Topology Aware Scheduling in the LSF Distributed Resource Manager

Chris Smith, Bill McMillan, and Ian Lumb

Platform Computing Corporation, Markham, Ontario, Canada
email: [csmith,billm,ilumb]@platform.com

ABSTRACT: Explicit job topology requirements provide a challenge for traditional batch queueing environments which aren't aware of the cc-NUMA characteristics of the Origin family of systems. This paper describes the integration of the LSF Distributed Resource Management framework with the IRIX cpuset API. The integration needs to address the issues of how to place jobs within a cpuset, how to decide which CPUs should be included in a cpuset based on the cpu and memory requirements of the job, and how to decide whether or not a particular machine can fulfill the job topology requirements at the time, or whether the job needs to wait for the topology requirements to be satisfied. LSF system was enhanced to provide an external scheduler interface to support topology scheduling and a job execution plug-in interface to support the binding of jobs to cpusets.

1 Introduction

The cache-coherent, non-uniform memory access (ccNUMA) architecture of the Origin family of systems has enabled the manufacture of very large configuration computer systems (up to 512 nodes boards for the Origin 2000, for 1024 processors, with up to 4GB per node board)[1]. This form of machine configuration allows programmers to run massively parallel programs, with very large memory requirements, using shared-memory programming semantics.

The NUMA aspects of the machine are not completely transparent to the programmer, though. While allowing processes to make use of very large process address spaces, the latency introduced by the non-uniform memory access model has made job runtime very dependent on the combination of CPUs which a job is scheduled onto. It has been noted that parallel programs can suffer from unpredictable runtimes based on the location where the shared memory segments for the program reside. It is possible for program pages to reside "far" from the processing of the data due to the first-touch page placement algorithm employed by IRIX[2].

There are many tools provided for programmers by which they can analyze their program dependencies (perfex), employ optimization algorithms on page placement (automatic page migration), and request specific memory and process mapping to node boards (topology specification using dplace)[3].

While the use of these tools allows a programmer to optimally run a job on a NUMA machine, one programmer is typically not the only programmer to make use of a large configuration machine. When multiple programmers

are contending for the memory and CPU resources of a computer, the goal of optimal job placement might conflict with the goal of optimally utilizing the machine's capabilities. In short, users will be competing for their share of the machine resources.

In order to address the contention over resources, systems will typically run resource management software to manage user access to the memory and CPUs of the system. SGI has provided some mechanisms to manage this issue[4]:

- The Miser batch queueing system. Miser allows users to "reserve" CPUs and memory for exclusive use for a user specified period of time. A user will submit a job to Miser, which will then choose a time slot when the job, with its particular CPU and memory requirements, can run.
- Cpusets. Cpusets are used to specify CPU and memory affinity for families of processes. They can be allocated both in "open" and "exclusive" modes, so that jobs will not only be contained on a subset of the system's CPUs, but they may be given exclusive access to those CPUs as well.

The problem with these mechanisms is that they are statically configured or configured ad-hoc. Miser queues are maintained by the system administrator and cannot be changed automatically to provide fairness between users, and cannot be changed automatically to rebalance the system between interactive and batch environments. Miser also doesn't attempt to address job topology requirements which can have a measureable impact on job performance.

Cpusets, on the other hand, can be created dynamically out of the available machine resources, but they aren't governed by any type of resource allocation policy that is evident in Miser and other resource managers.

In order to combine policy-based scheduling with enforcement of job topology and CPU requirements, we have implemented a topology aware scheduling mechanism for the LSF distributed resource manager.

2 Motivation

The current form of the LSF scheduler sees SMP machines as a collection of a number of CPUs along with a large amount of shared memory. Scheduling is done based on the number of "CPU slots" which have already been allocated to jobs, along with the current runtime load on the machine, but there is not necessarily a direct mapping of an LSF "CPU slot" to an actual CPU since the LSF scheduler cannot enforce a "CPU reservation" without the aid of the operating system.

The use of the IRIX cpuset API allows LSF to enforce it's CPU allocations for jobs. Not only does the cpuset provide "containment" so that an eight CPU job will only run on eight CPUs, but it provides a "reservation" so that those eight CPUs are guaranteed to be available only for the job which they were allocated to.

In addition to this basic view of the SMP, LSF does not inherently understand the ccNUMA characteristics of the machine. Two hosts with an identical number of free "CPU slots" would be considered equivalent, even though the job's topology requirements might fit more readily onto one of the two

hosts. This choice of host might have a significant impact on the runtime of the job.

