pjw: A "process jail warden" for HP-UX

Mark Bartelt
Center for Advanced Computing Research
California Institute of Technology
Pasadena, California
626 395 2522
626 584 5917 (fax)
mark@cacr.caltech.edu

1. Introduction

Many problems in computational science involve the use of large multiprocessor systems to run parallel applications. There are several common methods for implementing parallel codes. For example:

• MPI (see http://www-unix.mcs.anl.gov/mpi) is a standard which defines an API for exchange of messages between cooperating processes.

• OpenMP (see http://www.openmp.org) is a set of compiler directives and library routines for C, C++, and Fortran, which provide shared-memory programming on SMP and ccNUMA systems.

• Some programmers prefer to code parallel applications by making calls to pthreads (Posix threads) routines directly.

Regardless of the programming model used, one characteristic that all parallel applications have in common is that each of the N processes or threads does a portion of the total work. In a well-balanced application, each process or thread would perform 1/N of the total work. Periodic synchronization, either for processes or threads to exchange information with each other, or to enable a non-parallelizable portion of the code to run, is necessary.

A corollary of this is the fact that parallel codes run best when the individual processes or threads comprising a parallel job don't need to compete with unrelated resource-hungry processes (e.g. competing for CPU cycles on the same processor), so that when the N components of a parallel job start at the same time, they're likely to reach their synchronization points at approximately the same time. A single MPI process (or a {Posix|OpenMP} thread) which is slowed down by sharing timeslices with some other process will delay some or all of its N-1 siblings, if they've reached a point at which components need to synchronize with each other.

2. Computing Milieu

Caltech's Center for Advanced Computing Research (http://www.cacr.caltech.edu) supports scientists and engineers engaged in various fields of research in which high-end computing plays a major role.

Among CACR's computing resources are two 64-processor V2500 SCA servers. The V2500 SCA is a ccNUMA version of the V2500, supporting a maximum of four SMP nodes with a maximum of 32 PA8500 processors per node. (See http://docs.hp.com/hpux/pdf/A5532-90001.pdf; also (but apparently not online) the "*V2500 SCA HP-UX System Guide*", A5532-90003.)

One of CACR's V2500 SCA systems is a four-node system with 16 processors per node. The other is a two-node system with 32 processors per node.

Submission, queuing, and launching of jobs is handled by Platform Computing's LSF (Load Sharing Facility) software (see http://www.platform.com).

3. The Goal

A primary goal in configuring our V2500 systems was to provide optimal performance in a multi-job environment. For our systems' typical job mix, oversubscription (a situation in which N processes and/or

threads are competing for M processors, with M < N) invariably results in lower aggregate throughput. One typical example of the impact that oversubscription can have on the total runtime of parallel jobs is the following: Two MPI jobs (‘‘A’’, 16-way; ‘‘B’’, 8-way) were each run on an idle node. Five sequential runs of ‘‘A’’ were followed by five sequential runs of ‘‘B’’, and timings noted. Then five sequential runs of both were started simultaneously, both on the same node, to measure the impact of oversubscription.

Runtimes (wallclock seconds) for five consecutive runs of job ‘‘A’’:

A:	106.0	105.6	105.5	105.2	105.4	(total 527.7)

Runtimes for five consecutive runs of job ‘‘B’’:

B:	109.3	107.7	109.8	109.1	109.0	(total 544.9)

Runtimes for five consecutive runs of both jobs, with the two sets launched simultaneously, sharing the same 16-processor node:

A:	413.5	422.0	105.2	105.3	105.0	(total 1151.0)
B:	224.2	111.7	136.6	143.3	190.7	(total 806.5)

Job ‘‘B’’ ran more slowly by a factor of 1.5, on average. But the impact on ‘‘A’’ was even more severe: At first glance it appears that its slowdown was a bit more than a factor of two. But in fact, the final 344.5 seconds of its sequence of five runs occurred after the fifth run of ‘‘B’’ had completed. This means that during the time when both ‘‘A’’ and ‘‘B’’ were running simultaneously, the average runtime for ‘‘A’’ increased by more than a factor of four. And in fact, the total elapsed time necessary for both ‘‘A’’ and ‘‘B’’ to complete was longer than if the two sets of five runs had been done sequentially.

