A Simple Algorithm for Optimal Load Balancing on Hypercube Multiprocessors

Yuan-Shin Hwang Gene Eu Jan
Department of Computer Science
National Taiwan Ocean University
Keelung 20224
Taiwan
{shin@cs, b0199@mail}.ntou.edu.tw

Abstract

This paper proposes a simple yet optimal algorithm to distribute loads evenly on multiprocessor computers with hypercube interconnection networks. The algorithm is developed upon the well-known *dimension exchange method*. However, the error accumulation suffered by other algorithms based on the dimension exchange method is avoided by exploiting the notion of *block distributions*, which are commonly deployed for data distributions in parallel programming. The algorithm achieves perfect load balance over P processors with error of 1 and the worst-case time complexity of $O(M \log_2 P)$, where M is the maximum load assigned to each processor initially. Furthermore, perfect load balance is achieved over subcubes as well — once a cube is balanced, if the cube is decomposed into two subcubes by the lowest bit of node addresses, then the difference between the numbers of the total tasks of these subcubes is at most 1.

1 Introduction

Distributing tasks evenly on processors is essential for multiprocessor computers, since load imbalance leads to low utilization of some processors. Load balancing on hypercubes has been studied extensively [1, 2, 5, 6, 7, 9, 10, 11]. The problem is often called the *token distribution problem* [1].

Most load balancing algorithms for hypercube multiprocessor computers are developed based on the dimension exchange method [2, 9]. The main advantage of this approach is that every processor can redistribute tasks to its neighboring processors without the information of global distributions of tasks. However, it can not always reach the perfect load balancing. Specifically, if *error* ξ is defined as the difference between the maximum number of tasks at any processors and the minimum number of tasks at any processors, then on P processors tokens will be balanced with error $\xi = \log_2 P$ in the worst case. In order to balance loads evenly on multiprocessor systems, the information of global load distributions must somehow be available to processors. Techniques have been proposed to gather the global load distributions to achieve the perfect load balancing. However, preprocessing must be applied before tasks are redistributed based on the information [1].

This paper proposes a simple yet optimal algorithm to distribute loads evenly on hypercube multiprocessor computers. The features of this algorithm are twofold:

- It does not require preprocessing to gather the global load distributions in order to achieve the prefect load balancing. The global view of load distributions can be acquired as the load redistribution is proceeded. The notion of *block distributions*, which are commonly deployed for data distributions in parallel programming [3, 4], is exploited to provide the global view of load distributions.
- The algorithm achieves perfect load balance over P processors with error of 1 and the worst-case time complexity of $O(M \log_2 P)$, where M is the maximum load assigned to each processor initially. Furthermore, perfect load balance is achieved over sub-

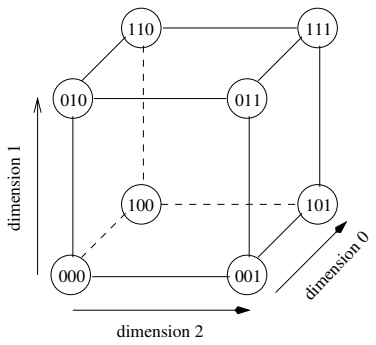


Figure 1: A 3-Dimensional Hypercube

cubes as well. Let H^0_{n-1} and H^1_{n-1} be the (n-1)-dimensional subcubes of an n-dimensional cube H_n, i.e. the lowest bit of the node address of every node is 0 in H^0_{n-1}, and 1 in H^1_{n-1}. Once H_n is balanced, the difference of the numbers of total tasks on H^0_{n-1} and H^1_{n-1} is at most 1. In addition, the same feature of perfect load balancing among subcubes holds for H^0_{n-1} and H^1_{n-1} as well.

The rest of this paper will be organized as follows. Section 2 briefly presents some background information for the load balancing algorithm. Section 3 will outline the algorithm that balances the tasks on a hypercube, and the time complexity of the algorithm will be analyzed in Section 4. Related work will be presented in Section 5, and Section 6 concludes this paper.

