

Scalable Parallel Application Launch on Cplant™

Ron Brightwell Lee Ann Fisk*

Sandia National Laboratories
Scalable Computing Systems Department
PO Box 5800
Albuquerque, NM 87185-1110
[bright,lafisk]@cs.sandia.gov

ABSTRACT

This paper describes the components of a runtime system for launching parallel applications and presents performance results for starting a job on more than a thousand nodes of a workstation cluster. This runtime system was developed at Sandia National Laboratories as part of the Computational Plant (Cplant™) project, which is deploying large-scale parallel computing clusters using commodity hardware and the Linux operating system. We have designed and implemented a flexible runtime system that allows for launching parallel jobs on thousands of nodes in a matter of seconds. The interactions of the components are described, and the key issues that address the scalability and performance of the runtime system are discussed. We also present performance results of launching executables of varying sizes on more than a thousand nodes.

Keywords

massively parallel, runtime system, workstation cluster

1. INTRODUCTION

One of the challenges in any massively parallel processing system is providing a runtime environment that allows for fast startup of parallel jobs. Since the primary goal of parallel computing is to reduce the amount of time required to achieve a solution, it is critical for a large-scale parallel machine to start jobs as efficiently as possible. While most suppliers of large-scale parallel computing platforms emphasize delivering performance to an application once it is running, few address the time spent getting the application started. On many systems, this time can be significant.

This limitation also exists in the arena of commodity clusters. The typical method of using a shell script that loops over UNIX remote shell or secure shell commands to start processes on remote nodes in a cluster has severe inherent performance and scalability limitations. However, for many reasons, the problem is not as apparent.

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000

©2001 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a non-exclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SC2001 November 2001, Denver ©2001 ACM1-58113-293-X010011 \$5.00

Most clusters are not concerned with running a single parallel job across several hundred or a thousand nodes. Because the typical cluster has a small set of users running on only tens or a few hundreds of nodes, these scalability problems are not as evident.

Large-scale commodity clusters running a single application over several hundred or thousands of nodes are becoming prevalent. These machines are intended to be low-cost alternatives to traditional vendor-supplied supercomputers, and they are required to run hundreds of different applications from possibly hundreds of different users. The usage requirements for these machines and the amount of resources spent on them are significant enough to make fast job launching a key component to their success.

The Computational Plant (Cplant™)[4] project at Sandia National Laboratories includes several such large-scale machines. Our two largest production clusters, a 592-node cluster and a 1024-node cluster, will soon be combined into a single large machine with a theoretical peak compute performance of greater than 1.7 trillion floating-point operations per second (TFLOPS). The system software on these machines is modeled after the software that Sandia designed and developed for the Puma[12] operating system, which is the basis for the lightweight compute node kernel in use on the 9000 processor ASCI/Red Intel TeraFLOPS[11] machine.

In the next section, we present the components of the runtime system that allow large parallel jobs to start on a Cplant™ cluster in a matter of several seconds. Section 3 describes some support tools for the runtime system. We continue with Section 4, which outlines the choices that affected the design and interaction of these components and describes some of the considerations inherent in these choices. Performance results for launching various executables on several hundred nodes are presented in Section 5. We provide an overview of related work in Section 6 and conclude with a discussion of future work in Section 7. Section 8 provides a summary of the contributions of this paper.

2. Cplant™

The Computational Plant is a large-scale, massively parallel computing resource composed of commodity computing and networking components. The main goal of the project is to construct a commodity cluster capable of scaling to the order of ten thousand nodes to provide the compute cycles required by Sandia's critical applications. Because of this scalability requirement, Cplant™ has been designed to address scalability in every aspect of the hardware and software architectures.

The CplantTM machines employ the partition model of resource provision[8] that was initially developed by Intel on their early parallel platforms. This model divides the machine into several different partitions that provide specialized functionality. The main partitions are service, compute, and I/O.

The service partition provides a full-featured UNIX environment where users can log in and perform the usual UNIX commands, such as compiling codes, editing files, or sending email. The service partition is also where the users launch parallel applications into the compute partition, view status of running jobs, or debug compute node applications. The service partition is what many workstation clusters call the “front end.” The service partition is usually composed of several different machines, and for the current CplantTM clusters, we employ a load balancing name server to place new logins on the least loaded machine.

The largest portion of the machine is the compute partition, which is dedicated to delivering processor cycles and interprocessor communications for parallel applications. Nodes in the compute partition are space-shared such that a group of nodes is dedicated to running only a single application. Compute nodes typically provide only a small subset of UNIX functionality in order to maximize the resources given to parallel application processes. User logins directly to compute nodes are prohibited.

