A Simple Load Balancing Scheme for Task Allocation in Parallel Machines

Larry Rudolph
Department of Computer Science
Hebrew University, Jerusalem, Israel
and currently visiting
IBM TJ Watson Research Center
Yorktown Heights, NY

Miriam Slivkin-Allalouf
John Bryce Ltd.
Science Based Industries
PO BOX 23838
Jerusalem, Israel

Eli Upfal
Department of Applied Mathematics
Weizman Institute, Rehovot, Israel
and
IBM Almaden Research Center
San Jose, Ca

Abstract

A collection of local workpiles (task queues) and a simple load balancing scheme is well suited for scheduling tasks in shared memory parallel machines. Task scheduling on such machines has usually been done through a single, globally accessible, workpile. The scheme introduced in this paper achieves a balancing comparable to that of a global workpile, while minimizing the overheads. In many parallel computer architectures, each processor has some memory that it can access more efficiently, and so it is desirable that tasks do not mirgrate frequently.

The load balancing is simple and distributed: Whenever a processor accesses its local workpile, it performs a balancing operation with probability inversely proportional to the size of its workpile. The balancing operation consists of examining the workpile of a random processor and exchanging tasks so as to equalize the size of the two workpiles. The probabilistic analysis of the performance of the load balancing scheme proves that each tasks in the system receives its fair share of computation time. Specifically, the expected size of each local task queue is within a small constant factor of the average, i.e. total number of tasks in the system divided by the number of processors.

1 Introduction

The scheduling of the activities of a parallel program on a parallel machine can significantly influence its performance. A poor scheduling policy may leave many processors idle while a clever one may consume an unduly large portion of the total CPU cycles. The main goal is to provide a distributed, low cost, scheme that balances the load across all the processors.

Scheduling schemes are only important when the system is allowed to schedule and map tasks to processors. Many users of today's parallel machines demand maximum performance and are therefore willing to invest an Herculerian effort in doing the mapping and scheduling themselves. On the otherhand, shared memory machines as well as a few message passing ones wish to relieve the user of this job and allow development of more portable code. We focus on shared memory, asynchronous parallel machines, but believe that our results are more widely applicable.

Many load balancing schemes for parallel processing have been studied, although most make different assumptions or require a detailed knowledge of either the architecture or the application. For example, when adaptive meshes are used in applications such as Particle-in-cell methods, finite difference and finite element calculations, load balancing schemes are concerned with how to move boundries to equalize the number of particles allocated to each processor ([DG89]). Here information is exchanged at natural synchronization points.

Independent of the application, some schemes try to find the global average load and the least and most loaded node in the system using a minimum of communication steps. The strategy is to query only a small number of neighbors at periodic intervals in the hope of keeping its own information up-to-date (e.g. [CK79],[BKW89],[JW89]). Sometimes a central server is considered ([BK90]). There have also been attempts to use simulated annealing ([FFKS89]).

Schemes are often directly related to the machine architectures. There have been many schemes proposed for hypercubes ([DG90],[DG89]) and either balance

Permission to copy without fee all or part of this matertial is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 089791-438-4/91/0007/0237 $1.50

only between local neighbors or recursively balance between adjacent subcubes.

Similar to our scheme, Hong et al. [HTC89] try to achieve global balance by equalizing the loads between pairs of processors. The balancing, however, occurs at fixed, predetermined intervals and they are interested in the effects of changing the interval times. Another approach that also uses random polling [KR89] has the processors poll only when they are idle.

In contrast, our work is concerned with tightly coupled shared memory parallel machines where the tasks may all be part of the same parallel program. We assume that communication between PEs is uniform and inexpensive although it appears that our results can also apply to restricted communication architectures, such as hypercubes. Moreover, we are able to prove a bound of the performance of the balancing scheme.

Parallel shared-memory machines designed in the past often provided a flat global memory and so there was a minimal cost in moving a task from one processor to another. The single global workpile is sufficient for these architectures (e.g. see [R87]). Each processor (PE) chooses the next task in the global workpile and executes it for a given time-quantum. If the task does not terminate within the time-quantum, it is returned to the end of the global workpile and the PE chooses the next task. Thus, unless the number of tasks matched the number of PEs, it was unlikely that task would be resumed on the same PE. The global workpile with time-slicing gives the appearance that there are many more PEs executing in parallel than may physically exist in the system. The work-load of the tasks is also evenly balanced among all the PEs and no PE will be idle while there is a task ready to be executed.