2 Background

2.1 Token Distribution Problem

The problem of distributing tasks evenly over multiprocessors is often called the token distribution problem if the following assumptions are taken

- there are no dependences between tasks (i.e. tokens)
- every task takes a unit-time to execute.

Therefore, perfect load balancing is achieved when the error is at most 1 (i.e. $\xi \leq 1$). In other words, each processor will have either $\lceil N/P \rceil$ or $\lfloor N/P \rfloor$ tasks, where N is the total number of tasks and P is the number of processors.

2.2 Hypercubes

An an n-dimension hypercube H_n consists of $P = 2^n$ processors (see Figure 1). Each processor P_m ($0 \leq m < P$) is identified by an n-bit binary address with the form $m \equiv d_0 d_1 \cdots d_i \cdots d_{n-1}$, where $d_i = 0 \,|\, 1$ ($0 \leq i < n$) and d_0 is the most significant bit (MSB). Processor $P_{d_0 d_1 \cdots d_i \cdots d_{n-1}}$ is connected to each processor in the set of processors $\{P_{d_0 d_1 \cdots \overline{d_i} \cdots d_{n-1}} \,|\, 0 \leq i < n\}$, where $\overline{d_i}$ denotes the complement of bit d_i. As a result, processor $P_{d_0 d_1 \cdots d_i \cdots d_{n-1}}$ is connected to processor $P_{d_0 d_1 \cdots \overline{d_i} \cdots d_{n-1}}$ at dimension i.

2.3 Dimension Exchange Method

This method goes through all dimensions, from dimension 0 to dimension $n-1$, and balances loads by redistributing the tasks across the links of each dimension.

```
for i = 0 to n − 1 do
    S1. balance the hypercube across dimension i
end do
```

Before load balancing is performed, each processor of the hypercube can be viewed as a 0-dimensional subcube H_0 with balanced load (i.e. tasks are distributed evenly over processors in the subcube since there is only one processor). After iteration 0 is executed, each pair of 0-dimensional subcubes H^0_0 and H^1_0 that are connected by a link of dimension 0 will be combined into a 1-dimensional subcube H_1. Furthermore, the tasks of H^0_0 and H^1_0 will be redistributed through the link, and as a result H_1 is balanced. The same process will go on until load balancing is performed on all dimensions to form an n-dimensional hypercube with balanced load.

To balance 1-dimensional subcubes H_1 by redistributing tasks of 0-dimensional subcubes H^0_0 and H^1_0 results in balanced 1-dimensional subcubes with error $\xi \leq 1$. In other words, if H^0_0 and H^1_0 have x and y tasks respectively, after redistribution, one of H^0_0 and H^1_0 will have $\lceil (x+y)/2 \rceil$ tasks and the other have $\lfloor (x+y)/2 \rfloor$ tasks. If $x+y$ is odd, the one of the two subcubes has one more task than the other. This extra task is usually called the *loose* task [1], and it can be placed at either of two sub-

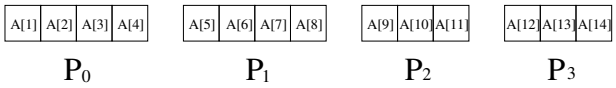


Figure 2: A Block Distribution over 4 Processors

cubes and load balancing is still preserved.

The error will accumulate as a sequence of load balancing operations are performed through all dimensions, as redistributing tasks through every dimension might generate loose tasks. Specifically, after redistributing tasks across links of n dimensions, the hypercube will be balanced with error $\xi \leq n$.

2.4 Block Distributions

In order to perform load balancing on hypercube multiprocessors by the dimension exchange method without accumulating errors across dimensions, this paper exploits the notion of *block distributions*, which are commonly used to specify data distributions of arrays in parallel programming languages such as Fortran D [3] and HPF [4].