The CplantTM runtime system is also modeled after the runtime system of ASCI/Red. This runtime system is dependent upon an underlying high-performance system area network, not only for supporting application message passing via a user-level library, such as MPI[10], but also for supporting compute node allocation, application launch, parallel I/O, and debugging tools. CplantTM clusters use the Myrinet [3] gigabit network with a Sandia-designed message passing interface called Portals[5]. All communication between the runtime system components is implemented over Portals.

This runtime environment is composed of four functional components that work together to start and manage parallel jobs. The *process manager* controls the resources on an individual compute node. It is responsible for starting an application process and providing it with the basic information needed for the process to be part of a parallel job. The *allocator* chooses which compute nodes to assign to a specific job. The *launcher* is the component with which users invoke a parallel job. Finally, the *job scheduler* is responsible for applying a policy for queueing and running batch jobs on the system. The following describes these components in greater detail.

2.1 Process Manager

The process manager component of the runtime system is called the Process Control Thread¹, or PCT. A PCT runs on each compute node in the cluster and is responsible for managing the processor and memory resources on the node it controls. The PCT’s implement a space-shared system, where each compute node processor runs a single parallel application process.

The PCT provides the application process on a node with the user’s

¹The term “thread” is a misnomer for the current CplantTM environment. In the Puma operating system, the PCT was a user-level thread that was able to perform some privileged operations. The current Linux implementation of the PCT is a heavyweight daemon process.

environment as well as the environment needed to participate in a parallel application. It is responsible for starting the application process, redirecting UNIX signals to the application process, attaching a debugger to the application process, terminating the application process, and recovering resources after the application process terminates.

A compute node is added to the machine when the PCT on the node contacts the allocator to let it know that the compute node’s resources are available for hosting an application. During application launch, the allocator contacts each of the PCT’s in the parallel job to let it know that it has been allocated and that it will be contacted by the application launcher.

The PCT’s participating in the parallel job launch form a spanning tree that allows for efficient group communications, such as broadcasts and reductions. Efficient communication allows the PCT’s in a large job to quickly relay global information to compute node processes.

Once a PCT has been communicated all of the information necessary to start the parallel application process, it starts the process, and then waits for requests from either the job launcher or from the application process. The PCT tries to yield as much of the compute cycles as possible to the application process, while still trying to service requests in a timely fashion.

When a PCT is not hosting a parallel application process, it performs some minimal health checks on the compute node. In particular, it checks the available memory on the node and the size of the RAM disk. Should either of these fall below a threshold for hosting an application process, the PCT logs the problem, sends a message to the allocator to let it know it is unable to host an application, and exits.

2.2 Bebopd

The allocator component of the runtime system is the bebopd². Bebopd runs on a node in the service partition and is responsible for allocating compute nodes to parallel jobs as requested by the job launcher. Bebopd is also responsible for providing status information about the compute partition, such as the number of free compute nodes and which nodes have been allocated to jobs.

Each PCT contacts bebopd upon startup to make it aware of the available resources. The launcher also contacts bebopd to reserve nodes for the parallel job. Bebopd then contacts the PCT’s to insure that each node is ready to participate in the parallel job. Bebopd then passes the launcher a list of the available compute nodes on which the parallel job will run. Once the PCT’s have finished hosting the parallel job, they contact bebopd to update their availability status.

Currently the algorithm used to allocate compute nodes is naive. Each compute node is assigned a physical node identifier, which is an integer from 0 to $n - 1$. The allocator starts with the smallest available node id and searches the list of node id’s in ascending order to find free nodes. This allocation scheme does not take into account the network topology of the cluster. Ideally, the allocator would find nodes that are close together³ to reduce network con-

²Better Engineered Bag Of Pc’s Daemon

³The exact definition of close is a combination of the physical network topology and a combination of other factors, such as the ex-

tention. The allocator on ASCI/Red does this quite easily, since the routes and topology of the machine are fixed. The task is more challenging for Cplant™ machines, since each machine may have a different topology or even have varying topologies, such as in the case of a machine that is grown and pruned periodically. We have inserted hooks into our allocator to allow for more intelligent allocation schemes, and we are investigating different algorithms for optimal job placement.

Bebopd is also a single point of failure in the cluster. Should the bebopd die or become uncommunicative, the ability to launch jobs or get compute partition status is lost. Initially we imagined the need distributed allocator that would be responsible for subsets of the machine. These distributed bebopd's would communicate with each other to satisfy a request for nodes, and the loss of a single bebopd would result in the loss of only a subset of the nodes in the machine. We have not implemented this distributed allocator because failures of the current bebopd implementation have been infrequent, even during times of heavy usage. We have also implemented the bebopd so that it checkpoints its state and can be restarted after a failure with little loss of information about the current state of the machine.

