GENERATING ARTIFICIAL LOAD PATTERNS ON MULTI-PROCESSOR PLATFORMS

Turkey Alsalkini
Department of Computer Science, Heriot-Watt University, Edinburgh, UK
ta160@hw.ac.uk

Greg Michaelson
Department of Computer Science, Heriot-Watt University, Edinburgh, UK
G.Michaelson@hw.ac.uk

ABSTRACT

It is desirable to evaluate system performance under repeatable conditions. Typically, parallel systems are evaluated on dedicated platforms with little or no external impact on load. For dynamic systems, however, such as those that adapt to changing conditions, it is necessary to generate both predictable and realistic patterns of load. Furthermore, on shared non-dedicated systems, there is a need to compensate for the unpredictable loads introduced by other users.
We have developed a novel load function which may be instantiated to generate predictable patterns of load across multiple processors. Our function can both generate idealised load patterns, and record and playback real load patterns. Furthermore, it can dynamically maintain a required load pattern in the presence of external real-time load changes, which makes it particularly suitable for experimentation on shared systems. We discuss the design of the load function and show that it can generate dynamic, adaptive and precise load, with minimal impact on system load. We then illustrate its use in the experimental evaluation of static/dynamic load balancing, load stealing and mobile skeletons.

KEYWORDS

Load, Multi-processor, Multi-core, Evaluation, Balancing, Mobile Computing.

1. INTRODUCTION

In recent years, communication networks and computer environments have offered resources which are distributed across a large number of systems and are shared by a large number of users. The demand for resources in such computational environments is irregular, so the load may be unpredictable.

In real world systems, there is a fundamental difference in behaviour between dedicated systems, where the parallel system is dedicated to execute the scheduled tasks, and non-dedicated systems, where multiple tasks can share the resources, thus the system may have a volatile loads. In a dedicated system, we may derive good estimates of the time that jobs take. In contrast, in a shared system, task execution and impact on system resources are highly unpredictable. However, on shared distributed multi-processor systems there may be substantial differences in execution time for a given program depending on what other programs are running concomitantly, so the performance maybe unpredictable.

We are exploring a mechanism to generate CPU loads to degrade system performance on heterogeneous architectures and control the resource usage. In this paper, we discuss the implementation of a novel load function which may be instantiated to apply predictable patterns of load in a dedicated system to simulate different load scenarios that may occur in a shared distributed non-dedicated system.

2. BACKGROUND AND RELATED WORK

Heterogeneous architecture software needs to be tested and validated, but resource usage depends on the computing demands from other user processes. These unpredictable loads from other users make the

environment unpredictable. Thus, the experimental environment for such software needs to be adaptable to reflect changing conditions. Some simulation tools, for example SimGrid (Casanova et al, 2008), can be used to explore varying loads but simulating the behaviour of such complex systems is very difficult and may be impossible in some situations (Cappello et al, 2006). For better results, it is more efficient if these experiments run on real environments under controlled conditions. Thus, a load generator is needed to mimic such conditions by producing a desired amount of load across the environment. Ideally, the load generator would produce defined levels of load on CPU, memory, cache and network.

For example, KRASH (Perarnau et al, 2010) is a tool for reproducible generation of system-level CPU load on many-core machines. It creates a dynamic and precise load but only for multicore systems. Stress (Waterland, 2012) is a workload generator for stressing the CPU, memory, I/O and the disk. This tool spawns a fixed number of processes with some calculations for stressing the CPU. The load generated using this tool is non-dynamic and non-accurate. Another method to reduce the CPU performance is down-scaling the CPU frequency which is used to reduce CPU power consumption (Makineni et al, 2003). Moreover, cpulimit (Marletta, 2012) is a tool to limit the CPU usage of a process. It controls the CPU time dynamically and quickly without changing the scheduling settings but it does not handle multicore systems. Wrekavoc (Dubuisson et al, 2009) is a tool for heterogeneity simulation which enables users to limit the resources available to their application.

Our load function has been constructed to generate CPU load, as the CPU is a significant element in high performance computing. This function is able to generate a dynamic, precise, thoughtful and systematic load on shared/distributed memory architecture. Hence, we can prepare and replicate real experimental conditions by applying various patterns of loads.