The global workpile scheme is not well suited for more advance architectures in which there is a more complex memory hierarchy and where the amount of local state is significant. Here, it is preferable for a task to resume its execution by the same processor that executed it previously. A local workpile associated with each PE will provide this continuity, but it suffers from a potential unequal distribution of the work (tasks) among the processors. One PE may have a very long workpile while another may have a very short or empty one. Unless the system employes some load balancing technique, tasks will then execute at different rates, inversely proportional to the average length of their workpiles.

We first state our assumptions and requirements for a parallel task system. Our load balancing scheme is then presented. Its behavior and performance is then analyzed and the results of simulations are presented. Finally, we show that there are cases in which a task allocation scheme based on local workpiles with load balancing performs better than one based on global workpiles.

2 Foundations

Each processing element consists of a CPU, registers, local memory for local computation and I/O ports which will enable it to work independently on a task. We consider the task as an indivisible entity, the smallest viable computational unit.

We assume a parallel processor that is executing a parallel program. A *task* is a piece of code that is to be executed, possibly in parallel with other tasks. We assume that tasks are generated dynamically by the processors and can be executed by any of the processors. All existing, ready-to-run tasks in the system are either being executed by some processor or are are maintained in a data structure that we call a *workpile*. We use this term instead of the traditional term of *task queue* since a strict FIFO order is not required. Tasks are inserted to and deleted from the workpile.

The workpile can be organized two different ways: as a single, global workpile accessible to all processors or as set of local workpiles, one associated with each processor. Of course hybred schemes are also possible although not considered in this paper. All tasks are assumed to have the same priority; multiple priority levels can be implemented with the schemes presented here used within each distinct priority level.

Since tasks may closely interact and since there may be more tasks than the number of processors, a preemptive scheduling strategy is required. That is, a processor executes a task for a period of time, usually called a *time-quantum*, or until the task voluntarily gives up control.

2.1 The Workpile as a Global Queue

When the task allocation scheme is based on a single global queue, all insertions and deletions of tasks are directed to the same data-structure. Each idle processor removes a task from the workpile and begins to process that task. The process executes the task until the task either terminates, suspends, or a quantum of time has elapsed. In the latter case, the task is returned to the global workpile for continued processing at a later time. The processor is then considered idle and thus chooses another task from the global workpile and the process continues. If the workpile is empty, the processor retries after a short idle period. Access to the global workpile is sequential in that only one processor may be adding or removing a task from the global workpile at any time. If multiple PEs simultaneously attempt access, they are serialized in an arbitrary fashion.

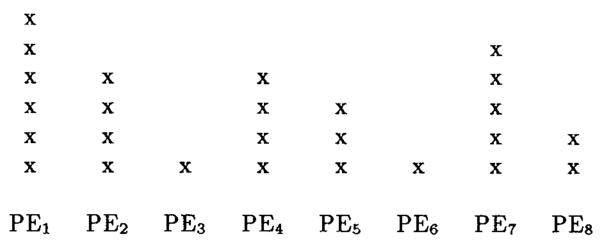

Figure 1: Each PE has a workpile associated with it. It inserts and deletes only to this workpile. A load balancing mechanism may move tasks from one workpile to another. PE$_i$ executes a load balancing operation with probability $1/l_i$ where l_i is the length of the i workpile.

This scheme appears to provide the best load balancing since it is never the case that a processor remains idle while there is a task ready to be executed. But this is true only in a very limited setting. First, since many processors may concurrently access the global workpile, some of these access may have to be delayed. Second, if the number of tasks is only slightly larger than the number of processors, there will be too many needless context switches. Finally, it is highly unlikely that a task will be rescheduled on the same processor that previously executed it and if the processors have the ability to keep a significant amount of task state in their local memory then this migration will be expensive.

2.2 The Workpile As A Set Of Local Queues

An alternative to the global workpile is for each processor to maintain its own local workpile of tasks. Each processor chooses the next task only from its own local workpile (Figure 1). The local workpile is organized as a FIFO queue so that the tasks are processed in a round-robin fashion. If the local workpile is empty, the associated processor remains idle.

Newly created tasks are also placed into the local workpile of the processor executing the creation operation. This allows an optimization in certain situations. Many parallel languages, and or operating systems, allow a task to be spawned or created with an associated *multiplicity* count, m. In a single call, m distinct tasks are created, each one assigned a different index in the range 0 to $m-1$. The optimization consists of placing only a single templete into the workpile and as each task begins execution, the full task control block is created. If the tasks require only a small amount of CPU time, this optimization will be significant.

