Managing the APAC NF Altix cluster

Kevin Pulo
APAC National Facility,
ANU Supercomputer Facility,
Australian National University,
ACT, Australia, 0200.
Email: kevin.pulo@anu.edu.au

Abstract

The APAC National Facility has set system management goals of providing an environmant that allows consistent, high performance for all jobs while maintaining very high utilisation. The Facility's newly installed SGI Altix cluster presents a number of challenges in terms of achieving these goals. At a minimum the topology of a cluster of large NUMA SMP nodes must be respected in scheduling and job placement decisions. Even more challenging has been the requirement to overcome deficiencies and limitations in the proprietary, closed source MPI job launch used on Altix clusters.

In this paper we present the techniques and policies implemented by APAC NF on this system to ensure consistently good performance under this diverse and competitive workload. This includes issues encountered in the PBS-based batch queueing system, the SGI MPT-based MPI system and general system configuration and admistration.

1 Introduction

In April 2004, the Australian Federal Government announced funding of $29m for the next stage of APAC. SGI won the tender to replace the 5-year-old APAC National Facility (NF) "SC" system with an Altix 3700 Bx2 cluster. This new system has a *Non-Uniform Memory Access* (*NUMA*) architecture, which is significantly different to the previously *Uniform Memory Access* systems at APAC NF. As such, APAC NF has invested substantial development effort into ensuring that this system is of sufficiently high quality.

1.1 System details

The system is an Altix 3700 Bx2 with a total of 1680 64-bit Intel Itanium 2 1.6 GHz processors, 3.6 Tb of memory, 120 Tb of storage and Brocade 24000 fibre channel switches. The SGI NUMAlink 4 interconnect is used for high-speed, low-latency transfer within the system, with an additional Gigabit Ethernet "management" network. The individual hosts (or "partitions", in SGI terminology) consist of 16 NUMA nodes each, where a NUMA node consists of 2 CPUs, a 4 or 8 Gb memory node, and a shared memory bus ("SHUB"). Thus, each host has 32 CPUs and 16 memory nodes (64 or 128 Gb). NUMAlink is used as the interconnect between NUMA nodes both within each partition and between them. There are 52 compute hosts (`ac1`–`ac52`), along with a 4 NUMA node interactive login front-end host (`ac`) and a 4 NUMA node test host. The overall system is referred to as "AC" ("Altix Cluster"). It ranked 26th on the recently announced 25th Top500 list [1] with LINPACK performance of 8974 GFlops.

30Tb of global storage is available to all partitions using SGI's CXFS clustered filesystem, and a further 70Tb is divided locally amongst the partitions using dual fibre-channel.

The operating system is SUSE Linux Enterprise Server (SLES) 9, with SGI ProPack 4.

1.2 Motivation

The APAC NF operates with a large user-base of over 700 users in over 200 distinct projects, and as such, experiences a widely diverse workload. Compute cycles on the systems are offered under a competitive grant system, and on previous systems it is possible for job queues to be very long (for example, several thousand CPU hours on a 500 CPU machine). Thus, it is important for users to obtain repeatable and consistent benchmark results, in order to effectively plan the use of their grants. APAC NF uses the walltime of jobs, rather than the CPU time, as this is the only effective way to measure the "overall" performance of the job (particularly in terms of turnaround time for users). The NUMA architecture of the SGI Altix system means that without appropriate controls, performance benchmarks can easily vary by 20% between identical runs due to poor memory placement and "interference" between jobs, and this figure can increase to as much as 50% for memory bandwidth limited applications. Clearly, users would be unable to make efficient use of their grants when the length of their jobs cannot be predicted with reasonable accuracy. For this reason, consistent and reproducable performance is of extremely high priority (perhaps even more so than good performance).

The main difference with a NUMA architecture (compared to SMP) is that memory is divided into

memory nodes. Each memory node is associated with a NUMA node, and thus, with the 2 CPUs in that node (on our system). Memory accesses by a CPU may be *local* (to addresses in the same NUMA node) or *remote* (to addresses in other NUMA nodes). By using *directory based* cache coherency [2], local memory accesses require only the local SHUB, whereas remote accesses involve other SHUBs and the interconnect. As such, remote memory accesses incur higher latency than local ones. This means that it is desirable for the memory being used by a thread running on a CPU to be local to that CPU.

Most NUMA systems allocate pages of memory according to the *first touch* placement strategy, whereby memory allocated by a process executing on a particular CPU is placed on that CPU's local memory node. If the process is later migrated to another CPU, then accesses to the previously allocated memory become remote accesses (with the associated penalty). Most modern NUMA-aware Unix operating systems (including Linux) support the dynamic migration of processes between CPUs (since this is effectively identical to the SMP case), but do not support the dynamic migration of pages between memory nodes. The two most common other memory placement strategies are *round-robin* and *random*, both of which effectively reduce the NUMA system to the UMA case. However, the hardware cache coherency used in SGI Altix systems is such that these page placement strategies are not feasible.

In the absence of any controlling mechanisms, the memory layout of an application running on a fully-loaded host in the cluster depends entirely on the particular CPU and memory load of the host during the lifetime of the application. Clearly, in this situation one expects that only very rarely will applications have significant fractions of their pages in local memory, and have no other applications remotely using their local memory node (which, of course, reduces the SHUB bandwidth available to the local application). This is the source of the unpredictable (and therefore unrepeatable) performance on unmanaged NUMA systems.

1.3 cpusets

One way to deal with this problem is to set the kernel-level CPU and memory node *affinity* of a process. SGI provide a utility known as `dplace` for achieving this. However, the problem with affinity is that it is a recommendation for the kernel, and the kernel may violate it when it deems necessary.

A better solution is to use what are known as *cpusets*. A cpuset is a subset of the CPUs and memory nodes in a NUMA system. The list of CPUs and memory nodes is expressed as a comma-separated list of ranges, for example, `0-3,11,13,15`. cpusets are "hard", in the sense that a process in a cpuset cannot use CPUs or memory nodes that are not listed in the cpuset. A cpuset may have *sub-cpusets*, thus, the system has a hierarchical tree of cpusets. A sub-cpuset may only use CPUs and memory nodes that are listed by its parent cpuset. This allows for an easier, more natural partitioning of the system. A filesystem metaphor is used for cpusets, with the root cpuset named `/` and containing all the CPUs and memory nodes in the sytem. In fact, cpusets are implemented in Linux using a *virtual filesystem* (*vfs*), usually mounted at `/dev/cpuset`. In this filesystem, a directory corresponds to a cpuset, and each file corresponds to a property of the cpuset that may be read or set. This also has the significant advantage of utilising the standard filesystem permissions model to control access to cpusets and their properties.

