On Trading Task Reallocation for Thread Management in Partitionable Multiprocessors

Lixin Gao Arnold L. Rosenberg Ramesh K. Sitaraman
Department of Computer Science
University of Massachusetts
Amherst, Mass. 01003, USA
{gao, rsnbrg, ramesh}@cs.umass.edu

Abstract

Most general-purpose multiprocessors are time-shared among multiple users. When a user arrives, s/he requests a submachine of size appropriate to his/her computation; the processor-allocation algorithm then assigns him/her a portion of the multi-processor of the requested size. This study is motivated by the fact that, as successive users arrive, use a portion of the multiprocessor, and depart, various individual processors in the multiprocessor may find themselves managing quite disparate numbers of threads. This load-imbalance is clearly undesirable, and can be rectified by reallocating (or migrating) the tasks periodically so as to balance the processors thread-loads. However, task reallocation is an expensive operation and must be performed infrequently, if ever. This paper establishes that there is a predictable trade-off between the frequency of task reallocation and the imbalance in the processor loads.

The processor-allocation algorithms devised in this paper are applicable to any hierarchically-decomposable multiprocessor, even though we state all our results for a tree-based multiprocessor. We devise a deterministic processor-allocation algorithm for an N-processor tree-machine that achieves a maximum load of

$$\min\left\{(d+1), \left\lceil \frac{1}{2}(\log N + 1)\right\rceil\right\} L^*,$$

where L^* is the optimal load achievable for the task sequence, and d is the reallocation parameter. We prove a lower bound by showing that no deterministic algorithm with reallocation parameter d can achieve within a factor of

$$\min\left\{\left\lceil \frac{1}{2}(d+1)\right\rceil, \left\lceil \frac{1}{2}(\log N + 1)\right\rceil\right\}$$

from the optimal load for all task sequences. Next, we present a randomized processor-allocation algorithm for an N-processor tree-machine that does not reallocate tasks and achieves a load of at most

$$\left(\frac{3\log N}{\log\log N} + 1\right) L^*,$$

where L^* is the optimal load of the task sequence; and, we show that no randomized processor-allocation algorithm without reallocation can achieve within a factor of

$$\frac{1}{7}\left(\frac{\log N}{\log\log N}\right)^{1/3}$$

from the optimal load for all task sequences.

1 Introduction

Most general-purpose multiprocessors are time-shared among multiple users. When a user arrives, s/he requests a submachine of size appropriate to his/her computation; the processor-allocation algorithm then assigns him/her a virtual submachine of the requested size. This study is motivated by the fact that, as successive users arrive and receive submachines, the various actual processors of the multiprocessor may find themselves managing quite disparate numbers of threads. The more heavily loaded processors are thus burdened by the nontrivial—and nonproductive—overhead of managing many threads as shown in [4, 5]. One avenue to alleviating this situation is to allow the processor-allocation algorithm to reallocate users' tasks so as to balance the numbers of threads across the machine's processors. This solution does not come without cost: process reallocation can require extensive communication cost (e.g., moving checkpointing states) and memory space (for the checkpointing). Therefore, before one advocates frequent reallocation as a remedy for load imbalances, one would do well to understand the impact of periodic reallocation on the load-balance of the multiprocessor. This paper is devoted to studying this impact in an environment in which users arrive and depart at unpredictable times and request submachines of unpredictable sizes. The main results of this paper establish that there is, in fact, a predictable tradeoff between the frequency of process reallocation and the maximum imbalance in processor thread-load.

We consider a hierarchically decomposable multiprocessor that consists of processing elements (PEs) that communicate over an interconnection network. Independent users arrive over time and request real-time service. Upon arriving, each user requests a submachine of fixed size and topology; for instance, if the multiprocessor were a hypercube, then all user requests would be for subcubes. Since there is no bound on the number

of active users, distinct users may well be assigned to overlapping portions of the multiprocessor at the same time. We call the number of distinct users allocated to a PE at any moment the *load* of the PE at that moment, i.e., the number of threads the PE has to manage at that moment.

> Note that PE-load often admits another interpretation also. When tasks allocated to a single PE are time-shared in a round-robin fashion, the worst slowdown ever experienced by a user is proportional to the maximum load of any PE in the submachine allocated to it.

Our focus here is on studying avenues for minimizing the maximum loads of PEs, i.e., the maximum numbers of threads that the PEs have to manage. Now, of course, there is some level of PE-load that is inevitable, even if the processor allocation algorithm were to balance the processor loads evenly at all times. It is this inevitable load level that we shall use as the benchmark against which to measure our process-allocation algorithms.

As we remarked earlier, allowing process reallocation is one natural avenue for keeping the load down, for it allows one to take advantage of user departures that have already occurred. Indeed, we show that, if one were to allow process reallocation at every step, then one could easily guarantee minimum load at every step. The main focus of this paper is to quantify the benefits in load level of periodic process reallocation in an on-line allocation algorithm. Specifically, if the multiprocessor has N PEs, then we choose a parameter d, and we allow a reallocation whenever the cumulative sizes of the tasks that have arrived since the last reallocation reaches dN. Note that the case $d = 0$ corresponds to the constantly reallocating algorithm, while the case $d = \infty$ corresponds to an algorithm that never reallocates. Our results capture the tradeoff between the two cost measures: the frequency of reallocation, as exposed by the parameter d, and the complexity of managing threads, as exposed by the maximum load of any PE.

