Enhancing an Open Source Resource Manager with Multi-core/Multi-threaded Support

Susanne M. Balle and Daniel J. Palermo

Hewlett-Packard Corp.
High Performance Computing Division
Susanne.Balle@hp.com, Dan.Palermo@hp.com

Abstract. Current resource managers do not have adequate node allocation and distribution strategies to efficiently schedule jobs on multi-core multi-threaded systems. Clusters composed of multiple cores per processor as well as multiple processors per node are presenting new challenges to users when trying to run their program efficiently on this class of machines. New allocation algorithms and their respective user interfaces need to be devised to ensure minimum contention for cache and memory, reduced on-chip contention, etc. as well as evaluate trade-offs between resource contentions.

1 Introduction

Multi-core processing is a growing industry trend as single-core processors rapidly reach the physical limits of possible complexity and speed. Both AMD and Intel are currently marketing dual-core processors. Quad cores are currently available and CPUs with larger core count will be available in 2008.

In a clustered environment, an important factor contributing to an application's performance is the layout of its processes onto the nodes, i.e. workload balancing. Developers currently worry about finding the optimal layout for their application on CPUs and nodes. With the introduction of multi-core and hyper-threaded processors, the level of complexity increases considerably. A common problem with today's resource managers e.g. LSF [3], PBSpro [1], etc. is that they consider cores to be "normal" CPUs without taking into consideration locality between cores and memory, resources shared between cores, or other architectural features of the system. For users using the default node allocation scheme, the lack of extra support in the resource managers can result in a performance degradation and variability.

The goals of this paper are to illustrate the problems that developers encounter when running their applications in a multi-socket multi-core clustered environment and to present solutions to these problems. We discuss how we enhanced an open-source resource manager, SLURM, to better support multi-core computing environments. SLURM is an integral part of HP's XC Cluster offering. The development of SLURM is a joint effort between Lawrence Livermore National Laboratory (LLNL), Hewlett-Packard, Bull, and LinuxNetworX. The HP

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2007, LNCS 4942, pp. 37–50, 2008.

XC cluster [5] is a scalable, Linux-based Supercomputing Utility, enabling high system utilization. It leverages industry-leading technology from open source, HP and its partners.

The paper starts with a brief description of multi-core architectures. We then introduce the SLURM resource manager and describe the current state of multi-core support in resource managers. We show the problems users encounter when using multi-core systems and present our solutions to these problems. We show performance data illustrating that both a standard benchmark (HPLinpack) and an ISV application (LSDyna) achieve sizeable performance improvements using the features available in the new multi-core aware SLURM.

2 Multi-core Architectures

To achieve higher processor density and higher performance per watt, processor architectures are focusing on multi-core designs. CPU vendors have exploited the traditional methods to increase performance, such as higher clock speeds and improved internal parallelism, to the fullest and must now rely on providing resources for explicit parallelism. Several multi-core processor architectures that are becoming available in HP systems are discussed below.

The dual-core AMD Opteron provides two 64-bit Opteron cores on a single die. Each core within a dual-core Opteron has two levels of cache with none of the levels shared between cores. L1 is a split 64KB/64KB instruction/data cache, and L2 is a unified 1MB cache. In 2007, AMD plans to offer a 4-core version of the Opteron which will also likely have a shared level 3 cache.

The dual-core Intel Xeon provides two 64-bit Xeon Extended Memory 64 Technology (EM64T) cores on a single die. Each core has two levels of cache with none of the levels shared between cores. L1 is a split 64KB/64KB instruction/data cache, and L2 is a unified 2MB cache. The Intel Xeon EM64T also supports 2-way Hyperthreading within each core resulting in one chip providing 4 logical processors to the operating system. In 2007, Intel released a 4-core version of the Xeon which, with Hyperthreading, would provide 8 logical processors in a single package.

The Intel Itanium Processor Family (IPF) incorporates a dual-core design in the next major revision of the processor code named Montecito. Compared with earlier releases, the Montecito release of IPF will provide two Itanium 2 cores on a single die, a separate L2 I-cache, increased cache capacity, extra functional units for shifting and population count, and Hyperthreading. Hyperthreading in Montecito provides hardware support for 2 logical threads per physical core providing a total of 4 logical processors to the operating system.