The main properties a cpuset has are:

- CPU list and memory node list
  - Controls which CPUs and memory nodes may be used by processes in this cpuset.
  - The CPU and memory node lists of child cpusets must be subsets of their parent's.
- CPU and memory node exclusivity flag
  - If set, the CPUs and memory nodes in the cpuset may not be shared with any other cpuset (except ancestor cpusets).
  - Can only be set if the sibling cpusets are also CPU or memory node exclusive.
- Task (process) list
  - Lists the process IDs of processes in the cpuset.
  - Every process must be in exactly one cpuset.
  - Child processes inherit the cpuset of their parent.
  - The system `init` process may be in the `/` cpuset, or may be placed into a `/boot` cpuset, so that system daemons (and the like) may be separated from other processes.
- Destroy on release
  - Controls whether or not the cpuset will be removed when the last process in it exits.

Since cpusets are implemented using a virtual filesystem, they are very easily manipulated, either programmatically with normal file IO operations, or manually using the shell. For example, a typical shell session may be

```
# cd /dev/cpuset
# cat cpus
0-7
# cat mems
0-3
# mkdir half
# cd half
# /bin/echo 0-3 > cpus
# /bin/echo 0-1 > mems
# cat cpus
0-3
```

```
# cat mems
0-1
# /bin/echo 24652 > tasks
# /bin/echo 24653 > tasks
# cat tasks
24652
24653
#
```

This creates a new cpuset called `half`, which consists of the first 4 CPUs and 2 memory nodes of an 8 CPU, 4 memory node system, and then moves the processes with pids 24652 and 24653 into this cpuset. (The command `/bin/echo` is used because the `bash` "`echo`" builtin does not report errors correctly.) The cpuset a particular process belongs to can be found by querying the `cpuset` file within the process's `/proc` entry, i.e.

```
# cat /proc/24652/cpuset
/half
#
```

Thus, cpusets are an ideal way to separate batch jobs from one another when they are running on the same host. The batch queueing system is modified to create a cpuset for the job at startup, with the correct CPUs and memory nodes, and places the initial job process into this cpuset. These cpusets are not CPU or memory node exclusive, which allows a job to share CPUs and memory nodes with suspended jobs. Limited permissions are given to the user of the job, namely, they own the cpuset directory (allowing them to create sub-cpusets) and the `tasks` file (allowing them to move processes between sub-cpusets and the main job cpuset).

2 Batch-queueing issues

Over a number of years, APAC NF has developed a unique batch scheduling system to support the diverse workload presented by the userbase. It is based on OpenPBS [3] with numerous enhancements to support the following features:

- Tight control by the PBS system of all user processes running on the compute nodes.
- Detailed resource requests, allocations and limiting.
- A model of *suspending* some jobs to run others. Generally multiple jobs each using a small number of CPUs are suspended to run a job requesting a larger number of CPUs, but preemption may happen in other circumstances as well (e.g. high priority queues).
- An "equal access" scheduler to ensure that no usage group is disadvantaged, e.g. jobs cannot be suspended indefinitely.
- Maintain high system utilisation while still allowing large parallel jobs to run easily (mixing "capability usage" with "capacity usage").
- Minimise inter-job interactions as much as is possible.
- Provide as repeatable performance as possible, and as high performance as possible, in the face of full system utilisation.

Until now, this development occurred on more traditional clusters with smaller SMP nodes. The following modifcations have been prompted by features of the SGI Altix system (specifically, large CPU count SMP nodes (up to 512 CPUs), NUMA nodes with two CPUs, and fairly complex network hierarchy and locality at multiple scales, both *within* and between SMP nodes).

- PBS can be dynamically configured to hold specified CPUs free for system activity and daemons. A common cause of performance degradation of large tightly coupled parallel jobs is the unsynchronised perturbations to CPU access from system load. AC is currently configured with the first CPU on each partition as a "system" CPU. If a user requires all CPUs of a partition (for a large threaded application), that CPU can be dynamically configured in.
- The scheduler collects memory resource information down to the NUMA node level, both for what is physically available and for what each job is using on each NUMA node. This ensures that, even in the presence of cpusets, memory is not oversubscribed leading to excessive paging. PBS uses a job attribute called the `exechost` to specify where the job is executed, and this has been generalised to incorporate NUMA node information.
- As mentioned in Section 1.3, all jobs are placed in cpusets consisting of the CPUs and memory nodes assigned to the job by the scheduler. Jobs with small memory requirements (less per CPU than physically available) are given memory nodes corresponding to the NUMA nodes of the CPUs allocated to the job. Larger memory jobs are given a broader memory footprint but only such that they share memory nodes with jobs of a small number of CPUs (actually, jobs within the same subnode, as described below). The non-exclusivity of memory nodes is necessary even for running jobs in this case. Large memory jobs do not share memory nodes with large parallel jobs (currently, a "large parallel job" is one of more than 8 CPUs). To avoid another perturbation to parallel scalability, large parallel jobs never share memory nodes.
- Where possible, jobs are NUMA node aligned so as to minimise job interaction, i.e. jobs of an even number of CPUs are allocated whole NUMA nodes.
- Like many networks, the NUMAlink network is based on a quad-tree. Given that the base NUMA node contains two CPUs, there is a hierachy of building blocks of 8, 32, 128 and 512 CPUs. By

employing a metric that reflects this topological hierachy, the scheduler is able to localise job placement. The APAC NF scheduler employs a scoring system with cutoffs to decide which jobs can suspend which others under what circumstances. On the Altix system, these scores are modified to include a contribution describing the *spread* of a job in this metric. The score cutoffs are set such that jobs are reasonably localised in the network hierachy.

- To avoid fragmentation and to respect the interconnect topology (even within a host), the scheduler actually allocates CPUs to large parallel jobs in units of the fundamental 8-CPU (4 NUMA node) building blocks referred to here as *subnodes*. In many respects, subnodes take the place of hosts in scheduling:
  - They can be given scheduling properties to limit what sort of jobs they can run (e.g. `singlecpu`, `onlyparallel`, ...).
  - Queues can be targetted to only use certain subnodes.
  - Subnodes can be individually drained and taken offline.
  - Memory resources for jobs with overlapping memory nodes are constrained to within a subnode.

