The Load Distribution Problem in a Processor Ring

Francis C.M. Lau[*]
Department of Computer Science and Information Systems
The University of Hong Kong, Hong Kong

September 1998, August 2001

Abstract

Given a global picture of the system load and the average load, the *load distribution problem* is to find a suitable schedule, consisting of the amount of excess load to transfer along every edge, so that the system load can be balanced in minimal time by executing the schedule. We study this problem for the ring topology. We discuss some existing algorithms, show how they fall short of being able to generate optimal schedules, and present a simple algorithm that would generate an optimal schedule for any given system load instance. This simple algorithm relies on an existing algorithm to create a search window in which the optimal solution is to be found.

Keywords: interconnection networks, load balancing, load distribution, multicomputers, parallel algorithms, performance analysis, scheduling.

1 Introduction

We consider the problem of dynamic load balancing in distributed-memory parallel computers. One systematic approach to the problem is to divide the load balancing procedure into the following phases: load measurement, calculation of load average, generation of load distribution schedule, and load redistribution [8, 6, 1]. In the load measurement phase, the load in a processor is measured and captured in some abstract load index (usually the number of units of workload in the processor); these abstract loads across all the processors are then used in a computation of the average load in the second phase; in the third phase, based on the average load, a set of instructions are generated which dictate how much load each processor should give away or receive along each of its links; these load distribution instructions collectively constitute what we refer to as a *load distribution schedule*. A good schedule would lead to an efficient redistribution of load in the final phase. The *load distribution problem* can be stated as: Given the current load situation (the set of all abstract load indices) and the average load, find a load distribution schedule so that the load redistribution phase would take minimum time to complete.

Our primary objective is to find the optimal schedule. Optimizing the time it takes (by an algorithm) to find this schedule is secondary. Nonetheless, as shown in the experimentation section, the algorithm we propose for finding optimal schedules is reasonably efficient.

[*]Correspondence: F.C.M. Lau, Department of Computer Science and Information Systems, The University of Hong Kong, Hong Kong / Email: fcmlau@csis.hku.hk / Fax: (+852) 2559 8447

There are basically two approaches to computing the average load: decentralized and centralized. The GDE (Generalized Dimension Exchange) method [7, 8] is an example of the decentralized approach, and the DDE (Direct Dimension Exchange) method [6] is an example of the centralized approach. In terms of time performance, the centralized approach has some clear advantages because the computation can be carried out in a single processor, thus saving the cost of much message passing which is characteristic of the decentralized approach. But the decentralized approach would be more reliable. The GDE method actually does not compute the average explicitly; with no knowledge of the load average, it starts constructing the load distribution schedule in some iterative fashion immediately after the load measurement phase. By contrast, the DDE method explicitly computes the average load. This paper is more in line with the latter—we assume that the average load is available as an input parameter for the generation of the load distribution schedule.

We show that an algorithm proposed in the DDE paper provides no guarantee on the time performance of the load distribution schedules generated by the algorithm [6]. A good load balancing procedure really cannot do without an efficient load distribution schedule because the redistribution of load via messages could account for a substantial portion of the overall load balancing time. We propose an algorithm that would solve the load distribution problem by generating an optimal schedule for any given system load instance.

An example of a distribution schedule is shown in Fig. 1 where six processors are connected into a ring. Each processor is identified by the number of units of workload it has (its load index). The subscript indicates the processor's position within the ring. Let v_i be the processor at position i. For convenience v_1 is drawn twice (one in brackets) so that v_6 appears to be its left-neighbor. The box around v_6 will be explained later. Above every link is an integer corresponding to the amount of load to be sent from one processor to the next. For example, v_1 has to send two units of load to v_2, and three units to v_6. These integers above the links collectively form the distribution schedule. This schedule takes at least two timesteps to execute

The schedule:

$$7_1 \xrightarrow{2} 0_2 \overset{0}{--} 3_3 \xrightarrow{1} 1_4 \overset{0}{--} 1_5 \xleftarrow{1} \boxed{0_6} \xleftarrow{3} (7_1)$$

$$\Downarrow$$

After execution of the schedule:

$$2_1 - -2_2 - -2_3 - -2_4 - -2_5 - -2_6 - -(2_1)$$

Figure 1: An example of a load distribution schedule.

because v_6 has to send one unit of load to v_5 but v_6 has zero load initially. It is not difficult to see that this is an optimal schedule in the sense that no other schedules can balance the load here in less than two timesteps.

In this paper, we concentrate on the ring topology which is a building block for many other more complex structures including the torus. The work presented here should serve as a good foundation on which to carry on the study of the problem for other structures. We assume the following system model.

- It takes one time unit for a processor to send a message to one of its direct neighbors.

- Messages are delivered using the store-and-forward mode.

- A message is large enough to carry any amount of workload being transferred according to the distribution schedule.
- A processor operates in the all-port mode (*i.e.*, all of its links can be active sending or receiving simultaneously).

This model is the simplest and least-restrictive. It is not unrealistic, especially for those applications where workload can be compactly encoded. For these applications, messages carrying various amounts of workload during the redistribution phase would be sufficiently small in size to justify the message-size assumption above. On the other hand, if messages are small, the store-and-forward mode would differ very little from the wormhole mode because of the network latency and startup times. But even for this simple model, the load distribution problem is non-trivial. A future extension of the present work could put a limit on the size of a message, like what is done in gossiping research [2]. When a message can carry at most one unit of load at a time, the problem degenerates into the *token distribution problem* (TDP) [5]. Many solutions exist for the TDP, such as the one by JáJá and Ryu [3], which fits into our model of first computing the average and then redistributing the load.

Section 2 defines the problem and introduces the notations. Section 3 presents two possible modes of operation in which a schedule can be executed. Section 4 studies two existing algorithms. Section 5 proposes a simple algorithm that can generate optimal schedules. Section 6 discusses the results from experiments we carried out for the algorithms presented in the previous sections.