3 SLURM: Simple Linux Utility for Resource Management

A resource manager is a tool that dispatches jobs to resources according to specified policies and constrained by specific criteria. SLURM is an open-source

resource manager designed for Linux clusters of all sizes. The overview of the SLURM architecture presented in this section is an edited excerpt of the documentation available on the SLURM website [2] and the paper by Jette and Grondona [4].

SLURM provides three key functions: First, it allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (typically a parallel job) on a set of allocated nodes. Finally, it arbitrates conflicting requests for resources by managing a queue of pending work.

SLURM consists of a slurmd daemon running on each compute node, a central slurmctld daemon running on a management node and six command line utilities: srun, scancel, sinfo, squeue, scontrol, and smap which can run anywhere in the cluster.

The six SLURM commands have the following functionalities:

- srun is used to submit a job for execution, allocate resources, attach to an existing allocation, or initiate job steps.
- scancel is used to cancel a pending or running job or job step. It can also be used to send an arbitrary signal to all processes associated with a running job or job step.
- scontrol is the administrative tool used to view and/or modify SLURM state.
- sinfo reports the state of partitions and nodes managed by SLURM. It has a wide variety of filtering, sorting, and formatting options.
- squeue reports the state of jobs or job steps. It has a wide variety of filtering, sorting, and formatting options. By default, it reports the running jobs in priority order and then the pending jobs in priority order.
- smap reports state information for jobs, partitions, and nodes managed by SLURM, but graphically displays the information to reflect network topology.

SLURM uses a general purpose plug-in mechanism to select various features such as scheduling policies and node allocation mechanisms which allows it to be flexible and utilize custom as well as default methods. When a SLURM daemon is initiated, it reads a configuration file to determine which of the available plug-ins should be used.

4 Current State of multi-core/multi-threaded Support

In early 2006, we investigated current resource managers to understand if they had the adequate scheduling and node allocation strategies to efficiently schedule jobs on multi-core/multi-threaded systems. At the time of the investigation, we were not aware of any resource manager which has multi-core multi-threaded support. We investigated PBSpro 7 [1] and concluded that it allocates

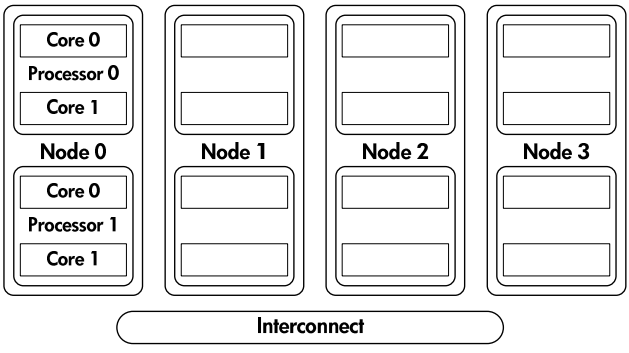

Fig. 1. Four-node dual-core dual-processor cluster

processors as its finest granularity. The latest version of LSF 6.2 [3] has an option which allows it to allocate resources at the core level but without taking into consideration the existing characteristics between cores. In versions of SLURM prior to v1.2 [2], the lack of multi-core support meant that it viewed cores as CPUs and did not make a distinction at the processor level.

Below we illustrate the problem that users typically encountered when using the default block distribution in SLURM (prior to v1.2) on a four-node dual-socket dual-core cluster.

Figure 1 shows a four-node cluster (Nodes 0 to 3) with dual-socket dual-core nodes. SLURM allocates resources at the node/CPU granularity level which means that for this example it would view each node as having four CPUs. An 8-process job is initiated using SLURM by typing: "srun -n 8 a.out". If the above system contained single- core sockets, the default layout would result in having processes 0 and 1 on Node 0, processes 2 and 3 on Node 1, processes 4 and 5 on Node 2 and processes 6 and 7 on Node 3. In a dual-core environment, the default node allocation would be processes 0 and 1 as well as processes 2 and 3 on Node 0 and processes 4, 5, 6, and 7 on Node 1 leaving Node 2 and Node 3 unallocated for this job. The latter allocation is not the expected default layout for the non-expert user and can result in detrimental performance for certain types of applications such a memory bound applications. The two processes on the cores within the same processor have to share hardware resources which could result in resource contention.