3. DESIGN AND IMPLEMENTATION

3.1 Load and scheduling

Typical parallel applications are composed of many processes or threads (Herlihy et al, 2008). These threads use the CPU cores which are the smallest computing elements in a computing system. The operating system scheduler assigns a CPU core to threads. These threads are competing for accessing the core at the same time. Then, the scheduler has to choose which thread should run on the core using scheduling policy. The schedulers try to balance fairly resource usage amongst running threads. Therefore, the scheduler will use time-slicing by assigning time intervals of the core execution to all threads intend to run on the core where the time intervals assigned to the threads depends on their priorities and the scheduler policy. We can conclude that the core load is the ratio of unavailable time slices to the total time slices.

The load function creates threads which are scheduled on a regular basis. The time slices assigned to these threads depend on the amount of load in the load pattern.

CPU load profiling is a method to measure the system load. The load function is able to measure and record the load for the whole system, nodes and cores, where it can use the load pattern later for mimicking the whole system. An alternative approach is to instrument a model of the system (Lublin et al, 2003).

3.2 Load Function Design

The load function is designed using a master/worker model: see Fig 1. Here, the master is responsible for managing and controlling the workers which are distributed over all the nodes in the system, as localised measurement entities and load generators. We will name the load generator thread the *loader*, the worker thread the *local controller* and the master the *global controller*. Generating load on a CPU means making it unavailable for processing other work. In other words, generating the CPU load involves creating and running threads/processes on the CPU cores. A loader is an intensive thread which runs on one CPU core and has to be controlled to adjust the amount of load, either by the thread itself or by another thread, the local controller. This thread will run frequently to monitor and manage the loader threads with no change in their priorities. However, the frequency of running the local controller thread should be balanced to avoid extra

load on the CPU and to give precise load control. Our function cooperates with the scheduler and runs with the regular policy without changing any priorities.

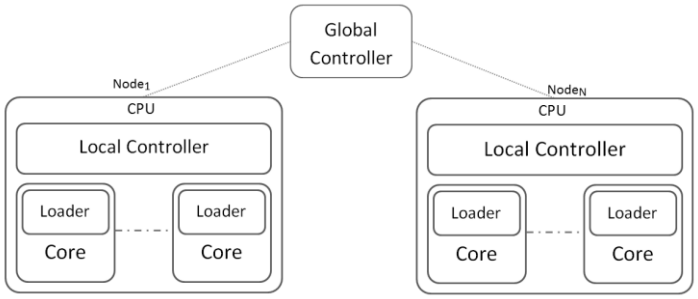


Figure 1. Load function design

The load function will run on multiprocessor systems. When the global controller runs, it will create a local controller on each machine. Thereafter, the local controller will create a loader thread for each core in the node. After that, it will assign a core to a thread to guarantee that the thread is running only on one core. Depending on the load pattern and for generating a precise load, the local controller will check frequently each loader to make sure that it is loading the core with the desired amount of load.

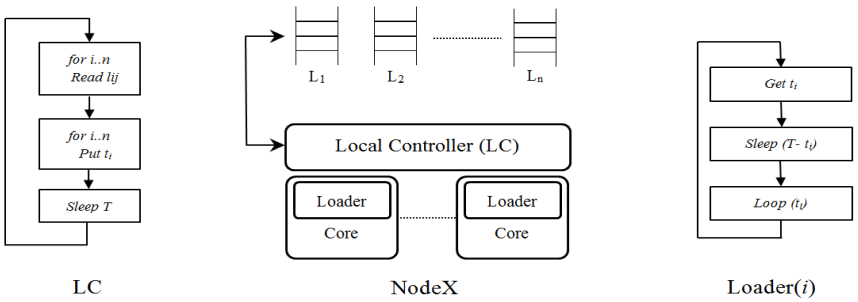
As we observed above, the core load is the ratio of unavailable time slices to the total time slices in a period of time. So that, applying a load on a core means making some time slices on the core unavailable. Regarding the dynamic load, the local controller will change the number of unavailable time slices in the core depending on the load profile. In contrast, for adaptive load, the local controller will take into account the current external real-time load changes and generate the remaining amount of load to reach the desired load.

A variant of load injectors uses a supervisor model, for example KRASH (Perarnau et al, 2010) and Wrecavoc (Dubuisson et al, 2009).