2.3 Yod

The parallel job launcher component of the runtime system is yod⁴. Yod contacts bebopd to allocate a set of nodes, and then communicates with the primary PCT to move the user's environment and executable out to the compute nodes.

Once a job has started, yod serves as an I/O proxy for all UNIX standard I/O functions, including file I/O. Parallel applications on Cplant™ are linked with a library that redefines all of the standard I/O library routines. This library implements a remote procedure call interface to yod to perform I/O operations. If a compute node process calls `open()`, the library makes a remote request to yod via message passing. Yod then handles the file open operation locally and sends the result to the compute node process. This method of handling terminal and file I/O is sufficient for standard I/O functions and small input or output files. However, because there is only a single yod process servicing requests from all of the compute node processes in a parallel job, yod becomes a bottleneck when trying to handle many simultaneous I/O requests or large data transfers. A separate parallel I/O capability is provided for such operations.

Yod also disseminates some UNIX signals that it receives out to the processes running in the parallel job. When yod receives a signal, it sends a message to the primary PCT, which fans the message out to the other PCT's in the job and delivers the desired signal to the application process. This feature can be very useful for operations such as user-level checkpointing and killing jobs.

Yod has many command-line options, most of which have been the same throughout successive implementations on the nCUBE, Intel Paragon, the Intel TeraFLOPS, and now the Cplant™ clusters. Because yod is the main interface for application developers, it was important to keep the functionality of yod as similar to previous machines as possible so that Cplant™ would be familiar to our users

act routes from one node to another. While two nodes may be on the same switch, the route between the nodes may traverse other switches.

⁴The name yod is derived from the launcher on the nCUBE, which was *xnc*. Each subsequent letter of *xnc* results in yod.

and allow applications to be ported more easily. The following describes some of these options.

- size [n]** This option indicates the number of processes in the parallel job. If no *n* is specified, it is assumed to be 1.
- l [node list]** This option can be used to specify the exact nodes to use in a job. A comma-separated list of physical node id's (-l 1,2,3,4,10) or a range of node id's (-l 1..4,10).
- attach** This option displays a list of allocated nodes and asks for user confirmation before letting the application processes proceed to their main function. This option can be used to attach a debugger to the individual processes during the load.
- batch, -interactive** These options indicate whether or not the application has been run interactively or under the control of the batch scheduler. Certain runtime semantics change when running in batch mode. For example, in batch mode, if one of the processes in the job dies, the entire job is terminated.
- file [name]** Redirect yod status messages to the specified file.
- D** This option turns on load debugging information. Yod will display the sequence of load information.
- d [stage]** This option turns on specific debugging information in yod. For example, -d io displays the details of application I/O requests to yod.

The first argument that yod does not recognize is assumed to be either the executable to be launched or a loadfile containing a list of several executables to be launched. A loadfile can contain different executables each with its own size and command line arguments. For example, a loadfile containing the following

```
# comment
-sz 10 exec1 arg1 arg2
-sz 50 exec2 arg3 arg4
-sz 100 exec3 arg5 arg6
```

would launch a single parallel job composed of the three different executables, each with its own command-line arguments. Yod can support launching up to five different executables in a single parallel job. It is also possible to list the same executable with different command line arguments in a loadfile.

2.4 Batch Scheduler

We have made several enhancements[1] to the open version of the Portable Batch System (PBS)[2] that allow it to reliably and scalably schedule jobs on thousands of nodes. We also made several changes to our runtime components in order to integrate PBS scheduling.

In contrast to a typical cluster environment where a PBS component runs on every compute node, we restrict PBS components to run only on nodes in the service partition. This restriction provides several benefits: it reduces the likelihood of running into scalability limitations as the number of compute nodes increases, keeps the compute node cycles under our control, and removes the requirement of supporting UNIX sockets on the compute nodes.

The compute nodes are an abstract resource managed by PBS with help from the runtime system. We have added a *size* resource that is analogous to the “-size” command-line argument to yod. This value represents the number of compute nodes requested by a batch job. The bebopd regularly updates the PBS server with the number of compute nodes available for PBS jobs. So if a compute node goes out of service, the PBS size resource will be updated accordingly. The PBS MOM component puts the size in the batch job’s environment so that yod can pass it along to the compute node allocator. The *nodes* resource in PBS represents the service nodes on which the job scripts are executed.

