June 1989

UILU-ENG-89-2219
ACT-107

COORDINATED SCIENCE LABORATORY
College of Engineering
Applied Computation Theory

LOAD BALANCING IN MULTIPROCESSOR SYSTEMS

Michael C. Loui
Milind A. Sohoni

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-89-2219 (ACT #107)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ONR N00014-85-K-0570
8c. ADDRESS (City, State, and ZIP Code) 800 N. Quincy Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Load Balancing in Multiprocessor Systems

12. PERSONAL AUTHOR(S)
Loui, Michael C. and Sohoni, Milind A.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	June, 1989	10

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	load balancing, scheduling, multiprocessor, shared memory

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We present an algorithm for dynamic load balancing in a multiprocessor system that minimizes the number of accesses to the shared memory. The algorithm assumes no information, probabilistic or otherwise, regarding task arrivals or processing requirements. For k processors to process n tasks, the algorithm incurs $O(k \log k \log n)$ potential memory collisions in the worst case. The algorithm itself is a simple variation of the strategy of visiting the longest queue. The key idea is to delay reporting task arrivals and completions, where the delay is a function of dynamic loading conditions.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Load Balancing in Multiprocessor Systems

Michael C. Loui

Milind A. Sohoni

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue, Urbana, IL 61801

June 5, 1989

Abstract

We present an algorithm for dynamic load balancing in a multiprocessor system that minimizes the number of accesses to the shared memory. The algorithm assumes no information, probabilistic or otherwise, regarding task arrivals or processing requirements. For k processors to process n tasks, the algorithm incurs $O(k \log k \log n)$ potential memory collisions in the worst case. The algorithm itself is a simple variation of the strategy of visiting the longest queue. The key idea is to delay reporting task arrivals and completions, where the delay is a function of dynamic loading conditions.

Supported by the Office of Naval Research under Contract N00014-85-K-0570.

Introduction

This paper studies load balancing in a multiprocessor environment. We consider a shared memory model similar to the one described in [3]. Briefly, there are k processors, each with unlimited individual memory, and a segment of shared memory. Tasks arrive, or are generated by ongoing tasks, at unpredictable instants, and their computation times are also unpredictable. We assume all tasks are independent of each other and may be assigned in any order to any processor. The k processors must perform these tasks promptly, and at the same time, with minimum interference from each other. Any load balancing algorithm must keep track of the loading conditions of the processors and assign a suitable number of available tasks whenever a processor becomes free. This dynamic information is maintained in the shared memory. We assume, as in [3], that all processors execute the same load balancing algorithm and access the shared memory asynchronously. This leads to the possibility of interference or *collisions*. The goal of our algorithm is to minimise this interference, while keeping all processors busy. As in [3], we assume that the number of potential collisions is the sole performance criterion.

Here, we present an algorithm that performs dynamic task scheduling. For k processors to process n tasks, our algorithm incurs $O(k \log k \log n)$ collisions in the worst case. This is an improvement on algorithm in [3], which incurs $O(k^{\varepsilon} \log n)$ $(\varepsilon>1)$ collisions in the worst case.

The Problem

One approach to the problem is to retain all tasks (or their descriptors) in the shared memory. Each processor is assigned only one task, and on its completion the processor accesses the shared memory to acquire another task. There are two disadvantages to this strategy. First, if a task generates a new task, then the new task must be "reported" or "put" into the shared memory. Second, at the completion of a task each processor must access the shared memory to obtain a new task. Both events can lead to collisions. If there were n tasks to be processed, there could be $2n$ (i.e., $O(n)$) collisions. Hence when reassigning tasks to processors one must assign not one but a certain number depending on the dynamic conditions. But if several tasks are being assigned to processors, then idle processors must not only look in the shared memory but also examine the loads at other processors. Here, by the *load* of a processor we mean the number of tasks assigned to it. Idle processors must then "visit" overloaded processors and relieve some of their load. This "visit", as it is interference, is also counted as a collision. Thus in our worst case analysis, the possibility of interference or collision is termed as a *collision*. >From our analysis above, any scheduling algorithm must handle two issues:

(i) *Once a processor finds itself idle, it must arrange to "visit" another processor for sharing its load. Thus there must be a policy to decide which processor to visit.*

(ii) *Once processor* p_i decides to visit processor p_j, efficient data structures should ensure that load sharing is done speedily. Processor p_i must also decide what fraction of processor p_j's load it should take.

