

On Runtime Parallel Scheduling for Processor Load Balancing

Min-You Wu
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
wu@cs.buffalo.edu

Abstract— Parallel scheduling is a new approach for load balancing. In parallel scheduling, all processors cooperate to schedule work. Parallel scheduling is able to accurately balance the load by using global load information at compile-time or runtime. It provides high-quality load balancing. This paper presents an overview of the parallel scheduling technique. Scheduling algorithms for tree, hypercube, and mesh networks are presented. These algorithms can fully balance the load and maximize locality

1. Introduction

Static scheduling balances the workload before runtime and can be applied to problems with a predictable structure, which are called static problems. Dynamic scheduling performs scheduling activities concurrently at runtime, which applies to problems with an unpredictable structure, which are called dynamic problems. Static scheduling utilizes the knowledge of problem characteristics to reach a well-balanced load [1, 2, 3, 4]. However, it is not able to balance the load for dynamic problems. In addition, the requirement of large memory space to store the task graph restricts the scalability of static scheduling. Dynamic scheduling is a general approach suitable for a wide range of applications [5, 6, 7]. It can adjust load distribution based on runtime system load information. However, most runtime scheduling algorithms utilize neither the characteristics information of application problems, nor the global load information for load balancing decisions. System stability usually sacrifices both quality and quickness of load balancing.

Parallel scheduling is a promising technique for processor load balancing. In parallel scheduling, all processors cooperate to schedule work. Parallel scheduling utilizes global load information and is able to accurately balance the load. It provides high-quality, scalable load balancing. Some parallel scheduling algorithms have been introduced in [8, 9, 10, 11].

Parallel scheduling can be applied to static problems. Most existing scheduling algorithms for static problems running on a single processor are not scalable to massively parallel computers

because storing the task graph requires large memory space. To speed up scheduling and to relax the demand of memory space, static scheduling can be parallelized. Kwok and Ahmad have developed a parallel algorithm [12]. Wu has parallelized the MCP algorithm [13].

Parallel scheduling can also be applied to dynamic problems. 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 [11]. It starts with a system phase which schedules initial tasks; it is followed by a user computation phase to execute the scheduled tasks and possibly to generate new tasks. In the next system phase, the old tasks that have not been executed will be scheduled together with the newly generated tasks. In each system phase, a parallel scheduling algorithm is applied to balance the load.

In this paper, we discuss the parallel scheduling methodology. This paper is devoted particularly to a kind of scheduling that only schedules ready jobs or tasks. That is, the objects to be scheduled are a set of jobs or tasks that are ready to execute. Scheduling algorithms for tree, hypercube, and mesh networks will be presented. These algorithms are primarily designed for dynamic problems with randomly arrived or dynamically generated jobs or tasks. These algorithms can fully balance the load, maximize locality, and significantly reduce communication overhead compared to other existing algorithms.

This paper is organized as follows. In section 2, we discuss the optimal scheduling problem. The parallel scheduling algorithms for tree, hypercube, and mesh topologies are presented in section 3. The properties of these algorithms are described in section 4 and performance is presented in section 5. Previous works are discussed in section 6, while section 7 concludes the paper.

2. The Optimal Scheduling Problem

The objective of scheduling is to schedule works so that each processor has the same work load. Thus, we need to estimate the task execution time. The estimation can be application-specific, leading to a less general approach. Sometimes, such an estimation is difficult to obtain. Due to these difficulties, each task is presumed to require equal execution time, and the objective of the algorithm becomes to schedule tasks so that each processor has the same number of tasks. Inaccuracy caused by grain-size variation can be corrected in the next system phase. An algorithm with estimated time of tasks could improve load balancing to some extent. However, since the algorithm is more complex, the scheduling overhead increases which may overwrite this benefit [11].

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. When $w_i > w_{avg}$, the node must determine where to send the tasks.

In a communication step, many communications can be performed simultaneously. The time spent on the load balancing activity depends on the number of communication steps and the time taken for each step. For a parallel scheduling algorithm that utilizes global information, the number of communication steps can be of the order of $\log N$, where N is the number of processors [14]. The average time of each communication step depends on both 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 [15] 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* = ∞ , *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 maximum-flow algorithm yields a solution to the problem. Figure 1 shows a load distribution in an eight-node hypercube network. The graph constructed for Figure 1 is given in Figure 2, where $w_{avg} = 8$. The minimum cost algorithm [15] generates a solution as shown in Figure 3.

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 [15]. 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. For a topology other than trees, we need to find a heuristic algorithm.

3. Parallel Scheduling Algorithms

In this section, we present parallel scheduling algorithms for the tree, hypercube, and mesh topologies. The common feature of these algorithms is that the total number of tasks is obtained by

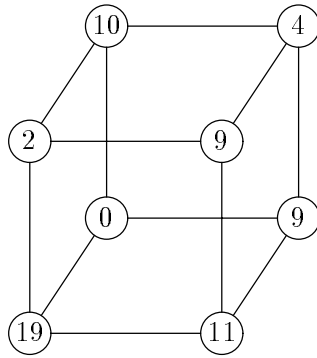


Figure 1: A load distribution in a 3-dimensional hypercube.

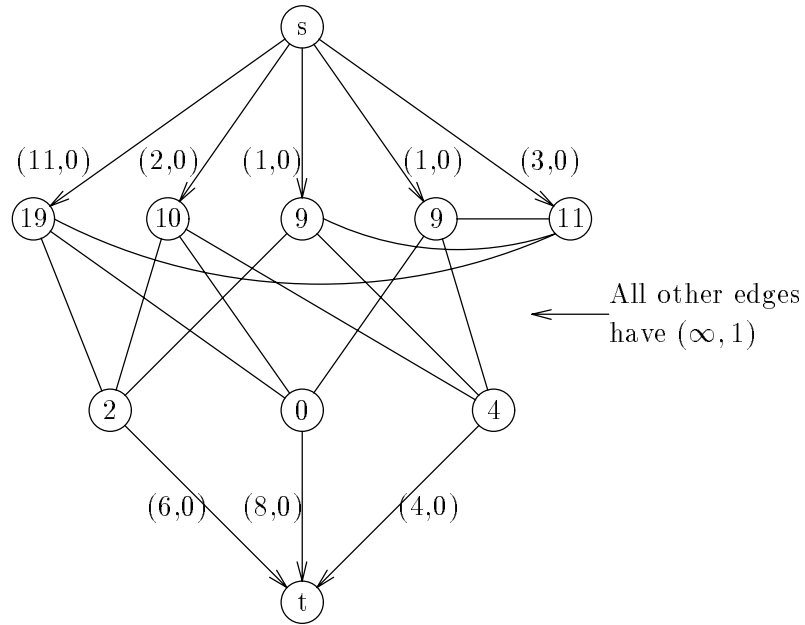


Figure 2: Graph for optimal scheduling problem (Figure 1).