2 Preliminaries

Refer to Fig. 1. The collection of load indices constitute the (initial) *load instance*, an instance of the *load distribution problem*; the set of integers above the edges, which dictates the amount of load to be transferred along every edge, is the *load distribution schedule* (or simply *schedule*). The schedule is a solution to the load distribution problem. After the execution of the schedule, a balanced system load emerges.

All processor indices are from the cyclic set $\{1, 2, \ldots, n\}$, where n is the number of nodes. For convenience, we assume that a processor index is always in modulo.[1] We denote the initial load in v_i by l_i, and the excess load to be transferred between v_i and v_{i+1} by s_i. Define $L^+(i)$ to be the amount of load that v_i will receive from its neighbor(s), and $L^-(i)$ the amount it will give away. For example, in the schedule shown in Fig. 1, $L^+(1) = 0, L^-(1) = 2 + 3 = 5$ and $L^+(6) = 3, L^-(6) = 1$.

We call a node a *negative* node if it has some excess load to send to its left-neighbor but not its right-neighbor; a *positive* node if it has some excess load to send to its right-neighbor but not its left-neighbor; or a *rich* node if its excess load is to be sent to both its left- and right-neighbor. Note that a negative or positive node is not necessarily an overloaded load in the given load instance; the excess load it will be sending could actually be in transit from some node to some destination node. A rich node is definitely an overloaded node. Referring again to Fig. 1, v_1 is a rich node, v_3 is a positive node, and v_6 is a negative node.

A schedule $\{s_i\}$ is a valid schedule if and only if by performing the operation $l_i = l_i + L^+(i) - L^-(i)$, $l_i = l_{avg}$, for all i, where l_{avg} is the average load.[2] Applying the above operation to the schedule in Fig. 1,

[1] That is, v_i is actually $v_{i \bmod n}$, if $i \bmod n \neq 0$; otherwise v_n.

[2] For simplicity, we assume that $l_{avg} = \sum l_i/n$ is a whole number throughout this paper; the case where l_{avg} is non-integer is a simple extension.

which is a valid schedule, we have $l_i = l_{avg} = 2$ for all i. By this definition of a valid schedule, there are infinitely many valid schedules for any load instance. The proof of the following lemma is trivial.

Lemma 1 *For a load instance, if $\{s_i\}$ is a valid schedule, then $\{s_i + h\}$ is also a valid schedule, where h is any integer.*

For example, letting $h = 3$ and applying to the schedule in Fig. 1, we have a new schedule as shown in Fig. 2. Nevertheless, in a set of infinitely many valid schedules, only a finite number of them can achieve

$$7_1 \xrightarrow{5} \boxed{0_2} \xrightarrow{3} \boxed{3_3} \xrightarrow{4} \boxed{1_4} \xrightarrow{3} \boxed{1_5} \xrightarrow{2} 0_6 \overset{0}{--} (7_1)$$

Figure 2: Another valid schedule.

optimal time when executed. The objective of this paper is to propose an algorithm that would generate one of these time-optimal schedules. The proposed algorithm would start off with some initial schedule $\{s_i\}$, and then determine an h value so that $\{s_i + h\}$ is optimal.

We use a further characteristic to classify the nodes. A node v_i is "in deficit" if $l_i < L^-(i)$—we call such a node a *red* node; all other nodes are *green* nodes. In Fig. 1, v_6 is a red node, and the others are green nodes. In Fig. 2, v_2, v_3, v_4, v_5 are red nodes, v_1, v_6 are green nodes. We put a red node in a box for easy identification. A red node is either a positive or a negative node, but cannot be both. A red node is not something to be welcome in a load instance because it does not have enough excess load initially to quickly send away to its neighbor, implying there could be some delay being incurred, as we will see in the next section.

3 Single- and Multi-send Modes

A schedule can be executed in either the *single-send* mode or the *multi-send* mode. In the single-send mode of operation, each node will receive at most one message and will send at most one message along an incident edge during the entire execution of the schedule. As a result, a red node must wait until it has received a message (carrying some load) from its neighbor (and the node turns green) before it can send its own. The worst scenario is when red nodes are clustered together in a load instance, forming "chains" of red nodes. In such a chain, the waiting is compounded. For example, for the schedule in Fig. 1, it takes two timesteps to execute the schedule: one timestep for the only red node, v_6, in the initial load instance to turn green, and another timestep to let v_6 send its message carrying one unit of load. Whereas for the schedule in Fig. 2, five timesteps are needed to complete the execution because of the chain of red nodes, $v_2, \cdots, v_5$: v_5 has to wait for v_4 to send it some load, v_4 has to wait for v_3, and so on. Denote the number of timesteps to execute a schedule S in the single-send mode by $T_s(S)$. We have the following.

Proposition 1 *Given a schedule to be executed in the single-send mode, $T_s = 1+$ the length of the longest chain of red nodes.*

Therefore, $T_s(S)$ depends on the length of the longest chain of red nodes, and not the total number of red nodes in a load instance. For example, Fig. 3 shows two schedules for the same load instance, where the

$$\begin{array}{rl}(a): & 9 \xrightarrow{2} 1 \xrightarrow{1} 3 \xrightarrow{2} 0 \overset{0}{--} 2 \overset{0}{--} 1 \xleftarrow{1} \boxed{0} \xleftarrow{3} \boxed{0} \xleftarrow{5} (9) \\ (b): & 9 \xrightarrow{3} \boxed{1} \xrightarrow{2} 3 \xrightarrow{3} \boxed{0} \xrightarrow{1} 2 \xrightarrow{1} 1 \overset{0}{--} 0 \xleftarrow{2} \boxed{0} \xleftarrow{4} (9)\end{array}$$

Figure 3: Two schedules for the same load instance: (a) with two red nodes, but $T_s = 3$; (b) with three red nodes, but $T_s = 2$.

one that has more red nodes (Fig. 3(b)) actually would finish faster than the one that has fewer red nodes (Fig. 3(a)), the reason being the longest chain in the latter is longer than that in the former.