There are several advantages and disadvantages to the local workpile organization. The advantages are

```
Balancing Task for PE $i$
    lock workpile$_i$
    $r$ = random number in the range $[1,p]$ but not $i$
    lock workpile$_r$
    if | size of workpile$_i$ - size workpile$_r$ | $> \tau$
    then move task from the end of the longer
           to the end of the smaller so as to
           equalize their workpile sizes.
    unlock workpile$_i$ and workpile$_r$
```

Figure 2: The load balancing task

that tasks may be inserted and removed in parallel and tasks remain on the same processor that first executed them. On the otherhand, there may be many idle processors. Indeed, without any explicit *load balancing*, all the tasks might be on a single local workpile. Two types of load balancing are possible, initial allocation where newly created tasks are inserted onto any of the local workpiles and continuous balancing where tasks may be moved from one local workpile to another whenever they are not executing. Any combination of these two are possible; we focus on the latter case.

3 The Balancing Scheme for Local Workpiles

The most striking feature of our load balancing scheme is that it is adaptive, distributed, and very simple. Each processor makes its own local, independent decision as to when to balance and with whom. It is suprising that although no processor has global knowledge of the system load, the expected load on each node is about the average load thoughout the system.

The main problem related to the function of controlling and balancing the workpiles is the conflict between the desire to control and the unwillingness to have one single master processor. Also it is not desirable that each processor check the situation of the whole system during each time period. Rather the load-balancing is done dynamically with minimal interference.

Load balancing has traditionally been exectued in one of two different ways. In the *periodic solution* a time period is fixed throughout the system during which load balancing is performed. In the *when necessary* or *polling* solution, whenever a PE is idle it polls the other PEs to share in their work.

We propose using a different approach to scheduling the load balancing work. There is no set time during which a processor executes the load balancing task. Instead, the load or size of the local workpile dictates the frequency of its execution. Let $l_{i,t}$ be the number of tasks on the workpile associated with processor i at time t. Let t be the time at which processor i is accessing its scheduling code. At time t, before scheduling the next task from its local workpile, processor i flips a coin and executes the load balancing task with probability $1/l_{i,t}$. If $l_{i,t} = 0$, then the processor delays a certain amount before executing the balancing task once again. In our implementation, we use an exponential backoff scheme in this case.

A processor with many tasks in its local workpile will thus execute the load-balancing task infrequently while one with a short workpile will frequently try to load-balance. The fraction of time that a processor will invest in load balancing is inversly proportional to its own workload.

The load-balancing task simply chooses some other PE at random and tries to equalize the load between the two workpiles (see Figure 2). If the difference in load between the two workpiles is greater than some lower limit, tasks are then migrated from the heavier loaded workpile to the lighter one. If the other workpile is currently being accessed, then either the PE may give up or else wait until the workpile becomes free. Our implementations suggest that there is little difference between these strategies.

There are several implementation details that require further explanation:

- The simplist way of choosing another local workpile with which to balance is by choosing one at random. Each processor begins with a unique seed for its random number generator.
- Concurrent access to a local workpile is not allowed.
- Local workpiles are accessed in FIFO order.
- The load of a workpile is measured as the number of tasks on the workpile.
- A system-wide threshold value, τ, is fixed such that only task workpiles whose length differs by more than τ will perform a balancing operation.
- A balancing operation consists of moving tasks from one workpile to another in order to equalize their lengths. Tasks from the end of the longer workpile are moved to the end of the shorter workpile.
- No starvation occurs. Since workpiles are accessed in FIFO order a task the does not migrate will eventually be executed. If it is migrated, it is moved to a position closer to the head of the queue.

In summary, in a heavily loaded system, there is little load balancing and in a lightly loaded system, load balancing is frequently attempted. This also holds true for each individual processor. Furthermore, since the sizes of the workpiles are roughly equal, there is very little task migration. It is also very rare that some processor will attempt to access a workpile will some other processor is currently accessing it.

4 Results

Since there is no single, generally accepted performance criteria for task scheduling, we present several different analyses of the behavior of our scheme. Of prime importance is how well does our scheme perform in a real system with real workloads. Many factors, such as memory access time, network traffic, type of application, and low level implementation details influence the performance. We have therefor measured and compared the performance of our scheme and the global workpile one using a simulation system of a parallel computer with sample application programs. The results are reported in the first subsection. Simulations are not sufficent, however, to prove the general applicability of our result. The second subsection, therefore, presents analytical results proving that each tasks in the system receives computation time that is proportional to its size in the entire system, that is, p/T, where p is the number of processors and T is the total number of active tasks in the system. The final subsection explains the effect of the scheduling and load balancing and presents an example where the decentralized scheme outperforms the centralized one.