Related work. There has been a significant amount of prior work in processor allocation; all such work view the computational load as a sequence of tasks, each requiring certain computational resources. A number of prior studies in [12, 9, 10, 11, 13, 14, 18] allocate processors considering topology constraint of each task. In [12, 9, 10, 11], they consider the problem of subcube recognition for hypercube machines, but they do not formally analyze their algorithms; further [12, 10] give task reallocating strategies, but there is no formal measure on the tradeoff between the reallocation frequency and resulting fragmentation. The studies in [13, 14, 18] allocate parallel machines under the assumption that each task has the exclusive use of its assigned processors and that the tasks can be delayed for arbitrarily long periods of time before they are serviced. They evaluate the makespan for a set of tasks, instead of the response time for each single task, thus forsaking the issue of real-time service. There are a number of studies, e.g., [2, 19, 8] and references therein, on the on-line problem of allocating tasks to a set of servers. However, in their model, servers are independent; therefore, topology is not considered to be an issue. Further, the algorithms in [8] preempts tasks at any time without considering the cost involved. In all the above mentioned work except [2], machines are never truly shared, in the sense that no two users are allocated to share the same processor at the same time. Therefore, thread management is not considered to be an issue. However, as in many real-world parallel machines such CM-5 [17] and SP2, multiple users could share the same processor at the same time.

Our problem. To the best of our knowledge, our study is the first attempt at quantifying the complexity of thread management when multiple users requesting real-time service share a a hierarchically decomposable multiprocessor. Motivated by practical considerations, we assume that our on-line allocation algorithms have no apriori knowledge of the duration of each task, or of any future task arrivals or departures. We do, however, allow periodic process reallocation. In fact, we prove (in Section 3) that load achieved by a constantly reallocating algorithm is, in fact, is exactly the optimal load for any task sequence. (The *optimal load* for a task sequence refers to the inevitable load that some processor must experience even if the load is evenly balanced at all times.) We evaluate the performance of our on-line algorithm with periodic reallocation via the ratios between our algorithm's maximum loads over time and the optimal load for the worst task sequence. We explore the tradeoff between the performance of the on-line algorithm and the periodicity of reallocation used in the algorithm. In this paper, we concentrate on multiprocessors having the topology of a complete binary tree (cf. [3, 6]). The results also hold for any hierarchically decomposable machine such at CM-5 and SP2 (cf. [17]). The processor allocation algorithms developed in this paper also apply to other networks such as the butterfly, the hypercube and the mesh.

A roadmap. In Section 2, we give a formal definition of the problem. In Section 3, we present a constantly reallocating algorithm that achieves the optimal load for any task sequence. In Section 4, we present a deterministic on-line algorithm for an N-PE multiprocessor with reallocation parameter d that achieves a load of at most $\min\{(d+1), \lceil \frac{1}{2}(\log N + 1)\rceil\}L^*$, where L^* is the optimal load of the task sequence. We close the section by proving that no deterministic on-line algorithm with reallocation parameter d can achieve a load within a factor of $\min\left\{\lceil \frac{1}{2}(d+1)\rceil, \lceil \frac{1}{2}(\log N + 1)\rceil\right\}$ from the optimal load for all task sequences. Our upper and lower bounds are tight within a factor of 2. In Section 5, we present a randomized on-line algorithm without reallocation for an N-PE multiprocessor that achieves a load of at most $\left(\frac{3\log N}{\log\log N} + 1\right)L^*$, where L^* is the optimal load of the task sequence; and, we show that no randomized on-line algorithm without reallocation can achieve within a factor of $\frac{1}{7}\left(\frac{\log N}{\log\log N}\right)^{1/3}$ from the optimal load for all task sequences.

2 Model and Definitions

The Parallel Machine. For most of the paper, we consider an N-PE tree machine T (see [3, 6]), which is a parallel machine having the topology of an N-leaf complete binary tree whose leaf nodes hold processing elements (PEs) and whose internal nodes hold communication switches.

Submachines An M-PE submachine is an M-PE complete binary subtree of T.

Tasks. Each task t of size $s(t)$ requires a submachine with $s(t)$ PEs; the size of a task is a power of 2 and is known as soon as it arrives, but its execution time is not known. As soon as it arrives, a task t must be assigned an $s(t)$-PE submachine of T; once assigned, the task cannot be migrated to another submachine of T except during reallocation.

Task Sequence. A *task sequence* σ is a sequence of task-arrival or task-departure events that are ordered by time of occurrence. A task is *active* from its arrival time to its departure time. The size of sequence σ at time τ, denoted $S(\sigma;\tau)$, is the cumulative size of tasks active at time τ. Let $|\sigma|$ be the time of the last arrival. The size of sequence σ, denoted $s(\sigma)$, is the maximum over time τ (τ varying from 0 to $|\sigma|$) of the cumulative size of the tasks active at time τ:

$$s(\sigma) = \max_{0<\tau\leq|\sigma|} S(\sigma;\tau)$$

Load. The *load* for a PE u of T at time τ, denoted $\lambda(u;\tau)$, is the number of tasks that are assigned to node u and are active at time τ.

Allocation Algorithms. An *allocation algorithm* must select an $s(t)$-PE submachine in T, assign task t to it at t's arrival time, and deallocate it at t's departure time. The load of an algorithm A on task sequence σ at time τ, denoted $L_A(\sigma;\tau)$, is the maximum load of all the PEs of T at time τ:

$$L_A(\sigma;\tau) = \max_{u\in T} \lambda(u;\tau)$$

The load of a deterministic allocation algorithm A on task sequence σ, denoted $L_A(\sigma)$, is the maximum load of T over all times:

$$L_A(\sigma) = \max_{0<\tau\leq|\sigma|} L_A(\sigma;\tau)$$

The load of a randomized allocation algorithm R on task sequence σ, denoted $L_R(\sigma)$, is the maximum expected load of T over all times:

$$L_R(\sigma) = \max_{0<\tau\leq|\sigma|} E(L_R(\sigma;\tau))$$

An *on-line* allocation algorithm must assign an arriving task t to an $s(t)$-PE submachine of T knowing only the quantity $s(t)$ and all previous task assignments; the assignment is made without any knowledge about future arriving or departing tasks. For example, one plausible greedy on-line algorithm would allocate task t to an $s(t)$-PE submachine of T that has the smallest load, allocating to the leftmost such submachine in case of a tie. Figure 1 shows the assignments by this algorithm on a 4-PE tree-machine for the following sequence

σ^*: t_1 arrives, t_2 arrives, t_3 arrives, t_4 arrives, t_2 departs, t_4 departs, t_5 arrives,

where t_1, t_2, t_3, t_4 are tasks of size 1, and t_5 is a task of size 2. We use dotted boxes to enclose tasks that are not active at time $|\sigma^*|$. Note that this on-line algorithm achieves a load of 2 for the given sequence.