In order to properly make use of the cpuset API on a large Origin class machine, LSF needs to understand how to properly choose the CPUs which will be in a cpuset. In some cases, it is better for a job to wait until the desired CPUs are available rather than be run on a set of CPUs which might increase the job runtime. Thus, the LSF scheduler needs to know how to decide whether a host can currently fulfill a job's topology requirements.

3 Design

The ultimate goal of the LSF integration with cpusets is to provide a mechanism such that an LSF job may be run in a cpuset. This might be so that a job can guarantee some CPU affinity to support a job topology requirement, or in order to allow a job to have exclusive access to it's allocated CPUs, avoiding The effect of thread migration and cache flushes. The integration must also support a mix of jobs, some of which run in a cpuset and some of which don't.

We investigated three approaches to cpuset integration.

- First, LSF could run jobs in statically defined cpusets. That is, a number of cpusets would be defined at system boot time for use by the resource management system. Both open and exclusive cpusets could be used in this case, but it would be difficult to support the multitude of combinations of cpuset options required by the wide range of user jobs. This approach would work in a situation where we wanted to coarsely separate the machine into batch and interactive sections.
- A second approach would be to allocate cpusets dynamically using a first-fit algorithm. This approach would allow each job to run in its own exclusive cpuset, thus providing enforcement for LSF's per job CPU allocation. It would also allow the optional use of cpuset options such as MEMORY_EXCLUSIVE or MEMORY_MANDATORY on a per job basis. The problem with this approach is that a first-fit algorithm would not provide optimal allocation of CPUs based on a job's specific topology requirements. The advantage of this solution is that the LSF scheduler would not require any changes, since it can already schedule based on the availability of CPU slots.
- The third approach is an extension of the second method whereby we introduce the ability to schedule jobs based on cpuset topology requirements. That is, LSF scheduler is enhanced to make a job placement decision, or whether to wait to run a job, based on the availability of a particular CPU topology.

While the third approach is more complicated to implement, discussions with prospective cpuset users highlighted the importance of CPU placement for parallel jobs on Origin servers.

In order to integrate LSF with the cpuset mechanism we needed to identify the events during a job lifecycle where interaction is required. Five different key events were identified:

1. Expression of the topology requirement. A mechanism is required which will allow a user to define the topology requirements for the job so that a scheduler can make a host placement decision. This mechanism needs to support the existing cpuset allocation options, as well as provide a means for extending the expression without changing the underlying mechanism.
2. Scheduling to a host which could satisfy the topology requirement. Since LSF inherently does not understand ccNUMA architecture, and since the underlying architecture might be changed, a scheduling mechanism was required which could be updated without needing to change the LSF scheduler itself. This mechanism could be subdivided into two distinct service functions:
   - The first is an information service which provides the current cpuset state and availability for a given host. This service needs to understand the underlying ccNUMA topology of the host.
   - The second is a centralized scheduling service for matching user specified topology requests with currently available resource topologies.
3. Creation of the cpuset. After a host had been selected to run the job, and the job has been dispatched to this execution host, the cpuset for the job needs to be allocated according to the topology requirement for the job. The cpuset at this time must also be named in such a way that later actions can derive the name from known process state (e.g. the LSF job id).
4. Binding the processes of the job to the cpuset. Once the cpuset has been allocated, the job itself needs to be bound to this cpuset. Care must be taken to make sure that the entire process tree of the job is contained within the cpuset.
5. Cpuset deallocation. Once the job has completed, the resources used by the cpuset need to be released.

Underlying these requirements was the goal of making the scheme extensible. This was done so that future work on the cpuset mechanism, more optimized topology scheduling algorithms, or different underlying hardware topologies, could be done without needing to redesign or reimplement the framework supporting the NUMA scheduling.

4 Architecture

The implementation of the cpuset integration according to the above design criterion was done using a set of different architectural modules which interacted with each other.

4.1 Host-based Topology Daemon

A daemon runs on each host which supports the cpusets. This daemon is responsible for keeping track of the machine's hardware graph, as well as the current allocation of cpusets. By having a current snapshot of the hardware graph and in-use CPUs in one place, there is a single decision point as to whether a particular topology can be satisfied. This daemon considers boot

cpusets and cpusets manually created by the system managers, and adjusts its notion of CPU availability based on that information.