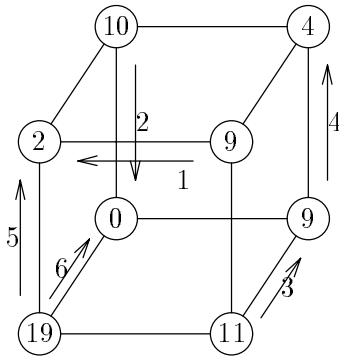


Figure 3: An optimal solution of Figure 1.

a parallel reduction operation so that the average number of tasks per node can be calculated [14]. A node will not send its tasks to other nodes unless the number of tasks exceeds the average. Therefore, only necessary tasks are migrated.

Before discussing individual algorithms for different topologies, we give a generic algorithm which is shown in Figure 4. The first step collects global information by using a *sum* reduction [14]. In step 2, the average number of tasks per node is calculated. If the number of tasks cannot be evenly divided by the number of nodes, the remaining R tasks are evenly distributed to the first R nodes so that they have one more task than the others. The values of w_{avg} and R are available to each node. In step 3, each node calculates its quota so that each node knows if it is overloaded or underloaded. The quotas for some subsets of nodes are also computed here for a particular topology. In step 4, tasks are exchanged to meet the quotas with minimum communication. Different algorithms are designed for different topologies.

Let w_i be the number of tasks in node i .

1. **Global Information Collection:** Perform the *sum* reduction of w_i to compute T and other information, where T is the total number of tasks.
2. **Average Load Calculation:** $w_{avg} = \lfloor T/N \rfloor$, $R = T \bmod N$.
3. **Quota Calculation:** Each node computes its quota q_i

$$q_i = \begin{cases} w_{avg} + 1 & \text{if } i < R \\ w_{avg} & \text{otherwise} \end{cases}$$

The quotas for some subsets of nodes are also computed.

4. **Task Exchange:** Each overloaded node determines where to send its excess tasks.
-

Figure 4: The Generic Parallel Scheduling Algorithm.

In the following subsections, we present three parallel scheduling algorithms: the Tree Walking Algorithm (TWA), the Cube Walking Algorithm (CWA), and the Mesh Walking Algorithm (MWA). The tree algorithm is an optimal algorithm in terms of the number of task-hops. The hypercube and mesh algorithms are heuristic algorithms.

3.1 Tree Walking Algorithm

When the network topology is a tree, the complexity of optimal scheduling can be reduced. The *Tree Walking Algorithm (TWA)* is shown in Figure 5, which is essentially the same as the one presented in [11]. In step 1, when the total number of tasks is counted with a parallel reduction operation, each node records the number of tasks in its subtree and its children's subtrees (if any). In step 2, the root calculates the average number of tasks per node and then broadcasts

the number to every node. In step 3, each subtree rooted at node i calculates its quota Q_i that indicates how many tasks are to be scheduled to the subtree. Q_i can be calculated directly as follows:

$$Q_i = w_{avg} * n_i + r_i$$

where

$$r_i = \begin{cases} 0 & \text{if } i \geq R \\ n_i & \text{if } i \leq R - n_i \\ R - i & \text{if } R - n_i < i < R \end{cases}$$

Each node keeps records of Q_i and Q_j , where node j is node i 's child (if any). In step 4, the workload is exchanged so that at the end of the system phase each node has the same number of tasks as its quota.

Tree Walking Algorithm (TWA)

Assign each node an *order* i according to the preorder traversal; and n_i , the number of nodes of its subtree, where $i = 0, 1, \dots, N - 1$, and $N = n_0$ is the total number of nodes in the system. Node i has its parent node p_i and also has a child vector $c_{i,0}, c_{i,1}, \dots, c_{i,m_i-1}$ to give its m_i children's node numbers.

1. **Global Information Collection:** Perform a *sum* reduction of w_i :

$$W_i = \sum_{j=i}^{i+n_i-1} w_j$$

2. **Average Load Calculation:** $T = W_0$, $w_{avg} = \lfloor T/N \rfloor$, and $R = T \bmod N$.
3. **Quota Calculation:** The quota of each node q_i is computed:

$$q_i = \begin{cases} w_{avg} + 1 & \text{if } i < R \\ w_{avg} & \text{otherwise} \end{cases},$$

Also, the quota for each subtree is computed:

$$Q_i = \sum_{j=i}^{i+n_i-1} q_j$$

4. **Task Exchange:** Each node computes $\eta_i^L = W_i - Q_i$, and $\eta_{i,j}^R = Q_{c_{i,j}} - W_{c_{i,j}}$, where $j = 0, 1, \dots, m_i - 1$.
 - 4.1) For node i with $\eta_i^L < 0$ receive tasks from node p_i .
 - 4.2) For node i and $j = 0, 1, \dots, m_i - 1$ if $\eta_{i,j}^R < 0$ receive tasks from node $c_{i,j}$.
 - 4.3) For node i with $\eta_i^L > 0$ send η_i^L tasks to node p_i .
 - 4.4) For node i and $j = 0, 1, \dots, m_i - 1$ if $\eta_{i,j}^R > 0$ send $\eta_{i,j}^R$ tasks to node $c_{i,j}$.

Figure 5: The Tree Walking Algorithm

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

Proof: Assume node j is a child of node i such that $c_{i,l} = j$, η_j^L in node j is equal to $-\eta_{i,l}^R$ in node i . Thus, if $\eta_j^L < 0$, node j will receive $|\eta_j^L|$ tasks from node i . Similarly, $\eta_{i,l}^R$ in node i is equal to $-\eta_j^L$ in node j . Thus, if $\eta_{i,l}^R < 0$, node i will receive $|\eta_{i,l}^R|$ tasks from node j . Therefore, after execution of TWA, the number of tasks in node i is

$$w'_i = w_i - \eta_i^L - \sum_{j \text{ is child of } i} \eta_{i,j}^R = w_i - (W_i - Q_i) - \sum_{j \text{ is child of } i} (Q_j - W_j).$$

Because

$$\sum_{j \text{ is child of } i} W_j = W_i - w_i,$$

$$w'_i = w_i - (W_i - Q_i) - \sum_{j \text{ is child of } i} Q_j + (W_i - w_i) = Q_i - \sum_{j \text{ is child of } i} Q_j = q_i.$$

□

In this algorithm, steps 1 and 2 spend $2m$ communication steps, where m is the depth of the tree. The communication steps in step 4 is the distance from a leaf node to another leaf node, which is at most $2m$. Therefore, the total number of communication steps of this algorithm is at most $4m$. With a balanced tree, $m = \log N$, and the number of communication steps of this parallel algorithm on N nodes is $O(\log N)$.