4.1 Simulation Results

For two different application programs, we present the results of a simulation system of an ideal parallel computer. The simulation system makes many simplifications:

1. there is no difference in the costs of memory accesses.
2. no overhead in moving a thread from one processor to another.
3. no overhead in updating cache or page table entries.
4. no contention on the global workpile and insertion/deletion takes no time.
5. load balancing takes zero time.

All these assuptions, save the latter, favor the global workpile scheme since such overheads are higher in the global workpile scheme than in any other scheme.

We wish to argue that even without these benefits, the decentralized workpile scheme with adaptive load-balancing is competitive with the global workpile scheme. There are, of course, many situations that favor the centralized scheme. Thus, the ultimate choice depends on how costly are these measures on a particular machine.

We present the results of two different applications and four different schemes. The applications are a parallel version of quicksort and a master/slave skeleton program. In the first, a random splitting element is choosen, the array is subdivided into those elements larger and those smaller than the splitting element. The sort is recursively and in parallel applied to these two sets. The second application has a master process executing and then spawns off a number of slave tasks, 16 of them in our case, to work in parallel with the master. The slaves run for a short time. Somewhat later, the master once again spawns off a set of slave tasks. The process continues a number of times, 16 in our case. In both cases, the problem size is fixed and we vary the number of processors and measure the time to completetion.

Figures 5, and 5 summarize the results of the simulation that focuses only on the load-balancing features. There are two points to note. First, the scheduling schemes affect the performace even when one ignores the overheads of migration and concurrent access to the global workpile. This is important since it gives credence to the simulations and the same applications. Second, when one considers just the balance, the local workpile scheme gives a balance which is as good as the global workpile.

We also examined what happends when the first two simplifications are dropped. That is, we assumed that memory references can be either local, shared, or remote. We present results when these values are 1, 2, and 3, respectively. In the global workpile situation, all memory references are charged as shared. In the local workpile situation, memory references are either local or remote. Without loab balancing, tass are initially allocated to random processors and never moved. Their memory accesses always are charged as local. When there is migration, a task is first charged at the remote rate and after a while, it reverts to a local rate. We hope to capture the fact that initially, there is a high cost but after some time, the state can be transfered to the local memory. Figure 5 presents these results. We can see that the global workpile case is always twice as expensive as the others. This implies that there is not too much migration. It is also interesting to note that without load balancing, performace is as bad as the global case. The unbalanced work load wastes about half the processing power.

4.2 Analysis

We prove that the load balancing scheme indeed provides a good balance. The expected length of local workpiles is within a small constant factor times the average length. The key observation is that if the system starts in balanced, then it will stay that way. Of course this result depends on some reasonable assumptions concerning the creation and termination of tasks.

Denote by $L_{p,t}$ the load (= number of tasks) processor p has at the start of step t, and by $\Delta_{p,t}$ its change to its own load at time t ignoring load balancing changes, i.e. the number of new tasks processor p generates at step t minus the number of tasks terminated at processor p at this step. $\Delta_{p,t}$ might have an arbitrary distribution, but we require that $|\Delta_{p,t}|$ is bounded by some constant δ.

Denote by $L_t = \sum_{p \in V} L_{p,t}$ the total load of the system at time t, and by $A_t = \frac{L_t}{n}$ the average load at time t; there are n processors. Let $P_{p,t}$ denote the probability that processor p initiates a load balancing procedure at time t. We prove that if the system starts balanced, and if $P_{p,t} \geq \frac{\theta}{L_{p,t}}$, for some constant $\theta > 1$, then the system achieves optimal load balancing up to a constant factor. In other words, the expected waiting time of processes in workpiles of different processors differ only by a constant factor.

Theorem 1: *There are constants α, and C, independent of the number of processors n, and the distribution of $\Delta_{p,t}$ (the schedule of tasks to the processors), such that when the system starts balanced, and the load balancing algorithm is executed, for each processor p, and at each step t,*

$$E[L_{p,t}] \leq \alpha A_t + C.$$

proof: To simplify the analysis we consider a weaker load balancing scheme in which $P_{p,t}$ (the probability that processor p initiates a load balancing procedure at time t) is always bounded by $\min[\frac{\mu}{A_t}, \frac{1}{2}]$ for some $\mu > 0$. Thus, we restrict the activity of the very lightly loaded processors. We also assume a very weak conflict arbitration scheme: If processor p is initiating a load balancing procedure at a given step, it ignores messages from other processors initiating load balancing at this step. Furthermore, if processor p did not initiate a load balancing procedure but was chosen by more than one processor, then p does not balance its load with any processor. The performance of the original algorithm clearly dominates the performance of this algorithm.