In short, it’s clear that avoiding oversubscription should be a high-priority goal, particularly when performance is important.

4. First Attempts

At first, it appeared that a number of features provided by HP-UX and LSF would help us to minimize the likelihood of oversubscription. Specifically:

• LSF provides hooks which give a system administrator control over how a job is launched: For example, environment variables can be set appropriately before a job begins running, and the method by which the job is actually started can be specified.

• HP-UX 11.10 (a special release for the V-class SCA systems) provides a command (*mpsched*) which can be used to request that a command be run in a particular ‘‘locality domain’’ (in the context of this discussion, a locality domain is the same as a physical node on an SCA-connected system).

• The *mpsched* command also permits one to specify scheduling policies for processes and threads. The scheduling policies determine where newly-created processes and threads will run:

RR	Each new {process|thread} will run in the next locality domain (round-robin order).
LL	A new {process|thread} will run in the least-loaded locality domain.
FILL	Launch in same node as parent, until one per processor; then spill to next node.
PACKED	All new {processes|threads} will run in the same node as the parent.

• HP’s implementation of MPI recognizes an environment variable (MPI_TOPOLOGY) which is used to specify how many MPI ‘‘worker’’ processes will run in each of a system’s locality domains. In addition, setting this variable also prevents an MPI job from starting if the number of MPI processes doesn’t match the number expected (i.e. as implied by MPI_TOPOLOGY).

With this in mind, it seemed that a reasonable strategy would be to have LSF launch jobs in the following way:

• After determining which node(s) a job should run on, set environment variables which specify the maximum number of threads a process is permitted to create, the node(s) on which MPI processes should run, etc. Mark these as ‘‘readonly’’ shell variables, in case a user’s job is a shell script which attempts to assign different values to them.

• Launch the job using *mpsched*, to ensure that it will start running on the correct node. Also, set the job's process and thread scheduling policies to PACKED, to ensure that all newly-created processes and threads will run on the same node as their parent.

• Keep track, via a job-to-node mapping, of which jobs have been launched on which nodes, and how many processors each job has requested, so that subsequent jobs can be placed on processors not already assigned to some other job(s), thereby avoiding oversubscription.

5. The Harsh Reality

It turned out that this approach failed, for two reasons. First, despite the fact that ...

• LSF was using *mpsched* to launch each job in an LSF-specified locality domain,

• MPI_TOPOLOGY was being set to force the job's MPI processes to run in that same locality domain,

• *mpsched*'s scheduling policies should have ensured that all new processes and threads would run in the same LSF-specified locality domain,

we have nonetheless observed situations in which two 16-way jobs were running on the same node (while another node had no active processes at all), despite the fact that the LSF batch system had specifically targeted the two jobs for different nodes. In other words, HP-UX occasionally decides, for unknown reasons, to migrate processes to nodes different from the ones requested.

Secondly, even if HP-UX had always behaved as it should it terms of honoring requests for process and thread placement, there are numerous methods by which users could easily circumvent our attempts to restrict the load they place on the system. For example, suppose that a user requests eight processors when submitting a batch job. The LSF job starter might set the MPI_TOPOLOGY variable to 0,0,8,0 (which tells MPI to run eight ''worker'' processes on node 2, and none on other nodes). Besides specifying the locality domain on which the MPI processes will run, this setting of MPI_TOPOLOGY also prevents the user from starting any N-way MPI job for which N does not equal 8. However, there would be nothing to prevent a user from requesting eight processors, and having the batch job be a shell script which would start two or more eight-way jobs, thereby acquiring more CPUs than LSF's job information would indicate, and thus oversubscribing the node on which the job was started.