  Large parallel shared memory jobs (e.g. OpenMP jobs) requiring over 8 CPUs are also allocated subnodes, but are, of course, restricted to a single host.

One of the advantages large SMP offers in the presence of job of suspension is job migration. Suspended jobs can, theoretically, be resumed on different CPUs when those CPUs become free. This can decrease the fraction of jobs that are suspended and provide better utilisation of the system. As pointed out in Section 1.2, although the threads of a job will migrate, Linux does not support migrating pages, so any migrated job will run suboptimally and interact badly with jobs running on its original CPUs. SGI have developed a manual page migration system call and user level utility for Linux that can be invoked by resource management systems to overcome this. One constraint on these is that the process being migrated must be suspended at the time — but that is exactly the scheduling scenario for job migration.

A number of issues arise from the underlying *XPMEM* mechanism that supports SGI's MPI (see next section). XPMEM pages are process pages pinned in memory to be used as message passing buffers. For applications which communicate a large fraction of their memory (e.g. FFT based applications), this can mean that much of the job's memory cannot be paged. Job suspension in APAC NF's PBS assumes that processes of suspended jobs can be paged out to free memory for the new job. SGI have provided a manual `/proc` based mechanism for unpinning these pages during job suspension. Note that XPMEM page unpinning is also a prerequisite for job migration.

The other issue arising from XPMEM is the difficulty of quantifying the memory use of a job. Previously, APAC NF's PBS has used the virtual memory summed over the jobs processes, taking into account that some virtual memory maps are shared between processes. However, for an N CPU XPMEM based job, there are around N^2 large (usually 2 GB) and generally very sparsely populated virtual memory maps, leading to a meaningless virtual memory sum. We have developed a heuristic for evaluating job virtual memory that ignores these maps. We have also implemented an accurate physical memory use counter for jobs. This counter ensures that all pages attributed to the job are only counted once, and also includes swapped out pages in the count. Since it has to walk the page tables of all job processes, it can be rather expensive and its usage is somewhat limited as yet.

3 MPI issues

The *Message Passing Interface*, more commonly known as *MPI* [4], is the defacto standard for distributed memory applications. It is very widely used, and all the distributed memory applications in use on the NF's facilities make use of it, including both commercial scientific applications and "home-grown" code written by users. Two prevalent freely available MPI implementations are *MPICH* [5] and *LAM* [6, 7], both of which are in use on the NF's *Linux Cluster* (*LC*).

MPI systems typically require the use of a specialised *launcher* to start MPI-enabled programs. This launcher initialises the computing environment, including the network, starting the necessary instances of the MPI program on the appropriate hosts in the cluster, and so on. MPI launchers are most frequently named `mpirun`, or sometimes `prun`.

3.1 MPT launcher operation

SGI's implementation of MPI which accompanies its Altix systems is known as the *Message Passing Toolkit* (*MPT*). MPT is a proprietary, closed-source MPI system that provides support for the use of high-speed interconnects with MPI, such as Infiniband and SGI's NUMAlink. We refer to the MPT launcher as `sgimpirun`.

The MPT MPI implementation is somewhat arcane, in that it uses a relatively different launching method, compared to other MPI implementations such as MPICH and LAM. Frequently, MPI launchers make use of the *rsh* (or, equivalently, *ssh*) system tool to start the MPI processes on the necessary hosts. MPT makes use of another proprietary SGI software package known as *Array Services* (*AS*) to start the MPI processes. An array services daemon, `arrayd`, runs as root on each host in the cluster, and is configured to be aware of the other hosts in the cluster. `arrayd` listens

on a standard TCP port and a Unix domain socket, and responds to both user and remote arrayd requests on these sockets.

`sgimpirun` accepts command line parameters in a variety of forms. The exact specification of the MPI job on the command line affects the way in which the job is started. The most notable forms are:

1. `sgimpirun -np` *n a.out*
   where *n* is the number of MPI processes desired, and *a.out* is the name of the MPI program.

2. `sgimpirun` *host1,host2,...* `-np` *n a.out*
   As above, except that *n* MPI processes will be started on each of the hosts listed.

3. `sgimpirun` *host1,host2,...* `-np` *n a.out* `:` *host3,host4,...* `-np` *m b.out*
   In this case, *n* instances of the *a.out* MPI program will be started on each of *host1*, *host2*, and so on, while *m* instances of *b.out* will be started on *host3*, *host4*, and so on. This syntax allows the use of the *Multiple Program, Multiple Data* (*MPMD*) paradigm of distributed computing, rather than the less flexible *Single Program, Multiple Data* (*SPMD*) paradigm, as in the previous command line forms.

Note that, if hostnames are specified, the "`-np`" option may be omitted. This permits command lines such as

```
sgimpirun host1 n a.out : host2 m
b.out : ...
```

The use of this "feature" is strongly discouraged, as it introduces ambiguity (specifically, the inability of the MPI launcher to distinguish between an *n* parameter which is intended to be the number of processes (i.e. "`-np`"), and an MPI program that happens to be named as an integer. The use of "`-np`" rather than the more common "`-n`" is non-standard enough, but omitting it completely makes scripts that call `sgimpirun` in this way even less portable.

There are two ways in which `sgimpirun` can spawn the actual MPI processes. If no hostnames are specified on the command line (such as in the first form listed above), then `sgimpirun` will directly spawn the MPI processes, that is, by using the `fork()` and `execve()` system calls. In this case, the parent process of the MPI processes is the `sgimpirun` process, and the MPI processes inherit various aspects of its environment (such as environment variables, rlimits and group id (gid)).

By contrast, if hostnames are present on the command line, then `sgimpirun` will use array services to spawn the MPI processes. In this case, the parent process of the MPI processes is the `arrayd` process, and as such, the environment inherited by the child MPI processes is that of the `arrayd` process. While array services ensures that most of the `sgimpirun` environment is reproduced by `arrayd` prior to executing the MPI processes, however, there are three notable and unfortunate exceptions. The first is that `arrayd` modifies the value of the `PATH` environment variable, resetting it to a "safe" default of

```
/usr/sbin:/usr/bsd:/sbin:/usr/bin:
/usr/bin/X11:/bin:
```