We prove by induction on t that $E[L_{p,t}] \leq \alpha A_t + C$. The theorem assumes that the system started balanced, thus the induction hypothesis holds for $t = 0$. Assuming that the claim holds for t we bound $E[L_{p,t+1}]$.

$E[L_{p,t+1}] = E[L_{t,p}] + E[\Delta_{p,t}] + E[C_1] - E[C_2]$, where C_1 gives the contribution to $L_{p,t+1}$ from load balancing with a processor chosen by p, and C_2 gives the load lost by a load balancing with a processor choosing p.

Clearly, p never receives more than half the load of the processor it chooses, the probability that p initiates a load-balancing procedure at step $t+1$ is bounded by $\frac{\mu}{A_t}$. If it initiates load balancing, the probability that it chooses processor x is $1/(n-1)$, and in that case it receives no more than $L_{x,t}/2$ tasks. Thus,

$$E[C_1] \leq E[\frac{\mu}{A_t}\frac{1}{n-1}\sum_{x \neq p}\frac{L_{x,t}}{2}] \leq \frac{\mu}{2}.$$

Bounding $E[C_2]$ is more complicated since we need a lower bound. Let M_t denote the set of processors with $L_{x,t} \leq 2A_t$. Clearly $|M_t| \geq n/2$. The probability that a processor $x \in M$ initiates load balancing is at least $\min[\frac{\theta}{2A_t}, \frac{\mu}{A_t}, \frac{1}{2}]$. The probability that processor x chooses p, p is not chosen by any other processor, and p is not initiating load balancing at this step is at least

$$(1 - \frac{\mu}{A_t})\frac{1}{n-1}\prod_{y \neq p,x}(1 - \frac{P_{y,t}}{n-1}) \geq \frac{1}{4(n-1)}.$$

Let $\gamma = \min[\theta/2, \mu]$, then

$$\begin{aligned} E[C_2] \quad &\geq \min \quad \left[E[\frac{1}{4(n-1)}\frac{\gamma}{A_t}\sum_{x \in M}\frac{L_{p,t} - L_{x,t}}{2}],\right. \\ &\qquad\qquad \left. E[\frac{1}{4(n-1)}\frac{1}{2}\sum_{x \in M}\frac{L_{p,t} - L_{x,t}}{2}]\right] \\ &\geq \min \quad \left[E[\frac{\gamma}{8A_t}(L_{p,t} - 2A_t)], E[\frac{(L_{p,t} - A_t)}{16}]\right]. \end{aligned}$$

Set $\alpha = 6+8\delta$, $\mu = \theta/2 = \alpha+1$, $C = 8\mu+16(\alpha+1)\delta$, and recall that $A_{t+1} \geq A_t - \delta$.

If $\frac{\gamma}{A_t} \leq \frac{1}{2}$ then

$$E[L_{p,t+1}] = E[L_{t,p}] + E[\Delta_{p,t}] + E[C_1] - E[C_2]$$

$$\leq \alpha A_t + C + \delta + \mu/2 - \gamma(\alpha - 2)/8 \leq \alpha A_{t+1} + C.$$

If $\frac{\gamma}{A_t} > \frac{1}{2}$ then

$$E[L_{p,t+1}] = E[L_{t,p}] + E[\Delta_{p,t}] + E[C_1] - E[C_2]$$

$$\leq \alpha A_t + C + \delta + \mu/2 - C/16 \leq \alpha A_{t+1} + C.$$

□

We measured how much the length of the task queues differed from the global average and plotted the resutls in Figure 5. We ran 10 master-slave applications together and measured how the local workpile lengths differed from the global average. At each time step, we computed the global average, A_t, and summed the square of the differences of the queue lengths from A_t. We averaged this value and then took the average over alltime steps. That is,

$$\frac{1}{T}\sum_{t \in T}\frac{1}{n}\sum_{i \in n}(L_{i,t} - A_t)^2$$

We see that the load balancing keeps the size of the workpiles very near the average. This value ranged between 2 and 3.

4.3 FIFO Not Always Best

We present another example when the decentralized scheme provides better scheduling than the centralized one. It is interesting since at first blush, the global queue seems to provide the best possible balance. We use the term queue in place of workpile in order to emphasize its FIFO nature. Let us assume p PEs and $p+1$ tasks. One of the tasks is much longer than the other and it takes h time quantum units to execute it. The other tasks are equal sized - each requiring s time quantum units, where $s << h$. First, consider the global task queue scheme: The time until all the short tasks complete requires $((p+1)/p) * s$ time quantum units. The remainder part of the long task has $h - s$ units of time. Thus, the whole execution takes

$$h - s + ((p+1)/p) * s = h + s/p$$

units.