The existing pre-defined distributions, namely block and cyclic, are not adequate in a multi-core multi-threaded environment since they do not make a distinction between sockets and cores/threads. The extra levels of logical processors (cores and threads) in the CPU/socket should allow the job scheduler to create more elaborate distribution schemes.

In the next section, we present the features we are currently adding to SLURM as part of the multi-core support.

5 Identified Problems

We interviewed users with experience running applications on multi-core and multi-threaded systems to understand the problems they experience as well as the features they seek. We researched prior and ongoing work in the Linux kernel community to allow the Linux kernel users to take better advantage of the new architectures. We were especially interested in the different level of affinities in the Linux kernel and its schedulers.

There are three major features which have been introduced in the Linux 2.6 kernel that are of interest when discussing multi-core support and CPU affinity. These features include:

- The new "O(1)" job scheduler
- Kernel support for both process and memory affinity
- On-going work in page migration

At the end of our investigation phase, we identified three major enhancements that would make SLURM more attractive to users by increasing its usefulness in a multi-core environment.

The enhancements are:

- Need for a flag or feature allowing users to "pin" processes to logical processors (node, socket, core or thread). HPLinpack and LSDyna experiments, carried out on a four node dual-core Opteron cluster, confirm the interviewees' conclusions that pinning the processes to the cores does give better as well as repeatable timing results. For the non-expert users, we need to provide higher-level flags and have the pinning happen automatically.
- Need for new node allocation and distribution strategies for clusters of multi-core systems. This includes being able to specify a block or a cyclic distribution at the core level and at the thread level as well as adequate default distributions for the cores and the threads.
- Need for an adequate multi-core user interface which support the new multi-core enhancements. The user interface consists of low-level expert flags as well as high-level flags. The latter allows users to take advantage of the lower level logical processors (core or thread) without having to understand the system architecture in detail.

6 Pinning Processes to Cores

Process affinity is at the heart of the SLURM multi-core support since the Linux kernel job scheduler is not able to optimally assign processes to cores for high-performance technical computing applications. OS job schedulers, originally designed to achieve maximum throughput for oversubscribed workloads in single-core systems, when used for perfectly subscribed jobs in multi-core systems typically try to distribute load across available processors sometimes migrating processes to temporarily idle cores instead of staying where the cache

is hot. Users have observed that by pinning processes to cores, performance can be increased. This is a result of avoiding the overhead of migrating processes to cores with cold caches, thereby staying close to the caches that already contain the data a process is accessing. This is true even on systems comprised of multiple single-core processors.

Linux provides several affinity controls to pin or bind processes to cores. We have selected the sched_setaffinity(...) and the sched_getaffinity(...) interfaces provided in the Linux 2.6 kernel to implement our pinning feature. These interfaces allow users, or in our case SLURM on behalf of a user, to set and get a process's CPU affinity mask. Setting a process's CPU affinity allows SLURM to specify which scheduling queues are allowed for a given process thereby providing them with the ability to bind a process to a specific core or set of cores. One of sched_setaffinity(...)'s parameter is the affinity bit mask where a 1 indicates that a process may use the core (ID represented by the bit position). A mask of all 1's indicates the process may use any core (at the discretion of the OS). A mask containing a single 1 indicates a process is bound to a specific core.

We have added the ability to SLURM for users to bind their tasks to particular logical processor (sockets, cores per socket, and threads per core) as well as sets of logical processors using high-level flags as well as an expert low-level flag (see Section 9). We developed the SLURM affinity task plug-in using the task plug-in infrastructure. The former encompasses all the CPU affinity pieces needed, including automatic mask generation when using the high-level flags, to support the process to core pinning functionality in SLURM. We allow users to take advantage of this new feature in two ways. They can either provide the affinity mask directly to SLURM via the –cpu_bind flag or by using high-level flags (see Section 9).

One complication that arises when binding processes to cores is that different combinations of hardware, BIOS, and operating systems result in different core numbering schemes. Given, the fact that the core indexing varies among different systems, it is not possible to devise a single set of masks that would achieve the same results on all systems. When specifying masks directly with –cpu_bind users must be aware of the core mapping in order to achieve a desired placement across the processors and cores. The higher-level switches use an abstraction layer which will make the logical processor to physical core mapping transparent to users (see Section 10).