3.3 The Implementation

We implement our load function using C+MPI (Snir, 1998) while we use the POSIX library to create and manage the threads (Nichols et al, 1996). In this implementation, we use some features of the Linux kernel (2.6 or later). Here, the function has two main tasks: recording the loads of the machines and generating CPU loads.

To record the load, a monitor thread is created to record all information about the machine using the `/proc` virtual file system. The information is collected every second by default, or according to input



L_i: The load pattern for a core.
n: Number of cores.
l_{ij}: The amount of load for the core i, at time j.
t_i: The amount of time to be spent at the core i.
T: The occurrence time.

Figure 2. Load function implementation

configurations. After that, the information will be sent to the global controller to create the load pattern for the current system.

To generate the CPU load, see Fig 2, the load function creates a local controller on each machine on the system, and the local controller creates a loader thread for each core. The global controller uses a data parallel model to divide the load pattern amongst the machines which in turn distribute the machine loads over the cores. A core should be assigned to a loader thread using `cpuset`, a Linux feature which can be used to restrict the thread execution on a specific core or cores (In POSIX, this is implemented in thread affinity)(Nichols et al, 1996). The loader thread is a simple infinite loop with conditions to keep the loader controlled by the local controller so that it has minimal impact on cache and memory. The local controller will run once per second by default, or according to input configurations. Each time, the local controller will adjust the amount of load for each loader depending on the load pattern. The local controller is not attached to a specific core so that it does not matter where it runs. If the scheduler is fair, the local controller will run on time. But under highly loaded conditions where all cores are at 100% utilization, some delay in running the local controller might happen, but this has no effect because the local controller sets only the sleep period.

4. EVALUATION

In general, the load function may change the loads of arbitrary processors across a cluster or Grid, according to the load pattern with which it is instantiated. Our function has been designed to meet the requirements:

- *Reproducibility*: The load function is able to generate the desired load on the system regardless of environmental conditions (number of machines, number of cores, other user processes…).
- *Precise*: It can generate a precise CPU load to mimic real environments which are subject to variant amount of load.
- *Dynamic and Adaptive*: Under external real-time changes, a required load pattern can be dynamically maintained, which makes the load function specifically appropriate to be used in experimentations on shared systems.
- *Over-loading*: The load function can create any number of loaders on a core which this makes that core intensively loaded.
- *Non Intervention*: The load function has minimal impact on the system because the scheduling policy is not affected and the priorities of the current processes are not changed.

The load function was tested with a Beowulf cluster located at Heriot-Watt University. The cluster consists of 32 eight-core machines where each is an 8 core Intel(R) Xeon(R) CPU E5504, running GNU/Linux at 2.00 GHz with 4096 kb L2 cache and using 12GB RAM.

In the first experiment, see Fig 3, we validate the reproducibility for a real environment by running LINPACK (Dongarra et al, 2003) benchmark over 4 nodes. Then, we run the load function in the record mode to observe the load of the system. After that, we reproduce the recorded load pattern on the same nodes. The average error between the load pattern and the generated load is 0.62 sec with a standard deviation 0.105 sec.

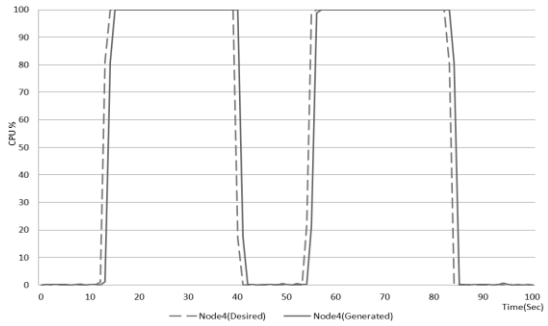


Figure 3. The required and the actual load in node4

To validate the dynamic and adaptive requirements, we propose a simple pattern of load. Here, during run-time the load function generates the load dynamically and adapts the load according to the changes which may happen during the experiment. The load function can run in either adaptive or non-adaptive mode. In adaptive mode the function will take into account the current system load while in non-adaptive mode the will generate the load regardless of the current load. In this experiment, we have 5 nodes and an external user which monopolises the first node for some time. In this case, the local controller in the first node will ask the loaders to reduce the artificial load to make the total load equal to the required load, see Fig 4.