Furthermore, the basic idea of attempting to restrict behavior by using readonly environment variables is flawed: Although *sh* and *ksh* use the same method for marking environment variables as readonly, *bash* uses a different method, so something marked as readonly by a *sh* script won't be recognized as such by a *bash* script. And *csh* and *tcsh* don't support readonly environment variables at all. So a user whose batch job is a {*bash*|*csh*|*tcsh*} script can set environment variables to values different from what the LSF job starter had set them to, thereby permitting more processes and threads to be created than the batch system intended. Or even in if a user's batch job were a {*sh*|*ksh*} script, the *env* command could be used to alter the value of readonly environment variables on a per-command basis.

6. What? No Processor Sets?

One solution, if the functionality had been available to us, would have been to arrange for the batch system to use processor sets to confine a job to a defined set of CPUs, thereby ensuring that no matter how many processes and threads a job might create, they would be limited to the N processors that the user had requested. Other operating systems (e.g. IRIX) provide this, but unfortunately HP-UX does not. Under IRIX, processor sets work as follows:

• A processor set is a group of CPUs to which access may be restricted.

• Each processor set has a configuration file (specifying the CPUs the processor set contains) and a name.

• Access to a processor set (permission to retrieve information about it, permission to run a process in it) is determined by a user's {read|execute} access to the configuration file.

• System calls exist to create/destroy processor sets, run processes in one, move processes out of one, etc.

It's easy to see how processor sets could be used to guarantee that batch jobs wouldn't interfere with each other: If someone requests N processors, the batch system could create an N-cpu processor set, and launch the job there. The user's job (and all new processes and threads that it might create) would be confined to its processor set ''jail''. So if the user created more processes or threads than originally requested, an oversubscription situation would still occur, but the user would be oversubscribing only against himself, and thus not interfering with others' jobs.

7. When in Doubt, Fake It!

But after a bit of thought, it became evident that even without operating system support for processor sets, a clever privileged daemon could provide equivalent (or at least sufficiently similar) functionality. Thus was born the ''process jail warden'' (*pjw*).

This is a program which monitors jobs launched by the LSF batch system, determines which processors each batch job will be permitted to use, and ensures that each job's processes and threads run only on the processors which the daemon has associated with the job.

The three critical things which enable this to work are:

• The *pstat()* system call permits a program to retrieve all important information about all the processes and lightweight processes (threads) which are currently active.

• The *mpctl()* system call can be used to lock a process or thread to a designated CPU.

• When the LSF batch system launches a job, a remote execution server (the process which causes the user's job to run) does a *setpgrp()* before spawning any child processes; furthermore, mapping between a job's process group and its LSF job ID is straightforward. This means that information returned by *pstat()* can be used to easily identify which processes and threads belong to which batch job.

8. How it Works

The daemon wakes up periodically, and does three things:

• It collects information about all current batch jobs, all processes, and all threads, updating the information it had the previous time it checked. New batch jobs may have been launched, in which case all the information about that job's processes and threads needs to be gathered. Existing batch jobs may have caused new processes or threads to be created, or old ones may no longer exist; in either case, tables of {process|thread} information need to be updated. Some batch jobs which were active before may have completed.

• For any new batch jobs which may have been launched, a set of processors needs to be assigned. For any old batch jobs which have completed since the previous time, whichever processors had been assigned to them need to be put back into the daemon's ''available CPU'' pool.

• For each active batch job, load balancing is done within the group of processors assigned to the job. This involves examining the state (e.g. RUN, SLEEP) of each process and thread. Those which aren't actually running needn't be locked to a processor. Those which are in a RUN state most of the time are locked to a CPU in a round-robin fashion within the group of processors assigned to the job.

9. Results

The *pjw* daemon does a good job of ensuring that batch jobs won't interfere with each other by competing for the same CPUs. It's possible that there may be brief periods of oversubscription between the time that the daemon goes to sleep and the next time that it wakes up to check on things. But these are negligible compared with the total amount of time that a typical job runs.

Of course, the daemon doesn't (and can't, and perhaps shouldn't) do anything about people oversubscribing against themselves, i.e. trying to use more processors than they had asked for. But now people who do this impact only their own job, and no longer interfere with others.