The single-send mode is easy to implement and the number of messages that are sent is minimal. An alternative to the single-send mode is the *multi-send* mode in which a red node would send away all it has in every timestep until it has sent enough of what the schedule requires. This is a "greedy" mode of operation, and the time to execute a schedule in this mode is expected to be better than the previous mode. For the schedule in Fig. 2, it takes three timesteps, instead of the previous five timesteps, if operating in the multi-send mode. A trace of the execution is shown in Fig. 4.

$$\begin{array}{rcl}t = 0 & : & \mathbf{7_1} \xrightarrow{\mathbf{5}} \boxed{0_2} \xrightarrow{3} \boxed{3_3} \xrightarrow{4} \boxed{1_4} \xrightarrow{3} \boxed{1_5} \xrightarrow{2} 0_6 \overset{0}{--} (7_1) \\ t = 1 & : & 2_1 \overset{0}{--} \mathbf{5_2} \xrightarrow{\mathbf{3}} \boxed{0_3} \xrightarrow{1} 3_4 \xrightarrow{2} 1_5 \xrightarrow{1} 1_6 \overset{0}{--} (2_1) \\ t = 2 & : & 2_1 \overset{0}{--} 2_2 \overset{0}{--} \mathbf{3_3} \xrightarrow{\mathbf{1}} 1_4 \overset{0}{--} 2_5 \overset{0}{--} 2_6 \overset{0}{--} (2_1) \\ t = 3 & : & 2_1 \overset{0}{--} 2_2 \overset{0}{--} 2_3 \overset{0}{--} \mathbf{2_4} \overset{0}{--} 2_5 \overset{0}{--} 2_6 \overset{0}{--} (2_1)\end{array}$$

Figure 4: An execution in the multi-send mode.

Given a schedule S to execute in multi-send mode, we use $T_m(S)$ to denote the number of timesteps needed for the execution. Studying what goes on in Fig. 4, we notice that $T_m = 3$ comes from the following "thread" of actions, which we indicate using bold type in the figure: v_1 sends five units to v_2 ($t = 0$), followed by v_2 sending three units to v_3 ($t = 1$), followed by v_3 sending one unit to v_4 ($t = 2$). We say that this is a thread of length three, involving one green node (v_1) and two red nodes (v_2, v_3) actively sending some load. Informally, a thread is a string of consecutive nodes leading to some s_i so that the collective effort of these nodes would be just enough to cover s_i. There are other threads in the schedule, but they are shorter than this one. For example, there is a thread from v_4 to v_6 which in two steps would complete the transfer (two units) required by s_5. Formally, a thread is defined as follows.

Definition 1 *When operating in the multi-send mode, a thread for a positive red node v_{k_m} is the shortest string of consecutive nodes $v_{k_1}, v_{k_2}, \ldots, v_{k_m}$—where $k_1 < k_2 < \cdots < k_m$, v_{k_1} is a green or a red node, and $v_{k_2}, \ldots, v_{k_m}$ are all red nodes—such that $s_{k_m} \leq l_{k_1} + l_{k_2} + \cdots + l_{k_{m-1}}$ if v_{k_1} is a red node, or $s_{k_m} \leq s_{k_1} + l_{k_2} + \cdots + l_{k_{m-1}}$ if v_{k_1} is a green node. A thread for a negative red node is defined similarly.*

Obviously, the minimum length of a thread is 2.

Proposition 2 *Given a schedule to be executed in the multi-send mode, T_m = the length of the longest thread.*

Fig. 5 shows a larger example, $n = 10$, in which we highlight the chains and two of the threads. The thread on the left hand side is in fact the longest thread, and hence $T_m = 4$. $T_s = 4$, for the single-send mode, because the longest chain consists of three nodes. By Definition 1 and Propositions 1 and 2, it is easy to see

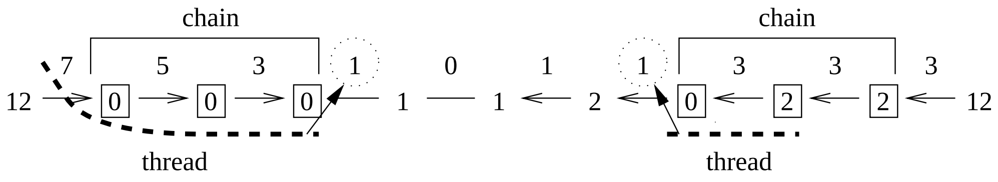

Figure 5: Chains and threads.

that the following is true.

Proposition 3 *Given a schedule S for any load instance, $T_s(S) \geq T_m(S)$.*

Therefore, it is confirmed that the single-send mode cannot be better than the multi-send mode in terms of time performance. The multi-send mode, however, has its disadvantages: it sends more messages than the single-send mode, and the termination procedure would likely be more complicated because a series of messages as opposed to one single message is being sent over an edge.

4 Some Existing Algorithms

The schedule in Fig. 2 actually can be generated using a rather intuitive algorithm which treats the ring as a linear array (*i.e.*, ignoring the wraparound link between v_n and v_1) [6]. The algorithm is shown in Fig. 6. We refer to this as the Linear algorithm. By not using all the edges in a ring, the Linear algorithm fails to make use of all the available bandwidth. But as we will show in the following, the schedules generated by the Linear algorithm can still be considered reasonable. Before we define reasonable schedule, we introduce the measure of *traffic*, which is equal to $\sum s_i$. Obviously, a schedule that generates less traffic is more preferable than one that generates more. But in practice, when there is a conflict between optimizing number of timesteps and optimizing traffic, the former is usually given priority. In fact, it is often the case that the reason for optimizing traffic is to minimize the time. The following lemma defines “unreasonable”

```
for $i = 1$ to $n - 1$:
    $s_i = \sum_1^i l_i - l_{avg} \times i$
```

Figure 6: The Linear algorithm.

schedule.