This daemon also performs the cpuset specific actions of allocation and deallocation (an operation which needs to be done as the root user). It supports protocol interfaces for performing queries, allocation and deallocations.

4.2 External Scheduler

An external scheduling module was implemented to keep the specifics of NUMA aware scheduling external to the main LSF scheduling loop. The interface between the LSF scheduler and this external scheduler is a functional interface. Among other operations, there is an operation so that the LSF scheduler can ask "does this host satisfy requirement X" without needing to understand any details about X. Another operation is to sort the list of hosts which can satisfy the requirement by which one provides the "best" topology.

Essentially, the external scheduler acts as a "request broker", since it translates the user-supplied topology requirement to an availability query to a host's topology daemon, which then interprets the request and makes a reply.

4.3 Job Execution "plug-in" Functions

LSF currently provides various site configuration hooks which are called during a job's execution lifecycle[5]. There is a pre-execution function, a "job starter" function (interposed between the slave batch daemon and the execution of the actual job), and a post-execution function. These mechanisms could have been used to implement the allocation, binding and deallocation of the cpuset, however this would have made the interface cumbersome for users wishing to perform other functions using these hooks.

A shared object is loaded into the slave batch daemon which provides entry points for pre and post job processing. One entry point is used to allocate a cpuset, one for binding the job to the cpuset, and another for deallocaing the cpuset. Using a shared object allows the implementation of these functions to change without requiring modifications to LSF's slave batch daemon.

4.4 Module Interactions

The modules use various ways to communicate with each other in order to implement the five actions identified in section 3. The interactions are diagrammed in figure 1 (on page 10). The numbers in the diagram are referenced in the interaction description below.

A job submission time (1 in the diagram), the user defines the topology requirement for the job by using the new "-extsched" command-line option to LSF's bsub command. This option attaches the topology requirement string to the job using the new bpost functionality from LSF 4.0. This attachment can then be read using the bread command (or the equivalent LSF library API).

An example of an LSF submission command which attaches a job (command-line "command") to an exclusive cpuset using CPUs 24 to 39 and 48 to 53. The MEMORY_MANDATORY option is specified to restrict memory allocations for

the processes from the nodes on which the CPUs reside.

```
bsub -n 32 -extsched \
'CPU_LIST=..;CPUSET_OPTIONS=..'\
command

where:
CPU_LIST=24-39,48-53
CPUSET_OPTIONS=\
CPUSET_CPU_EXCLUSIVE\
|CPUSET_MEMORY_MANDATORY
```

In order to schedule the request, the LSF scheduler (MBD) calls into the LSF scheduler (2) which then queries the topology daemon on a host to see if it can satisfy the job's topology requirements (3). These requirements are "bread" from the job, and then passed in the query message. The topology daemon then responds whether or not it can schedule the job. Since the external scheduler queries are performed within the LSF scheduler's normal scheduling run, the topology daemon query only represents one "filter" for job candidate hosts. This means that the external scheduler will coexist with LSF's current scheduling algorithms and mechanisms, such as scheduling based on resource requirements (e.g. the job needs one gigabyte of memory), the preemption algorithm, fairshare, and CPU slot reservation, to name a few.

After the scheduling run, a job will have a list of hosts which can satisfy the topology requirement for the job. This list is then passed to the external scheduler, which then reorders the list based on the cpuset which has "best" topology available. For the best-fit algorithm this would be the cpuset which offers the shortest radius between CPUs in the cpuset.

Once the execution host has been selected, and the job has been dispatched to the slave batch daemon (SBD) on that host to run (4), the slave batch daemon will call a plug-in function to allocate the cpuset. This function will ask the topology daemon(5) to create the cpuset on behalf of the job(6). The LSF job id will be passed to the topology daemon so that the cpuset can be named using the unique LSF job id, thus allowing the later stages of the process to properly identify the cpuset. The topology requirements will also be "bread" again so that the topology daemon can allocate the proper set of CPUs.

Another plug-in function is then called to bind the job to the cpuset (7). This plug-in function needs to be called in a process which will ultimately be a parent for the entire job. Since it is called from the slave batch daemon, this can be guaranteed.

Lastly, once the job has completed and it's processes exited, a plug-in function is called to ask the topology daemon to deallocate the cpuset.

5 Results

The use of IRIX cpusets with LSF has shown to have a positive effect on throughput benchmarks run by SGI in response to customer benchmark requests.

