A Load-Balancing Algorithm for N-Cubes

Min-You Wu and Wei Shu
Department of Computer Science
State University of New York at Buffalo

Abstract

A parallel scheduling algorithm for N-cube networks is presented in this paper. This algorithm can fully balance the load and maximize locality by using global load information. Communication costs are significantly reduced compared to other existing algorithms.

1 Introduction

Parallel scheduling is a promising technique for processor load balancing. In parallel scheduling, all processors are cooperated together to schedule work. Parallel scheduling utilizes global load information and is able to accurately balance the load. It provides high-quality, scalable scheduling. Some parallel scheduling algorithms have been introduced in [4, 3, 11, 15]. Parallel scheduling can be applied to problems with a predictable structure, which are called static problems [1, 18]. It can also be applied to problems with an unpredictable structure, which are called dynamic problems. The load of dynamic problems is usually balanced by dynamic scheduling, which can adjust load distribution based on runtime system load information [5, 6, 14]. However, most dynamic scheduling algorithms, when making a load balancing decision, utilize neither problem characteristics nor global load information. When parallel scheduling is applied at runtime, it becomes an incremental collective scheduling. It is applied whenever the load becomes unbalanced. All processors collectively schedule the workload. Such a system has been described in [15].

A category of scheduling sometimes referred to as prescheduling is closely related to the idea presented in this paper. Prescheduling schedules workload according to the problem input. Therefore, problems whose load distribution depends on its input and cannot be balanced by static scheduling can be balanced by prescheduling. Applying prescheduling periodically, the load can be balanced at runtime. Fox et al. first adapted prescheduling to application problems with geometric structures [7]. Some other works also deal with this type of problem [4, 2]. The project PARTI automates prescheduling for nonuniform problems [13]. The dimension exchange method (DEM) is applied to application problems without geometric structure [12, 3]. It was conceptually designed for a hypercube system but may be applied to other topologies, such as k-ary n-cubes [19]. It balances load for independent tasks with an equal grain size. The method has been extended by Willebeek-LeMair and Reeves [17] so that the algorithm can run incrementally to correct the unbalanced load due to varied grain sizes. Nicol has proposed a direct mapping algorithm which computes the total number of tasks by using sum-reduction [11]. However, it does not minimize the communication cost, nor eliminate communication conflict. An incremental scheduling for N-body simulation is presented in [8]. The task graph is rescheduled periodically to correct the load imbalance. However, its runtime scheduling has not been parallelized yet.

In this paper, we present a new parallel scheduling algorithm for N-cube systems. This algorithm can fully balance the load and maximize locality. It significantly reduces communication overhead compared to other existing algorithms. This paper is organized as follows. Section 2 reviews the DEM algorithm. In section 3, the optimal scheduling problem is discussed. In section 4, we present the new algorithm and analyze its optimality. Performance is presented in section 5, and section 6 concludes the paper.

2 Dimension Exchange Method (DEM)

DEM is a good scheduling algorithm. It has been shown that DEM outperformed other dynamic scheduling algorithms [17]. In DEM, small domains are balanced first and then combined to form larger domains until ultimately the entire system is balanced. The algorithm is described in Figure 1. All node pairs in the first dimension whose addresses differ in only the least significant bit balance the load between themselves. Next, all node pairs in the second di-
mension balance the load between themselves, and so forth, until each node has balanced its load with each of its neighbors. The number of communication steps of the DEM algorithm is $3d$, where $d$ is the number of dimensions [17].

**DEM**

for $k = 0$ to $d-1$
  node $i$ exchanges with node $j$ the current values of $w_i$ and $w_j$, where $j = i \oplus 2^k$
  if $(w_i - w_j) > 1$, send $\lfloor (w_i - w_j)/2 \rfloor$ tasks to node $j$
  if $(w_j - w_i) > 1$, receive $\lfloor (w_j - w_i)/2 \rfloor$ tasks from node $j$
  update the value of $w_i$

Figure 1: The DEM algorithm.