Lemma 2 *A schedule $\{s_i\}$ where $s_i > 0$ for all i, or where $s_i < 0$ for all i, cannot be an optimal schedule in terms of both time and traffic.*

Proof: Consider the case of $s_i > 0$ for all i (the case of $s_i < 0$ is similar). Let $h = \min(s_i)$. We claim that $\{s_i - h\}$ is a better solution. Let the given schedule be S, and the latter schedule S_h. In terms of

traffic, S_h puts out $n \times h$ units less than that of S. In terms of time, in the single-send mode, a red node in S could become a green node in S_h after its s_i is reduced by h, but a green node could never turn red. In the multi-send mode, a thread in S could become shortened in S_h because, referring to Definition 1, s_{k_m} is reduced by h, but none of $l_{k_1}, l_{k_2}, \ldots, l_{k_{m-1}}$ is reduced. $\Box$
Here is an example of such an unreasonable schedule:

$$\boxed{1} \xrightarrow{10} \boxed{1} \xrightarrow{10} \boxed{1} \xrightarrow{10} \boxed{1} \xrightarrow{10} \boxed{1} \xrightarrow{10} \boxed{1} \xrightarrow{10} (\,\boxed{1}\,)$$

The load instance is already balanced, but the schedule insists on having each node send 10 units to its neighbor. If running in the single-send mode, this schedule would deadlock, because none of the nodes has enough load to proceed! If running in the multi-send mode, this schedule would take 10 timesteps to execute because the longest thread here would span 10 nodes (some nodes being spanned twice).

A schedule generated by the Linear algorithm does not belong to the category of unreasonable schedules as defined by Lemma 2 because at least one of the s_i's, s_n, is equal to zero. According to the proof of Lemma 2, $\{s_i - h\}$ is a better schedule, but since $h = \min(s_i) = 0$ for any schedule generated by the Linear algorithm, a better schedule is the original schedule itself. We say that a schedule generated by the Linear algorithm is a reasonable schedule.

The Linear algorithm, however, provides on guarantee on the optimality of the execution time of the schedules it produces. Wu and Shu have given an algorithm which is based on the Linear algorithm and which minimizes the traffic [6]. We refer to this as the Traffic algorithm. The idea of this algorithm comes from a minimum-cost flow algorithm described in [4], which can be explained pictorially, using for example Fig. 7.[3] In the following discussion, when we say that $\{s_i\}$ is adjusted by h, we mean the new schedule $\{s_i - h\}$. Fig. 7 shows a schedule generated by the Linear algorithm being visualized as a sorted "bar chart". The bars that are upright (above the horizontal axis) correspond to positive s_i's—those emanating from positive nodes; the bars that are upside-down (below the horizontal axis) correspond to negative s_i's—those emanating from negative nodes; and s_i's that are zero appear as a small dot. The s_i's are displayed in sorted order with negative s_i's being treated as negative numbers in the sorting (*e.g.*, $s_5 = -1$ in Fig. 1). We denote the sorted set by $\{\bar{s}_i\}$, where $\bar{s}_1$ is the largest s_i, $\bar{s}_2$ the second largest s_i, and so on. If the number of positive $\bar{s}_i$'s, n_p, is greater than $\lfloor n/2 \rfloor$, the Traffic algorithm would apply an adjustment of $h = \bar{s}_{\lceil n/2 \rceil}$ to the schedule; similarly, for $n_n > \lfloor n/2 \rfloor$, where n_n is the number of negative $\bar{s}_i$'s, the Traffic algorithm would set h to $\bar{s}_{\lceil (n+1)/2 \rceil}$.

Lemma 3 *The Traffic algorithm yields a schedule that is traffic-optimal.*

Proof: Consider first the case of even n where $n_p > n/2$, as shown Fig. 7(a1). We use a vertical dashed line to divide $\{\bar{s}_i\}$ into the left and the right half. By adjusting $\{\bar{s}_i\}$ by $h = \bar{s}_{n/2}$ (represented by the dotted line in Fig. 7(a1), and the result is shown in Fig. 7(a2)), the traffic that is reduced in the left half minus the traffic that is increased in the right half is equal to the sum of the positive $\bar{s}_i$'s that are on the right half in the unadjusted schedule. This saving in traffic is the maximum possible with any value of h. The case of $n_n > n/2$ is symmetric to this case.

For the case of odd n where $n_p > \lfloor n/2 \rfloor$, as shown in Fig. 7(b1), the saving in traffic due to the adjustment by $h = \bar{s}_{\lceil n/2 \rceil}$ is equal to the sum of the positive $\bar{s}_i$'s that are on the right half plus the $\bar{s}_i$ that is

[3]A mathematical proof can be found in [6].

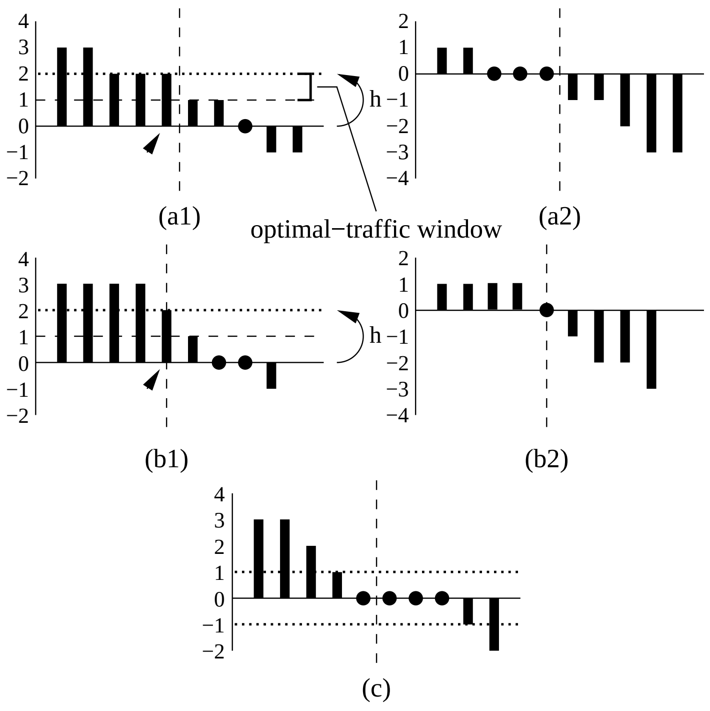

Figure 7: The idea of the Traffic algorithm.