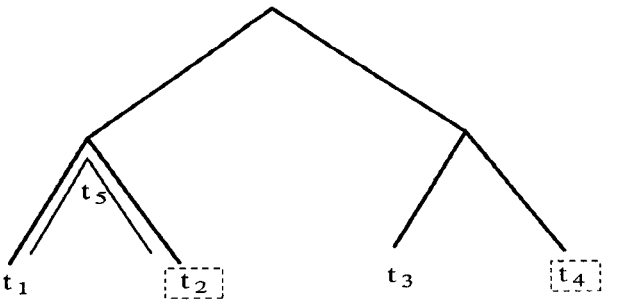

Figure 1: The assignments by the greedy on-line algorithm

A *d-reallocation* on-line algorithm is an on-line allocation algorithm that can reallocate tasks after the total size of tasks that have arrived since the last reallocation reaches dN. During a reallocation, every active task can be reassigned to a new submachine. A 0-reallocation on-line algorithm is an on-line allocation algorithm that can reallocate tasks at every time-step. An ∞-deallocation on-line algorithm is an on-line allocation algorithm that never reallocates tasks. For the example of task sequence σ^*, an 1-reallocation algorithm can reallocate tasks when the total size of tasks that have arrived since the last reallocation reaches 4. Therefore, it can reallocate t_3 to the position of t_2 at the time t_5 arrives. This results a load of 1 for task sequence σ^*.

Optimal Load. The *optimal load* L^* for a task sequence σ is the inevitable load that some processor must experience even if the load is evenly balanced at all times. The maximum cumulative size of active tasks at any point of time for a task sequence σ is $s(\sigma)$. Therefore, the optimal load L^* is equal to $\left\lceil \frac{s(\sigma)}{N} \right\rceil$. We compare the load achieved by our allocation algorithms to the benchmark of L^* – a good allocation algorithm is one that achieves a load close to L^* for every task sequence.

Our goal here is to devise deterministic and randomized on-line allocation algorithms that achieve load close to the optimal load L^*.

3 An Optimal 0-Reallocation Algorithm

In this section, we present an allocation algorithm A_C that reallocates tasks every time a task arrives, and achieves the optimal load of L^* for every task sequence.

Algorithm A_C:

Task Arrival When a task arrives, add it to the set of active tasks. Use reallocation procedure A_R (described below) to reallocate all active tasks.

Task Departure: When a task departs, the sub-machine allocated to it is de-allocated.

Note that algorithm A_C reallocates all the tasks during each task arrival. The procedure used for reallocation, A_R, takes a set of active tasks and maps them to (possibly) new positions within T. It is convenient to view procedure A_R as allocating tasks to many identical copies of the machine T. The procedure starts with one copy of the machine. It can create more copies if needed. A PE in each copy can be assigned to one task only and copies of T are ordered according to their time of creation. Each copy of the machine is emulated as a different thread on machine T. Thus, the load of T is at most the total number of copies.

We call a submachine of a copy of T, a *vacant submachine* if none of its PEs is assigned to a task. A *maximal vacant submachine* is a vacant submachine that is not properly contained in any other vacant submachine.

Reallocation Procedure A_R:
Sort the tasks in order of decreasing size.
For each task of size 2^x,
search for the first copy of T that contains a 2^x-PE vacant submachine. (If there is no such copy, create a new copy of T.)
Assign the task to the leftmost 2^x-PE vacant submachine in this copy.

Lemma 1 *For any task set of the total task size S, procedure A_R achieves load of $\lceil S/N \rceil$.*

Proof. We prove the lemma by the following claim.

Claim 1. Reallocation procedure A_R does not create a vacant submachine except possibly in the last copy.

Assume, for contradiction, that procedure A_R creates a vacant submachine in a copy κ other than the last copy. Let the size of a maximal vacant submachine in κ be 2^x. From the hierarchical structure of the tree machine T, there is a task of size at most 2^x assigned to the copy κ. Since procedure A_R assigns tasks in order of decreasing size, the tasks assigned to the last copy must have size of at most 2^x. This contradicts with the fact that procedure A_R always assigns a task to the first copy that contains a vacant submachine of the required task size.

Therefore, the number of copies created by procedure A_R is $\lceil S/N \rceil$. We conclude that the load of procedure A_R is $\lceil S/N \rceil$. □

Theorem 3.1 *For any task sequence σ, Algorithm A_C achieves the optimal load L^*.*

Proof. Since departures decrease load, it is sufficient to prove that the load is at most L^* after each arrival. At any time τ that a task arrives, Algorithm A_C reallocates all active tasks using the reallocation procedure A_R. From Lemma 1, the load achieved by A_C at time τ is $\lceil S(\sigma, \tau)/N \rceil$. Therefore, the load of Algorithm A_C is L^* for any task sequence σ. □

4 Deterministic On-line Allocation Algorithms

In this section, we present and analyze a deterministic d-reallocation on-line algorithm. We then give a lower bound on the performance of any deterministic d-reallocation on-line allocation algorithm.

4.1 A Deterministic On-line Allocation Algorithm

First, we present a greedy on-line allocation algorithm A_G that does not reallocate tasks. Subsequently, we use A_G in our d-reallocation on-line algorithm A_M.