Example 1:

An example is shown in Figure 6. The nodes in the tree are numbered by preorder traversal. At the beginning of scheduling, each node has w_i tasks ready to be scheduled. Values of W_i are calculated in step 1. The root calculates the value of w_{avg} and R :

$$w_{avg} = 4, R = 5.$$

Then, each node calculates the value of Q_i in step 3. The values of w_i , n_i , W_i , Q_i , η_i^L , and $\eta_{i,j}^R$ are shown as follows:

i	w_i	n_i	W_i	Q_i	η_i^L	$\eta_{i,0}^R$	$\eta_{i,1}^R$	$\eta_{i,2}^R$
0	1	9	41	41	0	-5	0	1
1	4	3	20	15	5	0	-6	-
2	5	1	5	5	0	-	-	-
3	11	1	11	5	6	-	-	-
4	7	2	9	9	0	2	-	-
5	2	1	2	4	-2	-	-	-
6	3	3	11	12	-1	1	-1	-
7	3	1	3	4	-1	-	-	-
8	5	1	5	4	1	-	-	-

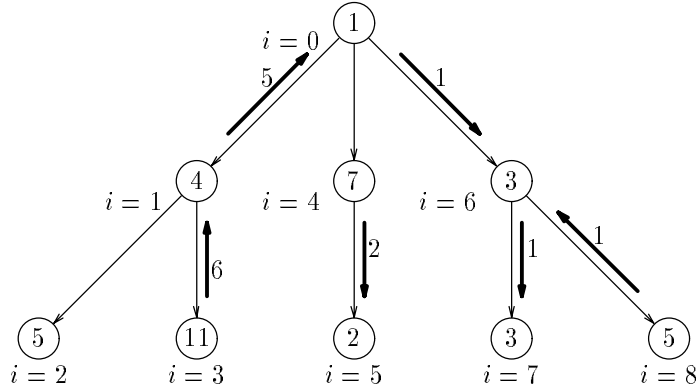


Figure 6: Example for the Tree Walking Algorithm.

The numbers of tasks to be exchanged between nodes are shown in Figure 6. At the end of scheduling, nodes 0–4 have five tasks each, and nodes 5–8 have four tasks each.

3.2 Cube Walking Algorithm

In this subsection, we study two algorithms designed for the hypercube topology: the DEM algorithm [8, 9] and the proposed *Cube Walking Algorithm (CWA)*.

In DEM, small domains are balanced first and then combined to form larger domains until ultimately the entire system is balanced. The “integer version” of DEM is described in Figure 7. 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 dimension 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 [16].

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 w_i

Figure 7: The DEM algorithm.

Example 2:

The DEM algorithm is illustrated in Figure 8. The load distribution before execution of the

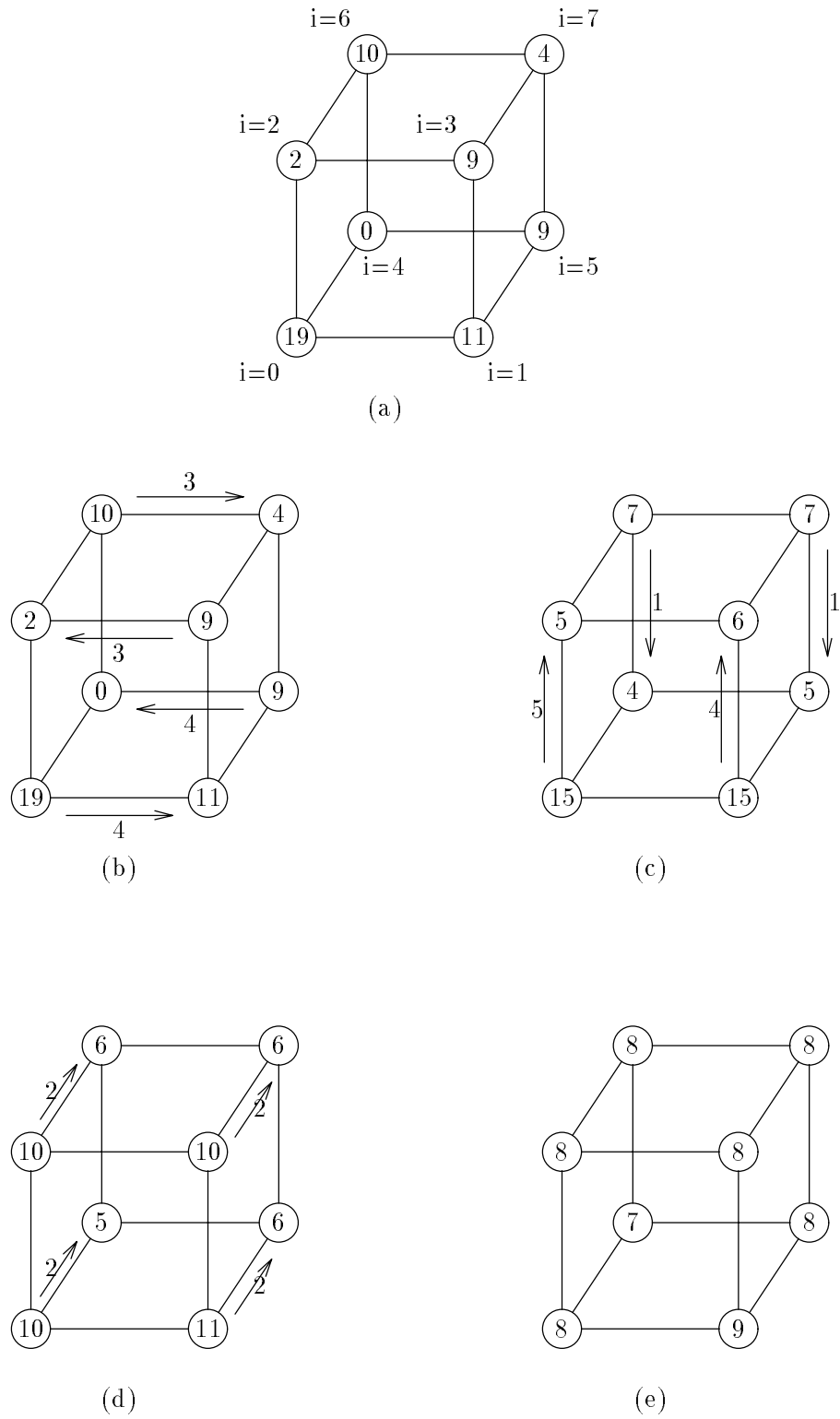


Figure 8: An example for the DEM algorithm.