**Example 1:**

The DEM algorithm is illustrated in Figure 2. The load distribution before execution of the DEM algorithm is shown in Figure 2(a). In the first step, nodes exchange load information and balance the load in dimension 0 as shown in Figure 2(b). Then, the load is balanced in dimension 1 as shown in Figure 2(c). After load balancing in dimension 2 (Figure 2(d)), the final result is shown in Figure 2(e). The load is not fully balanced because only integer numbers of tasks can be transmitted between nodes. There are a total of 33 task-hops.

After execution of the DEM algorithm, the load difference $D = \max(w_i) - \min(w_i)$ is bounded by $d$, the dimension of the hypercube [9]. Figure 3 shows an example where $D = 4$ for a 4-dimensional hypercube.

The DEM algorithm is simple and of low complexity. At each load balancing step, only node pairs exchange their load information. No global information is collected. Without global load information, it is impossible for a node to make a correct decision about how many tasks should be sent. Node pairs attempt to average their number of tasks anyway. A node may send excessive tasks to its neighbor. DEM is unable to fully balance the load and to minimize the communication cost.

**3 The Optimal Scheduling Problem**

The scheduling problem can be described as follows. In a parallel system, $N$ computing nodes are connected by a given topology. Each node $i$ has $w_i$
tasks when parallel scheduling is applied. A scheduling algorithm is to redistribute tasks so that the number of tasks in each node is equal. Assume the sum of $w_i$ of all nodes can be evenly divided by $N$. The average number of tasks $w_{avg}$ is calculated by

$$w_{avg} = \frac{\sum_{i=0}^{N-1} w_i}{N}.$$ 

Each node should have $w_{avg}$ tasks after executing the scheduling algorithm. If $w_i > w_{avg}$, the node must determine where to send the tasks.

For a parallel scheduling algorithm that utilizes global information, the number of communication steps can be of order of $\log N$, where $N$ is the number of processors. The average time of each communication step depends on the total number of tasks migrated and their traveling distances. The objective function is to minimize the number of task-hops:

$$\sum_k e_k,$$

where $e_k$ is the number of tasks transmitted through the edge $k$. In general, this problem can be converted to the minimum-cost maximum-flow problem [10] as follows. Each edge is treated as a bidirectional arc and given a tuple $(capacity, cost)$, where capacity is the capacity of the edge and cost is the cost of the edge. Set capacity = $\infty$, cost = 1, for all edges in the processor network. Then add a source node $s$ with an edge $(s, i)$ to each node $i$ if $w_i > w_{avg}$ and a sink node $t$ with an edge $(j, t)$ from each node $j$ if $w_j < w_{avg}$. Set capacity$_{si} = w_i - w_{avg}$, cost$_{si} = 0$, for all $i$, and capacity$_{jt} = w_{avg} - w_j$, cost$_{jt} = 0$, for all $j$. A minimum cost integral flow yields a solution to the problem. The graph constructed for Figure 2 is given in Figure 4, where $w_{avg} = 8$. The minimum cost algorithm [10] generates a solution as shown in Figure 5.

The complexity of the minimum cost algorithm is $O(N^2v)$, where $N$ is the number of nodes and $v$ is the desired flow value [10]. The complexity of its corresponding parallel algorithm on $N$ nodes is at least $O(Nv)$. This high complexity is not realistic for runtime scheduling. For certain topology, such as trees, the complexity can be reduced to $O(\log N)$ on $N$ nodes [16]. For a topology other than trees, we need to find a heuristic algorithm.