We briefly sketch Manber's algorithm here. We refer the reader to [3] for a complete discussion of the above. Our algorithm differs from Manber's only in its way of handling (i) above.

Each processor p_i has an individual area in the shared memory, called its *local segment*, in which p_i stores the tasks (or their descriptors) assigned to p_i. The data structure here is similar to a binary tree which allows addition, deletion, and splitting in constant (i.e., $O(1)$) time. The *split* operation splits the tree into two trees whose sizes are between 1/3 and 2/3 of the original. Whenever a processor finishes its current task, it accesses its local segment first. If the local segment is not empty, then the processor picks up a task from its local segment and deletes the task (or the task's descriptor) from its local segment. If the currently executing task generates more tasks, or if some external tasks arrive, then the processor puts the new tasks into its local segment. When a processor finds its local segment empty, however, it proceeds to the global memory to find a processor that has a non-empty local segment. This local segment is split, and the visitor takes a part of the host's local segment into its own local segment. We say that some of the host's tasks have *migrated* to the visitor.

Processor p_i is *busy* if it is processing a task or its local segment is non-empty; otherwise p_i is *idle*. A global data structure maintains the status (i.e., busy or idle) of each processor. The data structure of [3] is organised as an m-ary tree with each of the k processors as a leaf. Each internal node maintains the loading conditions of its leaves. Then finding a busy processor takes $O(k^{\varepsilon})$ collisions (see [3] for details). The number ε approaches 1 as m increases. Thus for k processors to finish n tasks, this scheme incurs $O(k^{\varepsilon} \log n)$ collisions in the worst case. Also, Manber established a lower bound of $\Omega(k \log n)$ collisions.

Our global data structure, which resides in the shared memory, is a Fibonacci heap [2]. As an abstract data structure on k objects $\{x_i\}$ with real weights $\{w_i\}$, a Fibonacci heap supports the following operations:

max: find the maximum weighted object.
incr(x, r): increment the weight of x by positive real r.
decr(x, r): decrement the weight of x by positive real r.

The *max* and *decr* operations take $O(1)$ (amortized) time, and the *incr* operation takes $O(\log k)$ time. Hence each operation involves $O(\log k)$ memory accesses. We use a Fibonacci heap F to maintain the processor loading information. Each processor p_i is an object, and its reported load R_i is the weight associated with p_i.

For ease of analysis, we first present a simple version of our algorithm. Once again, our algorithm differs from [3] only in how it finds a processor to visit.

Each processor is individually responsible for updating its loading information in F. To minimize updating collisions when two processors try to access F at the same time, however, the processors report only increases in their previously reported loads. Thus at any time, the *reported load is always an overestimate of the actual load.* When processor p_i completes a task, it first checks its local segment. If that is empty, then p_i accesses F and performs the operation *max* to obtain the name of the processor p_j whose reported load is the largest. After p_i visits p_j, processor p_i has between one-third and two-thirds of p_j's old load and p_j has the remainder. Processor p_i uses *incr* and *decr* to record the new loads of p_i and p_j in the F. Note that though visiting the most heavily loaded processor seems to be a good policy for minimising future visits, it entails maintaining fairly accurate loading conditions in the global memory. This in turn requires processors to *report* their loads, which is collision-prone. So then there are two types of collisions: *reporting* collisions and *visiting* collisions. We estimate them separately. Note that when a processor completes a task and deletes the task from its local segment, the access to the local segment is not counted as a potential collision.

Visiting Collisions

We begin with estimating visiting collisions in our scheme. First, a "static" result. Assume that there are n tasks, and these tasks are distributed over the local segments of the k processors. Further, no new tasks are generated within, or added to, the system, so *no reporting collisions occur.*

Theorem 1. In the static case, if there are n tasks in a k-processor system, then in the worst case the processors complete all tasks with $O(k \log k \log n)$ collisions.

Proof. First note that every operation on F takes $O(\log k)$ time, hence $O(\log k)$ collisions, in the worst case. We consider each such operation as a unit *step* and proceed to estimate the number of steps.