While resetting the `PATH` in this way is a prudent security practice for daemons that execute priviledged commands on behalf of users, it does not make sense for an MPI launcher. The second problem is that the `gid` (group id) of the MPI processes does not necessarily match that of the `sgimpirun` process. Specifically, if a user belongs to several groups, and has used a command such as `newgrp` to change into another group, the MPI processes are started by `arrayd` with the user's default `gid`. This is clearly an oversight — after forking, `arrayd` calls `setuid()` to change the `uid` to the user, but simply neglects to call `setgid()`, since both the `uid` and `gid` of `sgimpirun` are passed to `arrayd`. Finally, when `arrayd` runs the user process, it does so by executing (with `execve()`) a command of the form

```
/bin/sh -c "cd cwd ; exec command
args..."
```

where */bin/sh* is the user's shell, *cwd* is the current working directory of the `sgimpirun`, and every parameter is protected from shell interpretation by surrounding it with single quotes. If the use of cpusets has been enabled using the RM API (see below), then this command takes the slightly different form

```
/bin/sh -c "cd cwd ; exec
/usr/bin/cpuset -i cpuset -I command
-- args..."
```

where *cpuset* is the cpuset to place the MPI processes into. Since this spawning involves the user's shell, it has the potentially unfortunate side-effect of upsetting the user environment if shell initialisation files are sourced. For example, `tcsh` always reads the user's `~/.cshrc` file on startup. A better solution would be for the `arrayd` to simply call `chdir()`, followed by directly `exec()`ing the necessary `cpuset` (or user) command.

`sgimpirun` also uses array services to negotiate a unique *Array Session Handle* (*ASH*) for the MPI job. This is a 64-bit value (usually represented in hex) which is unique for the lifetime of the job across the cluster. The ASH of a process is inherited by its child processes, and setting or changing the ASH of a process requires root priviledges. This ensures that the ASH identifies all the processes relating to that job. The ASH is set for the `sgimpirun` itself, as well as any MPI processes started by `arrayd`. (This means that `sgimpirun` always requires `arrayd` to be running, even when `sgimpirun` directly spawns the MPI processes.) This ASH can then be used to control all the processes in the MPI job, for example, to send signals, determine process ids, and so on. This can be done from the command line using the `array` command, or by linking to the `libarray` dynamic library.

The actual startup procedure of the MPI processes themselves is the same regardless of whether

sgimpirun has spawned them directly or with array services. One MPI process is spawned on each host (and if multiple programs have been specified, then one for each program), and this process is known as a *shepherd* process. MPI programs are dynamically linked at compile time against the MPT libmpi shared library (i.e. users specify -lmpi at compilation). A substantial amount of initialisation occurs at the run-time loading of this dynamic library. If a user runs an MPI program without using sgimpirun, then when the code reaches the MPI_Init() function, a simple error message is output ("mpirun must be used to launch all MPI applications"). sgimpirun notifies the MPI processes that they are being run within the MPI launcher by setting two environment variables, MPI_ENVIRONMENT and MPI2_ENVIRONMENT. Typical values for these variables are

```
MPI_ENVIRONMENT="51c8a8c0 49757 0"
MPI2_ENVIRONMENT="1 0 0"
```

The value in MPI_ENVIRONMENT specifies the host and port that the shepherd is to connect to in order to communicate with sgimpirun. The first value is the little-endian hex encoded IP address (i.e. in this case, 192.168.200.81), and the second value is the port. The sgimpirun has created and is listening on this socket prior to spawning the shepherd processes. sgimpirun closes this socket once the correct number of shepherds have connected to it, and all communication between sgimpirun and the shepherds occurs over this connection (for example, the standard output and error of the MPI processes is sent back to the sgimpirun process for output to the user). Once the shepherd process has connected to the sgimpirun, it is able to determine how many MPI processes should be started on that particular host. These MPI processes are then started by the shepherd, which does so by calling fork()[1] from inside the dynamic library initialisation code. This is *not* followed by a call to execve() — the actual MPI processes continue through the rest of the libmpi initialisation, where they block until they receive notification from the sgimpirun that all of the MPI processes have been created. Following this, they finish the libmpi initialisation and continue into user code. They block at the implicit barrier in MPI_Init(), but otherwise, they are free to run without restriction.

3.2 NF Requirements

There are a number of requirements that an MPI launcher must fulfill in order to be useful on the AC. These are (in decreasing order of importance)

1. Interface with the batch queueing system, both to determine the resources within the cluster that are available to an MPI job, and so that the batch system may record accounting information about the MPI job. Users must not be permitted to make use of resources that have not been reserved by the batch queueing system. The batch system must be notified of all the processes involved in the MPI job, so that it can monitor the resources used and record this at the completion of the job.

2. Allow MPI programs to be run interactively (with a limited amount of resources), as well as within the batch queueing system. This facilitates the development and debugging of code.

3. All processes (including MPI processes) of all jobs should be contained within a job cpuset (one for each host allocated to the job). This inhibits MPI processes from interfering with other jobs on the system by allocating cpus or memory from NUMA nodes not assigned to the job.

4. Each MPI process (*rank*) should optionally be placed into its own sub-cpuset. This allows MPI processes to maintain optimal page placement locality even in the presence of suspended jobs. Suspended jobs filling the memory of some NUMA nodes might otherwise lead to the running job allocating pages on remote NUMA nodes. The use of sub-cpusets may cause some short-term paging of suspended jobs but helps to repeatably provide highest performance for long running jobs.

5. Run non-MPI programs, such as shell-scripts. This is useful for tasks such as the staging-in of input files prior to the "actual" MPI job (and the staging-out of output files), and so forth.

6. The user environment at the time of running the MPI launcher should be replicated in the environment of the MPI processes. Apart from being a consistency that users reasonably expect, it also allows, for example, user limits that are set at login to continue to apply to any subsequent MPI processes.

7. Ideally, the launcher should be as backward-compatible with existing commonly found mpirun and prun command line options.

Clearly, sgimpirun does not meet most of these requirements.

1. sgimpirun does not interface with any batch queueing system. Furthermore, the way MPI process startup occurs within the libmpi dynamic library initilisation means that it is not very feasible to "shoehorn" communication with the batch queueing system at the time of MPI process creation. The way that sgimpirun allows users to specify hostnames, and does not use cpusets, means that there is no possible way to restrict the resources available to users. Also, there is no way for the batch queueing system to be aware of MPI processes started by arrayd on other hosts in the cluster. Even MPI processes on the same host as the sgimpirun are difficult to track, since they

[1] In fact, the shepherd directly calls the Linux-specific clone() system call, rather than fork() or the clone() libc library function.

have `arrayd` as their parent process, and the only possibility is to use the ASH of the MPI job to find the processes.