Once such benchmark consisted of a mix of 50 MPI jobs (one was a hybrid MPI/OpenMP job)[6]. The mix as provided by the customer (Mix A) was not to be changed (in terms of ordering and number of cpus), included some 5 minute sleeps during job sub-

mission, and was run on a 128 processor, 128 gigabyte Origin 3000 machine. [1]

The ideal mix time (with no sleeps) was calculated to be 2:21:22, using the formula:

$$\frac{sum(jobwallclock * numprocs)}{numprocs}$$

The LSF configuration used included a job slot limit at the host level of 128 (one slot per cpu). To reduce dispatch delays the JOB_ACCEPT_INTERVAL for the cluster was 0 (meaning multiple jobs could be dispatched to a host on one scheduling run), and NEW_JOB_SCHED_DELAY for the queue was set to 0 (so that a scheduling run would be initiated as soon as a job was accepted by LSF). When cpusets were used, they were created with the cpuset option CPUSET_MEMORY_LOCAL.

Running through LSF without the cpuset integration yielded a runtime of 4:19:01. Of note was the variance in runtime between two identical jobs, one which ran in 39:34.36 and one which ran in 8:17.02.

Running through LSF with the cpuset integration yielded a runtime of 2:51:06. The two identical jobs mentioned previously ran with much less variance. Job 1 in 8:08.39 and job 2 in 8:03.69.

Running the benchmark again without the sleeps yielded a runtime of 2:35:47.

See table 1 for the summary of results, and table 2 (on page 11) for a breakdown of individual job runtimes with and without cpusets compared to the ideal runtime.

Mix condition	Runtime
Ideal time	2:21:22
Without cpusets	4:19:01
With cpusets	2:51:06
No sleeps	2:35:47

Tab. 1: Summary of Mix A runtimes

6 Future Work

A number of future enhancements can be made to the implemented topology scheduling framework:

- The external scheduler interface can be extended. Currently the interface supports functions to decide whether a host is suitable for providing the resources required by a job. Further interfaces could be defined at different points in the scheduler to allow external algorithms for determining which user should go next, or to help resolve external job dependencies (e.g. a file transfer has completed), or to call to another, 3rd party scheduler (e.g. Maui scheduler).
- The external scheduler interface could be publicized, and an SDK developed so that sites can program their own scheduling algorithms for LSF.
- Different algorithms for topology scheduling can be examined and implemented. For example, topology requirements could come from a dplace file, or an algorithm could be developed to schedule in a "hy-

[1] Due to the need to keep details of the customer request confidential, the actual applications run in the benchmark cannot be described in this paper.

percube" topology, rather than the "cluster" topology.

7 Concluding Remarks

We presented an architecture for topology aware scheduling within the LSF distributed resource manager, and the use of IRIX cpusets to enforce the topology and CPU allocation for jobs by the LSF scheduler. The integration was designed by examining which events during a job's lifecycle need to be modified to deal with the scheduling and allocation of cpusets, and discussed the architectural modules which implement the integration.

We found that it was very important to identify the events during a job's lifecycle where topology scheduling decisions and actions needed to be taken. By making this analysis, it was straightforward to design architectural components which could deal with each event, thus encapsulating each step in a smaller, more focused module, which was easier to implement.

The component approach also allowed us to design an external scheduler to extend the LSF scheduling capability. This was a better approach than "hard-coding" the understanding of ccNUMA architecture into the LSF scheduler, since the architecture may change, and is specific to a particular hardware and operating system platform. The external scheduler also provides the beginnings of a framework by which the LSF scheduler could be easily modified to support arbitrary external scheduling algorithms (not just NUMA schedulers), much improving LSF's extensibility.

As expected, the use of cpusets for running jobs decreased the runtime variability of parallel tasks. Moreover, the combination of LSF with cpusets yielded an improvement of at least 50% on a simple throughput benchmark, and with some submission improvement (no sleeps), the benchmark ran only 10% longer than the ideal mix time.

Finally, it is interesting to note the verification of some past LSF architectural improvements. The bpost/bread mechanism was introduced in LSF 4.0 in order to facilitate the integration of LSF with third-party software packages. By making use of this mechanism within the implementation of the external scheduler, we have verified the utility of the bpost/bread functionality, and set an example for how this functionality can be used.

Acknowledgements. We would like to acknowledge Dan Jones at FNMOC for his role in the development of some of the concepts inherent in topology scheduling during an implementation of similar functionality at FNMOC by SGI and Platform Computing.

The authors also would like to acknowledge Dave Anderson from SGI for providing the benchmarking data presented here in Section 5.