Next, consider the local task queue scheme with load balancing and a threshold $\tau = 1$. There are two diffent possiblities: The long task can be inserted into a queue with another short task (this happens with a probability of $1/p$) or the long task is alone and some other task queue has two short tasks. If the long task is alone in the queue the execution time is h. Otherwise it takes $h+s/2$. Thus, the whole execution is expected to require

$$1/p(h + s/2) + ((p-1)/p)h = h + s/(2p).$$

Since $s/(2p) < s/p$ the global task queue scheme does not provide the best scheduling.

5 Conclusions

We believe that the local workpile with load balancing is ideally suited for shared memory parallel processors. It gives the programmer the ability to program at a higher level of abstraction since he does not need to know the exact number of processors available. Moreover, the load balancing scheme adapts to the current load on the system.

It is interesting to consider how our scheme performs when faced with the popular paridigm of "loosly synchronous" programs and a multiprogrammed environment. Here, each program consists of a set of tasks that execute in a loosely synchronous fashion. That is, each task executed for a period of time and then participates in a communication or information exchange step with all the other tasks in the program. It is expected that the execution periods of the tasks are about equal. Now suppose there are several of these programs running concurrently. If there are enough processors for all the tasks, then our scheme will quickly approach that mapping. If the total number of tasks is much larger than the number of processors, then our scheme will cluster together tasks belonging to the same program since there tasks were initially placed on the same task queue.

Our experimental results showed that there is hardly any difference between the two schemes in terms of the distribution of work among the processors. Thus, architectures that encourage local state, would be better served with the local workpile scheme.

The key insight in the analysis is to demand that the system begin in a balanced state. It can be shown that the random selection is necessary to get our result. It each processor had chosen to balance only with a small set of neighbors, the it is possible to form little mountains or heaps with the peak located at a processor that continues to generate new tasks and the size of the workpiles decreasing as one gets further from the peak processor.

One of the weaknesses of the scheme is that the load-balancing tasks are executed with the same frequency whether the system is balanced or not. A possible improvement is in the dynamically setting the threshold value.

The scheme should be investigated for non-fully connected topologies such as the hypercube. It is our belief that it is nevertheless worthwhile to treat the hypercube as a completely interconnected network and to simply pay the extra cost when sampling or moving tasks between non-adjacent processors.

Another extention to this work involves treating newly created tasks differently from already executing ones. If is cheaper to move a task before it has executed. One should move such tasks first and only when that does not provided a good enough balance should other tasks be moved. Of course both types of migration can be based on the same load balancing scheme, only using different probabilities.

References

[BKW89] Baumartner, K., R. Kling, and B. Wah, "*Implementation of GAMMON: an efficient load balancing strategy for a local computer system,*" *Proceedings of the International Conference on Parallel Processing*, **Vol 2**, pp. 77-80, Aug. 1989.

[BK90] Bonomi, F. and A. Kumar, "*Adaptive optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler,*" *IEEE Transactions on Computers*, **Vol. 39**, pp. 1232-50, Oct. 1990.

[CK79] Y.C. Chow and W. Kohler, "*Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System,*" *IEEE Transactions on Computers*, **Vol. C-28**, pp. 334-361, May 1979.

[DG90] Dehne, F. and M. Gastaldo, "*A note on the load balancing problem for coarse grained hypercube dictionary machines,*" *Parallel Computing*, **Vol 16**, pp. 75-79, Nov 1990.

[DG89] Dragon, K, and J. Gustafson, "*A low cost hypercube load balancing algorithm,*" *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, **Vol 1**, pp. 583-589, March 1989

[FFKS89] Fox, G., W. Furmanski, J. Koller, and P. Simic, "*Physical optimization and load balancing algorithms,*" *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, **Vol 1**, pp. 591-594, March 1989

[HTC89] Hong, J., X. Tan, and M. Chen, "*Dynamic cyclic load balancing on hypercubes,*" *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, **Vol 1**, pp. 595-598, March 1989

[JW89] Juang, J. and B. Wah, "*Load balancing and ordered selection in a computer system with multiple contention buses,*" *Journal of Parallel and Distributed Computing*, **Vol 7**, pp 391-415, Dec. 1989.