By convention, the computation begins at step 0. Let

$R_i(t)$= the reported load of processor i at step t.

$L_i(t)$= the actual load of processor i at step t.

$m(t)= \max_j \{R_j(t)\}$.

Thus $m(t)$ is the largest weight in F at step t. If processor p_i visits processor p_j at step t_0, then p_i takes between one-third and two-thirds of p_j's load. In our notation,

$$R_i(t_0) = L_i(t_0) \leq \frac{2}{3} L_j(t_0-1) \leq \frac{2}{3} R_j(t_0-1),$$

$$R_j(t_0) = L_j(t_0) \leq \frac{2}{3} L_j(t_0-1) \leq \frac{2}{3} R_j(t_0-1).$$

Hence

$$R_i(t_0) \leq m(t_0-1) \text{ and } R_j(t_0) \leq m(t_0-1).$$

Now consider $m(t_0+k)$, i.e., the maximum reported load after k steps (k potential visiting collisions). We claim that $m(t_0+k) \leq \frac{2}{3} m(t_0)$. Observe that at step t_0 there can be at most k processors with reported load greater than $\frac{2}{3}\, m(t_0)$, and that the processor with the largest reported load is visited at each step. Hence in the k visits following t_0 all the processors with load greater than $\frac{2}{3} m(t_0)$ must be visited and our claim must hold. But $m(0) \leq n$, so the number of steps in the computation is bounded by $O(k \log n)$. □

In the static case above, there were no additional tasks or reporting collisions to complicate our analysis. Nevertheless, the bound on the visiting collisions holds even in the dynamic case when visiting collisions are interspersed with reporting collisions.

Theorem 2. Under dynamic conditions, in the worst case, k processors complete n tasks with at most $O(k \log k \log n)$ visiting collisions.

Proof. In the proof of Theorem 1, since no tasks were introduced into the system, $m(t)$ was a monotonic decreasing function. This is not true in the dynamic case. Let us assume that all the n tasks in the system at step 0 are *blue* in colour, and all tasks generated or added are *green*. So at any step in the computation each processor has a mix of blue and green tasks. Further, as a theoretical convenience, we assume that when p_i visits p_j, processor p_i takes equal proportions of blue and green tasks.

In addition to the previously defined functions $R_i(t)$, $L_i(t)$, $m(t)$, let us define $R'_i(t)$, $L'_i(t)$, $m'(t)$ considering only the blue jobs. Thus, for example, $R'_i(t)$ is the reported *blue* load of processor i at step t. Further, define $n'(t)$ as the number of blue tasks in the system at step t. Again, by a step we mean a single Fibonacci heap operation (which incurs $O(\log k)$ collisions in the worst case). So at any step t, $m'(t) \geq \frac{n'(t)}{k}$.

Our line of proof is as follows. We show that in βk steps after step t (for a constant $\beta \geq 1$ to be chosen later), either $n'(t)$ tasks finish execution or $m'(t+\beta k) \leq \frac{2}{3} m'(t)$. This would imply that some n jobs, blue or green, have been processed in $O(k \log n)$ steps following step t. Now at time t, there can be at most k processors with $R'_i(t) > \frac{2}{3} m'(t)$. If $m'(t+\beta k) > \frac{2}{3} m'(t)$, then clearly there must exist a processor p_j such that $R'_j(t) > \frac{2}{3} m'(t)$ and $R'_j(t+\beta k) > \frac{2}{3} m'(t)$. Since equal proportions of blue and green tasks migrate at every visit, we conclude that processor p_j has not been visited in the βk steps following t. As our policy is to visit the most heavily loaded processor, the βk visits must have involved processors whose loads were *larger* than $R'_j(t)$. Hence each visit must have involved a migration of at least $\frac{1}{3} R'_j(t) > \frac{2}{9} m'(t) \geq \frac{2}{9} \frac{n'(t)}{k}$ tasks. Also note that between successive visits by processor p_i, p_i must execute all tasks acquired at the previous visit. Now as there are k processors in the system, and there have been βk visits, at least $(\beta-1)k\ (\frac{2}{9}\frac{n'(t)}{k})$ tasks must have been processed in the meantime.