DEM algorithm is shown in Figure 8(a). In the first step, nodes exchange load information and balance the load in dimension 0 as shown in Figure 8(b). Then, the load is balanced in dimension 1 as shown in Figure 8(c). After load balancing in dimension 2 (Figure 8(d)), the final result is shown in Figure 8(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, whereas the optimal scheduling shown in Figure 3 has only 21 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 [17]. Figure 9 shows an example where $D = 4$ for a 4-dimensional hypercube.

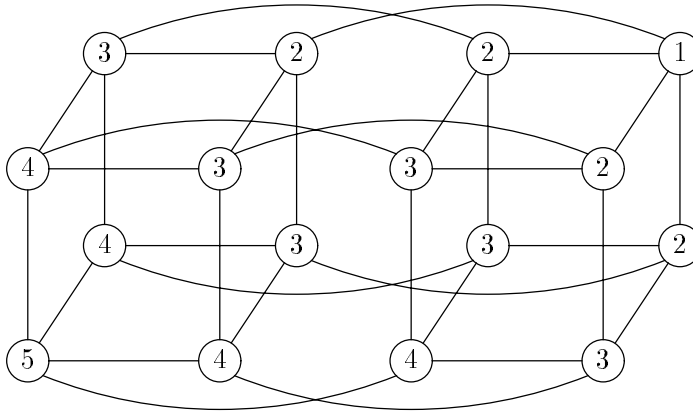


Figure 9: An example which shows that the number of tasks differs by 4 resulting from DEM.

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.

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 10. 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^k to obtain the values of w_i^{k+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^d 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:

$$q_i^k = w_{avg} * 2^k + r_i^k$$

where

$$r_i^k = \begin{cases} 0 & \text{if } i \wedge (N - 2^k) \geq R \\ 2^k & \text{if } i \vee (2^k - 1) < R \\ R - i \wedge (N - 2^k) & \text{otherwise} \end{cases}$$

where \wedge is the bitwise AND and \vee the bitwise OR. The δ vector is the difference of w and q , which stand for the number of tasks to be sent to or received from other subcubes.

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 \wedge the bitwise AND.

1. Global Information Collection:

Perform sum reductions. Each node computes its w vector, $k = 0, \dots, d$

$$w_i^0 = w_i, \quad w_i^k = w_i^{k-1} + w_{i \oplus 2^{k-1}}^{k-1}$$

2. Average Load Calculation: $T = w_i^d$, $w_{avg} = \lfloor T/N \rfloor$, $R = T \bmod N$.

3. Quota Calculation: Each node computes its vectors q_i^k and δ_i^k , $k = 0, \dots, d - 1$

$$q_i^0 = \begin{cases} w_{avg} + 1 & \text{if } i < R \\ w_{avg} & \text{otherwise} \end{cases}, \quad q_i^k = q_i^{k-1} + q_{i \oplus 2^{k-1}}^{k-1}$$

$$\delta_i^k = w_i^k - q_i^k$$

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 $\theta_i^k = \delta_i^k$ and $\gamma_i^k = 0$

For $j = k - 1$ to 0

$$\theta_i^j = \begin{cases} 0 & \text{if } \delta_i^j \leq \gamma_i^{j+1} \text{ and } i \wedge 2^j = 0 \\ \min(\delta_i^j - \gamma_i^{j+1}, \theta_i^{j+1}) & \text{if } \delta_i^j > \gamma_i^{j+1} \text{ and } i \wedge 2^j = 0 \\ \theta_i^{j+1} & \text{if } \delta_{i \oplus 2^j}^j \leq \gamma_i^{j+1} \text{ and } i \wedge 2^j \neq 0 \\ \max(\delta_i^j, 0) & \text{if } \delta_{i \oplus 2^j}^j > \gamma_i^{j+1} \text{ and } i \wedge 2^j \neq 0 \end{cases}$$

$$\gamma_i^j = \delta_i^j - \theta_i^j$$

Send θ_i^0 tasks as well as its θ vector to node $i \oplus 2^k$. Update its own w and δ vectors for each dimension $j = 0, 1, \dots, k - 1$: $w_i^j = w_i^j - \theta_i^j$, $\delta_i^j = \delta_i^j - \theta_i^j$.

4.2) For node i with $\delta_i^k < 0$, receive tasks as well as the θ vector from node $i \oplus 2^k$. Update its own w and δ vectors for each dimension $j = 0, 1, \dots, k - 1$: $w_i^j = w_i^j + \theta_{i \oplus 2^k}^j$, $\delta_i^j = \delta_i^j + \theta_{i \oplus 2^k}^j$.

Figure 10: The Cube Walking Algorithm.

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 $i = 0, 1, \dots, N/2 - 1$. 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 θ vector for every node in the overloaded subcube. The calculation of θ is a local operation without any communication. The value of δ of n' can be calculated by $\delta_{i \oplus 2^j}^j = w_i^{j+1} - w_i^j - q_{i \oplus 2^j}^j$. The γ 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 2: 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 δ_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 δ_i^k . That is, $\sum \theta_i^0 = \delta_i^k = \theta_i^k$. It can be proven by showing that

$$\theta_i^{j+1} = \theta_i^j + \theta_{i \oplus 2^j}^j.$$

There are three cases when assigning the value of θ :

Case 1: when $\delta_i^j < \gamma_i^{j+1}$, we have

$$\begin{aligned} \theta_i^j &= 0; \quad \text{and} \\ \theta_{i \oplus 2^j}^j &= \theta_{i \oplus 2^j}^{j+1}. \end{aligned}$$

$$\text{Hence, } \theta_i^j + \theta_{i \oplus 2^j}^j = \theta_{i \oplus 2^j}^{j+1} = \theta_i^{j+1}.$$

Case 2: when $\theta_i^{j+1} + \gamma_i^{j+1} \geq \delta_i^j > \gamma_i^{j+1}$,

$$\begin{aligned} \text{since } \delta_i^j - \gamma_i^{j+1} &\leq \theta_i^{j+1}, \text{ we have } \theta_i^j = \min(\delta_i^j - \gamma_i^{j+1}, \theta_i^{j+1}) = \delta_i^j - \gamma_i^{j+1}; \\ \text{since } \delta_{i \oplus 2^j}^j &= \delta_i^{j+1} - \delta_i^j = \gamma_i^{j+1} + \theta_i^{j+1} - \delta_i^j \geq 0, \text{ we have } \theta_{i \oplus 2^j}^j = \max(\delta_{i \oplus 2^j}^j, 0) = \delta_{i \oplus 2^j}^j. \end{aligned}$$

$$\text{Hence, } \theta_i^j + \theta_{i \oplus 2^j}^j = \delta_i^j - \gamma_i^{j+1} + \delta_{i \oplus 2^j}^j = \delta_i^{j+1} - \gamma_i^{j+1} = \theta_i^{j+1}.$$

Case 3: when $\delta_i^j > \theta_i^{j+1} + \gamma_i^{j+1}$,

$$\begin{aligned} \text{since } \delta_i^j - \gamma_i^{j+1} &> \theta_i^{j+1}, \text{ we have } \theta_i^j = \min(\delta_i^j - \gamma_i^{j+1}, \theta_i^{j+1}) = \theta_i^{j+1}; \\ \text{since } \delta_{i \oplus 2^j}^j &= \delta_i^{j+1} - \delta_i^j = \gamma_i^{j+1} + \theta_i^{j+1} - \delta_i^j < 0, \text{ we have } \theta_{i \oplus 2^j}^j = \max(\delta_{i \oplus 2^j}^j, 0) = 0. \end{aligned}$$

$$\text{Hence, } \theta_i^j + \theta_{i \oplus 2^j}^j = \theta_i^{j+1}. \quad \square$$

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 of this algorithm is $3d$.