in the middle. This saving in traffic is the maximum possible with any value of h. The case of $n_n > \lfloor n/2 \rfloor$ is symmetric.

For the case of neither $n_p > \lfloor n/2 \rfloor$ nor $n_n > \lfloor n/2 \rfloor$, as shown in Fig. 7(c), an adjustment by any h value will not improve the traffic. □

Unfortunately, traffic-optimality does not imply time-optimality. Consider the following example, where $n = 10$. The Traffic algorithm sets h to $\bar{s}_{s/2} = 2$.

$$\begin{array}{ll} \text{Linear :} & 5 \xrightarrow{3} \boxed{1} \xrightarrow{2} 1 \xrightarrow{1} 3 \xrightarrow{2} 3 \xrightarrow{3} \boxed{1} \xrightarrow{2} 0 \overset{0}{--} 1 \xleftarrow{1} 2 \xleftarrow{1} 3 \overset{0}{--} (5) \\ \text{Traffic :} & 5 \xrightarrow{1} 1 \overset{0}{--} 1 \xleftarrow{1} 3 \overset{0}{--} 3 \xrightarrow{1} 1 \overset{0}{--} 0 \xleftarrow{2} \boxed{1} \xleftarrow{3} \boxed{2} \xleftarrow{3} 3 \xleftarrow{2} (5) \end{array}$$

The total traffic for the Linear schedule and the Traffic schedule is 15 and 13 respectively. But T_s for the Linear schedule is 2, which is better than that for the Traffic schedule, which is 3. T_m for either schedule is 2. The best schedule (in both traffic and time) turns out to be the following, with $T_s = T_m = 1$.

$$5 \xrightarrow{2} 1 \xrightarrow{1} 1 \overset{0}{--} 3 \xrightarrow{1} 3 \xrightarrow{2} 1 \xrightarrow{1} 0 \xleftarrow{1} 1 \xleftarrow{2} 2 \xleftarrow{2} 3 \xleftarrow{1} (5)$$

This schedule could have been generated by the Traffic algorithm if the algorithm had set $h = \bar{s}_{n/2+1} = 1$ instead of $\bar{s}_{n/2}$. As a matter of fact, the solution to the even-n case is not unique. Refer to Fig. 7(a1) again—it is not difficult to see that any h value between the middle two $\bar{s}_i$'s (*i.e.*, $\bar{s}_{n/2} \leq h \leq \bar{s}_{n/2+1}$) would give rise to a traffic-optimal schedule. We refer to this range of h values as the *optimal-traffic window*. For odd n, on the other hand, the solution generated by the Traffic algorithm is unique.

Although it happens that optimal schedule in the above example could have come from setting an h value which is within the optimal-traffic window, it is not true in general that time-optimality implies traffic-optimality. The following is an example of an time-optimal schedule, whose h value is outside of the optimal-traffic window.

$$\begin{array}{rl} \text{Optimal :} & 34 \xleftarrow{22} 40 \xleftarrow{25} 90 \xrightarrow{22} 40 \xrightarrow{19} 50 \xrightarrow{26} 60 \xrightarrow{43} 30 \xrightarrow{30} 0 \xleftarrow{13} (34) \\ \text{Traffic+ :} & 34 \xleftarrow{41} \boxed{40} \xleftarrow{44} 90 \xrightarrow{3} 40 \overset{0}{--} 50 \xrightarrow{7} 60 \xrightarrow{24} 30 \xrightarrow{11} 0 \xleftarrow{32} (34) \end{array}$$

The optimal-traffic window for this problem is $[32, 35]$. The Traffic+ schedule was generated by setting $h = 32$, as opposed to setting $h = 35$ if the original Traffic algorithm had been used; the latter would result in more red nodes in the schedule. The Optimal schedule, however, was from setting $h = 13$, which is outside of the window. This schedule has a total traffic of 200, whereas that of the Traffic+ schedule is 162.

The following summarizes our findings.

Lemma 4 *For a random load instance, time-optimality and traffic-optimality are not equivalent.*

The Linear algorithm tries to generate schedules that are reasonable, and the Traffic algorithm tries to optimize the generated traffic, but neither of them aims at producing a schedule that is time-optimal. For certain load instances, optimizing the traffic could make the time worse, as we will see in Section 6.

As optimizing time rather than traffic is the primary objective, we present in the following an algorithm for finding the time-optimal schedule.

5 An Algorithm for Finding Time-optimal Schedules

Given an initial schedule, say a Linear schedule, to find the time-optimal schedule means adjusting the schedule with a suitable h value. This h value represents a good balance between two kinds of chains: negative chains and positive chains. A negative chain is one that contains only negative nodes, and a positive chain contains only positive nodes. A positive h value (*i.e.*, $\{s_i - h\}$) might shorten the length of existing positive chains (possibly destroying some) and at the same time extend the length of existing negative chains (possibly creating some new ones).

Define the *deficit* of a red node, say v_i, to be $(L^-(i) - l_i)$ if v_i is positive, and $-(L^-(i) - l_i)$ if v_i is negative. The deficits form a set. Let d_{max} and d_{min} be the largest and the smallest element in the set, respectively. If there is no red node, then the set is not defined, and the initial schedule is an optimal schedule; if there is only one red node, then $d_{max} = d_{min}$.