References

1. James Laudon and Daniel Lenoski: The SGI Origin: A ccNUMA Highly Scalable Server. From http://www.sgi.com/origin/images/isca.pdf.
2. Talbot & Kelly: Stable performance for cc-NUMA using first-touch placement and reactive proxies, *High Performance Computing Systems and Applications, J. Schaeffer (ed.)*, Kluwer Academic Publish-

ers, 1998.
3. *Origin2000 and Onyx2 Performance Tuning and Optimization Guide*, Document Number 007-3430-002, Silicon Graphics, Inc., 1998.
4. *IRIX Admin: Resource Administration*, Document Number 007-3700-003, Silicon Graphics, Inc., 2000.
5. LSF Administrator's Guide, Platform Computing Corporation, 2000.
6. Anderson, D. Private communication, May 2001.

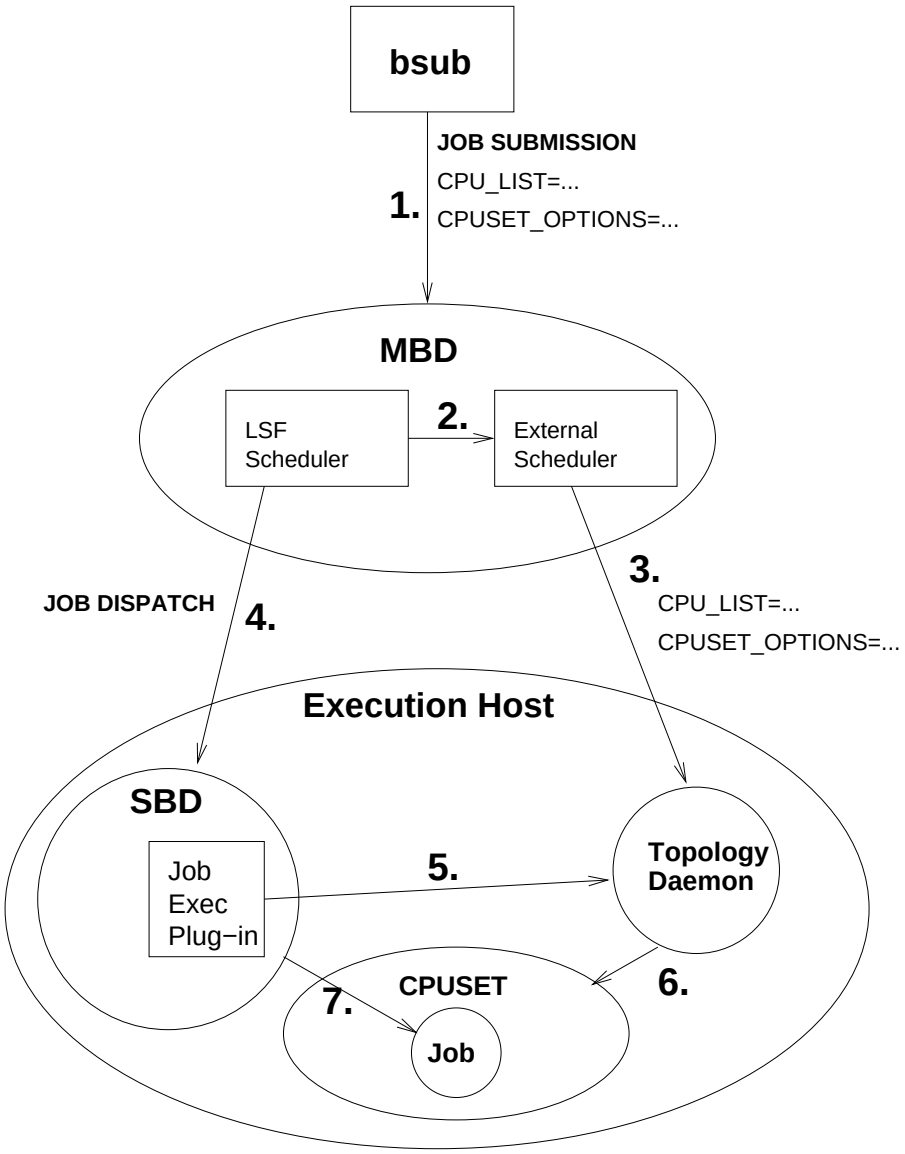

Fig. 1: Module interactions for topology scheduling

Code	**Time dedicated**	**Time without cpusets**	**Ratio w/o over dedicated**	**Time with cpusets**	**Ratio with over dedicated**
001_pgm_03	0:29:20	0:31:25	1.07	0:30:04	1.03
002_pgm_03	0:16:29	0:20:17	1.23	0:16:32	1.00
003_pgm_11	0:17:42	0:19:13	1.09	0:18:14	1.03
004_pgm_12	0:04:59	0:06:52	1.38	0:05:04	1.02
005_pgm_06	0:04:41	0:04:55	1.05	0:04:56	1.05
006_pgm_08	0:13:52	0:15:49	1.14	0:12:53	0.93
007_pgm_06	0:04:41	0:05:00	1.07	0:04:54	1.04
008_pgm_12	0:04:59	0:07:26	1.49	0:05:07	1.03
009_pgm_02	1:03:29	2:23:00	**2.25**	1:03:29	1.00
010_pgm_10	0:27:24	0:27:40	1.01	0:28:32	1.04
011_pgm_09	0:11:31	0:21:51	1.90	0:11:18	0.98
012_pgm_02	1:03:29	3:22:44	**3.19**	1:02:33	0.99
013_pgm_11	0:17:42	0:18:46	1.06	0:17:48	1.01
014_pgm_01	0:40:00	0:37:41	0.94	0:36:16	0.91
015_pgm_06	0:02:58	0:03:01	1.01	0:02:58	1.00