Example 3:

A running example of CWA is shown in Figure 11. 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^k , δ_i^k , θ_i^k , and γ_i^k are as follows:

Node i	d0				d1				d2			
	w_i^0	δ_i^0	θ_i^0	γ_i^0	w_i^1	δ_i^1	θ_i^1	γ_i^1	w_i^2	δ_i^2	θ_i^2	γ_i^2
0	19	11	6	5	30	14	9	5	41	9	9	0
1	11	3	3	0	30	14	9	5	41	9	9	0
2	2	-6	0	-6	11	-5	0	-5	41	9	9	0
3	9	1	0	1	11	-5	0	-5	41	9	9	0

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 , δ_i^k , θ_i^k , and γ_i^k are as follows:

Node i	d0				d1			
	w_i^0	δ_i^0	θ_i^0	γ_i^0	w_i^1	δ_i^1	θ_i^1	γ_i^1
0	13	5	5	0	21	5	5	0
1	8	0	0	0	21	5	5	0
4	6	-2	0	-2	18	2	2	0
5	12	4	2	2	18	2	2	0

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

When $k = 0$, nodes 3, 5, and 6 are overloaded. Their values of w_i^k , δ_i^k , θ_i^k , and γ_i^k are as follows:

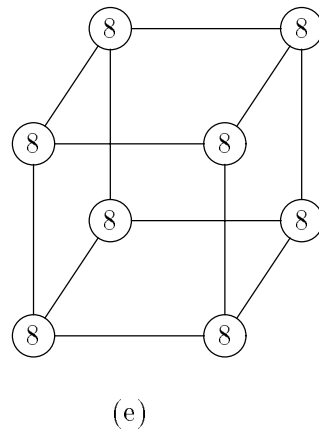
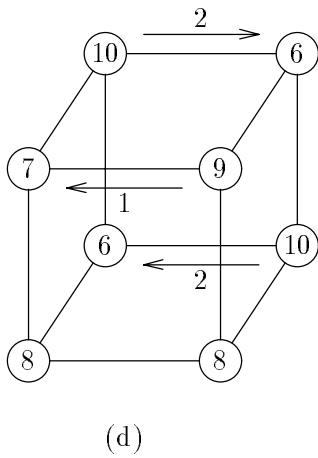
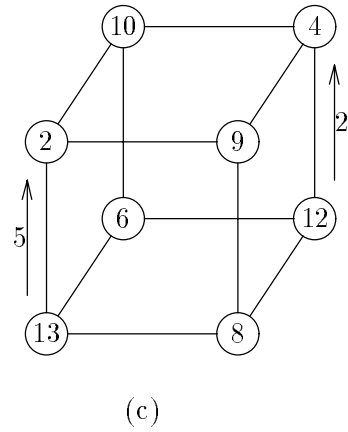
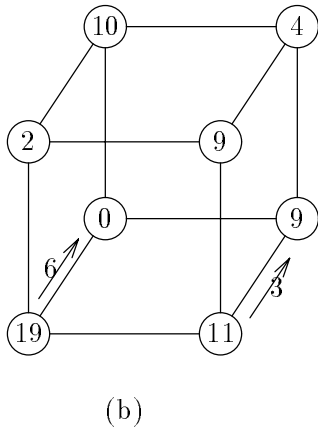
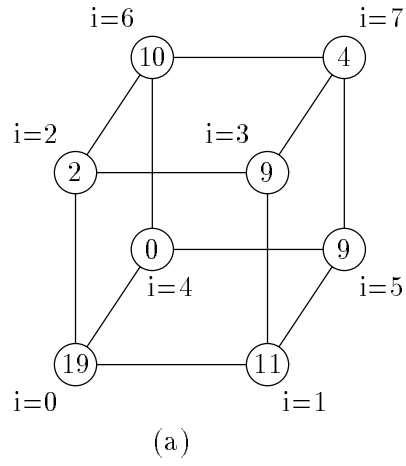


Figure 11: A running example of CWA.

Node i	d0			
	w_i^0	δ_i^0	θ_i^0	γ_i^0
3	9	1	1	0
5	10	2	2	0
6	10	2	2	0

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.

3.3 Mesh Walking Algorithm

A parallel scheduling algorithm for the mesh topology named *Mesh Walking Algorithm (MWA)* is shown in Figure 12. First, we scan the partial vector w along every row, and each node i records a w vector $w_{i,j}^0$, where $j = 0, \dots, i \bmod n_2$. Each node i ($i \bmod n_2 = n_2 - 1$) calculates the sum $w_i^1 = \sum_{l=0}^{n_2-1} w_{i,j}^0$. A *scan* operation is performed along these nodes, each of that keeps another vector $w_{i,j}^1$, where $j = 0, \dots, \lfloor i/n_2 \rfloor$. The values of w_{avg} and R are calculated at node $N - 1$, and spread to all nodes i ($i \bmod n_2 = n_2 - 1$). Consequently, these nodes spread the values of w_{avg} , R , W_i^1 , and $W_{i-n_2}^1$ along each row. Then, the vectors q^0 and δ are calculated in each node, as well as Q_i^0 , Q_i^1 , and $Q_{i-n_2}^1$. The values of Q_i^0 and Q_i^1 can be calculated directly by:

$$Q_i^0 = w_{avg} * (i \bmod n_2 + 1) + r_i$$

where

$$r_i = \begin{cases} (i \bmod n_2) + 1 & \text{if } i < R \\ 0 & \text{if } \lfloor i/n_2 \rfloor \times n_2 \geq R \\ R \bmod n_2 & \text{otherwise} \end{cases}$$

$$Q_i^1 = w_{avg} * \lfloor i/n_2 \rfloor * n_2 + r_i$$

where

$$r_i = \begin{cases} \lfloor i/n_2 \rfloor * n_2 & \text{if } \lfloor i/n_2 \rfloor * n_2 \leq R \\ R & \text{otherwise} \end{cases}$$