Algorithm A_G:

Task Arrival: When a task of size 2^x arrives, compute the loads for all 2^x-PE submachines of T. Assign the task to the leftmost submachine of size 2^x that has the smallest load.

Task Departure: When a task departs, the submachine allocated to it is de-allocated.

Theorem 4.1 *For any task sequence σ, the maximum load achieved by Algorithm A_G is at most $\left\lceil \frac{1}{2}(\log N + 1) \right\rceil L^*$, where L^* is the optimal load for σ.*

Proof. Since tasks of size N do not create a load imbalance, we assume that all tasks have size less than N. We establish the theorem via the following more detailed claim.

Claim. Algorithm A_G assigns a task of size $2^x < N$ to a submachine of the left subtree of T whose load is less than $\lceil (\frac{1}{2}x + 1)L^* \rceil$ or to a submachine of the right subtree of T whose load is less than $\lfloor (\frac{1}{2}x + 1)L^* \rfloor$

We verify the claim by induction on the size of the arriving task.

First, the base case: since the size of sequence σ is $s(\sigma)$, when a task of size 1 arrives, the cumulative size of active tasks is at most $s(\sigma) - 1$. By the pigeonhole principle, therefore, at least one PE of T has load less than L^*. Algorithm A_G assigns the task to such a PE, thus honoring the claim.

Suppose now that the claim is true for any task of size less than 2^x. When a task of size 2^x arrives, assume, for the sake of contradiction, that all 2^x-PE submachines of the left subtree of T have load at least

$\lceil (\frac{1}{2}x+1)L^* \rceil$, and all 2^x-PE submachines of the right subtree of T have load at least $\lfloor (\frac{1}{2}x+1)L^* \rfloor$. By the induction hypothesis, the following are true:

- For any 2^x-PE submachine of the left subtree of T, some PE has load at least $\lceil (\frac{1}{2}x+1)L^* \rceil$. Order the active tasks that are assigned to this PE by increasing size. Assume, for contradiction, that for some $i = 1, 2, \ldots, x$, the $\lceil \frac{1}{2}(i+1)L^* + 1 \rceil$th task has size less than 2^i. The task that arrives last among the first $\lceil \frac{1}{2}(i+1)L^* + 1 \rceil$ tasks is assigned to a submachine that has load at least $\lceil (\frac{1}{2}i+1)L^* \rceil$. This contradicts the induction hypothesis. Thus the $\lceil \frac{1}{2}(i+1)L^* + 1 \rceil$th task has size at least 2^i for all $i = 1, 2, \ldots, x$. Therefore, the cumulative size of the active tasks assigned to a 2^x-PE submachine of the left subtree of T is at least

$$L^* + \sum_{i=1}^{\lceil x/2 \rceil} \left\lceil \frac{L^*}{2} \right\rceil 2^{2i-1} + \sum_{i=1}^{\lfloor x/2 \rfloor} \left\lfloor \frac{L^*}{2} \right\rfloor 2^{2i}$$

- Similarly, we can conclude that the cumulative size of the active tasks assigned to a 2^x-PE submachine of the right subtree of T is at least

$$L^* + \sum_{i=1}^{\lceil x/2 \rceil} \left\lfloor \frac{L^*}{2} \right\rfloor 2^{2i-1} + \sum_{i=1}^{\lfloor x/2 \rfloor} \left\lceil \frac{L^*}{2} \right\rceil 2^{2i}$$

Combining these cases: when a task of size 2^x arrives, the cumulative size of active tasks assigned to T can be shown to be at least $L^*N \geq s(\sigma)$; specifically, the cumulative size of active tasks is

$$\left(L^* + \sum_{i=1}^{\lceil x/2 \rceil} \left\lceil \frac{L^*}{2} \right\rceil 2^{2i-1} + \sum_{i=1}^{\lfloor x/2 \rfloor} \left\lfloor \frac{L^*}{2} \right\rfloor 2^{2i} \right) \frac{N}{2^{x+1}} + \left(L^* + \sum_{i=1}^{\lceil x/2 \rceil} \left\lfloor \frac{L^*}{2} \right\rfloor 2^{2i-1} + \sum_{i=1}^{\lfloor x/2 \rfloor} \left\lceil \frac{L^*}{2} \right\rceil 2^{2i} \right) \frac{N}{2^{x+1}}$$

$$= \left(2L^* + \sum_{i=1}^{\lceil x/2 \rceil} L^* 2^{2i-1} + \sum_{i=1}^{\lfloor x/2 \rfloor} L^* 2^{2i} \right) \frac{N}{2^{x+1}}$$

$$= L^* \left(2 + \sum_{i=1}^{x} 2^i \right) \frac{N}{2^{x+1}} = L^* 2^{x+1} \frac{N}{2^{x+1}} = L^* N \geq s(\sigma)$$

The newly arriving task would increase this size, contradicting the fact that sequence σ has size $s(\sigma)$.

We conclude that, when a task of size 2^x arrives, there is a 2^x-PE submachine of the left subtree of T whose load is less than $\lceil (\frac{1}{2}x+1)L^* \rceil$ or a 2^x-PE submachine of the right subtree of T whose load is less than $\lfloor (\frac{1}{2}x+1)L^* \rfloor$. The claim follows.

Since a task has size at most $N/2$, the maximum load achieved by the greedy algorithm is at most $\lceil \frac{1}{2}(\log N + 1) \rceil L^*$. The theorem follows. □

Next, we present a basic algorithm A_B that will be used by the d-reallocation algorithm A_M. It is convenient to view the basic algorithm A_B as allocating tasks to many identical copies of the machine T. The algorithm starts with one copy of the machine. It can create more copies if needed. The PE of each copy can be assigned to one task only and copies of T are ordered according to their time of creation. As before, we call a submachine of a copy of T, a *vacant submachine* if none of its PEs is assigned to a task. A *maximal vacant submachine* is a submachine that is not properly contained in any other vacant submachine. Each copy of the machine is emulated as a different thread on machine T. Thus, the load of T at some point of time is at most the number of copies in existence at that point of time.