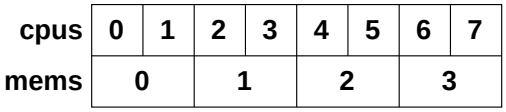
2. `sgimpirun` can certainly be run interactively as well as in batch jobs, but there is no way to limit the amount of resources.

3. `sgimpirun` does not use cpusets at all. If the `sgimpirun` is executed within a particular cpuset, and no hostnames are specified on the command line, then the MPI processes will also be in that cpuset. But if `arrayd` is used to start the MPI processes (either on the same host or other hosts in the cluster), then these MPI processes will be in whatever cpuset the `arrayd` is running in. There is no way to specify the cpuset to use on the command line to `sgimpirun` (despite the RM API supporting it, as described below). Even if cpusets were supported, there is no support for placing each individual MPI process into its own "fine-grained" cpuset. To see why this is important, Figure 1 shows an example of how remote memory accesses can occur when MPI processes are not restricted with cpusets, and how fine-grained cpusets improve the situation.

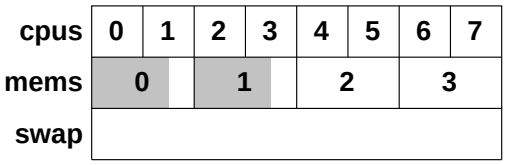
4. `sgimpirun` is unable to run non-MPI programs. If the program specified is not linked against `libmpi`, it will report an error. However, the program will be run on each host, and MPT only realises that it is not an MPI program when it terminates without having connected to `sgimpirun` on the socket specified by the `MPI_ENVIRONMENT` variable. This is a serious problem, as the non-MPI program or script may attempt tasks such as manipulating files, and may expect to be run n times, rather than just once (per host). Worse still is that if hostnames are specified to `sgimpirun`, the `stdout` and `stderr` of the non-MPI program may be lost by `arrayd`. `arrayd` redirects `stdout` and `stderr` to an error file in `/tmp/.arraysvcs`, but the contents of this file are not always reported back to `sgimpirun`[2]. This means that it is entirely possible for the non-MPI program to run to completion (success or error), without any output (normal or error) being reported to the user. This is not the case if no hostnames are specified — since `sgimpirun` directly spawns the MPI processes, they retain the `sgimpirun`'s `stdout` and `stderr`.

5. `sgimpirun` uses `arrayd` to spawn MPI processes, which has the known problem with the `PATH` environment variable and the `gids` of the processes,

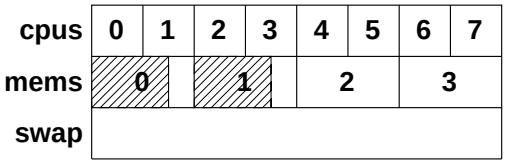
[2]It is not clear exactly why this is so. In addition, if two different MPI programs are specified on one host (i.e. `sgimpirun hosta -np n a.out : hosta -np m b.out`), then the same error file is used for both, and it is truncated when opened. This means that the output from the first shepherd will be lost when the second is started, and following that, the output from both will be mixed in the file.



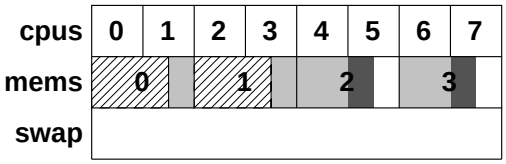
(a) Empty system



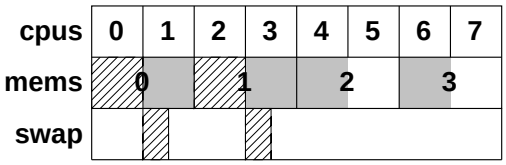
(b) Job α: 4 CPUs, 6 Gb



(c) Suspension of job α



(d) Remote memory used by job β



(e) Job α partially swapped

Figure 1: An example of the necessity of fine-grained cpusets for MPI jobs. Consider an 8 CPU, 16 Gb system (4 Gb per NUMA node, 2 Gb per CPU), shown in (a). A job α is run using 4 CPUs (0–3) and 6 Gb (75% of memory nodes 0–1), shown in (b). A second job, β, is now run using 8 CPUs and 8 Gb. This causes α to be suspended, shown in (c). The β processes running on CPUs 4–7 can fit the required 2 Gb into their local memory. However, memory nodes 0 and 1 each have only 1 Gb free (due to α), thus, half the memory used by β on CPUs 0–3 is remotely located on memory nodes 2 and 3, as shown in (d). The preferred situation would be to constrain each process to only the CPU and memory node it requires. This would force 1 Gb of job α to be swapped out of memory nodes 0 and 1, allowing all the memory for job β to be local.

as described above. Neither of these problems occur when `sgimpirun` directly spawns the MPI processes. The `PATH` issue causes obvious problems, where MPI programs may not be found when `sgimpirun` attempts to run them, despite being found by the shell from which the `sgimpirun` command is run. The `gid` issue causes problems with the APAC NF setup because users belong to multiple groups; their primary group is for their home institution, but all processes are set to run under a project group, of which a user may belong to several. To ensure that disk space used by users can be attributed to various projects, the filesystem quotas for the institution groups are set to no blocks or inodes[3]. If MPI processes are allowed to run with the user's primary `gid`, they will not be able to output to files, even though the `gid` of the `sgimpirun` has been correctly set by the login or batch job setup.

6. The command line syntax for `sgimpirun` is not compatible with existing (eg. MPICH) `mpirun` MPI launchers. In particular, the most common option, `-n`, is `-np` in `sgimpirun`. This means that even the most trivial of batch scripts (e.g. `mpirun -n 4 a.out`) would need to be modified.

3.3 Method

The limitations of `sgimpirun` described in the previous section are, for APAC NF, serious enough that `sgimpirun` is unusable for our purposes. Since MPT is closed-source software, we were required to overcome these limitations by writing our own replacement MPI launcher. This launcher makes use of the *RM API*, a dynamic library and corresponding external interface to the launching mechanisms used by `sgimpirun`. We refer to our launcher as `anumpirun`. The course of its development has seen it employ several mechanisms for launching MPI programs with the requirements of Section 3.2. These mechanisms can be grouped into three "generations" of `anumpirun`, each of which is now described.

3.3.1 First generation

The RM API provides several functions for initialising and running the MPI job. Using it requires linking against the `libxmpi` dynamic library. There are four important functions:

- `MPI_RM_batchargs()`: This takes the number of hosts, the number of processes for each host, and the name of the cpuset to use on each host.