In step 4, the first iteration $k = 1$ balances the load among rows. All nodes calculate the values of $\eta_{i,0,L}$ and $\eta_{i,0,R}$. If $\eta_{i,0,L} < 0$, row $r = \lfloor i/n_2 \rfloor$ will receive $|\eta_{i,0,L}|$ tasks from row $r - 1$. If $\eta_{i,0,R} < 0$, row $r = \lfloor i/n_2 \rfloor$ will receive $|\eta_{i,0,R}|$ tasks from row $i + 1$. If $\eta_{i,0,R} > 0$, the submesh from row 0 to row $r = \lfloor i/n_2 \rfloor$ is overloaded, and $\eta_{i,0,R}$ tasks need to be sent to row $r + 1$. Similarly, if $\eta_{i,0,L} > 0$, the submesh below row $r = \lfloor i/n_2 \rfloor$ is underloaded, and $\eta_{i,0,L}$ tasks need to be sent to row $r - 1$. Vector θ is calculated to determine how many tasks at each node need to be sent. The calculation of η and θ is a local operation without any communication. Variable $\gamma_{i,j,l}$ indicates how many tasks are to be reserved for the previous j nodes in the same row, and variable $\eta_{i,j,l}$ tells how many tasks remain to be sent out. The values of w and δ are updated. Iteration 0

Mesh Walking Algorithm (MWA)

Assume a $n_1 \times n_2$ mesh, the number of nodes is $N = n_1 \times n_2$.

1. Global Information Collection:

Perform *scan* operations and compute the w vectors:

$$w_i^0 = w_i, \quad w_i^1 = \sum \{w_j^0 \mid \lfloor j/n_2 \rfloor = \lfloor i/n_2 \rfloor\}, \quad w_i^2 = \sum_{j=0}^{n_1-1} w_{j \times n_2}^1$$

For $j = 0, 1, \dots, i \bmod n_2$, $w_{i,j}^0 = w_{i-(i \bmod n_2)+j}^0$; for $j = 0, 1, \dots, \lfloor i/n_2 \rfloor$, $w_{i,j}^1 = w_{j \times n_2}^1$.

2. Average Load Calculation: $T = w_i^2$, $w_{avg} = \lfloor T/N \rfloor$, $R = T \bmod N$.

3. Quota Calculation: Compute vectors q_i^k , Q_i^k , and δ_i^k , $k = 0, 1$

$$q_i^0 = \begin{cases} w_{avg} + 1 & \text{if } i \leq R \\ w_{avg} & \text{otherwise} \end{cases}, \quad q_i^1 = \sum \{q_j^0 \mid \lfloor j/n_2 \rfloor = \lfloor i/n_2 \rfloor\}$$

$$Q_i^0 = \sum \{q_j^0 \mid j \leq i \text{ and } \lfloor j/n_2 \rfloor = \lfloor i/n_2 \rfloor\}, \quad Q_i^1 = \sum \{q_j^1 \mid j \leq i \text{ and } j \bmod n_2 = 0\}$$

$$\delta_i^k = w_i^k - q_i^k$$

For $j = 0, 1, \dots, i \bmod n_2$, $\delta_{i,j}^0 = \delta_{i-(i \bmod n_2)+j}^0$.

4. Task Exchange: Let $o^1 = n_2$, $o^0 = 1$, $b_i^1 = \lfloor i/n_2 \rfloor$, $b_i^0 = i \bmod n_2$, $v_i^1 = i \bmod n_2$, $v_i^0 = 0$.

For $k = 1$ to 0 do

4.1)

$$W_i^k = \sum_{0 \leq j \leq b_i^k} w_{i,j}^k$$

Let $n_L = i - o^k$, $n_R = i + o^k$.

Initialize $\eta_{i,0,L} = \begin{cases} Q_{n_L}^k - W_{n_L}^k & \text{if } b_i^k > 0 \\ 0 & \text{otherwise} \end{cases}$, $\eta_{i,0,R} = W_i^k - Q_i^k$, $\gamma_{i,0,L} = \gamma_{i,0,R} = 0$

4.2) For $l = L, R$, for node i with $\eta_{i,0,l} < 0$, receive tasks as well as the θ vector from node n_l ; update its own w vector for $j = 0, 1, \dots, v_i^k$: $w_{i,j}^0 = w_{i,j}^0 + \theta_{n_l,j,l}$ and $\delta_{i,j}^0 = \delta_{i,j}^0 + \theta_{n_l,j,l}$.

4.3) For $l = L, R$, for node i with $\eta_{i,0,l} > 0$, calculate the number of tasks to be sent out.

For $j = 0, 1, \dots, v_i^k$

$$\theta_{i,j,l} = \begin{cases} \eta_{i,j,l} & \text{if } \delta_{i,j}^0 \geq \eta_{i,j,l} + \gamma_{i,j,l} \\ \delta_{i,j}^0 - \gamma_{i,j,l} & \text{if } \eta_{i,j,l} + \gamma_{i,j,l} > \delta_{i,j}^0 > \gamma_{i,j,l} \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_{i,j+1,l} = \gamma_{i,j,l} + \theta_{i,j,l} - \delta_{i,j}^0 \quad \text{and} \quad \eta_{i,j+1,l} = \eta_{i,j,l} - \theta_{i,j,l}$$

Send $\theta_{i,v_i^k,l}$ tasks as well as its θ vector to node n_l ; update its own w and δ vectors for $j = 0, 1, \dots, v_i^k$: $w_{i,j}^0 = w_{i,j}^0 - \theta_{i,j,l}$, $\delta_{i,j}^0 = \delta_{i,j}^0 - \theta_{i,j,l}$.

Figure 12: The Mesh Walking Algorithm.

balances the load in each row. The following lemma shows that at the end of the algorithm, each node has the same number of tasks as its quota.

Lemma 3: After execution of MWA, the number of tasks in each node is equal to its quota.