4 Cube Walking Algorithm

A good heuristic algorithm can be designed by utilizing global load information. Here we present a new parallel scheduling algorithm for the hypercube topology. The algorithm, called Cube Walking Algorithm (CWA), is shown in Figure 6. Let $w_i^0$ be the number of tasks in node $i$ before the algorithm is applied. The first step collects the system load information by exchanging values of $w_i^0$ to obtain the values of $w_i^1$. Each node records a $w$ vector, where $w_i^k$ is the total number of tasks in its $k$-dimensional subcube. Here, the $k$-dimensional subcube of node $i$ is defined as all nodes whose numbers have the same $(d-k)$-bit prefix as node $i$. The value of $w_i^k$ in each node is equal to the total number of tasks in the entire cube. In step 2, each node calculates the average number of tasks per node. A quota vector $q$ is calculated in step 3 so that each node knows if its $k$-dimensional subcubes are overloaded or underloaded. The vector $q$ can be
computed directly as follows:

\[ q_i^k = w_{avg} \cdot 2^k + r_i^k \]

where

\[ r_i^k = \begin{cases} 
0 & \text{if } i \land (N - 2^k) \geq R \\
2^k & \text{if } i \lor (2^k - 1) < R \\
R - i \land (N - 2^k) & \text{otherwise}
\end{cases} \]

where \( \land \) is the bitwise AND and \( \lor \) the bitwise OR. The \( \delta \) vector is the difference of \( w \) and \( q \), which stand for the number of tasks to be sent to or received from other subcubes.

In step 4, task exchanges are conducted among each dimension. We start with the cube of dimension \( d - 1 \). Recursively, we partition a cube of dimension \( k \) into two subcubes of dimension \( (k - 1) \). Each node \( n(i) \) is paired with the corresponding node \( n(i)' = n(i \oplus 2^k) \) in the other subcube. In this particular step, we only exchange tasks between \( n(i) \) and \( n(i)' \), where \( i = 0, 1, \ldots, N/2 - 1 \). And, we send tasks only in one direction — from the overloaded subcube to the other. In this way, an overloaded node does not necessarily commit itself to send tasks out since it may postpone the action. The decision is made globally within the subcube by calculating a \( \theta \) vector for every node in the overloaded subcube. The calculation of \( \theta \) is a local operation without any communication. The value of \( \delta \) of \( n' \) can be calculated by \( \delta_{i(2^k)} = w_i^{j+1} - w_i^j - q_{i(2^k)} \). The \( \gamma \) vector records the number of tasks reserved for subcubes of lower dimensions. The following lemma shows that at the end of the algorithm, each node has the same number of tasks as its quota.

**Lemma 1:** After execution of CWA, the number of tasks in each node is equal to its quota.

**Proof:** To show after iteration 0 the number of tasks in each node is equal to its quota \( q_i^0 \), we need to show that after iteration \( k \), each \( k \)-dimensional subcube has \( q_i^k \) tasks. Then, the subcube with \( \delta_i^k > 0 \) needs to send \( \delta_i^k \) tasks to the other subcube with \( \delta_i^k < 0 \). Because tasks are sent in one direction, the number of tasks sent from the overloaded subcube to the underloaded subcube must be equal to \( \delta_i^k \). That is, \( \sum \delta_i^k = \theta_i^k = \theta_i^k \). It can be proven by showing that

\[ \theta_i^{j+1} = \theta_i^j + \theta_i^{j+1}_{i(2^k)} \]

There are three cases when assigning the value of \( \theta \):

1. **Case 1:** when \( \delta_i^j < \gamma_i^{j+1} \)
   - \( \theta_i^j = 0 \)
   - \( \theta_i^{j+1}_{i(2^k)} = \theta_i^{j+1} \)

   Hence, \( \theta_i^j + \theta_i^{j+1}_{i(2^k)} = \theta_i^{j+1} \).

The Cube Walking Algorithm (CWA)

Assume the cube dimension is \( d \), the number of nodes is \( N = 2^d \). Let \( \oplus \) denote the bitwise exclusive OR and \( \land \) the bitwise AND.