A problem on many clusters choosing to schedule jobs with PBS has been the orderly termination of parallel applications started by PBS. This problem is exacerbated when the PBS MOM is managing the job script and not the parallel applications spawned by it. The PBS MOM on Cplant™ was enhanced to kill parallel applications when the time allocated to the job script has expired. It also kills parallel applications inadvertently left running by PBS jobs that have terminated.

We have made changes to PBS `qstat` command so that the size and walltime requests of a job are displayed as well as the total compute nodes in use by different jobs. We also display the amount of time a job has been queued.

In order to increase the reliability of the PBS components, we have added the ability to use non-blocking sockets for communication so that the PBS scheduler component will continue to run properly in spite of losing touch with a PBS MOM running on a service node. The patch implementing non-blocking sockets is available for download at [1] and has been incorporated into PBS at other sites. This addition has greatly increased the robustness of batch scheduling on the Cplant™ machines.

3. SUPPORT TOOLS

In addition to the runtime components that are involved in launching an application and servicing running applications, a tool is needed to look at what jobs are running on the machine and discover how many compute nodes are available. Most of the available tools for this are GUI-based. This creates problems when trying to scale up to several thousand nodes.

3.1 Compute Node Status and Job Management

The *pingd* utility is used to discover the status of the compute partition. With no arguments, it prints a single line of output for every compute node in the cluster. Each line contains the physical node number, its location in the machine (rack and node number), the job id, the process id and rank, the owner of the job, the elapsed time, and the PBS job number (if started by PBS). It displays the total number of nodes in the cluster, the total number that are busy, the total number that are free, and how many are not responding. It also lists the number of nodes under control of PBS and the number that are available interactively.

Because *pingd* displays a single line of output for every compute node, *pingd* does not scale past a few tens of nodes. We have a utility, called *showmesh*, which is a Perl script that condenses the output of *pingd* so that it fits more information into fewer lines of a terminal. Showmesh represents each compute node with an ASCII character, using different characters to indicate if a node

is being used, is free, or is unavailable. The Cplant™ version of showmesh is nearly identical to a tool by the same name available on ASCII/Red.

In addition to compute node status, *pingd* can also be used to kill parallel jobs. It is possible to start a yod process in the background and log out of a service node. However, when logging back into the cluster, the user is unlikely to be placed on the same service node where that background yod process is running. Users can use *pingd* to reset the compute nodes where a particular job is running, which essentially kills the parallel application processes and the associated yod process and makes those compute nodes available to host another job. *pingd* can also be used by privileged administrators to kill the PCT on a particular compute node, which logically removes it from the cluster.

3.2 Debugging

The Cplant™ runtime environment supports the TotalView source-level parallel debugger from Etnus, Inc. We have implemented a startup utility that can be used in conjunction with TotalView’s bulk server launch capability to launch the remote TotalView debug servers using the PCT’s spanning tree. Each debug server then connects back up to TotalView using TCP/IP. We are in the process of porting TotalView’s low-level channel-based communication layer to Portals to get rid of the dependency on TCP/IP.

For initial debugging support, we provided a crude level of debugging using tools based on the GNU debugger, *gdb*. The PCT supports launching the application process under control of *gdb*. Users can then debug individual compute node processes on a service node using a tool called *cgdb*. *cgdb* communicates debugger input to the PCT, which relays the input to the *gdb* process, and relays the output back to *cgdb*. Since there is no filesystem available on the compute node to the *gdb* process, no source-level debugging is available with this mechanism. Users can also gather backtrace information from *gdb* to help determine where a faulting application process was halted.

4. DESIGN CHOICES

The Cplant™ runtime system differs from most traditional runtime systems in a number of ways. The following discusses some of these differences and related issues.

Our runtime system is designed to efficiently move the executable(s) from the service partition into the compute partition. Many runtime environments use a globally mounted filesystem, such as NFS, to move executable(s). This many-to-one design, where thousands of clients are accessing a single file on a single server at the exact same time, is inherently non-scalable. NFS was not designed to handle this situation and will usually fail in some way or become unbearably slow under these circumstances.

The use of NFS across the compute partition also implies that the Internet Protocol (IP) stack is available. While our current compute node operating system (Linux) supports the IP stack, we assume that Portals is the only communication protocol supported on compute nodes. In order to get maximum performance out of the high-performance communication fabric, we only implement a single wire protocol. Portals has been designed to provide flexible building blocks for higher level protocols, such as those needed to perform file operations.

Following the model of ASCII/Red, Cplant™ compute nodes are

diskless and do not support virtual memory paging. Each compute node has a 16 MB RAM disk for storing the executable from which an application process is created, and all executables that run in the compute partition are statically linked.