Thus for $\beta=11/2$, in βk steps either $m'(t)$ decreases to two-thirds of its original value, or at least $n'(t)$ tasks are processed. Formally, either $m'(t+\frac{11}{2}k) \leq \frac{2}{3}m'(t)$ or $n'(t)$ tasks have been processed between steps t and $t+\frac{11}{2}k$. But $n'(t)=n$ is the number of blue tasks in the system at step t. Thus in $O(k \log n)$ steps n tasks were processed, though they need not all be blue. This proves the theorem. □

Reporting Collisions

Next, we estimate reporting collisions. First, note that our proofs of Theorem 1 and Theorem 2 were based on the assertion that *the reported load is always an overestimate of the actual load* of processors. This assertion, in turn, was based on the assumption that processors report every increase in their previously reported load. Under the above stated conditions we showed that it takes $O(k \log k \log n)$ collisions, in the worst case, to process n tasks. Hence if $O(k \log k \log n)$ reporting collisions suffice to introduce n tasks into the system, then our simple algorithm would incur $O(k \log k \log n)$ total collisions. But this would imply a "chunky" task arrival/generation process, which may not be a reasonable assumption in many situations. To accommodate this possibility, we relax our requirement that processors report every increase in load immediately. Instead, we ask that if a processor's current load is L, then it report the number $\lceil \log_\rho L \rceil$, where $\rho > 1$ is a constant to be determined later. If at a task arrival/generation this number does not change, then the processor does *not* report an increase. Thus our modified reporting rule is as follows: Do not report any load reductions; report increases *only* if $\lceil \log_\rho L \rceil$ changes, where L is the current load of the processor. Of course, if p_i visits p_j, then the new loading conditions of *both* processors are reported, but that has already been counted as a visiting collision.

Let us see how this new reporting policy affects our proofs of Theorems 1 and 2. The actual load in the modified reporting scheme is *at most* ρ times the reported load. Hence as long as $\frac{2}{3}\rho<1$, i.e., $\rho<\frac{3}{2}$, Theorems 1 and 2 hold.

First, a *static* result. Assume that there are n tasks in the k-processor system at time t_0. We analyse the worst case number of reporting collisions required to introduce an additional n' tasks. Further we assume that while we introduce these tasks, no visiting collisions occur. Let n_i be the load of processor p_i at time t_0, and of the n' new tasks, let n'_i be added to p_i. Then we have:

Theorem 3. In the static case, in a k-processor system, $O(k \log k \log n')$ reporting collisions suffice to add n' tasks into the system, in the worst case.

Proof. Clearly, in the static case, adding n'_i tasks to processor p_i takes no more than $\log_\rho (n'_i + n_i) - \log_\rho (n_i) \le \log_\rho n'_i$ reporting steps. Note that while there may be no visiting collisions, there is the possibility that p_i completes some of its assigned tasks. Under our reporting scheme, this could only lead to fewer reports, since more tasks may arrive at p_i before p_i reaches the next reporting level. Hence the total number of reports is less than $\sum_{i=1}^{k} \log n'_i \le k \log n'$. Therefore the number of steps required is $O(k \log n')$, and the number of possible collisions is $O(k \log k \log n')$. □

In the dynamic case, as a theoretical convenience, we assume that the tasks form a first-in first-out (FIFO) queue in front of their processors. Thus if tasks $b_0, b_1, \ldots, b_s$ are in processor p_1's queue, then b_0 entered the queue the earliest and b_s the latest. The next task arriving at or generated for p_1 sits after b_s. Further, we assume that when p_1 has $b_1, \ldots, b_{m'}, b_{m'+1}, \ldots, b_{m'+m}$ in its queue, and a visit by p_2 takes m tasks from p_1, then after the visit, p_1 has $b_1, \ldots, b_{m'}$ and p_2 has $b_{m'+1}, \ldots, b_{m'+m}$ as their respective queues. Thus visits do not disturb the order of the migrating tasks or of the tasks left behind. Let b_r be the mth task from the head of the queue at time t in some processor. Define the *level number*, $l(b_r, t) = j$ such that $\rho^j < m \le \rho^{j+1}$. Clearly, for each b, $l(b, t)$ only decreases with time. Further, since $\frac{2}{3}\rho < 1$, when task b migrates to a visiting processor, $l(b, t)$ decreases by at least 1 in this migration. Let $l_0(b)$ denote the level number of task b when it first entered the system. Then $l(b, t) \le l_0(b)$ for all tasks b, at any instant t. In the following theorem, we make a steady state assumption that is clarified in the proof. Why we need to make such an assumption is discussed in the next section.