The more precise load is generated, the better a real system is simulated. The local controller collects, generates and assigns the amount of load for a loader. Therefore, it is very important to run the local controller thread on time to have the required amount of load. This thread runs on any available core. There is no delay for executing the local controller if the load is less than 100% because there will be enough processing power to run it on an available core. See Table 1 which illustrates how the average error for generating varying amounts of load under 100%. We run the load function in adaptive and non-adaptive modes. Here, we notice that the average error in the adaptive mode is around 0.18 at the low loads while in the non-adaptive mode the average error is around 0.2 at the high loads.

Table 1. The precision of load generation by the load function

Loads	Adaptive Mode		Non-Adaptive Mode	
	Average Error	S-Deviation	Average Error	S-Deviation
1 %	0.151	0.149	0.099	0.123
2 %	0.156	0.119	0.062	0.083
5 %	0.168	0.143	0.116	0.080
10 %	0.184	0.085	0.068	0.049
25 %	0.080	0.046	0.053	0.053
50 %	0.075	0.063	0.088	0.031
75 %	0.081	0.041	0.120	0.043
90 %	0.051	0.074	0.157	0.052
98 %	0.099	0.050	0.175	0.061
99 %	0.038	0.058	0.181	0.063
100 %	0.056	0.207	0.204	0.173

If the load is more than 100%, the local controllers will compete for acquiring resource. An undesirable delay in executing the local controller will occur which affects only the next time interval. This delay depends on the total number of threads and the scheduling policy. The delay will not affect the generated load if the required load is over 100% but if the load pattern decreases the load then a slight error may appear.

The load function impact

When conducting an experiment, the load function runs at the same time to apply a load pattern for evaluating the solution. So it is important to ensure that running the function itself will not have a significant impact on overall system performance. To explore this, we use a matrix multiplication benchmark with the load function doing nothing.

As table 2 shows, we found that the effect on the system with a load function doing nothing is from 0.05% to 0.5%. We conclude that the load function has an insignificant impact on the overall performance of the system.

Table 2. The impact of the load function on the network traffic

Load	Nothing Running	00%	10 %	25 %	50%	75 %	100 %	150 %	200 %	250 %	300 %
Time(sec)	1.39	1.40	1.41	1.481	1.552	1.622	1.683	1.756	1.861	1.966	2.024

Because we are working across distributed memory architectures, it is also very important to check the function's impact on network performance. To investigate this, we use the NAS NBP DT benchmark (Bailey et al, 1991) which tests the data traffic over the network. We use 5 nodes from the cluster to run 80 processes from NAS DT with a shuffle communication graph alongside the load function generating different artificial loads.

Table 3 shows the minimal impact of the load function on the network performance.

Table 3. The impact of the load function on the system

	1000X1000	2000X2000	3000X3000	4000X4000	5000X5000
Time(Sec)	0.687	2.525	9.923	18.034	40.143
Time with load function	0.693	2.527	9.963	18.043	40.363
Percentage	0.873 %	0.079 %	0.403 %	0.05 %	0.548 %

Now, we explore the use of the load function in three parallel computing experiments. In this paper, we do not address the evaluation of these experiments themselves; rather we are evaluating tool use in very different contexts to control resource availability according to a load pattern.

4.1 Load Balancing

Load balancing attempts to balance the work load of all locations in a multicomputer (Casavant et al, 1988). In static load balancing, the work load is allocated at the start while in dynamic load balancing the work allocation depends on information collected from the workers. Thus, the behaviour and performance of an experiment in load balancing depends on the current load of the system. We implemented a matrix multiplication benchmark using the task/farm model in static and dynamic mode. In the static version (Shirazi et al, 1990) the tasks should be distributed evenly amongst all the workers. For dynamic load balancing, the distribution of tasks depends on the internal and the external load of all workers (Shivaratri et al, 1992). Then, we run both implementations for a 6000x6000 matrix with different task sizes over 5 nodes alongside with the load function with a load pattern illustrated in Fig 3 only for the first 3 nodes.

Fig 5 presents the results of running 100 tasks over 5 nodes. In this experiment, Fig 5 (A) and (C) show the change in task distribution between static and dynamic load balancing. Moreover, in Fig 5 (B) and (D) we can notice the effect of load on the execution time for both implementations.