In contrast to most other Linux-based clusters, we wanted to be able to explore the use of a lightweight kernel in the compute partition. Our initial plans were to research how some important characteristics of our previous lightweight kernels might be implemented in Linux. We also wanted to leave open the possibility of porting our lightweight kernel to Cplant™ compute nodes as well. This desire to explore a “lightweight” Linux or other lightweight kernel contributed to many of the design decisions for Cplant™. For example, we assume that the compute node operating system does not support a local filesystem. All file operations are fulfilled via message passing to the service or I/O partition where filesystem requests can be satisfied.

Our runtime system must be flexible enough to support a wide variety of programming models and high-level communication layers. Many of the runtime systems available for clusters are tailored to specific programming models or message passing systems. For example, the *mpirun* scripts and MPD[6] environment provided with MPICH[9] are specific to MPI. While the majority of our applications are written in MPI, we support other programming models as well. For example, we support more fault tolerant programming models where the loss of a process does not terminate the entire application.

We decided early on that the runtime system was key in detecting and reporting node failures in a large cluster of 1000 or more nodes. Although we have an on-going effort to implement scalable non-intrusive monitoring tools, failures often first become apparent to the runtime system via application load failures. The group formation and communication functions used by PCT’s when loading a parallel application are designed to timeout when another PCT is not responsive or malfunctioning. The problem nodes are detected by the runtime system and reported to yod, which displays this information to the user. The user can then notify the system administrators and attempt to load their application again. In fact, yod will automatically try again so that application load from a batch script will succeed. The surviving PCT’s in a failed job launch are returned to the pool of free nodes. In some cases the malfunctioning compute node is automatically removed from the pool of available nodes by the bebopd. This fault detection/recovery all occurs in about 20 seconds, which is the time required for the PCT’s to timeout while engaging in a collective message passing operation.

5. PERFORMANCE

One of the features available in yod is the ability to display timing information for various parts of the application load protocol. Yod can be invoked so that it calculates and prints times for various phases as it starts a parallel application. Included in this output is the total time required to start the job. The following phases of the load protocol are timed:

Allocate nodes: The time for yod to request and be given a list of compute nodes on which to run.

Initial message: Yod contacts all of the PCT’s in the job with an initial message to let them know which nodes are involved in the job.

Form group: Based on the initial load message from yod, the PCT’s form a spanning tree for broadcasting load data. Yod is contacted when all of the PCT’s have built this tree.

Pull arguments: The root PCT contacts yod to get the command line arguments for the application and then broadcasts this data to the rest of the PCT’s.

Pull environment: All of the environment variables associated with the shell in which yod runs are given to the root PCT which then broadcasts it to the other PCT’s.

Read file: Time needed for yod to read the executable image from the local filesystem on the service node.

Fanout: Time needed for the root PCT to read the executable out of yod’s memory and broadcast it to the other PCT’s.

Pull map: Yod sends a message to the root PCT telling it to pull the portal process ID map of the application and broadcast it. This is the time it takes for the root PCT to fetch the map from yod.

Log time: The time required to compile and communicate log entry data to the bebopd so that it can write an entry in a log file for the job.

Total time The total time the load has taken, from start of the yod command until all the application processes are ready to call `main()`.

5.1 Test Environment

In order to measure the performance of the runtime environment, we built a minimal executable that simply returns 0 in the `main` function. When compiled and statically linked, this executable is about 800 KB in size. We padded this file with zeros to build executables ranging from 2 MB to 14 MB in increments of 2 MB. Each executable was launched several times on varying numbers of nodes from one up to 1010 nodes. The timings for each executable on each node count were collected and averaged.

The Cplant™ machine used for testing is a 1024-node cluster name Ross/Antarctica. Each compute node is a Compaq DS10L, which is composed of a 466 MHz Alpha 21264 (EV6) with 256 MB of main memory. Each compute node has a 64-bit 33 MHz Myrinet LANai-7 network interface card. The nodes in this cluster are connected using 64-port Myrinet Mesh64 switches connected in a three-dimensional mesh topology that is torus in the X and Y dimensions but not in the Z direction. The Portals message passing layer achieves just over 100 MB/s peak asymptotic point-to-point bandwidth on this machine. At the time our data was gathered, only 1010 compute nodes were operational in the machine.

5.2 Results

Figure 1 shows the job launch times for 4 MB, 8 MB, and 12 MB executables out to 1010 nodes. As one would expect, the job launch times increase as the number of nodes increases, and the larger executables generally take more time. There are a few cases on smaller numbers of nodes where the size of the executable is not the determining factor in the launch time, but beyond 64 nodes, this appears to be the case. More importantly, the graph shows that a 1010-node job can be launched in less than 17 seconds for a 8 MB executable on a dedicated system.