Algorithm A_B:

Task Arrival: When a task of size 2^x arrives, search for the first copy of T that contains a 2^x-PE vacant submachine. (If there is no such copy, create a new copy of T.) Assign the arrival task to the leftmost 2^x-PE vacant submachine in this copy.

Task Departure: When a task departs, the submachine allocated to it is de-allocated.

Lemma 2 *For a task sequence σ in which the total size of the arrival tasks is S, Algorithm A_B achieves load of at most $\lceil S/N \rceil$. (Note that S is not the size of the task sequence but the sum of the sizes of all the arrivals in the sequence.)*

Proof. We prove the lemma by the following two claims.

Claim 1. Algorithm A_B never creates two maximal vacant submachines of the same size.

Assume, for contradiction, that Algorithm A_B creates two maximal vacant submachines of size 2^x, V_1 and V_2. The 2^{x+1}-PE submachine that contains V_1 is assigned a task of size at most 2^x, denoted by t_1; the 2^{x+1}-PE submachine that contains V_2 is assigned a task of size at most 2^x, denoted by t_2. Without loss of generality, assume V_1 is in front of V_2 in the search order of A_B. When task t_2 arrives, Algorithm A_B should find V_1 before it finds V_2. Therefore, Algorithm A_B should assign t_2 to vacant submachine V_1 or a vacant submachine in front of V_1 in the search order of A_B. This contradicts with the fact that t_2 is assigned to the 2^{x+1}-PE submachine that contains V_2. The claim follows.

Claim 2. When a task of size 2^x arrives, there is a 2^x-PE vacant submachine in first $\lceil S/N \rceil$ copies T.

Assume, for contradiction, that when a task of size 2^x arrives, there is no 2^x-PE vacant submachine in the first $\lceil S/N \rceil$ copies T. This implies that there is no maximal vacant submachine of size at least 2^x in the first $\lceil S/N \rceil$ copies T. By Claim 1, the cumulative size of maximal vacant submachines in the first $\lceil S/N \rceil$ copies

T is at most $\sum_{i<x} 2^i < 2^x$. This implies that more than $S - 2^x$ PEs are assigned tasks in the first $\lceil S/N \rceil$ copies. Therefore, when a task of size 2^x arrives, the total size of active tasks is more than $S - 2^x$. This contradicts the fact that the total size of active tasks in σ is at most S. The claim follows.

We conclude that the load of Algorithm A_B is at most $\lceil S/N \rceil$. □

Our d-reallocation on-line algorithm A_M that uses the greedy on-line algorithm A_G and the basic algorithm A_B is described as follows:

Algorithm A_M:

Task Arrival: If $d \geq \left\lceil \frac{1}{2}(\log N + 1) \right\rceil$, then allocate the task using greedy algorithm A_G. Otherwise, allocate the task using Algorithm A_B.

Task Reallocation: if $d \geq \left\lceil \frac{1}{2}(\log N + 1) \right\rceil$, perform no reallocation. Otherwise, if $d < \left\lceil \frac{1}{2}(\log N + 1) \right\rceil$ and the total size of arriving tasks since last reallocation is at least dN, reallocate all active tasks using reallocation procedure A_R.

Task Departure: When a task departs, the submachine allocated to it is de-allocated.

Theorem 4.2 *For any task sequence σ, the maximum load achieved by Algorithm A_M is within a factor of f from the optimal load L^*,*
where $f = \min\{d+1, \lceil \frac{1}{2}(\log N + 1) \rceil\}$.

Proof. When $d \geq \left\lceil \frac{1}{2}(\log N + 1) \right\rceil$, Algorithm A_M is exactly the same as Algorithm A_G.

By Theorem 4.1, the maximum load achieved by algorithm A_M is at most $\left\lceil \frac{1}{2}(\log N + 1) \right\rceil L^*$.

When $d < \left\lceil \frac{1}{2}(\log N + 1) \right\rceil$, we prove that the maximum load achieved by algorithm A_M is at most $d + L^* \leq (d+1)L^*$. At any time τ, let σ' be the sequence of events that occur before τ. Let σ_1 denote the sequence of events that occur before the last reallocation in σ', and σ_2 denote the sequence of events that occur after the last reallocation in σ'. Since Algorithm A_M reallocates all active tasks at time $|\sigma_1|$ using Algorithm A_B, we can use Lemma 1 to bound the load created by the active tasks in σ_1 to be at most $\lceil S(\sigma_1, |\sigma_1|)/N \rceil$, which is at most L^*. The total size of arriving tasks in σ_2 is at most dN. Using Lemma 2, the load created by active tasks in σ_2 is at most d. Thus, the total number of copies created by algorithm A_M is at most $d + L^*$. Thus the load of algorithm A_M is at most $(d+1)L^*$.

Therefore, the maximum load achieved by Algorithm A_M is at most fL^*. □

4.2 A Lower Bound for Deterministic On-line Algorithms

In this section, we prove a lower bound on the performance of any deterministic d-reallocation on-line algorithms for an N-PE machine. The lower bound presented in this section is tight to within a factor of two.

Theorem 4.3 *For any d-reallocation deterministic on-line algorithm, there exists a task sequence σ for which the algorithm incurs a load of at least fL^*, where L^* is the optimal load of σ and $f = \left\lceil \frac{1}{2}(\min\{d, \log N\} + 1) \right\rceil$.*

For any given deterministic d-reallocation on-line algorithm, we construct a sequence σ such that the algorithm incurs a load of at least fL^* on that sequence. We construct σ in $p \stackrel{\text{def}}{=} \min\{d, \log N\}$ phases as follows:

In phase 0, N tasks of size 1 arrive.