7 Node Selection and Allocation

SLURM uses a general purpose plug-in mechanism for selecting various features. The node selection plug-in also called select plug-in determines the set of nodes to be used for a given job based on the user's specified requirements. This type of plug-in is accessed within the slurmctld daemon.

SLURM currently supports two platform-independent node-selection plug-ins:

– select/linear – The default plug-in selects nodes assuming a one-dimensional array of nodes utilizing a best-fit algorithm. This plug-in does not allocate

individual processors, memory, etc. and is recommended for systems without shared nodes.
- select/cons_res – A plug-in that can allocate individual processors, memory, etc. within nodes. This plug-in is recommended for systems with programs sharing nodes.

It is important to notice that only one select plug-in can be specified for a given configuration and that the choice of select plug-in is done on SLURM start-up.

We researched the different SLURM components related to node allocation to figure out how to integrate the multi-core support. The conclusion of our investigation phase is that because the multi-core support should become an inherent part of SLURM it shouldn't be implemented as a separate node-selection plug-in.

We updated the select/linear plug-in, the select/cons_res plug-in as well as the node-selection infrastructure module with a mechanism that allows selection of logical processors at a finer granularity. The updated mechanism continues to support node level allocation and will support finer grained distributions within the allocated nodes. Users will be able to specify the number of nodes, sockets, cores and threads. Their applications should be distributed across a specified number of logical processors and the enhanced node allocation mechanisms will grant the requested layout if the resources are available.

8 New multi-core Process Distributions

The introduction of multi-core processors allows for more advanced distributions. Our first new approach is to create planes of cores. We plan to divide the cluster into planes (including a number of cores on each node) and then schedule first within each plane and then across planes. This approach results in new useful layouts. On a dual-socket node with quad-core sockets, this distribution could result in: as few as one plane (i.e. where the logical processors are scheduled by first filling up the nodes and then scheduling across the nodes) or as many as eight planes (always schedule across the nodes first). Two planes would provide better locality but potentially more contention for other resources. On the other hand, four planes (scheduling across the logical processors) would minimize contention for cache and memory.

We reused the srun "–distribution" flag to allow users to specify the new distribution methods (see Section 9):

"srun –distribution plane=<# of cores within a plane> <srun arguments>"

9 Enhanced User Interface

The increase in complexity of the processors and thereby of the systems makes it more difficult for users to specify a job's layout on the existing resources. Our

goal is for SLURM to provide an intuitive user interface to help users allocate their jobs in this environment. We have enhanced the user interface such that it is powerful for expert users, easy to use for naïve users as well as satisfactory for intermediate users. We preserve the existing interface and functionality to ensure backward compatibility.

As described above, several features required for adequate multi-core support have been identified as absent from SLURM. Adding these new features to SLURM required enhancing the user interface to providing users with the necessary functionality.

The enhancements impacting the user interface are:

- A low-level flag allowing users to explicitly "pin" processes to cores.
- High-level flags to automatically generate masks to pin processes to logical processors in terms of sockets, cores, and threads.
- A new multi-core "aware" distribution to give users more control over distributing tasks.
- A finer grained node description method which allows users to specify the number of sockets, cores and threads in allocation requests and constraints, and as displayed by sinfo, scontrol, and squeue.

Overview of New Srun Flags

It is important to note that many of these flags are only meaningful if the processes' affinity is set. In order for the affinity to be set, the task/affinity plugin must be first enabled in slurm.conf:

```
TaskPlugin=task/affinity      # enable task affinity
```

In the case where we set the affinity at a higher level than the lowest level of logical processor (core or thread) we allow the processes to roam within a specific socket or core. We introduce the notion of fat masks to describe affinity masks which bind a process to more than one logical processor.

Several new flags have been defined to allow users to better take advantage of the new architecture by explicitly specifying the number of sockets, cores, and threads required by their application. Figure 2 summarizes the new multi-core flags.

10 Motivation behind High-Level Srun Flags

The motivation behind allowing users to use higher level srun flags instead of –cpu_bind is that the later can be difficult to use. The high-level flags are easier to use than the low-level explicit binding –cpu_bind flag because:

- Affinity mask generation happens automatically when using the high-level flags.