Lemma 5 *The h value leading to the time-optimal schedule lies within the window $[d_{max}, d_{min}]$ if d_{max} and d_{min} are defined.*

Proof: Without loss of generality, suppose that d_{max} is positive (Fig. 8). If setting $h = d_{max}$, then all the positive red nodes would become green. Therefore, setting $h > d_{max}$ would not turn any more positive red nodes into green nodes, but might turn some green negative nodes and/or zero nodes into red nodes—*i.e.*, extending some of the existing negative chains. By symmetry, if d_{min} is negative (Fig. 8(b)), h cannot be smaller than d_{min}. If d_{min} is positive (Fig. 8(a)), then those red nodes with minimum deficit would turn green when h is set to d_{min}. Setting $h < d_{min}$ would not produce any more green nodes but might change some of those nodes with minimum deficit back to red.

Therefore, setting h to be outside of the said window might extend existing chains and/or creating new chains, and increase existing deficits, and hence T_s cannot be better. For T_m, refer to Definition 1 and consider positive threads without loss of generality. A thread is of either the form $s_{k_m} \leq l_{k_1} + l_{k_2} + \cdots + l_{k_{m-1}}$ if v_{k_1} is a red node, or the form $s_{k_m} \leq s_{k_1} + l_{k_2} + \cdots + l_{k_{m-1}}$ if v_{k_1} is a green node. For either form, the l's would not change. By choosing an h outside of the window, either s_{k_m} would become larger (for the first form above) or v_{k_1} would become a red node (for the second form). If s_{k_m} becomes larger, then it might take some extra l's to satisfy the condition—hence, a longer thread. If v_{k_1} becomes a red node, then it becomes a case of the first form; otherwise, if v_{k_1} remains as a green node, the length of the thread would not change because both s_{k_m} and s_{k_1} would be changed by the same amount by the choice of h. $\square$

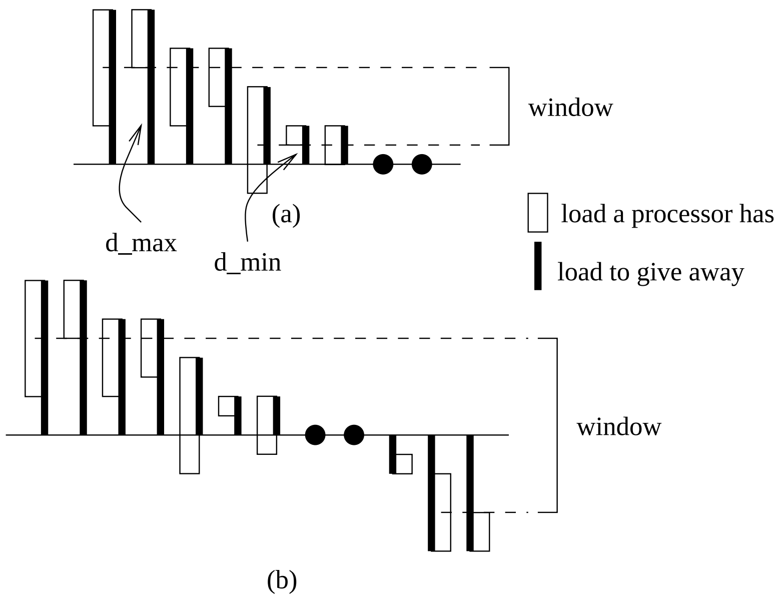

Figure 8: Finding the time-optimal schedule.

The above lemma lets us confine the searching for a solution to a smaller search space. The algorithm for finding an optimal schedule for the single-send mode is given in Fig. 9—we refer to this as the Optimal schedule. For the multi-send mode, simply substitute T_m for T_s in the algorithm.

```
generate a schedule $\{s_i\}$ using the Traffic algorithm;
$h = 0$;
if $d_{max}$ and $d_{min}$ are defined then
    compute $T_s$ for every $h$ in $[d_{max}, d_{min}]$;
    choose the $h$ whose $T_s$ is minimum;
the optimal schedule is $\{s_i - h\}$;
```

Figure 9: The Optimal algorithm.

The following lemma points out that the two times, T_s and T_m, are not related in general; minimizing one does not necessarily minimize the other.

Lemma 6 *A T_s-optimal schedule is not necessarily a T_m-optimal schedule, and vice versa.*

Proof: By example. Consider the following load instance ($n = 10$) and its T_s-optimal and T_m-optimal schedule.

$$\begin{array}{rl} T_s\text{-optimal}: & 10 \xrightarrow{3} \boxed{1} \xrightarrow{2} 3 \xrightarrow{3} \boxed{1} \xrightarrow{2} 2 \xrightarrow{2} 2 \xrightarrow{2} 0 \overset{0}{--} 0_8 \xleftarrow{\mathbf{2}} \boxed{0} \xleftarrow{4} \boxed{1} \xleftarrow{5} (10) \\ T_m\text{-optimal}: & 10 \xrightarrow{4} \boxed{1} \xrightarrow{3} \boxed{3} \xrightarrow{4} \boxed{1} \xrightarrow{3} \boxed{2} \xrightarrow{3} \boxed{2} \xrightarrow{3} \boxed{0} \xrightarrow{1} 0 \xleftarrow{1} \boxed{0} \xleftarrow{3} \boxed{1} \xleftarrow{4} (10) \end{array}$$

For the T_s-optimal schedule, $T_s = 3$, but $T_m = 3$ (consider s_8) is non-optimal. For the T_m-optimal schedule, $T_m = 2$ (because the longest thread is of length 2) but $T_s = 9$ is non-optimal. □