016_pgm_08	0:26:15	1:18:46	**3.00**	0:26:29	1.01
017_pgm_01	0:20:00	0:34:15	**1.71**	0:18:23	0.92
018_pgm_06	0:02:58	0:02:59	1.01	0:02:58	1.00
019_pgm_06	0:01:13	0:01:25	1.17	0:01:15	1.03
020_pgm_08	0:51:07	0:50:16	0.98	0:48:52	0.96
021_pgm_06	0:07:32	0:07:39	1.02	0:07:33	1.00
022_pgm_06	0:01:13	0:01:16	1.04	0:01:15	1.02
023_pgm_07	0:02:37	0:03:22	1.28	0:03:09	1.20
024_pgm_06	0:01:13	0:01:16	1.04	0:01:15	1.03
025_pgm_03	0:17:54	0:18:33	1.04	0:18:25	1.03
026_pgm_04	0:40:11	0:55:58	1.39	0:35:08	0.87
027_pgm_06	0:01:13	0:01:19	1.09	0:01:14	1.02
028_pgm_05	0:11:43	0:13:21	1.14	0:12:03	1.03
029_pgm_07	0:02:37	0:07:36	**2.91**	0:02:45	1.05
030_pgm_08	0:51:07	1:05:15	1.28	0:48:51	0.96
031_pgm_07	0:02:37	0:02:44	1.04	0:02:45	1.05
032_pgm_12	0:04:59	0:12:57	**2.60**	0:05:02	1.01
033_pgm_11	0:11:03	0:11:07	1.01	0:11:07	1.01
034_pgm_06	0:02:58	0:03:02	1.02	0:02:58	1.00
035_pgm_08	0:26:15	1:42:12	**3.89**	0:26:37	1.01
036_pgm_06	0:07:32	0:07:41	1.02	0:07:33	1.00
037_pgm_01	0:40:00	0:37:33	0.94	0:36:32	0.91
038_pgm_02	0:07:35	0:39:34	**5.22**	0:08:08	1.07
039_pgm_02	0:07:35	0:08:17	1.09	0:08:04	1.06
040_pgm_03	0:53:05	0:55:18	1.04	0:53:38	1.01
041_pgm_12	0:04:59	0:07:23	1.48	0:04:59	1.00
042_pgm_12	0:04:59	0:08:36	1.73	0:05:01	1.01
043_pgm_06	0:01:13	0:01:22	1.12	0:01:19	1.09
044_pgm_08	0:13:52	0:58:20	**4.21**	0:12:56	0.93
045_pgm_12	0:04:59	0:06:25	1.29	0:05:01	1.01
046_pgm_11	0:17:42	0:18:19	1.04	0:17:41	1.00
047_pgm_06	0:07:32	0:07:36	1.01	0:07:32	1.00
048_pgm_01	0:20:00	0:21:37	1.08	0:18:21	0.92
049_pgm_01	0:02:14	0:02:26	1.09	0:02:23	1.07
		Averages	1.53		1.01

Tab. 2: Breakdown of job runtimes for Mix A