Theorem 4. At time t, let processor p_1 have n tasks in its queue. Then, under the above scheme, and under the steady state assumption, no more than $O(\log k \ \log n)$ reporting collisions could have occurred while introducing the n tasks into the system.

Proof. Here again we consider Fibonacci heap operations as steps. Let $b_1, \ldots, b_n$ be in p_1's queue at time t. Let $0{=}m_0, m_1, \ldots, m_r$ be numbers such that $b_{m_j+1},\ldots,b_{m_{j+1}}$ originated at processor p_{σ_j} for all $0 \le j \le r-1$ where $p_{\sigma_j} \ne p_{\sigma_{j+1}}$. Call $b_{m_j+1}, \ldots, b_{m_{j+1}}$ the jth *segment* of p_1. Next, note that the 0th segment has incurred at least $r-1$ migrations due to visits. Since the level number decreases by at least 1 in every visit, we have $r \le l_0(b_1)$. Further, all segments except possibly the last one (i.e., $(r-1)$th), were acquired by p_1 in just one visit. This last segment may have recently arrived at p_1. As level numbers decrease in every migration, each segment of p_1 must have originated at a level higher than its current level .

For x, y, and $z>0$ such that $x \ge y$, we have $\log(x+z)-\log(x) \le \log(y+z)-\log(y)$, and hence $\lceil \log(x+z) \rceil - \lceil \log(x) \rceil \le \lceil \log(y+z) \rceil - \lceil \log(y) \rceil + 1$. Consequently

$$l_0(b_{m_{j+1}}) - l_0(b_{m_j+1}) \le l(b_{m_{j+1}}, t) - l(b_{m_j+1}, t) + 1.$$

But $l_0(b_{m_{j+1}}) - l_0(b_{m_j+1})$ is the number of possible reporting steps incurred in introducing the jth segment into the system at p_{σ_j}. So summing both sides of the above inequality we get:

$$\sum_{j=0}^{r-1} \left[l_0(b_{m_{j+1}}) - l_0(b_{m_j+1}) \right] \le \sum_{j=0}^{r-1} \left[l(b_{m_{j+1}}, t) - l(b_{m_j+1}, t) + 1 \right].$$

$$\le \lceil \log_\rho n \rceil + r \le \lceil \log_\rho n \rceil + l_0(b_1)$$

At this point we make the steady state assumption and say that $l_0(b_1)$ is $O(\log n)$. Thus $O(\log n)$ reporting steps occurred during the addition of n tasks to the queue of p_1. □

Discussion

Note that in our complexity analyses, each access to F was priced at the maximum possible of $O(\log k)$ shared memory accesses, and hence with the possibility of $O(\log k)$ collisions. Thus the Fibonacci heap is *not* essential to the result; any data structure allowing *max*, *incr*, and *decr* in $O(\log k)$ time will suffice. We chose Fibonacci heaps to optimise the performance as much as possible. It may be noted that non-amortized versions of Fibonacci heaps are now available [1].

We have shown that it takes $O(k \log k \log n)$ visiting collisions *following* step t to process the n tasks in the system at time t. Similarly we have shown that it takes $O(k \log k \log n)$ reporting collisions *preceding* time t to arrive at a configuration of n tasks at time t. We have *not* shown that over $O(k \log k \log n)$ collisions, the system completes n tasks; this is not true. The second assertion holds only when we assume steady loading conditions. The local steady state assumption of $l_0(b_1)=c \log n$ (c a constant) appears reasonable. The constant c measures how steady the loading is.

References

[1] Driscoll, J. R., Gabow, H. N., Shairman, R., and Tarjan, R. E., Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation, *Commun. ACM* **31** (1988) 1343-1354.

[2] Fredman, M. L., and Tarjan, R. E., Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* **34** (1987) 596-615.

[3] Manber, U., On maintaining dynamic information in a concurrent environment, *SIAM J. Comput.* **15** (1986) 1130-1142.