In the Optimal algorithm, we choose the Traffic algorithm instead of the simpler Linear algorithm to be used in generating the initial schedule which contains the search window. The reason is that the Linear algorithm could generate very large search windows for highly unbalanced load instances, such as the one shown in Fig. 10. The Optimal algorithm requires going through $\Delta = d_{max} - d_{min} + 1$ steps, each of which entailing examining n elements to determine the longest chain or thread. For the one in Figure 10, Δ is very close to the total number of load units in the system: the total load is 100 which is all concentrated in one node; $d_{max} = 98$ and $d_{min} = 1$, and $\Delta = 98$. Therefore, if we had used the Linear algorithm for the initial schedule in the Optimal algorithm, the worst-case complexity of the Optimal algorithm would be $\mathcal{O}(n \times L)$, where L is the total load of the system. Now, using the Traffic algorithm, at least extreme cases like the one just discussed would be much less of a problem. In fact, if we apply the Traffic algorithm to the load instance in Fig. 10, we get an optimal schedule right away because h would be equal to 45 and so the node holding 100 units would send 44 through one edge and 45 through another edge.

$$100 \xrightarrow{99} \boxed{0} \xrightarrow{98} \cdots \boxed{0} \xrightarrow{1} \boxed{0} \overset{0}{--} (100)$$

Figure 10: Worst scenario.

6 Simulation Experiments

We implemented the algorithms (the Linear algorithm, the Traffic algorithm, and the Optimal algorithm) presented in previous sections and conducted a number of experiments in order to answer the following questions. We refer to a schedule generated by the Linear algorithm as a Linear schedule, one by the Traffic algorithm a Traffic schedule, and a time-optimal schedule by the Optimal algorithm an Optimal schedule. In the following when we say optimal, we mean time-optimality.

1. What is the probability that a Linear schedule or a Traffic schedule is optimal? (If the probability is high, then the Linear algorithm or the Traffic algorithm perhaps is acceptable for real implementation.) For those non-optimal Linear or Traffic schedules, by how much are they worse than the optimal schedule?

2. Will the single-send mode be close enough in performance to the multi-send mode? (If yes, then the single-send mode being somewhat easier to implement should be preferred in real implementation.)

3. What is the time cost (in real seconds) of running the Optimal algorithm?

We considered five sizes of a ring, $n = 4, 10, 20, 30, 50$.[4] For each of these rings we generated $50,000$ random load instances, and for each instance, we applied the three algorithms to generate three schedules for comparison. The individual load assigned to a processor in a load instance ranges from 0 to 100 units. We measured a schedule's execution times—*i.e.*, T_s or T_m, or both.

The results from first set of experiments answer Question 1 above. Table 1 summarizes the results for the single-send mode, and Table 2 summarizes the results for the multi-send mode. The first column of the tables (Linear=Opt) corresponds to the number of Linear schedules, out of 50,000 schedules, whose execution time turned out to be optimal; the second column (Traffic=Opt) the number of Traffic schedules whose execution time turned out to be optimal; the third column (All) the number of times for which all three algorithms yield an optimal schedule; the fourth column (Opt) the number of times in which the schedule generated by the Optimal algorithm was the only optimal one, and we include the amount of extra traffic (+traffic) that the Optimal schedule would incur using the optimal traffic amount as a base; and the last column (%worse) corresponds to the average amount by which a non-optimal schedule (Linear or Traffic) is worse than the optimal schedule—a 100% means that the non-optimal schedule uses two times the optimal time to execute. The relationship between these various parameters is as follows.

$$\text{Opt} = 100\% - ([\text{Linear=Opt}] + [\text{Traffic=Opt}] - \text{All})$$

The following can be easily observed from the figures in the tables.

- The Linear algorithm is worse than the Traffic algorithm in terms of the probability of generating an optimal schedule, especially for large rings.
- Where the Linear algorithm would generate an optimal schedule but the Traffic algorithm would not (which is equal to [Linear=Opt] $-$ All) is a rare event.
- The amount of extra time (on top of the optimal time) that the non-optimal schedules would spend in their execution is quite substantial, especially for small rings in the single-send mode.
- The performance differences between the algorithms are less acute in the case of using the multi-send mode.
- The Optimal schedules generate only a little bit more traffic than the corresponding Traffic schedules.

The conclusion is that except for very small rings, the Optimal algorithm is the only reliable algorithm for generating optimal schedules, and that the Optimal schedule is substantially better than the non-optimal schedules generated by the other two algorithms.

The second set of experiments we conducted was for comparing the single-send and the multi-send mode of operation. The result is summarized in Table 3, which answers Question 2 above. Recall that we have proved that the single-send mode can never be better than the multi-send mode. The %worse column

[4]We picked all even n's (for the sake of easier programming); the behavior of the odd-n cases should be more or less the same.

n	Linear=Opt	Traffic=Opt	All	Opt / +traffic	%worse
50	2607 (5.21%)	17167 (34.33%)	1513 (3.03%)	31739 (63.48%) / 3%	87%
30	4116 (8.23%)	21574 (43.15%)	3090 (6.18%)	27400 (54.80%) / 3%	99%
20	6046 (12.09%)	25802 (51.60%)	5186 (10.37%)	23338 (46.68%) / 4%	109%
10	13664 (27.33%)	34485 (68.97%)	13281 (26.56%)	15132 (30.26%) / 5%	127%
4	36783 (73.57%)	46107 (92.21%)	36783 (73.57%)	3893 (7.79%) / 6%	107%

Table 1: single-send mode.

n	Linear=Opt	Traffic=Opt	All	Opt / +traffic	%worse
50	7271 (14.54%)	24172 (48.34%)	6211 (12.42%)	24768 (49.54%) / 7%	43%
30	10187 (20.37%)	29608 (59.22%)	9399 (18.80%)	19604 (39.21%) / 8%	42%
20	13189 (26.38%)	33782 (67.56%)	12589 (25.18%)	15618 (31.24%) / 10%	43%
10	17156 (34.31%)	39179 (78.36%)	16857 (33.71%)	10522 (21.04%) / 11%	46%
4	36783 (73.57%)	46107 (92.21%)	36783 (73.57%)	3893 (7.79%) / 6%	50%

Table 2: multi-send mode.

corresponds to the amount of extra time the optimal schedule in single-send mode would spend on average when compared with the schedule in multi-send mode; the #equal column corresponds to the number of cases, out of the 50,000 we simulated, in which the two times are equal. For large rings, the multi-send mode is the clear winner, but nonetheless, in those cases where the single-send mode loses out, the single-send mode is not substantially worse: even for the largest ring we tried, $n = 50$, the increase in execution time is only about 13%. Our conclusion is that the single-send mode, which is a simpler method, is viable for real implementation because its performance is not far from that of the multi-send mode.