In order to determine the effect of system load on the launch performance, we repeated these measurements during the day, when the

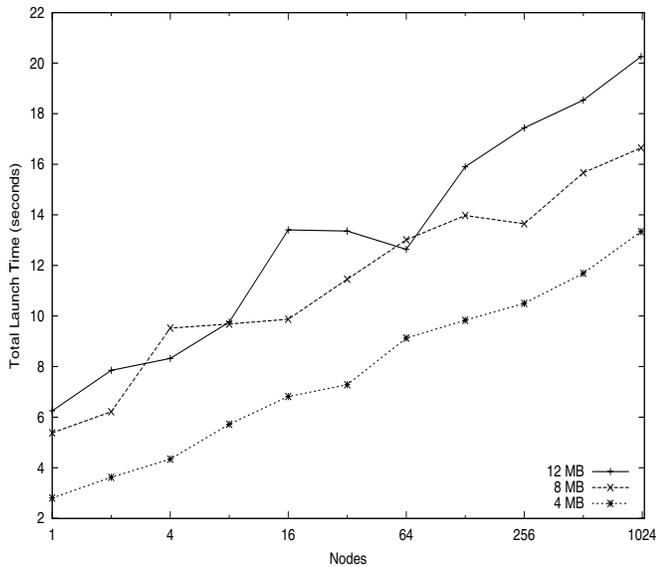


Figure 1: Parallel job launch times for the 4 MB, 8 MB and 12 MB executables on a dedicated machine.

machine is most heavily utilized. The scheduling policy on the machine during this time limits the number of interactive nodes to only 100. Figure 2 shows the launch performance out to 64 nodes. The impact of a loaded system appears to be minimal on job launching for such small numbers of nodes. Compared to the performance on a dedicated machine, the results for a loaded system are sometimes better than on a dedicated machine and are no worse than two seconds.

Since a majority of jobs run on the system are run under control of the batch scheduler, we also measured launch performance on a loaded system using PBS. Since the scheduling policy for the machine dictates that jobs using half of the available nodes are run during dedicated time, we limited these jobs to 256 nodes. Figure 3 shows the launch performance of jobs started under PBS. These numbers are very erratic, especially for jobs run on less than eight nodes. As with interactive jobs, launch times for batch jobs run during times of high usage are not significantly worse than launch times on a dedicated machine.

Table 1 shows the breakdown of the launch times for the 4 MB and 12 MB executables. The phases of the load protocol that increase significantly in time as the number of nodes increases are highlighted. These phases are: the time for the PCT's to form a group; the time to fan out the executable to the compute nodes; and the time to log the start of the job with the allocator. The other phases remain fairly constant as the number of nodes increases.

In forming the group for broadcasting data, each PCT sends a status message to yod indicating that the PCT is ready and willing to participate in job launch. This many-to-one implementation is inherently non-scalable, but was chosen because of its simplicity. It is easy for yod to recognize a non-responsive PCT or a PCT that is unable to participate in the job. A more complicated scheme may offer higher performance, but may sacrifice reliability and recognition of load problems.

The increased time to fan out the executable is expected. The per-

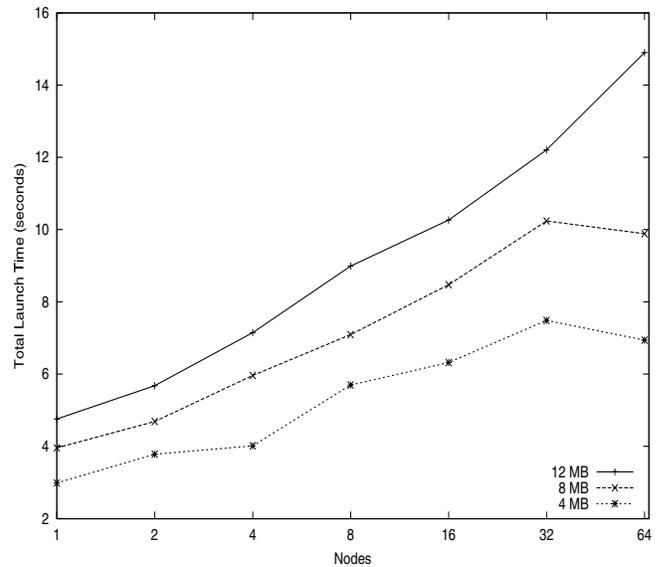


Figure 2: Parallel job launch times for the 4 MB, 8 MB and 12 MB executables on a loaded system.

formance of the broadcast is directly related to the performance of the network, and we would expect that future Cplant™ machines with higher performance networking hardware would perform better on this stage of application launch.