Definition 1 BLOCK DISTRIBUTIONS
Let $N(m)$ be the number of array elements on processor P_m $(0 \leq m < P)$. A distribution of N elements over P processors is called a block distribution if

$$N(m) = \begin{cases} \lceil N/P \rceil & \text{if } m < (N \,\%\, P) \\ \lfloor N/P \rfloor & \text{otherwise} \end{cases}$$

where % is the modulo function □

Figure 2 depicts the layout of 14 elements of array A on 4 processors in a block distribution.

The most significant feature of block distributions that is exploited in this paper is that the number of tasks on every processor of a subcube can be obtained analytically, if tasks are distributed over processors in the same way as array elements are partitioned in block distributions. In other words, each processor can easily figure out the numbers of tasks on all other processors according to Definition 1. Furthermore, the total number of tasks N and the number of processors in the subcube are the only pieces of information that every processor needs to know in order to figure out how tasks are distributed over processors.

Enforcing the tasks are partitioned over processors in block distributions in every subcube can greatly simplify the problem of load balancing. Suppose the total numbers of tasks on k-dimensional subcubes H_k^0 and H_k^1 are N_0 and N_1, respectively. The addresses of nodes on H_k^0 and H_k^1 have the form $\overbrace{xxx\cdots x}^{k}0d_{k+1}\cdots d_{n-1}$ and $\overbrace{xxx\cdots x}^{k}1d_{k+1}\cdots d_{n-1}$, respectively. The local addresses of processors in subcubes H_k^0 or H_k^1 are determined by $\overbrace{xxx\cdots x}^{k}$ part of node addresses. When tasks of H_k^0 and H_k^1 are balanced over edges of the dimension k, the only piece of information that H_k^0 and H_k^1 need to exchange before moving tasks is the number of total tasks in its own subcube, i.e. N_0 and N_1 respectively. Every node in H_k^0 can then easily figure out how the tasks are distributed in H_k^1 based on the information N_1: the first $(N_1 \,\%\, 2^k)$ processors in H_k^1 have $\lceil N_1/2^k \rceil$ tasks each and the rest processors have $\lfloor N_1/2^k \rfloor$ tasks each. Similarly, every node in H_k^1 knows exactly how tasks are distributed in H_k^0. Therefore, every node in H_k^0 can use this information to decide how to move tasks between its corresponding node in H_k^1, and vice versa.

3 Algorithm

This algorithm is developed upon the dimension exchange method. It goes through all the dimensions, from dimension 0 to dimension $n-1$, and distributes the tasks across the edges of every dimension. It first balances the load across the edges of dimension 0 to form subcubes of size 2^1 with balanced load. Similarly, the tasks of the subcubes of size 2^1 will then be distributed across the edges of dimension 1 to create balanced subcubes of size 2^2. This process will go on until the whole cube (i.e. 2^n) is balanced.

The algorithm works from dimension 0 to dimension $n-1$:

```
for i = 0 to n − 1 do
    S1. balance the hypercube across dimension i
    S2. redistribute loose tasks in each subcube
        to a block distribution
end do
```