[KR89] Kumar, V, and V. Rao, "*Load balancing on the hypercube architecture,*" *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, **Vol 1**, pp. 603-608, March 1989

[K89] Koller, J., "*The MOOS II Operating System and Dynamic Load Balancing*" *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, **Vol 1**, pp 599-602, March 1989

[R87] Raetz, G. "*Sequent general purpose parallel processing system,*" *Northcon/87*, pp. 7/2/1-5, Sept. 1987.

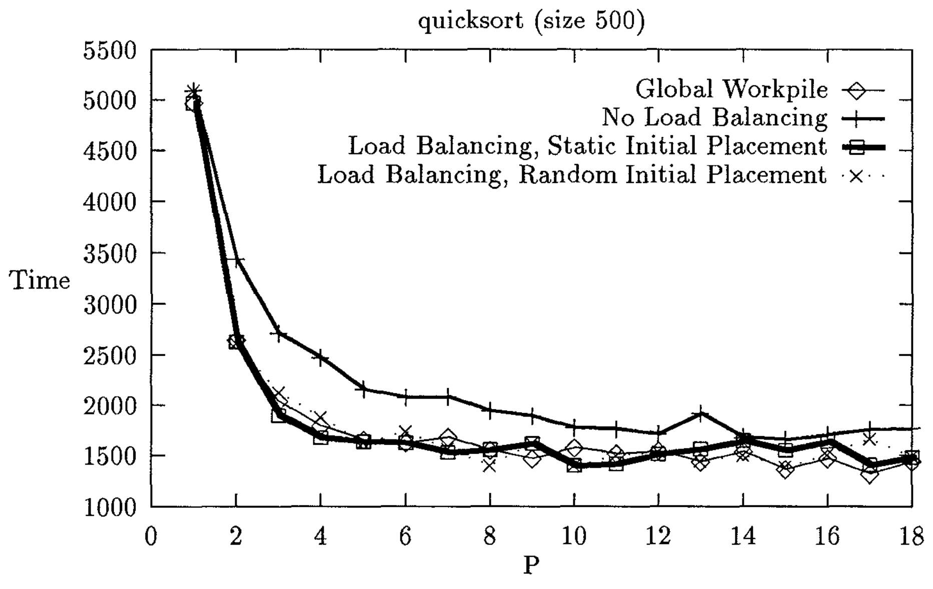

Figure 3: Quicksort application of size of 500. The scheduling strategy clearly affects the performance and there are times when the global workpile does not yield the best performance. Even without the local memory access advantages, we see that the load balancing scheme yields good performance.

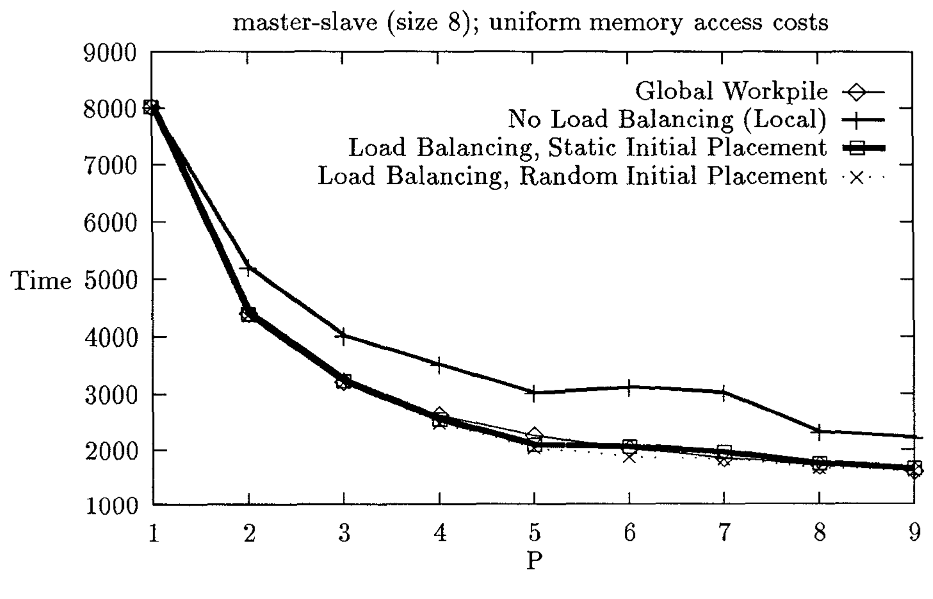

Figure 4: This data shows that when memory access costs are the same and the number of tasks is a little more than the number of processors, then our load balancing gives better results than then global workpile.