- `MPI_RM_parse()`: This takes an `argc` and `argv` pair of parameters, which are parsed for options using the `sgimpirun` syntax, an `env` parameter which is the environment for the MPI processes, and three file descriptors which are the `stdin`, `stdout` and `stderr` to use for the MPI processes.

- `MPI_RM_sethosts()`: This takes an array of the IP addresses of the hosts to launch the MPI processes on. This is where the bulk of the work is done: the RM API contacts the `arrayd` to start the processes on the correct hosts and when this function call returns, all of the MPI processes will have been started and are being held. It is possible at this point to obtain the process IDs of the MPI processes on each host.

- `MPI_RM_go()`: This takes no arguments, and it releases the MPI processes to run. It does not return until the MPI processes have completed (either successfully or with an error).

The operation of `anumpirun` is simple in concept, and follows these steps:

1. The command line is parsed to determine the number of processes, options, and MPI program command line. If `anumpirun` is running within a batch job, this is detected and the resources allocated to the job are obtained.

2. `MPI_RM_batchargs()` is called with the appropriate number of hosts and processes. The cpuset used is the current cpuset of `anumpirun`, with an additional `mpi` sub-cpuset if one exists.

3. A "fake" command line in the `sgimpirun` syntax is constructed and passed to `MPI_RM_parse()`, along with the current environment (i.e. `environ`) and the current `stdin`, `stdout` and `stderr`.

4. `MPI_RM_sethosts()` is called with the correct hosts for the batch job, or `localhost` otherwise. This creates and holds the MPI processes.

5. The process IDs of the MPI processes on each host are obtained from RM API.

6. Another instance of `anumpirun` is executed (using `system()`) with the `--launch-remote-setup` option and the list of pids.

7. This process then uses the RM API to run a secondary MPI job on the same hosts as the primary one. The MPI job run is `anumpirun` with the `--remote-setup` option, along with the list of pids for that host. The second separate process to do this is necessary because the RM API assumes that a process will only launch a single MPI job[4].

8. The remote setup `anumpirun` is a normal MPI program (that is, it makes use of `MPI_init()`, `MPI_Comm_rank()`, and so on), and it performs three tasks on each host:

[3] Actually, technical reasons mean that the quotas are 4 blocks and 1 inode, but this is still effectively a zero quota.

[4] This is despite the RM API ostensibly supplying a `void*` "*mpi_handle*". It also suggests that the RM API is probably not thread-safe.

(a) Informs the batch queueing daemon on that node of the pids which belong to the MPI job.

(b) Creates a fine-grained sub-cpuset for each MPI process and evenly divides the `cpus` and `mems` amongst them.

(c) Places each MPI process into its respective fine-grained sub-cpuset.

9. After the remote setup has completed successfully, `MPI_RM_go()` is called to allow the MPI job to run.

When `anumpirun` is run, it detects if the MPI program specified is an "MPI program" or not. This is acheived with the `ismpi` shell script, which uses `ldd` to determine if the program is linked against `libmpi`, and, if so, confirms that it is the MPT `libmpi`. If a program is found to be a non-MPI program, then `anumpirun` uses an internal MPI wrapper to run the non-MPI program (this is acheived by changing the user-specified MPI command line of *`command parameters`* to be `anumpirun --mpi-wrapper` *`command parameters`*). This wrapper is very simple — it obtains the MPI rank and size in the usual way, sets the `MPI_RANK` and `MPI_SIZE` environment variables, and then `exec()`s the non-MPI command line. The user is able to force `anumpirun` to interpret a program as an MPI program with the `-w` option, and force it to be a non-MPI program with `-W`. Special exceptions exist for the profiling and debugging programs `profile.pl`, `histx` and `strace`, which are known to be non-MPI programs and cause `anumpirun` to run `ismpi` on the actual MPI command, rather than running it on `profile.pl`, `histx` or `strace`.

`anumpirun` does not require the user to specify the `-n` option. In this case, when run in a batch job `anumpirun` will determine the maximum number of CPUs allocated to the job and use that as the `-n` option. The user may override this and specify a lower number of processes to use, in which case a warning is printed. The user cannot ask for more processes than CPUs, unless the `-O` (overcommit) option is specified. In this case, no more CPUs are used, but more processes are allowed, which is sometimes useful when it is known that some processes will remain idle for large periods of time. When running interactively (i.e. outside a batch job), the default for `-n` is 1, and the upper limit is the number of CPUs in the `mpi` cpuset. On the interactive login node of the AC, this is 8 CPUs.

Every invocation of `anumpirun` by a user is logged to a world-writable file. The information recorded is the time, hostname, process id, user id, batch job id, working directory, full command line, exit status and the location of the error that caused `anumpirun` to exit, if any. This information is very useful in tracking down bugs (particularly with several commercial packages, where `mpirun` is not run directly, but is executed by another (possibly binary only) program), and in monitoring the system to ensure that there are not many errors occurring.

Before calling the RM API, `anumpirun` sets the `MPI_DSM_CPULIST` environment variable. This is used to specify to MPT the initial placement of the MPI processes onto CPUs[5]. This is achieved by setting the CPU affinity of the processes, and as such, the processes may later change the CPU they are running on. Thus, this is simply an extra aid, and does not replace the use of fine-grained sub-cpusets. This is useful for ensuring that data structures created by the MPI processes on startup are placed onto the local memory node (through the use of the "first-touch" memory allocation strategy). This is particularly important for MPI-related data structures, which can severely impact the MPI performance of the application if they are not local. Since these data structures are created very soon after the creation of the MPI process itself, using this MPT environment variable is the best method.

As mentioned earlier, `arrayd` has two problems that interfere with the running of MPI programs: the resetting of the `PATH` environment variable, and the resetting of the group id. These are overcome by `anumpirun` by prepending onto the specified MPI commands the following:

```
env PATH='$PATH' nfnewgrp gid env
LD_LIBRARY_PATH='$LD_LIBRARY_PATH'
```

where the values *`$PATH`* and *`$LD_LIBRARY_PATH`* are replaced with their respective values by `anumpirun` (prior to calling the RM API). This first restores the value of the `PATH` environment variable. `nfnewgrp` is a modified version of the standard `newgrp` command, such that its command line syntax is

```
nfnewgrp gid command args...
```

This allows it to execute the given command with `execvp()`, rather than using `execl()` and the user's shell, which is the case with the standard `newgrp` command and causes potential problems with shell quoting. As with `newgrp`, it is installed setuid-root, since `setgid()` can only be called by root (even to change into a group that a user belongs to). In addition, `nfnewgrp` fails if the user does not belong to the target group, and the target group requires a password. The final step restores the value of the `LD_LIBRARY_PATH` environment variable. This is necessary because the kernel clears the `LD_*` environment variables when setuid programs are run, as a security precaution, and if it is not restored, then many dynamic libraries (such as the `libmpi` MPI library itself) may not be found. (In addition, when running interactively on the AC users' shells are a setuid-root program for accounting purposes (which subsequently spawns the user's "actual" shell, such as `bash` or `tcsh`). This is another setuid program which causes the `LD_*` environment variables to be cleared.)