We carried out the final set of experiments to obtain execution timings of the Optimal algorithm. We present only those for the single-send mode. The timings for the multi-send mode are approximately the same, since they both use the same window to search for an optimal solution for a given load instance. The result is shown in Table 4, which answers Question 3 above. The implemented algorithm begins with a random load instance, executes the Traffic algorithm to create an initial schedule, calculates the deficits, and then tries every h value within the search window. We ran the algorithm on a SPARCserver with a single SPARC chip running at 167 MHz. The measured execution times range from 0.23 milliseconds for small workload (up to 40 units/processor) and small ring ($n = 10$) to 22.2 milliseconds for large workload (up

n	%worse	#equal
4	0%	50000
10	3.79%	33000
20	7.94%	9185
50	13.39%	87

Table 3: Comparing single-send and multi-send mode.

n	max. load / proc.	mean total load	window (Linear)	time (msec)
10	200	1170	79 (95)	1.05
20	200	1860	182 (191)	4.41
50	200	4690	386 (391)	22.2
10	100	470	39 (48)	0.54
20	100	760	90 (95)	2.2
50	100	2090	192 (195)	11.25
10	40	170	16 (19)	0.23
20	40	300	36 (38)	0.95
50	40	770	79 (80)	4.77

Table 4: Runtimes of the Optimal algorithm for the single-send mode.

to 200 units/processor) and large ring ($n = 50$). We also measured the mean window size for each of the runs. For comparison, we include also the window size for using the Linear algorithm instead of the Traffic algorithm for the initial schedule. It appears that the Traffic algorithm as well as the Linear algorithm would generate larger window sizes (relative to the total workload) as the size of the ring increases. For $n = 10$, the window size is equal to about one-third to one-half of the maximum initial load of a processor, whereas for $n = 50$, the window size is twice the maximum initial load. We note also that the Traffic algorithm has only a slight advantage over the Linear algorithm in terms of the search window size.

We claim that the algorithm is reasonably efficient. Consider for instance the case of $n = 20$ in the middle section of Table 4. The load balancing procedure would spend at least $\mathcal{O}(n)$ timesteps in computing the average and then broadcasting the schedule to all the nodes, which could translate easily into several milliseconds or more in a processor ring using state-of-the-art routing hardware (where latencies and startup times are in the order of tens or hundreds of microseconds). Therefore, the 2.2 milliseconds to compute the schedule should not be a major concern. Also, the SPARC processor running at 167 MHz is a rather modest piece of hardware. The overhead thus added to the load balancing procedure should not in any way affect the performance of the application which initiates load balancing only occasionally.

7 Concluding Remarks

We have shown that the problem of finding an optimal schedule for load distribution is non-trivial even for a structure as simple as a ring. Results of our experiments clearly indicate that a non-optimal schedule could be much worse than an optimal schedule in terms of execution time, and that only the proposed Optimal algorithm, among the three algorithms discussed, is reliable as far as producing an optimal schedule is concerned.

The Optimal schedule is based on the Traffic algorithm which is a “reasonable” algorithm. But the Traffic algorithm does not seem to be able to yield a narrow-enough window in which to search for the solution. The size of the window is an important parameter that affects the time spent in executing the Optimal algorithm. Future pursuits could try to replace the Traffic algorithm by a smarter algorithm that would produce a smaller window for searching. For more specific load instances or load instances exhibiting a certain pattern, there might exist methods that could zero in on a solution more directly instead of having to exhaustively search through a window. An example of such a method can be found in [9]. Our solution for the ring does not seem to adaptable to work for other structures that are composed of rings, such as the torus, because in a torus there are many paths that join a pair of nodes (instead of just two in a ring).

Acknowledgement

The author wishes to thank the referees for their constructive comments.

References

[1] Corradi, A., Leonardi, L., and Ambonelli, F. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency* **7** (1999), 22–31.

[2] Gargano, L., Vaccaro, U., and Rescigno, A. Communication complexity of gossiping by packets. *J. of Parallel and Distributed Computing* **45** (1997), 73–81.

[3] JáJá, J. and Ryu, K.W. Load balancing and routing on the hypercube and related networks. *J. of Parallel and Distributed Computing* **14** (1992), 431–435.

[4] Lawler, E.L. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[5] Peleg, D. and Upfal, E. The token distribution problem. *SIAM Journal on Computing* **18** (1989), 229–243.

[6] Wu, M.-Y. and Shu, W. DDE: A modified dimension exchange method for load balancing in k-ary n-cubes. *J. of Parallel and Distributed Computing* **44** (1997), 88–96.

[7] Xu, C.Z. and Lau, F.C.M. The generalized dimension exchange method for load balancing in k-ary n-cubes and variants. *J. of Parallel and Distributed Computing* **24** (1995), 72–85.

[8] Xu, C.Z. and Lau, F.C.M. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, Boston, 1996.

[9] Yau, J.C.K. Efficient solutions for the load distribution problem. M.Phil. thesis, Department of Computer Science and Information Systems, The University of Hong Kong, 1999.