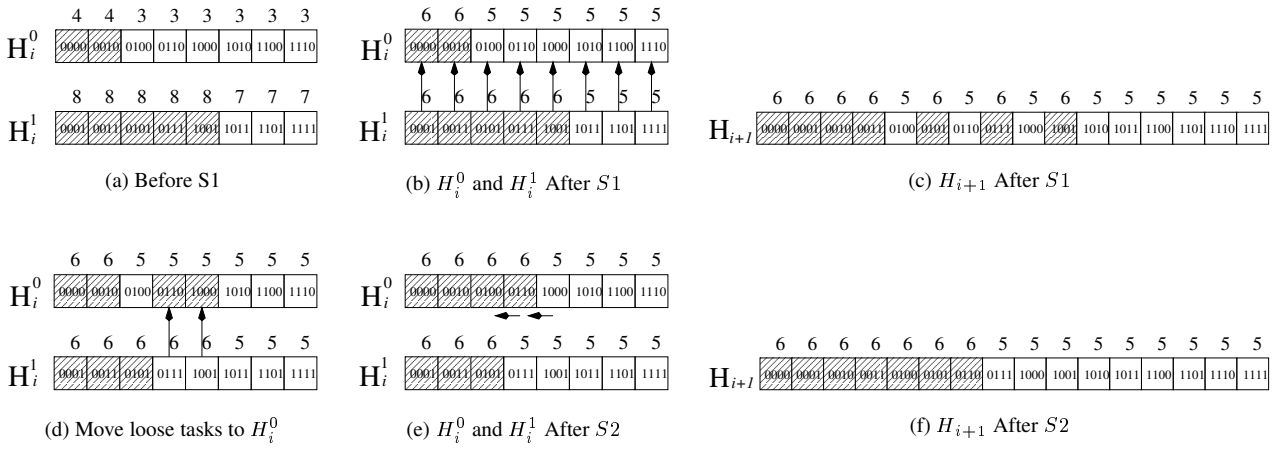


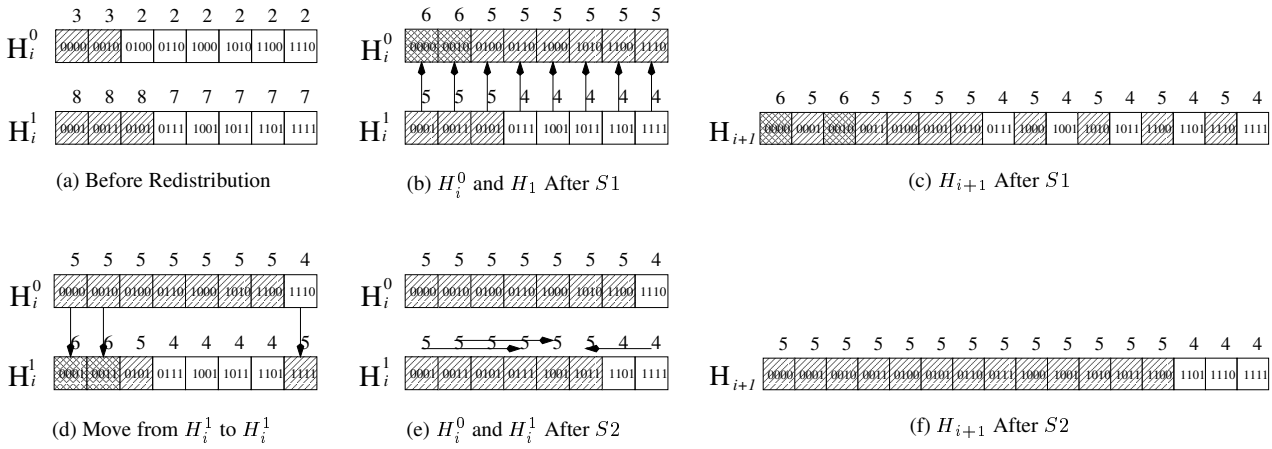
Figure 3: When $\lfloor N_0/P_i \rfloor + \lfloor N_1/P_i \rfloor$ Is Even

At each iteration i, the tasks of two corresponding i-dimensional subcubes, say H_i^0 and H_i^1 (each has $P_i = 2^i$ processors), will be redistributed over the edges of dimension i. Suppose the numbers of total tasks on H_i^0 and H_i^1 are N_0 and N_1, respectively. A cube H_{i+1} with $N_0 + N_1$ tasks will be formed by merging H_i^0 and H_i^1 after the load is balanced by steps $S1$ and $S2$. There are two possible cases.