[5]The `MPI_DSM_DISTRIBUTE` environment variable is similar, but simply uses CPUs 0–n. This does not support hybrid applications that use threading and MPI, where each MPI process may require more than 1 CPU.

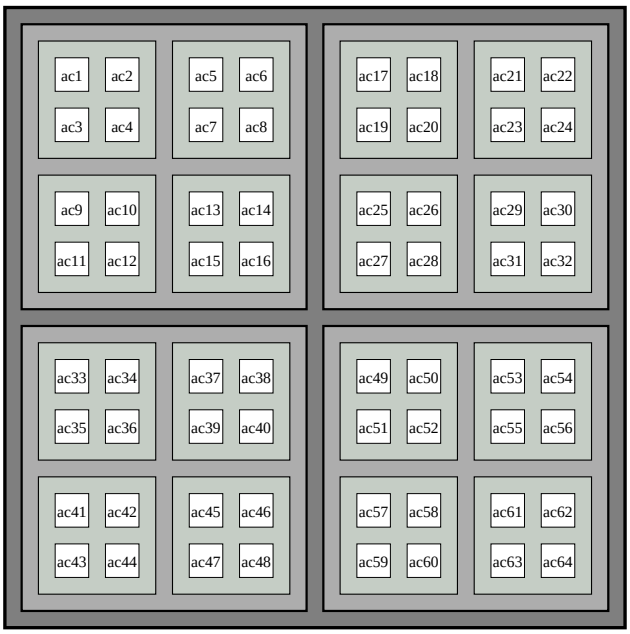


Figure 2: Layout of the sub-arrays used. A sub-array exists for each host (the white boxes labelled from ac1 to ac64), for each pair of hosts within the grouping of 4, and for each group of 4 hosts. This continues recursively until the array containing the entire cluster is obtained.

3.3.2 Second generation

The main problem with the first generation `anumpirun` was that there would often be problems starting jobs. These problems are MPT problems, and are directly related to `arrayd`. In particular, it is apparent that `arrayd` does not scale well. When creating a new ASH (which must be unique by definition), `arrayd` must contact every other host in the array (cluster). This can cause a large amount of network traffic, especially since each MPI job required two MPI jobs to be started (the "actual" MPI job itself and the "remote setup" job). On a partial subset of 20 hosts of the AC, with CPU 0 of each host reserved for system daemons such as `arrayd`, we found that it was still possible for jobs to fail to start, most likely based on slow responses from `arrayd`. Also, `arraydhas` the problem that a timeout of 45 seconds is encountered whenever any host in the cluster (or rather, when any `arrayd`) is down. These timeouts can, on occasion, cause jobs to fail to startup. Some problems have been observed with `arrayd` becoming deadlocked in futexes, however, we believe that these bugs have been resolved.

The second generation of `anumpirun` attempts to alleviate this problem by reducing the pressure on `arrayd`. This is achieved by two main changes:

1. The use of sub-arrays.
2. Avoiding the "remote setup" phase.

The main problem is that `arrayd` contacts every host in the cluster, even for a job that involves only a single host (such as the local host). To avoid this problem, we use an automatically generated `arrayd.conf` configuration file for `arrayd`. Array services supports the concept of multiple arrays, with each array being composed of a separate list of hosts. Our script generates arrays according to the natural topological hierarchy of the AC. We start with an array for each host, and then create an array for each group of 4 hosts, and an array for each pair of hosts within the grouping of 4. This procedure then recursively continues until an array is created for the entire cluster. This situation is shown diagrammatically in Figure 2, which shows the groupings used for 64 hosts. For example, in the upper-left quadrant, the following arrays will be created:

- For each group of 4 hosts:
  - `ac1_to_ac4`: ac1, ac2, ac3,ac4
  - `ac5_to_ac8`: ac5, ac6, ac7, ac8
  - `ac9_to_ac12`: ac9, ac10, ac11, ac12
  - `ac13_to_ac16`: ac13, ac14, ac15, ac16
- For each pair of these groups:
  - `ac1_to_ac8`:
    ac1, ac2, ac3, ac4, ac5, ac6, ac7, ac8
  - `ac1_to_ac4_ac9_to_ac12`:
    ac1, ac2, ac3, ac4, ac9, ac10, ac11, ac12
  - `ac1_to_ac4_ac13_to_ac16`:
    ac1, ac2, ac3, ac4, ac13, ac14, ac15, ac16
  - `ac5_to_ac12`:
    ac5, ac6, ac7, ac8, ac9, ac10, ac11, ac12
  - `ac5_to_ac8_ac13_to_ac16`:
    ac5, ac6, ac7, ac8, ac13, ac14, ac15, ac16
  - `ac9_to_ac16`:
    ac9, ac10, ac11, ac12, ac13, ac14, ac15, ac16
- For all 4 of these groups:
  - `ac1_to_ac16`:
    ac1, ac2, ac3, ac4, ac5, ac6, ac7, ac8, ac9, ac10, ac11, ac12, ac13, ac14, ac15, ac16

For the 52 hosts of AC, this gives a total of 172 arrays, which has not been found to noticably slow `arrayd`. Each array requires a unique name, which is constructed from the list of hosts with sequences of consecutive hosts replaced with "`to`", as shown in the above list[6].

As required by array services, each array is given a unique identifier (to allow the generation of unique ASH values). We then wrote a small static `libarrayset` library, which parses the `arrayd.conf` file, and, given a list of hostnames, finds the smallest array that contains all of those hosts. This is used by `anumpirun` to determine the best array to use for a job, which is passed to RM API with the `-a` option in

[6] `arrayd` has an 8kb limit on the length of the array name, and the array name cannot begin with a digit.

MPI_RM_parse. The batch queueing system also uses libarrayset to determine which array to use when suspending, resuming or killing a job. Contacting only the most relevant arrayds has a number of positive effects:

1. Job startup is quicker.
2. arrayds are less susceptible to high loads.
3. Job startup is not unnecessarily delayed when unrelated hosts are down.