Logging the start of the job with the allocator happens in two stages. First, yod compiles relevant information about the job, including the name of the user who launched the job, the start time, the command line, the number of processors, and a list of the nodes used. This information is then sent to the allocator, which in turn writes it to a log file. Further instrumentation is needed to accurately determine which operation takes more time as the number of nodes increases, but it is likely that the message to bebopd is not the culprit. Yod contains code that condenses the node list to find contiguous ranges of nodes in order to minimize the string representing the node list. It is likely that this compression consumes more time as the number of nodes is increased. More investigation is needed to measure the performance of this step of the load protocol to determine alternate methods of logging the start of a job.

6. RELATED WORK

There are many different runtime systems for parallel computing environments. While many of these systems encompass the different components described above, few are described in the detail necessary to analyze the scalability and performance limitations of launching a parallel application. Even fewer provide performance data for direct comparison. Two such projects that do provide some of this data are discussed below.

The Berkeley NOW project implemented a software layer called GLUNIX (Global Layer UNIX)[7] on top of the Solaris operating system that implemented parallel application launch as well as several other cluster management and runtime features. GLUNIX was able to start a 100-node parallel job in 1.3 seconds. In [7], a detailed breakdown of the time needed for various tasks in job startup is given as well as a graph of startup times from 1 to 100 nodes. The size of the executable used to gather this data is not

Phase	64 nodes		128 nodes		256 nodes		512 nodes		1010 nodes	
	4 MB	12 MB								
Allocate nodes	1.80640	1.45061	1.73317	2.16547	1.52729	1.84222	1.37249	1.52991	2.42833	1.95759
Initial message	0.20285	0.18077	0.20817	0.22959	0.22507	0.24333	0.26238	0.25770	0.48100	0.44273
Form group	0.08424	0.25412	0.09686	0.33935	0.10729	0.29819	0.10255	0.32139	0.14809	0.35136
Pull arguments	0.00085	0.00069	0.00089	0.00093	0.00075	0.00079	0.00077	0.00080	0.04060	0.05088
Pull environment	0.00724	0.00872	0.01399	0.01726	0.02234	0.02239	0.03708	0.05960	0.08491	0.03483
Read file	0.02431	0.07423	0.03432	0.39283	0.03317	0.07491	0.03292	0.07483	0.07035	0.02570
Fanout	2.33431	5.78789	2.62261	7.72959	2.98661	8.40935	3.29289	9.33005	3.53032	9.61980
Pull map	0.12685	0.06555	0.06449	0.03264	0.10784	0.02024	0.07240	0.03997	0.08824	0.04030
Log time	4.54698	4.18872	5.06261	5.00124	5.58053	6.53498	6.51932	6.92485	6.47246	7.73651

Table 1: Breakdown of parallel job launch times (in seconds) for the 4 MB and 12 MB executables on a dedicated machine.

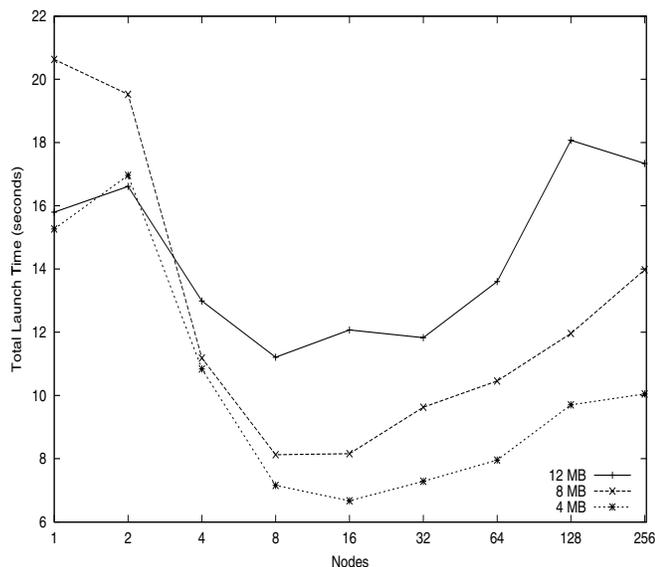


Figure 3: Parallel job launch times for the 4 MB, 8 MB and 12 MB executables on a loaded system.

given. GLUNIX was limited in scalability by the maximum number of file descriptors allocated to a single process, and could only support parallel programs with 341 or fewer processes.