In phase i, task departures are followed by task arrivals:

(1) For each 2^{i-1}-PE submachine T_{i-1}:
Compute $l(T_{i-1})$: the maximum load of all PEs in submachine T_{i-1}
Compute $L(T_{i-1})$: the cumulative size of the active tasks that use a PE in submachine T_{i-1}.
Compute $Q(T_{i-1}) \stackrel{\text{def}}{=} 2^i l(T_{i-1}) - L(T_{i-1})$
For each 2^i-PE submachine T_i:
Let T_i^L be the left subtree of T_i and T_i^R be the right subtree of T_i.
If $Q(T_i^L) > Q(T_i^R)$, then have all active tasks assigned to T_i^R depart;
If $Q(T_i^L) \leq Q(T_i^R)$, then have all active tasks assigned to T_i^L depart.

(2) Letting S be the cumulative size of current active tasks, have $\lfloor (N - S)/2^i \rfloor$ tasks of size 2^i arrive.

We use a potential argument to prove our lower bound. The potential of a submachine in T at the end of phase i is defined as follows:

- The potential of each 2^i-PE submachine T_i at the end of phase i: $P(T_i, i) \stackrel{\text{def}}{=} 2^i l(T_i) - L(T_i)$
- For $j \geq i$, the potential of each 2^j-PE submachine T_j at the end of phase i: $P(T_j, i) \stackrel{\text{def}}{=} \sum_{T_i \subset T_j} P(T_i, i)$

The potential of a submachine is a measure of its fragmentation. We first bound the increase in potential at each phase.

Lemma 3 *For all i, $P(T, i) - P(T, i-1) > \frac{1}{2}(N - 2^{i-1})$.*

Proof. At the beginning of phase i, the potential of a 2^i-PE tree T_i is

$$\begin{aligned} & P(T_i, i-1) \\ &= P(T_i^L, i-1) + P(T_i^R, i-1) \\ &= 2^{i-1} l(T_i^L) - L(T_i^L) + 2^{i-1} l(T_i^R) - L(T_i^R) \end{aligned}$$

Suppose that algorithm A assigns k tasks of size 2^i on submachine T_i, at phase i. Then:

If $Q(T_i^L) \leq Q(T_i^R)$, then all active tasks assigned to T_i^L depart; hence,

$$P(T_i, i) = 2^i \left(l(T_i^R) + k\right) - \left(L(T_i^R) + 2^i k\right) = Q(T_i^R)$$

Similarly, if $Q(T_i^L) > Q(T_i^R)$, then $P(T_i, i) = Q(T_i^L)$.

In either case, therefore,
$P(T_i, i) = \max\left\{Q(T_i^R), Q(T_i^L)\right\} \geq \frac{1}{2}(Q(T_i^R) + Q(T_i^L))$.

It follows that the potential of T_i at phase i is increased by

$$\begin{aligned} & P(T_i, i) - P(T_i, i-1) \\ \geq\ & \frac{1}{2}\left(Q(T_i^R) + Q(T_i^L)\right) - P(T_i, i-1) \\ =\ & \frac{1}{2}(L(T_i^L) + L(T_i^R)) \end{aligned}$$

Summing the potential increase for all 2^i-PE submachines of T, we thus find that the potential increase of T is

$$\begin{aligned} P(T, i) - P(T, i-1) &= \sum_{T_i \subset T} (P(T_i, i) - P(T_i, i-1)) \\ &= \sum_{T_i \subset T} \frac{1}{2}(L(T_i^L) + L(T_i^R)) \\ &\geq \frac{1}{2}(N - 2^{i-1}). \end{aligned}$$

The final inequality is true because we choose the number of task arrivals at phase $i-1$ so that the total size of active tasks is at least $N - 2^{i-1}$ at the beginning of phase i. □

Proof of Theorem 4.3. Since the total size of task arrivals at each phase is at most N, the total size of task arrivals in the task sequence is at most pN, which is at most dN since $p = \min\{d, \log N\}$. Therefore, reallocation can not occur during this task sequence. Using Lemma 3, the potential at the end of each task sequence is

$$P(T, p-1) \geq \frac{1}{2}N(p-1) - 2^{p-1} + 1$$

By definition, $P(T, p-1) = l(T)N - L(T)$. By the construction of σ, at the end of the sequence,

$$L(T) \geq N - 2^{p-1}.$$

We conclude that $l(T) \geq \left\lceil \frac{1}{2}(p+1) \right\rceil$,
where $p = \min\{d, \log N\}$. □

5 Randomized On-line Allocation Algorithms

In this section, we present a simple randomized on-line algorithm that does not reallocate tasks, and we analyze its maximum expected load. When reallocation is disallowed, we see that a simple application of randomization allows one to "beat" any deterministic algorithm. We close the section with a nontrivial lower bound on the maximum expected load incurred by any randomized on-line algorithm that does not reallocate tasks. Note that the results presented in this section do not utilize reallocation. The question of utilizing reallocation together with randomization is an area for future study.

5.1 A Randomized On-line Allocation Algorithm

The following oblivious randomized algorithm allocates tasks independent of the current loads of PEs in the machine and does not use reallocation.

Algorithm A_R:

Task Arrival: When a task of size 2^x arrives, assign it to any 2^x-PE submachine of T with probability $2^x/N$.

Task Departure: When a task departs, the submachine allocated to it is de-allocated.

The following lemma due to Hoeffding [15] is needed to analyze our algorithm.