Low-level (explicit binding)	
–cpu_bind=map_cpu:<list>	Specify a CPU ID binding for each task
–cpu_bind=mask_cpu:<list>	Specify a CPU ID binding mask for each task
–cpu_bind=rank	Bind by task rank
High-level (automatic mask generation)	
–sockets-per-node=<S>	Number of sockets in a node to dedicate to a job (minimum or range)
–cores-per-socket=<C>	Number of cores in a socket to dedicate to a job (minimum or range)
–threads-per-core=<T>	Number of threads in a core to dedicate to a job (minimum or range)
-B <S>[:<C>[:<T>]]	Combined shortcut option for –sockets-per-node, –cores-per-socket, –threads-per-core
–cpu_bind=sockets	Auto-generated masks bind to sockets (implied -B level)
–cpu_bind=cores	Auto-generated masks bind to cores (implied -B level)
–cpu_bind=threads	Auto-generated masks bind to threads (implied -B level)
Application hints	
–hint=compute_bound	use all cores in each physical socket
–hint=memory_bound	use only one core in each physical socket
–hint=[no]multithread	[don't] use extra threads with in-core multi-threading
New Distributions	
-m / –distribution	Distributions of: block | cyclic | hostfile | **plane=x** | **[block|cyclic]:[block|cyclic]**
New Constraints	
–minsockets=<MinS>	Nodes must meet this minimum number of sockets
–mincores=<MinC>	Nodes must meet this minimum number of cores per socket
–minthreads=<MinT>	Nodes must meet this minimum number of threads per core
Task invocation control	
–ntasks-per-node=<N>	number of tasks to invoke on each node
–ntasks-per-socket=<N>	number of tasks to invoke on each socket
–ntasks-per-core=<N>	number of tasks to invoke on each core

Fig. 2. New srun flags to support the multi-core/multi-threaded environment

– The length and complexity of the –cpu_bind flag vs. the length of the combination of -B and –distribution flags make the high-level flags much easier to use.

Also as illustrated in the example below it is much simpler to specify a different layout using the high-level flags since users do not have to recalculate mask or logical processor IDs. The new approach is effortless compared to rearranging the mask or map.

Given a 32-process MPI job and a four quad-socket dual-core node cluster (8 cores per node), we want to use a block distribution across the four nodes and then a cyclic distribution within the node across the physical processors. We have had several requests from users that they would like this distribution to be the default distribution on multi-core clusters. Below we show how to obtain the wanted layout using 1) the new high-level flags and 2) –cpu_bind.

10.1 High-Level Flags

Using SLURM's new high-level flag, users can obtain the above described layout on HP XC with:

mpirun -srun -n 32 -N 4 -B 1:1 –distribution=block:cyclic a.out
or
mpirun -srun -n 32 N 4 -B 1:1 a.out
(since –distribution=block:cyclic is the default distribution)

Where -N is the number of nodes to dedicate to a job. The -B 1:1 option requests binding with at least socket per node (with no maximum specified) and at least one core per socket (with no maximum specified). With cores shown as c0 and c1 and processors shown as p0 through p3, the resulting task IDs in a single node in the cluster are:

	c0	c1
p0	t0	t4
p2	t2	t6

	c0	c1
p1	t1	t5
p3	t3	t7

The computation and assignment of the task IDs is transparent to the user. Users don't have to worry about the core numbering (Section 6) or any setting any CPU affinities. By default CPU affinity will be set when using multi-core supporting flags.

10.2 Low-Level Flag –cpu_bind

Using SLURM's –cpu_bind flag, users must compute the logical processor IDs or taskset masks as well as make sure they understand the core numbering on their system. Another problem arises when core numbering is not the same on all nodes. The –cpu_bind option only allows users to specify a single mask for all the nodes. Using SLURM high-level flags remove this limitation since SLURM will correctly generate the appropriate masks for each requested nodes.