The Multi-Purpose Daemon (MPD)[6] environment for managing parallel programs was designed and implemented recently at Argonne National Laboratory to solve the problem of unreasonably long MPI job startup times on their large Linux cluster. MPD uses two levels of TCP-connected daemons to start parallel jobs. Each level of daemons is connected in a ring and control messages are passed from daemon to daemon around this ring. The choice of a ring topology is justified by experiments that have shown the design to be feasible for a thousand daemons. For example, experiments have shown that a message (of unspecified size) could make 1024 hops around the ring in less than 0.4 seconds. Performance tests on their large Linux cluster using an executable of unspecified size demonstrated that a 211-node job started and ran to completion in about 2 seconds for a single process per node and in about 3.5 seconds for two processes per node. These results do not include the time required to distribute the executable to the compute nodes.

We had hoped to perform a direct comparison between the Cplant™ application launch and that of a typical cluster environment. Unfortunately, several factors hampered a direct comparison. For example, the default MPICH environment assumes that the executable to be launched is available on a global filesystem such as NFS. It

is non-trivial to measure the amount of time that it takes for the executable to “move” out to each compute node. Likewise, the mpirun utility of MPICH does not perform dynamic allocation of nodes. Rather, it takes as input a file containing a list of hosts on which to start jobs. In this case, the static allocation model leads to greater performance, but also leads to a less robust system. Comparing individual parts of the load protocol is difficult, since few runtime systems are implemented similarly. Likewise, it is difficult to compare total launch times given the disparate functionality that different runtime systems provide.

7. FUTURE WORK

There are currently many ongoing and planned projects to enhance the capability of the Cplant™ runtime system and add new functionality. We are working on enhancing the ability to the allocator to place jobs on the machine in a way that reduces network contention. We are also working on support for the MPI-2 dynamic process creation functions, which will allow a compute node application to spawn and communicate with another compute node application. We are currently working on extending the runtime environment to support running multiple application processes on a compute node to better utilize multiprocessor nodes. Our current environment is implemented for single processor nodes and only supports running a single application process per node. We are also planning to support on a library interface to the runtime system so that third party applications can be written that launch jobs and interact with the runtime system components. We hope to be able to measure the impact that these projects have on the performance and scalability of the Cplant™ runtime system.

8. SUMMARY

The ability to launch large-scale parallel jobs quickly is critical to the usability of any massively parallel computing platform. As part of the Cplant™ project, we have designed and implemented a flexible runtime system that allows for starting a parallel job on more than a thousand nodes in a matter of several seconds. Our runtime environment also provides many desirable features beyond that of traditional job starting software provided in typical Linux clustering environments. The performance analysis that we have conducted will help us to judge the impact of future enhancements to the runtime system, and has helped to validate the ability of our design and implementation to scale to several thousand nodes.

9. ACKNOWLEDGMENTS

The Computational Plant project at Sandia involves many people from many different organizations, all of whom have contributed to the overall success of the project. The authors would like to acknowledge the contributions of the other members of the Scalable Computing Systems department. We would also like to thank the members of the Scientific Computing Department for their assistance and patience with scheduling dedicated time on Ross. A

special thanks is also extended to Laura Grit for her hard work in collecting and compiling the performance data.

10. REFERENCES

- [1] <http://www.cs.sandia.gov/doc/pbs/pbs.html>.
- [2] *Portable Batch System*. <http://www.openpbs.org>.
- [3] N. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet-a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995.
- [4] R. B. Brightwell, L. A. Fisk, D. S. Greenberg, T. B. Hudson, M. J. Levenhagen, A. B. Maccabe, and R. E. Riesen. Massively Parallel Computing Using Commodity Components. *Parallel Computing*, 26(2-3):243–266, February 2000.
- [5] R. B. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Riesen. The Portals 3.0 Message Passing Interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.
- [6] R. Butler, W. Gropp, and E. Lusk. A Scalable Process-Management Environment for Parallel Programs. In J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Proceedings of the 7th European PVM/MPI User's Group Meeting*, pages 168–175, Balatonfured, Hungary, September 2000.
- [7] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. *Software: Practice and Experience*, 28(9):929–961, July 1998.
- [8] D. S. Greenberg, R. B. Brightwell, L. A. Fisk, A. B. Maccabe, and R. E. Riesen. A System Software Architecture for High-End Computing. In *Proceedings of SC'97*, 1997.
- [9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
- [10] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.
- [11] Sandia National Laboratories. *ASCI Red*, 1996. http://www.sandia.gov/ASCI/TFLOP/Home_Page.html.
- [12] L. Shuler, C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.