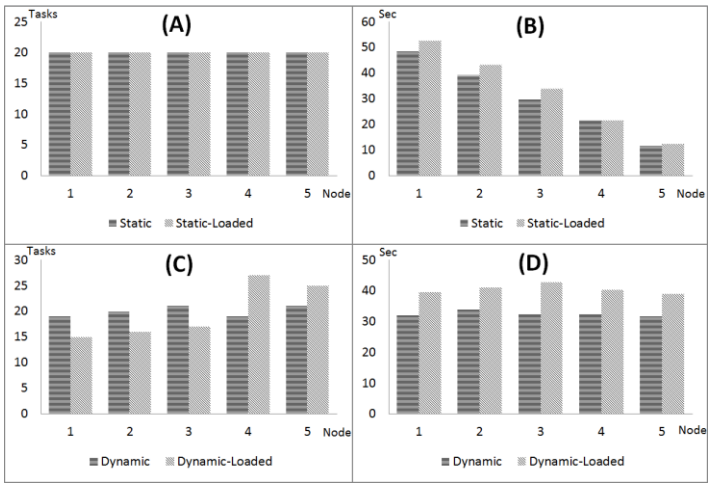


Figure 5. Load balancing (Static/Dynamic) under load changes

4.2 Work Stealing

Work stealing is a thread scheduling technique for shared-memory multiprocessors where a thread steals works from other threads (Blumofe et al, 1999). For this experiment, we use one node which has 8 cores. We run 8 threads over 8 cores where each thread has a pool of tasks and these pools are shared between all threads. We repeated the running 9 times with changing the number of unavailable cores.

Table 4 illustrates the effect of changing the load on task distribution. Here, the tasks should be evenly distributed amongst the cores if they have the same amount of load. Increases in load are uniform. In the table, bold numbers refer to the number of tasks processed on each loaded core. We can see that as more and more cores are loaded, the tasks are redistributed to maintain overall balance between loaded and unloaded cores.

Table 4. Work Stealing with the number of processed tasks on each core.

	Number of Loaded Cores								
Core	0	1	2	3	4	5	6	7	8
1	256	**166**	**158**	**146**	**128**	**147**	**171**	**205**	**256**
2	256	269	**160**	**149**	**128**	**146**	**171**	**205**	**256**
3	256	269	290	**149**	**128**	**146**	**170**	**205**	**256**
4	256	269	289	319	**128**	**147**	**171**	**206**	**256**
5	256	269	288	320	384	**146**	**171**	**205**	**256**
6	256	269	287	322	383	438	**171**	**205**	**255**
7	256	269	289	321	384	439	511	**206**	**257**
8	256	268	287	322	385	439	512	611	**256**

4.3 Mobility

We have been exploring a novel mobile skeleton (Alsalkini et al, 2012) whose individual workers may move their tasks dynamically across multi-processor systems. Such skeletons are self-mobile and able to mitigate the deleterious performance effects of external load on individual processors by dynamically moving tasks across processors.

In contrast, (Deng et al, 2010), have been exploring autonomous mobility skeletons where the whole skeleton moves. We contend that our approach gives much finer control of resource use.

Here, we consider a mobile skeleton for a ray tracer benchmark that generates the image for 100 rays for 150,000 objects in the scene. A ray tracing algorithm produces an image using imaginary rays of light from the viewer's eye through pixels in an image plane to the objects in the scene.

Fig 6 (A) shows the applied pattern of load while Fig 6 (B) gives the actual behaviour for moving the task between the workers. The decision of moving the task has been taken by the mobile skeleton which it mainly depends on the load on the current worker and the other workers.

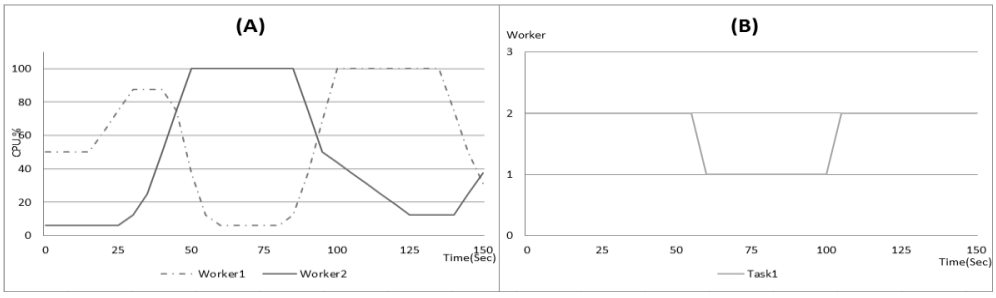


Figure 6. The load pattern applied to the ray tracer and its impact on moving tasks between workers.