3.1 $\lfloor N_0/P_i \rfloor + \lfloor N_1/P_i \rfloor$ Is Even

At $S1$, if $\lfloor N_1/P_i \rfloor > \lfloor N_0/P_i \rfloor$, $(\lfloor N_1/P_i \rfloor - \lfloor N_0/P_i \rfloor)/2$ tasks will be moved across each edge of dimension i from H_i^1 to H_i^0. Otherwise, $(\lfloor N_0/P_i \rfloor - \lfloor N_1/P_i \rfloor)/2$ tasks will be moved from H_i^0 to H_i^1. In other words, the loose tasks will not be moved at this step. For example, the tasks of H_i^0 and H_i^1 (each has 8 processors) shown in Figure 3(a) are redistributed by moving 2 (i.e. $(7-3)/2$) tasks from each processor in H_i^1 to its corresponding processor in H_i^0, as shown in Figure 3(b). Note that each box in Figure 3 represents a processor, the number on top of each box denotes the number of tasks currently allocated in the processor, and arrows depicts the direction of task flows. Furthermore, the processors that are marked by slashed lines in Figure 3 own a loose task each.

As shown in Figure 3(a), tasks are initially partitioned over processors of H_i^0 and H_i^1 in block distributions. However, the tasks in the cube H_{i+1} that is formed by merging H_i^0 and H_i^1 after $S1$ are not in a block distribution, as shown in Figure 3(c). Therefore, at step $S2$ the loose tasks must be redistributed to achieve a block distribution in H_{i+1}. Such a distribution can be achieved by transferring 2 (i.e. $\lceil (N_1 \,\%\, P_i - N_0 \,\%\, P_i)/2 \rceil$) loose tasks from consecutive processors starting from $\lfloor (N_1 \,\%\, P_i + N_0 \,\%\, P_i)/2 \rfloor$ in H_i^1 to the corresponding processors in H_i^0, as depicted in Figure 3(d), and then moving the loose tasks to consecutive processors starting at $(N_0 \,\%\, P_i)$, as shown in Figure 3(e). The cube H_{i+1} after the step $S2$ is displayed in Figure 3(f) and the tasks are partitioned in a block distribution. In summary, if $(N_0 \,\%\, P_i) > (N_1 \,\%\, P_i)$, $\lfloor (N_0 \,\%\, P_i - N_1 \,\%\, P_i)/2 \rfloor$ loose tasks on processors starting at $\lceil (N_1 \,\%\, P_i + N_0 \,\%\, P_i)/2 \rceil$ in H_i^0 will be moved to H_i^1 and then be shifted to consecutive processors with addresses starting at $N_1 \,\%\, P_i$. Otherwise, $\lceil (N_1 \,\%\, P_i - N_0 \,\%\, P_i)/2 \rceil$ loose tasks on processors starting $\lfloor (N_1 \,\%\, P_i + N_0 \,\%\, P_i)/2 \rfloor$ in H_i^1 will be transferred to H_i^0 and moved to processors starting at $N_0 \,\%\, P_i$. Furthermore, the redistribution of loose tasks can be viewed as a data concentration operation on hypercube, since the loose tasks on a set of consecutive processors are moved to another set of consecutive processors.



(a) Before Redistribution (b) H_i^0 and H_1 After $S1$ (c) H_{i+1} After $S1$

(d) Move from H_i^1 to H_i^1 (e) H_i^0 and H_i^1 After $S2$ (f) H_{i+1} After $S2$

Figure 4: When $\lfloor N_0/P_i \rfloor + \lfloor N_1/P_i \rfloor$ Is Odd

3.2 $\lfloor N_0/P_i \rfloor + \lfloor N_1/P_i \rfloor$ Is Odd

3.2.1 $\lfloor N_0/P_i \rfloor < \lfloor N_1/P_i \rfloor$

At $S1$, $\lfloor (\lfloor N_1/P_i \rfloor - \lfloor N_0/P_i \rfloor)/2 \rfloor$ tasks will be moved across each edge of dimension i from H_i^1 to H_i^0. In addition, if $(N_0 \,\%\, P_i) \leq (N_1 \,\%\, P_i)$, then one more task will be moved from H_i^1 to H_i^0. For example, the tasks of H_i^0 and H_i^1 shown in Figure 4(a) are redistributed by moving 3 (i.e. $\lfloor (7-2)/2 \rfloor + 1$) tasks from each processor in H_i^1 to its corresponding processor in H_i^0, as shown in Figure 4(b). Note that each processor that are marked by crossed lines in Figure 4 owns an extra loose task, i.e. one more loose task than the processors with slashed lines, each of which has one loose task.

