. Performance portability in the physical parameterizations of the Community Atmosphere Model. *International Journal of High Performance Computing Applications*, 19(3):187–202, August 2005.

[23] C. S. Chang, S. Klasky, J. Cummings, R. Samtaney, A. Shoshani, L. Sugiyama, D. Keyes, S. Ku, G. Park, S. Parker, N. Podhorszki, H. Strauss, H. Abbasi, M. Adams, R. Barreta, G. Bateman, K. Bennett, Y. Chen, E. D'Azevedo, C. Docan, S. Ethier, E. Feibush, L. Greengard, T. Hahm, F. Hinton, C. Jin, A. Khan, A. Kritz, P. Krsti, T. Lao, W. Lee, Z. Lin, J. Lofstead, P. Mouallem, M, Nagappan, A. Pankin, M. Parashar, M. Pindzola, C. Reinhold, D. Schultz, K. Schwan, D. Silver, A. Sim, D. Stotler, M. Vouk, M. Wolf, H. Weitzner, P. Worley, Y. Xiao, E. Yoon, and D. Zorin. Toward a first-principles integrated simulation of tokamak edge plasmas. *Journal of Physics: Conference Series*, 125(012042), July 2008.

[24] K. S. Perumalla. Parallel and Distributed Simulation: Traditional Techniques and Recent Advances. In *Proc. of the Winter Simulation Conference*, pages 84–95, 2006.

[25] K. S. Perumalla. μsik – A Micro-Kernel for Parallel and Distributed Simulation Systems. In *Proc. of the Workshop on Parallel and Distributed Simulation*, pages 59–68, 2005.

[26] K. S. Perumalla. Scaling Time Warp-based Discrete Event Execution to 10^4 Processors on a Blue Gene Supercomputer. In *Procs of the ACM Conference on Computing Frontiers*, pages 69–76, 2007.

[27] K. S. Perumalla and R. M. Fujimoto. Virtual Time Synchronization over Unreliable Network Transport. In *Proc. of the Workshop on Parallel and Distributed Simulation*, pages 129–136, 2001.