On a Four dual-socket dual-core node Cluster with Block Core Numbering The cores are shown as c0 and c1 and the processors are shown as p0 through p3. The CPU IDs within a node in the block numbering are: (this information is available from the /proc/cpuinfo file on the system)

	c0	c1
p0	0	1
p2	4	5

	c0	c1
p1	2	3
p3	6	7

resulting in the following taskset mapping for processor/cores and task IDs which users need to calculate:

taskset mapping for processor/cores

	c0	c1
p0	0x01	0x02
p2	0x10	0x20

	c0	c1
p1	0x04	0x08
p3	0x40	0x80

task ids

	c0	c1
p0	t0	t4
p2	t2	t6

	c0	c1
p1	t1	t5
p3	t3	t7

The above maps and task IDs can be translated into the following mpirun command:

```
# mpirun -srun -n 32 -N 4 –cpu_bind=mask_cpu:1,4,10,40,2,8,20,80 a.out
                                or
# mpirun -srun -n 32 -N 4 –cpu_bind=map_cpu:0,2,4,6,1,3,5,7 a.out
```

On a four dual-socket dual-core node cluster with cyclic core numbering. On a system with cyclically numbered cores, the correct mask argument to the mpirun/srun command looks like: (this will achieve the same layout as the command above on a system with block core numbering.)

```
# mpirun -srun -n 32 -N 4 –cpu_bind=map_cpu:0,1,2,3,4,5,6,7 a.out
```

Block map_cpu on a system with cyclic core numbering. If users do not check their system's core numbering before specifying the map_cpu list and thereby do not realize that the new system has cyclic core numbering instead of block numbering then they will not get the expected layout.. For example, if they decide to re-use their mpirun command from above:

```
# mpirun -srun -n 32 -N 4 –cpu_bind=map_cpu:0,2,4,6,1,3,5,7 a.out
```

they get the following unintentional task ID layout:

	c0	c1
p0	t0	t2
p2	t1	t3

	c0	c1
p1	t4	t6
p3	t5	t7

since the processor IDs within a node in the cyclic numbering are:

	c0	c1
p0	0	4
p2	2	6

	c0	c1
p1	1	5
p3	3	7

The important conclusion is that using the –cpu_bind flag is not trivial and that it assumes that users are experts.

11 Performance Results

To evaluate the performance of binding processes to cores we examined the behavior of the HPLinpack benchmark using the HPL program as well as of an ISV application, LSDyna. The HPLinpack benchmark solves a dense system of linear equations and is used to rank the Top 500 supercomputer sites. LS- Dyna is a general-purpose, explicit and implicit finite element program used to analyze the nonlinear dynamic response of three-dimensional inelastic structures. Both benchmarks were run on a system with a 2.6 Linux kernel.

Table 1 shows the performance of HPLinpack measured with and without affinity controls. The taskset command uses the sched_get,setaffinity(...) interfaces hereby allowing processes to be bound to a specific core or to a set of cores.

Binding processes to sockets or cores using taskset improves the application's performance. The greatest benefit is seen when binding to specific cores (0x1, 0x2, 0x3, 0x4 versus 0xf). The underlying functionality provided by taskset (sched_setaffinity) is the same as used in both HP MPI and SLURM (activated via the –cpu_bind option). The SLURM process binding supports non-HP MPI programs as well. It is our goal that HP MPI will use the SLURM multi-core support including the process binding on platforms such as XC where SLURM is available.

We also evaluated the effects of CPU binding on a larger, long-running application by examining the performance of LSDyna using a data set simulating a 3-car collision. Without any CPU binding, the overall run-time of the application on a single 2GHz DL385 dual-socket dual-core (Opteron) system is roughly 2 days and 6 hours. In Table 2, the performance of LSDyna is shown using different numbers of cores and nodes and various bindings.

On a dual-processor dual-core node, running LSDyna without binding which allows the OS to schedule the job across the cores, is 5.16% faster than forcing the job to bind to cores on the same socket. Binding the job to cores on the separate sockets, however, only improved performance slightly at 0.36% which means that we are likely reproducing what the OS was doing automatically. This demonstrates how the memory bandwidth is shared between cores on the same socket. Separating the two processes onto separate sockets for a memory bound application, improves performance by increasing the overall memory bandwidth.