Now the cube H_{i+1} shown in Figure 3(c) is ready to redistribute its loose tasks to achieve a block distribution. Suppose $(N_1 \,\%\, P_i) > (N_0 \,\%\, P_i)$. At $S2$, H_i^0 will first transfer $x = \min(N_0 \,\%\, P_i,\, P_i - N_1 \,\%\, P_i)$ loose tasks from consecutive processors (marked by crossed lines) starting from $\max(0, (N_0 \,\%\, P_i - (P_i - N_1 \,\%\, P_i)))$ in H_i^0 to the corresponding processors in H_i^1, and $y = \lfloor (N_0 \,\%\, P_i + N_1 \,\%\, P_i + P_i)/2 \rfloor - (x + N_1 \,\%\, P_i)$ from the tail processors in H_i^0 to H_i^1, as depicted in Figure 4(d). Then H_i^1 will move the x tasks from the head of subcube to consecutive processors starting at $(N_1 \,\%\, P_i)$ and the y tasks from the tail of subcube to consecutive processors starting at $(N_1 \,\%\, P_i + x)$, as shown in Figure 4(e). The cube H_{i+1} after the step $S2$ is displayed in Figure 4(f).

Similarly, if $(N_1 \,\%\, P_i) \leq (N_0 \,\%\, P_i)$, H_i^1 will first transfer $x = \min(N_1 \,\%\, P_i,\, P_i - N_0 \,\%\, P_i)$ loose tasks from consecutive processors (marked by crossed lines) starting from $\max(0, (N_0 \,\%\, P_i - (P_i - N_1 \,\%\, P_i)))$ in H_i^1 to the corresponding processors in H_i^0, and $y = \lceil (N_0 \,\%\, P_i + N_1 \,\%\, P_i + P_i)/2 \rceil - (x + N_0 \,\%\, P_i)$ from the tail processors in H_i^1 to H_i^0. Then the moved loose tasks will be concentrated to the processors starting at $(N_0 \,\%\, P_i)$ in H_i^1 to form a block distribution.

3.2.2 $\lfloor N_0/P_i \rfloor > \lfloor N_1/P_i \rfloor$

At $S1$, $\lfloor (\lfloor N_0/P_i \rfloor - \lfloor N_1/P_i \rfloor)/2 \rfloor$ tasks will be moved across each edge of dimension i from H_i^0 to H_i^1. In addition, if $(N_0 \,\%\, P_i) > (N_1 \,\%\, P_i)$, then one more task will be moved from H_i^0 to H_i^1. The rest will be to redistribute the loose tasks of cube H_{i+1} as described in Section 3.2.1 to achieve a block distribution.

4 Analysis

If each processor is assigned at most M tasks before the load balancing algorithm is performed, then at each iteration i there are at most $M/2$ tasks to be moved across the edges of dimension i at $S1$. Therefore, the time complexity contributed by $S1$ will be $O(M \log_2 P)$ in the worst case. Furthermore, the operations that are per-

formed at $S2$ are essentially data concentration operations. Since each concentration operation on a hypercube takes at most $\log_2 P$ moves [8], $S2$ of the algorithm takes $\log_2 2 + log_2 4 + \cdots + log_2 P = O(log_2^2 P)$ in the worst case. Therefore, the worst-case time complexity of this algorithm is $O(M \log_2 P + \log_2^2 P)$. Furthermore, if $M \gg \log_2 P$, then the time complexity could be dominated by the factor $O(M \log_2 P)$.