1. **Global Information Collection:**
   Perform sum reductions. Each node computes its \( w \) vector, \( k = 0, \ldots, d \)
   \[ w_i^0 = w_i, \quad w_i^k = w_{i(2^k)} + w_{i(2^k-1)} \]

2. **Average Load Calculation:**
   \[ T = w_i^d, \quad w_{avg} = \lceil T/N \rceil, \quad R = T \mod N. \]

3. **Quota Calculation:** Node \( i \) computes its vectors \( q_i^k \) and \( \delta_i^k \), \( k = 0, \ldots, d - 1 \)
   \[ q_i^0 = \begin{cases} 
   w_{avg} + 1 & \text{if } i < R \\
   w_{avg} & \text{otherwise}
\end{cases}, \quad q_i^k = q_i^{k-1} + q_i^{k+1}_{i(2^k-1)} \\
   \delta_i^k = w_i^k - q_i^k \]

4. **Task Exchange:** For \( k = d - 1 \) to 0 do
   4.1) For node \( i \) with \( \delta_i^k > 0 \), compute the number of tasks to be sent out
   - Initialize \( \theta_i^j = \delta_i^k \) and \( \gamma_i^j = 0 \)
   
   For \( j = k - 1 \) to 0
   
   \[ \theta_i^j = \begin{cases} 
   0 & \text{if } \delta_i^j \leq \gamma_i^{j+1} \\
   \min(\delta_i^j - \gamma_i^{j+1}, \theta_i^{j+1}) & \text{if } \delta_i^j > \gamma_i^{j+1} \\
   \theta_i^{j+1} & \text{if } \delta_i^{j+1} \leq \gamma_i^{j+1} \\
   \max(\delta_i^j, 0) & \text{if } \delta_i^{j+1} > \gamma_i^{j+1} \\
   \gamma_i^j & \text{if } \delta_i^{j+1} > \gamma_i^{j+1} \quad \text{and } i \land 2^j \neq 0
   \end{cases} \]

   Send \( \theta_i^j \) tasks as well as its \( \theta \) vector to node \( i \oplus 2^j \).

   Update its own vectors \( w_i^j = w_i^j - \theta_i^j, \delta_i^j = \delta_i^j - \theta_i^j \),
   for \( j = 0, 1, \ldots, k - 1 \).

   4.2) For node \( i \) with \( \delta_i^k < 0 \), receive tasks as well as the \( \theta \) vector from node \( i \oplus 2^k \).

   Update its own vectors \( w_i^j = w_i^j + \theta_i^j, \delta_i^j = \delta_i^j + \theta_i^j \),
   for \( j = 0, 1, \ldots, k - 1 \).

---

Figure 6: The Cube Walking Algorithm.
Case 2: when \( \delta^j_{i+1} + \gamma^j_{i+1} \geq \delta^j_i + \gamma^j_i \),

since \( \delta^j_i - \gamma^j_i \leq \delta^j_{i+1} \),

\[ \theta^j_{i+1} = \min(\delta^j_i - \gamma^j_i, \delta^j_{i+1}) = \delta^j_{i+1} \]

since \( \delta^j_{i+2} = \delta^j_{i+1} + \delta^j_i = \gamma^j_{i+1} + \delta^j_{i+1} - \delta^j_i \geq 0 \),

\[ \theta^j_{i+2} = \max(\delta^j_{i+2}, 0) = \delta^j_{i+2} \]

Hence,

\[ \theta^j_i + \theta^j_{i+2} = \delta^j_i - \gamma^j_i + \delta^j_{i+2} = \delta^j_{i+1} - \gamma^j_{i+1} = \theta^j_{i+1} \]

Case 3: when \( \delta^j_i > \delta^j_{i+1} + \gamma^j_i \),

since \( \delta^j_i - \gamma^j_i > \delta^j_{i+1} \),

\[ \theta^j_i = \min(\delta^j_i - \gamma^j_i, \delta^j_{i+1}) = \delta^j_{i+1} \]

since \( \delta^j_{i+2} = \delta^j_{i+1} + \delta^j_i = \gamma^j_{i+1} + \delta^j_{i+1} - \delta^j_i < 0 \),

\[ \theta^j_{i+2} = \max(\delta^j_{i+2}, 0) = 0 \]

Hence, \( \theta^j_i + \theta^j_{i+2} = \theta^j_{i+1} \).

In this algorithm, step 1 spends \( 2d \) communication steps for exchanging load information, where \( d \) is the dimension of the cube. Step 4 spends \( d \) communication steps for load balancing. Therefore, the total number of communication steps is \( 3d \).