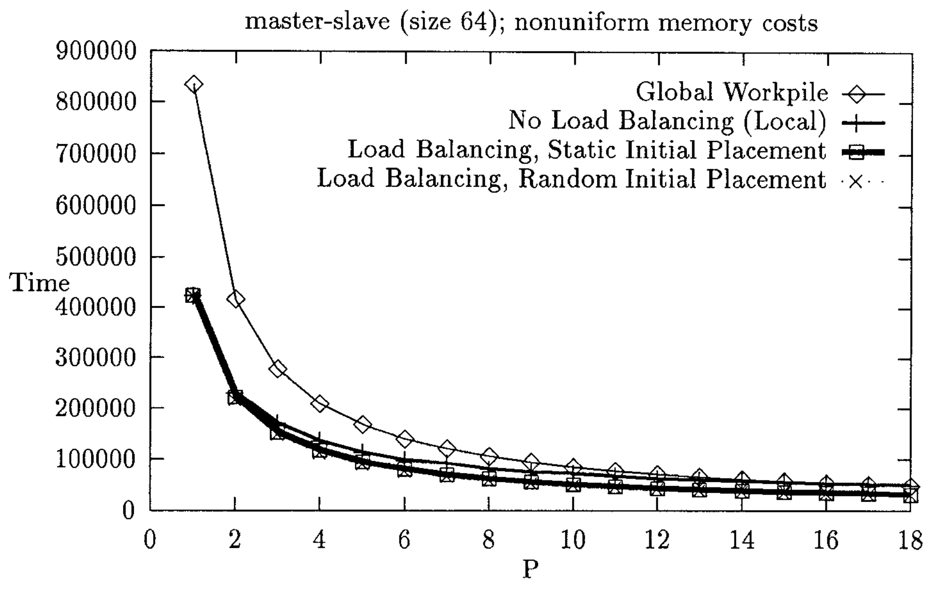

Figure 5: A memory access to local memory is charged 1 unit and to remote meory is charged 3 units. Shared memory access are charged 2 units. After a task migrates, its 10 access are charged at the highest rate of 3 units. Each task executes a total of 64 accesses. In this application, the master task spawns 64 slave tasks. After all these slaves finish, another batch is started. The results shows that the global workpile is about twice as bad due to the fact that each memory access time is twice as expensive. Thus we see that very few of the accesses are remote. Without load balancing performance degrades due to the poor scheduling even though all the accesses are charged 1 unit.

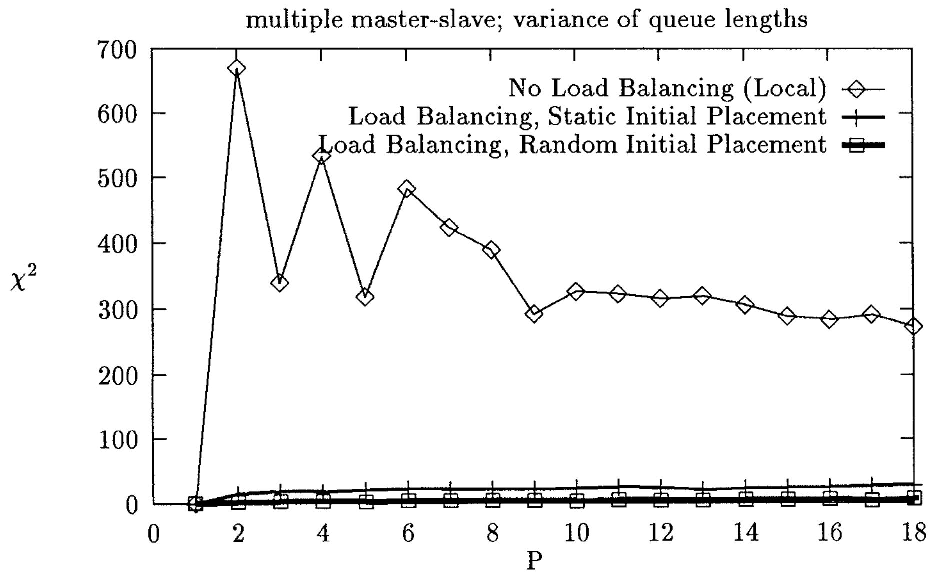

Figure 6: We ran 10 master-slave applications together and measured how the local workpile lengths differed from the global average. We plotted the following $\frac{1}{T}\sum_{t\in T}\frac{1}{n}\sum_{i\in n}(L_{i,t}-A_t)^2$, where T ranges over all time values, n is the number of processors, $L_{i,t}$ is the length of workpile at processor i at time t, and A_t is the average number of tasks in the system at time t. We see that the load balancing keeps the size of the workpiles very near the average.