5 Related Work

Ranka et al. and Cybenko introduced the dimension exchange method for load balancing on hypercube [2, 9]. It is a simple but usually heuristic and has been adapted by many researchers to perform parallel load balancing. However, its worst-case error ξ is n on a n-dimensional hypercube. Plaxton developed a couple of load balancing algorithms based on the similar approach, and hence the error would be n as well. Recently, Rim et al. adapted the dimension exchange method to distribute quantized loads with $\xi \leq \lceil log_2 P/2 \rceil$ [10].

Raghavendra et al. proposed a distributed a distributed load balancing algorithm DBALANCE based on the concentration operation [7]. It is required that the set of overloaded processors S_o and the set of underloaded processors S_u are known before DBALANCE can be performed to distribute tasks evenly. Furthermore, the total number of tasks N must be multiple of the number of processors P, i.e. $|S_o| = |S_u|$ where $|S_o|$ and $|S_u|$ denote the numbers of overloaded and underloaded processors respectively.

Chlebus et al. improved the load balancing algorithms developed by Plaxton to avoid the error accumulation over dimensions [1]. Similar to the algorithm presented in this paper, the worse-case performance of the algorithm is bounded by $O(M \log_2 P + \log_2^2 P)$. However, Chlebus's algorithm must perform preprocessing to globally determine where loose tasks should be placed before tasks are redistributed by dimension exchanging. The algorithm in this paper does not require preprocessing to gather global information of task distributions. Exploiting the block distributions in parallel programming languages provides a convenient way to represent global task distributions. Furthermore, perfect load balancing is achieved not only on processors of the hypercube but also on subcubes. In other words, once a cube is balanced, if the cube is decomposed into two subcubes by the lowest bit of node addresses, then the difference between the numbers of the total tasks of these subcubes is at most 1.

6 Conclusions

This paper proposed a simple yet optimal algorithm that could distribute loads evenly on hypercube multiprocessors with the worst-case time complexity $O(M \log_2 P)$. The algorithm could achieve perfect load balance over P processors of a hypercube with error of 1. More importantly, it could achieve perfect load balance over subcubes as well — if the cube is decomposed into two subcubes by bit 0, then the difference between the numbers of the total tasks in the subcubes is at most 1.

References

[1] B.S. Chlebus, J.D.P. Rolim, and G. Slutzki. Distrubuting tokens on a hypercube without error accumulation. In *Proceedings of the 1996 International Parallel Processing Symposium*, pages 573–578, 1996.

[2] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[3] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Department of Computer Science Rice COMP TR90-141, Rice University, December 1990.

[4] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1–2):1–170, 1993.

[5] J. Já Já and K.W. Ryu. Load balancing and routing on the hypercube and related networks. *Journal of Parallel and Distributed Computing*, 14:431–435, 1992.

[6] C.G. Plaxton. Load balancing, selection and sorting on the hypercube. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 64–73, 1989.

[7] S. Raghavendra, C.S. Chalasani, and R.V. Boppmana. Improved algorithms for load balancing in circuit switching hypercubes. In *Proceedings of the 5th International Parallel Processing Symposium*, pages 537–542, 1991.

[8] Sanjay Ranka and Sartaj Sahni. *Hypercube Algorithms with Applications to Image Processing and Pattern Recognition*. Springer Verlag, 1990.

[9] Sanjay Ranka, Youngju Won, and Sartaj Sahni. Programming a hypercube multicomputer. In *IEEE Software*, pages 69–77, 1988.

[10] Hwakyung Rim, Ju-Wook Jang, and Sungchun Kim. An efficient dynamic load balancing using the dimension exchange method for balancing of quantized loads on hypercube multiprocessor. In *Proceedings of 1999 IPPS/SPDP*, pages 708–712, 1999.

[11] J. Woo, , and S. Sahni. Load balancing on a hypercube. In *Proceedings of the 5th International Parallel Processing Symposium*, pages 525–530, 1991.