**Example 2:**

A running example of CWA is shown in Figure 7. At the beginning of scheduling, each node has \( w_i^0 \) tasks ready to be scheduled. Values of \( w_i^k \) are calculated at step 1. The values of \( w_{avg} \) and \( R \) are as follows:

\[ w_{avg} = 8, \ R = 0. \]

Then, each node calculates the values of \( q_i^k \) at step 3. Because \( R = 0 \), every node has the same quota vector:

\[ \{8, 16, 32\}. \]

At step 4, when \( k = 2 \), the subcube \( \{0,1,2,3\} \) is the overloaded one. The values of \( w_i^0, \delta_i^k, \theta_i^k \), and \( \gamma_i^k \) are as follows:

<table>
<thead>
<tr>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w_i^0 )</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Thus, node 0 sends six tasks to node 4, and node 1 sends three tasks to node 5. Now, the loads between subcubes \( \{0, 1, 2, 3\} \) and \( \{4, 5, 6, 7\} \) have been balanced. Each subcube has 32 tasks.

When \( k = 1 \), subcubes \( \{0,1\} \) and \( \{4,5\} \) are overloaded. The values of \( w_i^k, \delta_i^k, \theta_i^k \), and \( \gamma_i^k \) are as follows:

<table>
<thead>
<tr>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w_i^1 )</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Thus, node 0 sends five tasks to node 2, and node 5 sends two tasks to node 7. The loads between subcubes \( \{0, 1\}, \{2, 3\}, \{4, 5\}, \{6, 7\} \) have been balanced. Each subcube has 16 tasks.

When \( k = 0 \), nodes 3, 5, and 6 are overloaded. Their values of \( w_i^k, \delta_i^k, \theta_i^k \), and \( \gamma_i^k \) are as follows:
Finally, node 3 sends one task to node 2, node 5 sends two tasks to node 4, and node 6 sends two tasks to node 7. This results in a balanced load, each node having eight tasks. The total number of task-hops is 21.

Now, we discuss the scheduling quality, locality, and communication costs of the CWA algorithm. The next theorem shows that this algorithm is able to fully balance the load. If the number of tasks can be equally divided by the number of nodes, each node will have the equal number of tasks; otherwise, the number of tasks in each node differs by one.

**Theorem 2:** The difference in the number of tasks in each node is at most one after execution of CWA.

**Proof:** From Lemma 1, the number of tasks in each node is equal to its quota after execution of CWA. Since the quota is either $w_{avg}$ or $w_{avg} + 1$, the difference in the number of tasks in each node is at most one.

This algorithm also maximizes locality. **Local tasks** are the tasks that are not migrated to other nodes, and **non-local tasks** are those that are migrated to other nodes. Maximum locality implies the minimum number of non-local tasks. In Lemmas 2 and 3 and Theorem 3, we assume that the number of tasks $T$ is evenly divided by $N$, the number of nodes. When $T$ is not evenly divided by $N$, the algorithms are nearly-optimal. The following lemma gives the minimum number of non-local tasks.

**Lemma 2:** To reach a balanced load, the minimum number of non-local tasks is

$$\sum_i \max(w_{avg} - w_i, 0).$$

**Proof:** Each node where $w_i < w_{avg}$ must receive $(w_{avg} - w_i)$ tasks from other nodes for a balanced load. Therefore, a total of $\sum_i \max(w_{avg} - w_i, 0)$ tasks must be migrated between nodes.

The next theorem proves that CWA maximizes locality.

**Theorem 3:** The number of non-local tasks in the CWA algorithm is

$$\sum_i \max(w_{avg} - w_i, 0).$$

**Proof:** In CWA, each node sends tasks only when its weight is larger than $w_{avg}$ and no more than $(w_i - w_{avg})$ tasks are sent out. Thus, in all nodes at least $\sum_i \min(w_i, w_{avg})$ tasks are local. Therefore, the number of non-local tasks is no more than

$$N \times w_{avg} - \sum_i \min(w_i, w_{avg}) = \sum_i (w_{avg} - \min(w_i, w_{avg}))$$

$$= \sum_i \max(w_{avg} - w_i, 0).$$

As stated in Lemma 2, these algorithms minimize the number of non-local tasks and maximize locality. □

CWA is a heuristic algorithm and in general is not able to minimize the communication cost. However, for a system with less than or equal to four nodes, the algorithm minimizes the communication cost.