Lemma 4 (Hoeffding) *Given N independent Bernoulli trials with respective probabilities $p_1, \ldots, p_N$, with mean $\mu = \sum p_i$, if $m \geq \mu + 1$ is an integer, then the probability of at least m successes in the N trials is at most $(\mu e/m)^m$.*

Theorem 5.1 *For a task sequence σ, the maximum expected load of Algorithm A_R is at most fL^*, where L^* is the optimal load of σ, and $f = \left(\frac{3\log N}{\log\log N} + 1\right)$.*

Proof. At any time τ, there are at most $s(\sigma)$ active tasks: number them $t_1, \ldots, t_{s(\sigma)}$. Let $p_i(u) = s(t_i)/N$ denote the probability that task t_i is assigned to a PE u.

Since $\sum_{i=0}^{s(\sigma)} s(t_i) \leq s(\sigma)$,

$$\mu = \sum_{i=1}^{s(\sigma)} p_i(u) \ \leq s(\sigma)/N \leq L^*$$

By Lemma 4, the probability that Algorithm A_R assigns more than kL^* tasks to PE u is at most $\left(\frac{k}{e}\right)^{-kL^*}$. Thus, when $k = \frac{3\log N}{\log\log N}$, the probability that Algorithm A_R assigns more than kL^* tasks to PE u is at most N^{-2}. Thus, the probability that the algorithm assigns more than kL^* tasks to some PE in T is at most N^{-1}.

Letting L denote the load of T, the expected load of T can be bounded as follows.

$$\begin{aligned} & E(L) \\ & \leq kL^*(1 - Prob\{L > kL^*\}) + NL^* Prob\{L > kL^*\} \\ & \leq kL^* + L^* \end{aligned}$$

The theorem follows. □

5.2 A Lower Bound for Randomized On-line Algorithms

In this section, we provide a lower bound on the maximum expected load of any randomized on-line algorithm that does not reallocate tasks.

Theorem 5.2 *For any randomized on-line algorithm, there exists a task sequence for which the maximum expected load is at least ℓL^*, where the optimal load of the sequence is L^*, and $\ell = \frac{1}{7}\left(\frac{\log N}{\log\log N}\right)^{1/3}$.*

Our strategy for the proof is to construct a random task sequence and show that any on-line algorithm (random or deterministic) performs "poorly" on it. This will imply that for every randomized on-line algorithm, there exists a fixed task sequence such that this algorithm performs "poorly" on it.

Consider the following random task sequence σ_r that consists of $\frac{\log N}{2\log\log N}$ phases:

At Phase i:

1. $N/(3\log^i N)$ tasks of size $\log^i N$ arrive.
2. With probability $1 - (1/\log N)$, each task of size $\log^i N$ departs.

Lemma 5 *With high probability, $s(\sigma_r) \leq N$.*

Proof Sketch. Let x_i^k be the indicator random variable for the event that the kth task of size $\log^i N$ does not depart; the x_i^k are independent random variables.

$$x_i^k = \begin{cases} 1 & \text{with probability } 1/\log N \\ 0 & \text{with probability } 1 - (1/\log N) \end{cases}$$

The size $s(\sigma_r)$ can be expressed as a weighted sum of the variables, x_i^k. We now use Chernoff bounds [16] in a standard fashion to derive the high probability bound. □

For any j, let $l(T'_j, i)$ denote the load of a $(\log^j N)$-PE submachine T'_j of T at the beginning of phase i. Define the following potential functions.

- The potential of each $(\log^i N)$-PE submachine at the beginning of phase i: $P'(T'_i, i) \stackrel{\text{def}}{=} l(T'_i, i)\log^i N$.
- For $j \geq i$, the potential for each $(\log^i N)$-PE submachine T'_j at the beginning of phase i: $P'(T'_j, i) \stackrel{\text{def}}{=} \sum_{T'_i \subset T'_j} P'(T'_i, i)$.

We bound the potential increase at each phase in the following lemma.

Lemma 6 *For any on-line task allocation algorithm, if the load of T is less than ℓ after the arrivals at phase i, then $P'(T, i+1) - P'(T, i) > N/(120\ell^2)$ with probability at least $1 - N^{-6}$.*

Proof. Say that no more than ℓ tasks of size $\log^i N$ are assigned to the same $(\log^i N)$-PE submachine at phase i. we have the following two claims.

Claim 1. At least $N/(12\ell\log^{i+1} N)$ $(\log^{i+1} N)$-PE submachines of T are assigned to at least $\frac{1}{4}\log N$ tasks of size $\log^i N$. The claim follows from the pigeon-hole principle. □

Claim 2. For each $(\log^{i+1} N)$-PE submachine T'_{i+1} of T that are assigned to at least $\frac{1}{4}\log N$ tasks of size $\log^i N$, either (a) $P'(T'_{i+1}, i+1) - P'(T'_{i+1}, i) \geq \log^{i+1} N$ with probability at least $1/(9\ell)$ or (b) $P'(T'_{i+1}, i+1) - P'(T'_{i+1}, i) \geq \log^{i+1} N/(8\ell)$

We prove Claim 2 by proving the following two cases. Let h be the number of size-$(\log^i N)$ submachines of T'_{i+1} having load strictly less than $l(T'_{i+1}, i)$ We consider the following two cases:

Case A. $h < (\log N)/(8\ell)$

Since the load on any node is less than ℓ, and there are $\frac{1}{4}\log N$ tasks of size $\log^i N$ in T'_{i+1}, there are at least $(\log N)/(4\ell)$ size-$(\log^i N)$ submachines of T'_{i+1} that are assigned to at least one task of size $\log^i N$. There are $\log N - h > \log N - (\log N)/(8\ell)$ size-$(\log^i N)$ submachines of T'_{i+1} that have load $l(T'_{i+1}, i)$. Therefore, at least $(\log N/(8\ell))$ size-$(\log^i N)$ submachines of T'_{i+1} are assigned at least one task of size $\log^i N$ and have load $l(T'_{i+1}, i)$ at the beginning of phase i. We conclude that, with probability

$$1 - \left(1 - \frac{1}{\log N}\right)^{(\log N)/8\ell} > 1 - e^{-1/8\ell} > \frac{1}{9\ell}$$

the load of T'_{i+1} is increased by at least 1, which means $P'(T'_{i+1}, i+1) - P'(T'_{i+1}, i) \geq \log^{i+1} N$.