The other improvement included in the second generation of anumpirun is the removal of the "remote setup" phase of MPI job startup. This required another mechanism for informing the batch system of the MPI processes, and for creating the fine-grained sub-cpusets and placing the MPI processes into them (at a stage that is early enough in their startup). This proved to be a challenge, due to the way MPT starts the MPI processes by directly fork()ing in the shepherd process,

First, anumpirun uses a *per-shepherd* wrapper. This takes command line arguments to specify the main cpuset, whether to notify the batch system, the ranks of the MPI processes it needs to deal with, followed by the actual MPI command line itself. This wrapper is able to inform the batch system (if necessary) and create the fine-grained sub-cpusets before exec()ing the MPI shepherd. The batch system is instructed to track all the processes below this one in the process tree.

However, the issue of placing the MPI processes into these sub-cpusets still remains. This is acheived by using the LD_PRELOAD dynamic linker environment variable to preload the lib_mpt_child_setup.so library. This library doesn't provide any functions, rather, it relies on the fact that its initialisation code is executed during the startup of each child process. This means that when the shepherd spawns the MPI processes, the lib_mpt_child_setup.so initialisation code is able to determine the relative rank of the process and place it into the respective sub-cpuset. Since the library has been preloaded, this happens before the initialisation of any other library (such as libmpi). An environment variable is set by the per-shepherd wrapper to its process id, allowing the lib_mpt_child_setup.so initialisation to recognise when it is initialising the shepherd process, and hence only perform operations when the parent process id matches the process id in the environment variable. The relative rank is determined by using the array services library to obtain the list of processes with the ASH of the current process. This list of pids is in order of process creation. The parent process is found in the list, and then the number of processes following this (that have the same parent process id) are counted until the current process is found, and this number determines the relative rank of the process, and therefore which sub-cpuset it should be placed into. Finally, the library unsets the LD_PRELOAD environment variable, in case any of the MPI processes spawn child processes.

Removing the remote setup stage helps to lower the pressure on the arrayds. It also has the additional benefit of approximately halving the startup time for MPI jobs.

3.3.3 Third generation

Unfortunately, we continued to observe MPI job startup failures and delays with the second generation of anumpirun (although their incidence was greatly reduced). The third generation of anumpirun completely does away with launching MPI processes via arrayd.

An unfortunately problem with the RM API is that it always launches the MPI program with arrayd, and there is no way to force it to directly spawn the processes, such as when using sgimpirun with no hostnames. However, there is an undocumented function in the RM API,

```
int mpirun(int argc, char *argv[])
```

which performs identically to sgimpirun (in fact, it is likely that the implementation of sgimpirun is merely `int main(int argc, char *argv[]) { return mpirun(argc, argv); }`). This can be used to directly launch MPI programs without using arrayd, and without requiring a publicly executable version of sgimpirun installed on the system. (Allowing users access to sgimpirun would be problematic, since they could potentially run their code on any host, at any time.)

However, there are two main hurdles with this strategy. The first is that the RM API *still* contacts arrayd when directly spawning the MPI processes, in order to assign an ASH to the processes. This has been shown to cause problems when a batch job is suspended during its communication with arrayd, as the arrayd sometimes blocks on these communications, and is therefore unable to service other requests[7]. The second problem is that the lib_mpt_child_setup.so initialisation code relies on the processes being placed in an ASH in order to determine their relative ranks. For these two reasons, it is necessary that the RM API continue to have an arrayd available, even if it is not used to spawn the MPI processes.

The solution, then, is to have one arrayd instance per MPI job. A program called aide was written which will start an arrayd for an MPI job. Since arrayd must be run as root, aide is installed setuid-root. It finds an available port that arrayd can use and starts an instance of arrayd using this port and

[7]This problem was avoided in the first and second generation anumpiruns by having the batch system suspend, resume and kill MPI jobs with array services, rather than by sending the signals directly to the processes. This means that the arrayd must process the signal request, and as it is not possible for the arrayd to be simultaneously processing the signal request and the job startup request, this problem is avoided. However, part of the rationale behind directly spawning MPI processes (as opposed to using arrayd), is that the batch system will then once again be able to send signals directly to the processes in question.

a minimal `arrayd.conf` file. It then drops root priviledges, sets the `ARRAYD_PORT` environment variable accordingly, and executes the given command line. When either the `arrayd` or the user program exit, `aide` kills the other and cleans up the various lock files. However, `arrayd` listens for connections both on an `AF_INET` TCP port, and an `AF_UNIX` local Unix domain socket. The Unix domain socket is sufficient in this case, since no remote connections are expected or desired for the per-job `arrayd`. `aide` uses `LD_PRELOAD` to preload the `lib_no_inet_bind.so` dynamic library. This has been written to intercept calls to the `bind()` C library function. When `bind()` is called with an non-`AF_INET` parameter, it calls the actual `bind()` function and returns the result, as usual. However, when it is called with a `AF_INET` parameter, it merely returns success (0) without calling the actual `bind()` function. This allows the underlying program to believe that it has successfully bound the socket to the TCP port, when in fact it has not, and no connections will ever be returned by `accept()` or `select()` calls on the socket. This is effectively a "user-space firewall" around `arrayd`.

Thus, when `anumpirun` previously called the RM API `mpirun()` function, it now executes `aide`, which in turn executes `anumpirun` with additional parameters indicating that the `mpirun()` function can be called. This allows each MPI job to have its own `arrayd` which is treated as part of the batch job (i.e. it is suspended, resumed and killed just like other processes in the batch job).

4 Conclusion

This paper has presented the techniques and policies implemented by the APAC National Facility to support its new SGI Altix cluster. These have been required to ensure that the diverse and competitive user workload obtains consistently high performance, whilst maintaining high system utilisation through the use of job suspension. The primary issues that have been addressed are those in the PBS-based batch queueing system, and those relating to the SGI MPT-based MPI system.

References

[1] 25th Top500 Supercomputer List, June 2005. `http://www.top500.org/lists/2005/06/`.

[2] Dean Roe Michael Woodacre, Derek Robb and Karl Feind. The SGI® AltixTM 3000 Global Shared-Memory Architecture, 2003. `http://sc.tamu.edu/whitepapers/altix/altix_shared_memory.pdf`.

[3] OpenPBS v2.3: The Portable Batch System Software, April 1999. `http://www.OpenPBS.org/`.

[4] W. Gropp, E. Lusk, and A Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 2nd edition, 1999.

[5] 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.

[6] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[7] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.