**Lemma 3:** The CWA algorithm minimizes the communication cost in a system with two or four nodes.

**Proof:** The communication cost in a system is minimized if there is no negative cycle [10]. In a system of two nodes, there is no cycle. In a system of four nodes, only a path consisting of at least three edges can form a negative cycle. With CWA, the longest path has two edges. Therefore, there is no negative cycle. □

The DEM algorithm does not minimize the communication cost for four nodes because there may be a path consisting of three edges.

**5 Performance Study**

CWA is a heuristic algorithm. Its optimality needs to be studied with simulation. For this purpose, we consider a test set of load distributions. In this test set, the load at each processor is randomly selected, with the mean equal to the specified average number of tasks. The number of processors varies from 4 to 256. The average number of tasks (average weight) per processor varies from 2 to 100. The average weight is made to be an integer so that the load can be fully balanced.

First, we compare CWA to DEM. CWA can fully balance the load but DEM cannot in most cases. We run the DEM algorithm on 1,000 test cases. When the number of processors increases, there are less fully-balanced cases. For 32 processors there are a few cases, and for 64 processors, there is no fully-balanced case in this test set.

An important measure of a scheduling algorithm is its locality. The CWA algorithm sends only necessary tasks to other processors so that it maximizes locality. The DEM algorithm results in unnecessary task migration. Here, we study locality of the DEM algorithm. Because DEM is not able to fully balance the
load for all cases, only the fully-balanced cases are selected. Each result is the average of the fully-balanced cases in 1,000 test cases. The normalized locality is measured by

\[ \frac{T_{DEM} - T_{OPT}}{T_{OPT}} \]

where \( T_{DEM} \) is the total number of non-local tasks in the DEM algorithm, and \( T_{OPT} \) is the minimum number of non-local tasks. Figure 8 shows the normalized locality on 4, 8, and 16 processors. Because few fully-balanced cases exist on more than 16 processors, they are not reported here.

Figure 8: Normalized locality of DEM

Next, we compare the load balancing overhead. DEM is very simple so that the runtime overhead for load balancing decision is small. However, unnecessary task migration leads to a large communication overhead. Compared to the time spent on the load balancing decision, communication time is the dominate factor. CWA, on the other hand, although needing more time to make an accurate load balancing decision, involves less communication overhead. The normalized communication cost is measured by

\[ \frac{C_{DEM} - C_{OPT}}{C_{OPT}} \]

and

\[ \frac{C_{CWA} - C_{OPT}}{C_{OPT}} \]

where \( C_{DEM}, C_{CWA}, \) and \( C_{OPT} \) are the number of task-hops of the DEM, CWA, and optimal algorithms, respectively. Figure 9 compares the normalized communication costs on 4, 8, and 16 processors. Each result is the average of the DEM fully-balanced cases in 1,000 test cases. The number of task-hops of CWA on four processors is the minimum. It can be seen that the communication costs of DEM are much larger than those of CWA. Figure 10 shows the normalized communication costs of CWA on 64 and 256 processors. Each data presented here is the average of 100 different test cases.

Figure 9: Normalized communication costs of DEM and CWA

Figure 10: Normalized communication costs of CWA
6 Conclusion

In this paper, we described a parallel scheduling algorithm for N-cubes. In this algorithm, all processors cooperate together to collect load information and to exchange workload in parallel. With parallel scheduling, it is possible to obtain a high quality load balancing with a fully-balanced load and maximized locality. Communication costs can be reduced significantly.

Acknowledgements

The author would like to thank Xin He for his helpful discussion. This research was partially supported by NSF grant CCR-9505300.

References