Case B. $h \geq (\log N)/(8\ell)$

$$P'(T'_{i+1}, i+1) - P'(T'_{i+1}, i) \geq h\log^i N \geq \log^{i+1} N/(8\ell).$$

Using claim 1 and claim 2, and using Chernoff bounds, we conclude the theorem. □

Lemma 7 *The sequence σ_r results in a load of at least ℓ for any on-line allocation algorithm, with probability at least $1 - N^{-5}$, where $\ell = \left(\frac{\log N}{240\log\log N}\right)^{1/3}$.*

Proof. If the load of T never reaches ℓ during any phase, then by Lemma 6, with probability at least $1 - (\log N)/N^6$,

$$P'\left(T, \frac{\log N}{2 \log\log N}\right) \geq \frac{N}{120\ell^2} \frac{\log N}{2 \log\log N}$$

By the definition of the potential functions, the load of T is at least $\frac{\log N}{240\ell^2 \log\log N} = \ell$ with probability at least $1 - N^{-5}$. □

Proof of Theorem 5.2. By Lemma 7, the load of any on-line algorithm (deterministic or randomized) on the random sequence σ_r is at least ℓ with probability at least $1 - N^{-5}$. Using Lemma 5, the optimal load L^* of sequence σ_r is 1 with probability at least $1 - N^{-6}$. We conclude that the load of any randomized on-line algorithm on sequence σ_r is at least a factor ℓ away from its optimal load with probability at least $1 - N^{-4}$. Hence, the maximum expected load of any deterministic on-line algorithm on the random sequence σ_r is at least a factor ℓ away from its optimal load. Therefore, we conclude that for any randomized on-line algorithm there exists a fixed task sequence for which the maximum expected load incurred is at least a factor ℓ away from the optimal load. □

Acknowledgments. It is a pleasure to acknowledge helpful and stimulating conversations with Vittorio Scarano and Tom Leighton. The first author was supported in part by NSF Grants CCR-92-12567 and CCR-94-10077. The second author was supported in part by NSF Grant CCR-94-10077. The third author was supported in part by NSF Grant CCR-94-12567.

References

[1] B. Awerbuch, R. Gawlick, T. Leighton, Y. Rabani (1994): On-line Admission Control and Circuit Routing for High Performance Computing and Communication. *35th IEEE Symp. on Foundations of Computer Science,* 412-423.

[2] Y. Azar, A. Broder, A. Karlin and E.Upfal (1994): Balanced Allocations. *26th ACM Symp. on Theory of Computing,* 593-602.

[3] S.N. Bhatt, F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1995): Salvage-embeddings of complete trees. *SIAM J. Discrete Math.*

[4] R. Blumofe and C.E Leiserson (1993): Space-efficient scheduling of multithreaded computations. *25th ACM Symp. on Theory of Computing,* 362-371.

[5] R. Blumofe and C.E Leiserson (1994): Scheduling multithreaded computations by work stealing. *35th IEEE Symp. on Foundations of Computer Science.*

[6] S.Browning (1980): The Tree Machine: A highly Concurrent Computing Environment. Ph.D. Thesis, CalTech.

[7] A. Borodin, N. Linial, and M. Saks. (1987): An Optimal On-line Algorithm for Metrical Task Systems. *19th ACM Symp. on Theory of Computing,* 373-382.

[8] T. Brecht, X. Deng, N. Gu (1995): Competitive dynamic processor allocation for parallel applications. *7th IEEE Symp. on Parallel and Distr. Processing,* 448-455.

[9] M. Chen and K. Shin (1987): Processor Allocation in an N-Cube Multiprocessor Using Gray Codes. *IEEE Trans. Comp. C-36,* 1396-1407.

[10] M. Chen and K. Shin (1990): Subcube Allocation and Task Migration in Hypercube Multiprocessors. *IEEE Trans. Comp. C-39,* 1146-1155.

[11] S. Dutt and J. P. Hayes (1991): Subcube Allocation in Hypercube Computers. *IEEE Trans. Comp. C-40,* 341-352.

[12] G. Chen and T. Lai (1989): Virtual Subcubes and Job Migration in a Hypercube. *International Conference on Parallel Processing,* 73-76.

[13] A. Feldmann, J. Sgall and S. Teng (1991): Dynamic Scheduling on Parallel Machines. *32th IEEE Symp. on Foundations of Computer Science,* 111-120.

[14] A. Feldmann, M. Kao, J. Sgall and S. Teng (1993): Optimal Online Scheduling of Parallel Jobs with Dependencies. *25th ACM Symp. on Theory of Computing,* 642-651.

[15] W. Hoeffding(1956): On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics,* 27:713-721.

[16] T. Hagerup and C. Rub (1989): A Guided Tour of Chernoff Bounds. *Inf. Proc. Let.,* 305-308.

[17] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S.-W. Yang, R. Zak (1992): The network architecture of the connection machine CM-5. *4th ACM Symp. on Parallel Algorithms and Architectures,* 272-285.

[18] D. Shmoys, J. Wein and D. Williamson (1991): Scheduling Parallel Machine On-line. *32th IEEE Symp. on Foundations of Computer Science,* 305-308.

[19] J. Turek, W. Ludwig, J.L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelsohn, P.S. Yu (1994): Scheduling parallelizable tasks to minimize average response time. *6th ACM Symp. on Parallel Algorithms and Architectures.*

SPAA'96, Padua, Italy
© 1996 ACM 0-89791-809-6/96/06 ..$3.50