Table 1. HPLinpack with N=32381 run on 16 cores (4 nodes x 2 sockets x 2 cores)

Configuration	CPUs	Time (sec)	% Speedup
No affinity control used	16	467.16	
taskset 0xf	16	481.83	-3.04%
taskset 0x1; 0x2; 0x4; 0x8	16	430.44	8.53%
–cpu_bind=map_cpu:0,1,2,3 -B 1:1	16	430.36	8.55%

Table 2. Performance comparison of LSDyna with and without CPU binding

Cores	Nodes	Binding	CPU binding option low-level flag high-level flag	Time (sec)	Time (D:H:M:S)	Speedup	% Speedup vs. no binding
1	1	No		194,809	2:06:06:49	1.00	
1	1	Yes	–cpu_bind=map_cpu:0 -B 1:1	194,857	2:06:07:37	1.00	-0.02%
2	1	No		104,994	1:05:09:54	1.86	
2	1	Yes	–cpu_bind=map_cpu:0,1 -B 1:1	110,702	1:06:45:02	1.76	-5.16%
2	1	Sockets	–cpu_bind=map_cpu:0,2 -B 1:1-1	104,620	1:05:03:40	1.86	0.36%
2	2	No		102,336	1:04:25:36	1.90	
2	2	Yes	–cpu_bind=map_cpu:0 -B 1:1	100,266	1:03:51:06	1.94	4.72%
8	2	No		33,616	0:09:20:16	5.80	
8	2	Yes	–cpu_bind=map_cpu:0,1,2,3 -B 1:1	31,996	0:08:53:16	6.09	5.06%
8	4	No		28,815	0:08:00:15	6.76	
8	4	Yes	–cpu_bind=map_cpu:0,1 -B 1:1	28,532	0:07:55:32	6.83	0.99%
8	4	Sockets	–cpu_bind=map_cpu:0,2 -B 1:1-1	26,081	0:07:14:41	7.47	10.48%

Examining an “8 core 2 node” run (resulting in all 4 cores utilized on both nodes), we conclude that binding processes to cores runs 5.06% faster. Once a job’s processes use all available cores, the additional OS processes result in a total number of “runable” processes than is greater than the number of cores. In this case, binding becomes necessary to avoid performance degradation.

When examining the placement of the processes on a loaded system, processes in the unbound run were seen to continually move between the cores which is due to the fact that OS processes cause the total number of runable processes to exceed the number of available cores. With binding, however, the processes stay on the selected core even when the system is fully utilized system.

On a less loaded system, unbound processes tend to stay on the core where they are initiated. This is the reason why the relative improvement of binding drops off with an “8-core 4-node” run (only two cores are utilized per node, leaving two cores idle). Compared to an “8-core 4-node” run with no binding, binding to cores on the same socket ran only 0.99% faster. We expect this run to represent the binding the OS did automatically. Since the system was only half utilized we saw very little migration between sockets. Binding to cores on separate sockets, however, ran 10.48% faster due to improved memory bandwidth since only one core was used in each of the two sockets.

12 Conclusion

The multi-core support we added to SLURM is available starting with SLURM version 1.2. It allows users to use SLURM's default settings on clusters of multi-core nodes without suffering any application performance degradation or large variation in application runtime from run to run. We considered different approaches as well as investigated several features which would help SLURM better support job launching and resource management in a multi-core environment. We decided to add support for all three types of users i.e. expert, intermediate, and naïve users rather than focusing on a single type. We believe that the high-level flags are easy to use and will allow even novice users to take close to full advantage of the new architecture. The process binding is generated automatically when users use the high-level switches thereby allowing users to access very sophisticated functionalities without having to understand the underlying hardware in detail.

Acknowledgment

The authors thank Chuck Schneider, John Eck, Teresa Kaltz, Dick Foster and Kirby Collins from the Solution Engineering group for input on the multi-core/hyperthreading support. We thank Ron Lieberman for many discussions on his experiences with multi-core support in HP MPI. We thank Dan Christians and Chris Holmes for sharing their discussions on the –procmask flag and finally we thank Moe Jette (SLURM project leader) and Danny Auble, Chris Morrone, and Mark Grondona from LLNL for their collaboration around SLURM integration and enhancements.

References

1. Altair pbs professional 7.0, `http://www.altair.com/software/pbspro.htm`
2. Slurm website (June 2007)
3. Platform, L.S.F.: Platform lsf administrator's primer v6.2 documentation (2006)
4. Yoo, A.B., Jette, M.A., Grondona, M.: Slurm: Simple linux utility for resource management. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 44–60. Springer, Heidelberg (2003)