5. CONCLUSION

We have presented a new tool which helps in evaluating experiments that depend on changes in the load in multi-processor and multi-core environments. This tool is implemented as a load function which we have shown to have minimal impact in an experimental setting. Overall, we can conclude that the load function is highly effective in a dedicated system for simulating patterns of load changes in a shared system.

We think that our load function is of far wider applicability. For example, it might be used in a homogeneous setting to simulate a heterogeneous environment by giving differential constant loads to the processing elements with the same characteristics. It might also be used to simulate different patterns of system component failure by giving processing elements infeasibly large loads.

ACKNOWLEDGEMENT

The authors would like to acknowledge Dr. Peter J.B. King for his support in reviewing this work. The load function is available for download at http://www.macs.hw.ac.uk/~ta160/load-function/.

REFERENCES

Alsalkini, T. and Michaelson, G., 2012. Dynamic Farm Skeleton Task Allocation through Task Mobility. *In: 18th International Conference on Parallel and Distributed Processing Techniques and Applications*. Las Vegas, USA, pp. 232-238.

Bailey, D.H. et al, 1991, The NAS parallel benchmarks. Tech. rep., RNR-91-200, *NAS Systems Division*.

Blumofe, R.D. and Leiserson, C.E., 1999, Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, Vol. 46, No. 5, pp 720-748.

Cappello, F. et al, 2006. Grid'5000: A Large Scale and Highly Reconfigurable Experimental Grid Testbed. *Int. J. High Perform. Comput. Appl.* Vol. 20, No. 4, pp 481-494.

Casanova, H. et al, 2008. SimGrid: A Generic Framework for Large-Scale Distributed Experiments. *In: Proceedings of the Tenth International Conference on Computer Modeling and Simulation.* Washington, DC, USA, pp. 126-131.

Casavant, T. and Kuhl, J., 1988. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on*, Vol. 14, No. 2, pp 141-154.

Deng, X.Y. et al, 2010. Cost-driven Autonomous Mobility. *Computer Languages, Systems and Structures*, Vol. 36, No. 1, pp 34-59.

Dongarra, J.J. et al, 2003. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, Vol. 15, No. 9, pp 803-820.

Dubuisson, O et al, 2009. Validating Wrekavoc: A Tool for Heterogeneity Emulation. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, pp 1-12.

Haight, F.A., 1967. *Handbook of the Poisson Distribution*. John Wiley & Sons, New York.

Herlihy, M. and Shavit, N., 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Lublin, U. and Feitelson, D.G., 2003. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput,*Vol. 63, No. 11, pp 1105-1122.

Makineni, S. and Ravi, I., 2003. Measurement-based analysis of TCP/IP processing requirements. *In: 10th International Conference on High Performance Computing (HiPC)*, Hyderabad, India.

Marletta, A. *cpulimit*. http://cpulimit.sourceforge.net/, May 2012.

Nichols, B. et al, 1996. *Pthreads Programming*, O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Perarnau, S. and Huard, G., 2010. KRASH: Reproducible CPU Load Generation on Many-Core Machines. *IEEE International Parallel and Distributed Processing Symposium (IPDPS),* Atlanta, Georgia, USA, *pp* 1-10.

Shirazi, B. et al, 1990. Analysis and evaluation of heuristic methods for static task scheduling. *Journal of Parallel and Distributed Computing*, Vol. 10, No. 3, pp 222-232.

Shivaratri, N. et al, 1992. Load distributing for locally distributed systems. *Computer*, Vol. 25, No. 12, pp 33-44.

Snir, M. et al, 1998. *MPI-The Complete Reference. Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA.

Waterland, A. *stress*. http://weather.ou.edu/ apw/projects/stress/, July 2012.