Proof: In iteration 1,

$$\eta_{i,v_i^1+1,l} = \eta_{i,0,l} - \sum_{j=0}^{v_i^1} \theta_{i,j,l} \quad (1)$$

$$\gamma_{i,v_i^1+1,l} = \gamma_{i,0,l} + \sum_{j=0}^{v_i^1} \theta_{i,j,l} - \sum_{j=0}^{v_i^1} \delta_{i,j}^0 \quad (2)$$

When $\delta_{i,j}^0 \geq \eta_{i,j,l} + \gamma_{i,j,l}$, $\eta_{i,j+1,l} = \eta_{i,j,l} - \eta_{i,j,l} = 0$. When $\gamma_{i,j,l} < \delta_{i,j}^0 < \eta_{i,j,l} + \gamma_{i,j,l}$, $\eta_{i,j+1,l} = \eta_{i,j,l} - \delta_{i,j}^0 + \gamma_{i,j,l} \geq 0$. And when $\delta_{i,j} \leq \gamma_{i,j,l}$, $\eta_{i,j+1,l} = \eta_{i,j,l} - 0$. Since $\eta_{i,0,l}$ is larger than or equal to 0,

$$\eta_{i,j+1,l} \geq 0 \quad (3)$$

Because of (1) and (3)

$$\sum_{j=0}^{v_i^1} \theta_{i,j,l} \leq \eta_{i,0,l} \quad (4)$$

Assume for some $j' \leq v_i^1$, $\delta_{i,j'}^0 \geq \eta_{i,j',l} + \gamma_{i,j',l}$. Then $\eta_{i,j'+1,l} = \eta_{i,j',l} - \eta_{i,j',l} = 0$. Because of (1), $\sum_{j=0}^{j'} \theta_{i,j,l} = \eta_{i,0,l}$. Since each $\theta_{i,j,l}$ for $j > j'$ is equal to 0, we have

$$\sum_{j=0}^{v_i^1} \theta_{i,j,l} = \eta_{i,0,l} \quad (5)$$

Assume there exists no $j' \leq v_i^1$ so that $\delta_{i,j'}^0 \geq \eta_{i,j',l} + \gamma_{i,j',l}$. When $\eta_{i,j,l} + \gamma_{i,j,l} > \delta_{i,j}^0 > \gamma_{i,j,l}$, $\theta_{i,j,l} = \delta_{i,j}^0 - \gamma_{i,j,l}$ and $\gamma_{i,j+1,l} = \gamma_{i,j,l} + (\delta_{i,j}^0 - \gamma_{i,j,l}) - \delta_{i,j}^0 = 0$. When $\delta_{i,j}^0 \leq \gamma_{i,j,l}$, $\theta_{i,j,l} = 0$ and $\gamma_{i,j+1,l} = \gamma_{i,j,l} - \delta_{i,j}^0 \geq 0$. Therefore,

$$\gamma_{i,v_i^1+1,l} \geq 0 \quad (6)$$

From (1) and (2), and $\gamma_{i,0,l} = 0$

$$\eta_{i,v_i^1+1,R} + \gamma_{i,v_i^1+1,R} = \eta_{i,0,R} - (\delta_i^1 - \sum_{j=0}^{v_i^1} \theta_{i,j,L}) \quad (7)$$

Because of (4) and $\eta_{i,0,L} + \eta_{i,0,R} = \delta_i^1$

$$\eta_{i,0,R} - \delta_i^1 + \sum_{j=0}^{v_i^1} \theta_{i,j,L} \leq 0 \quad (8)$$

Because of (3), (6), (7), and (8)

$$\eta_{i,v_i^1+1,R} = 0 \quad (9)$$

$$\gamma_{i,v_i^1+1,R} = 0 \quad (10)$$

and

$$\sum_{j=0}^{v_i^1} \theta_{i,j,L} = \delta_i^1 - \eta_{i,0,R} = \eta_{i,0,L}$$

From (1) and (9)

$$\sum_{j=0}^{v_i^1} \theta_{i,j,R} = \eta_{i,0,R}$$

Because $\eta_{i,0,L} = -\eta_{i-n_2,0,R}$ and $\eta_{i,0,R} = -\eta_{i+n_2,0,L}$, if $\eta_{i,0,L} < 0$, node i will receive $|\eta_{i,0,L}|$ tasks from node $i - n_2$. And if $\eta_{i,0,R} < 0$, node i will receive $|\eta_{i,0,R}|$ tasks from node $i + n_2$. Therefore, after iteration 1,

$$w_i^{1'} = w_i^1 - \eta_{i,0,L} - \eta_{i,0,R} = q_i^1$$

The weight w_i^0 is updated. In iteration 0, $\theta_{i,0,l} = \eta_{i,0,l}$. Therefore, after iteration 0, the number of tasks in each row is

$$w_i^{0'} = w_i^0 - \eta_{i,0,L} - \eta_{i,0,R} = q_i^0 \quad \square$$

In this algorithm, step 1 spends n_2 communication steps to collect load information along each row and n_1 communication steps to collect load information across rows. Broadcasting and spreading operations spend $n_1 + n_2$ communication steps. Step 4 spends at most $n_1 + n_2$ communication steps for load balancing. Therefore, the total communication steps of this algorithm is $3(n_1 + n_2)$.

Example 4:

A running example of MWA is shown in Figure 13. The total number of tasks is computed by parallel reduction. The values of w_{avg} and R are calculated:

$$w_{avg} = 8, \quad R = 0.$$

Because $R = 0$, every node has the same q_i^0 and q_i^1 , which are 8 and 32, respectively. The values of Q_i^1 are also calculated:

$$Q_{0-3}^1 = 32, \quad Q_{4-7}^1 = 64, \quad Q_{8-11}^1 = 96, \quad Q_{12-15}^1 = 128.$$

The values of w_i^0 and δ_i^0 are listed as follows:

Node	w_i^0	δ_i^0	Node	w_i^0	δ_i^0	Node	w_i^0	δ_i^0	Node	w_i^0	δ_i^0
0	7	-1	1	12	4	2	6	-2	3	16	8
4	17	9	5	3	-5	6	0	-8	7	15	7
8	2	-6	9	13	5	10	5	-3	11	5	-3
12	5	-3	13	6	-2	14	4	-4	15	12	4

Nodes in the same row have the same values of w_i^1 , δ_i^1 , W_i^1 , $\eta_{i,0,l}$, and $\eta_{i,0,R}$ which are listed as follows:

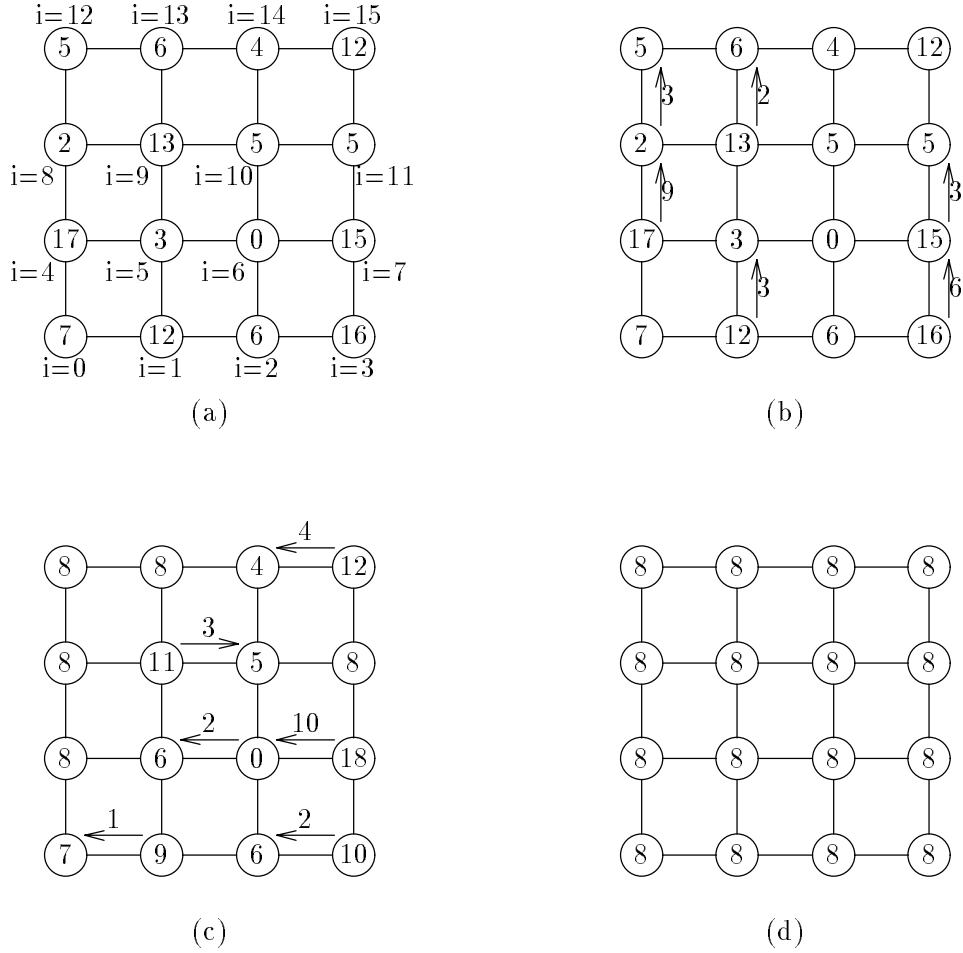


Figure 13: A running example of MWA.

Node	w_i^1	δ_i^1	W_i^1	$\eta_{i,0,L}$	$\eta_{i,0,R}$
0-3	41	9	41	0	9
4-7	35	3	76	-9	12
8-11	25	-7	101	-12	5
12-15	27	-5	128	-5	0

In iteration 1, every $\eta_{i,0,L} \leq 0$. Therefore, tasks are sent in one direction for this example. The values of $\delta_{i,j}^0$, $\eta_{i,j,R}$, $\gamma_{i,j,R}$, and $\theta_{i,j,R}$ are listed as follows:

Node	$\delta_{i,0}^0$	$\eta_{i,0}$	$\gamma_{i,0}$	$\theta_{i,0}$	$\delta_{i,1}^0$	$\eta_{i,1}$	$\gamma_{i,1}$	$\theta_{i,1}$	$\delta_{i,2}^0$	$\eta_{i,2}$	$\gamma_{i,2}$	$\theta_{i,2}$	$\delta_{i,3}^0$	$\eta_{i,3}$	$\gamma_{i,3}$	$\theta_{i,3}$
3	-1	9	0	0	4	9	1	3	-2	6	0	0	8	6	2	6
7	9	12	0	9	-2	3	0	0	-8	3	2	0	13	3	10	3
11	3	5	0	3	5	2	0	2	-3	0	-3	0	0	0	0	0

In row 0, node 1 sends three tasks to node 5, and node 3 sends six tasks to node 7. When nodes

in row 1 receive the tasks and θ vectors, they update their δ . Then, they calculate the θ vectors. Node 4 sends nine tasks to node 8, and node 7 sends three tasks to node 11. Finally, nodes in row 2 update their δ and calculate the θ vectors. Node 8 sends three tasks to node 12, and node 9 sends two tasks to node 13. The task exchange is shown in Figure 13(b). The number of tasks in each row now is equal to its quota q_i^1 , which is 32.

In iteration 0, W_i^0 is calculated from updated values of w . The values of Q_i^0 are:

$$Q_{0,4,8,12}^1 = 8, \quad Q_{1,5,9,13}^1 = 16, \quad Q_{2,6,10,14}^1 = 24, \quad Q_{3,7,11,15}^1 = 32.$$

The values of η vectors are as follows:

Node	$\eta_{i,0,L}$	$\eta_{i,0,R}$	Node	$\eta_{i,0,L}$	$\eta_{i,0,R}$	Node	$\eta_{i,0,L}$	$\eta_{i,0,R}$	Node	$\eta_{i,0,L}$	$\eta_{i,0,R}$
0	0	-1	1	-1	0	2	0	-2	3	-2	0
4	0	0	5	0	-2	6	-2	-10	7	-10	0
8	0	0	9	0	3	10	3	0	11	0	0
12	0	0	13	0	0	14	0	-4	15	-4	0

Nodes exchange tasks, as shown in Figure 13(c), according to the values of θ which is equal to η . This results in a balanced load, and each node has eight tasks. The total number of task-hops is 48.

4. Properties of the Scheduling Algorithms

In this section, we discuss the scheduling quality, locality, and communication costs of the TWA, CWA, and MWA algorithms. The next theorem shows that these algorithms are able to fully balance the load. If the number of tasks can be equally divided by the number of nodes, then each node will have the equal number of tasks; otherwise, the number of tasks in each node differs by one.

Theorem 1: The difference in the number of tasks in each node is at most one after execution of TWA, CWA, or MWA.

Proof: From Lemmas 1, 2, and 3, the number of tasks in each node is equal to its quota after execution of TWA, CWA, or MWA. 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. \square

These algorithms also maximize 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 maximum number of local tasks and the minimum number of non-local tasks. In Lemmas 4 and 5 and Theorems 2 and 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 4: 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. \square

The next theorem proves that these three algorithms maximize locality.

Theorem 2: The number of non-local tasks in the TWA, CWA, or MWA algorithm is

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

Proof: At any time when executing the TWA, CWA, or MWA algorithm, the number of tasks in each node is not less than $\min(w, w_{avg})$. In TWA, each node receives tasks before sending tasks. In CWA or MWA, 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 4, these algorithms minimize the number of non-local tasks and maximize locality. \square

TWA is an optimal scheduling algorithm. The next theorem proves that TWA minimizes the number of task-hops and communication.

Theorem 3: The TWA algorithm minimizes $\sum_k e_k$, the total number of task-hops, and the total number of communications.

Proof: For an edge k that connects a subtree i and its parent, if $Q_i \geq W_i$, $e_k = Q_i - W_i$, which is the minimum number of tasks to be transmitted from its parent to the subtree. Similarly, if $Q_i < W_i$, $e_k = W_i - Q_i$, which is the minimum number of tasks to be transmitted from the subtree to its parent. Therefore, $\sum_k e_k$, the total number of task-hops, is minimized.

For each subtree i , if $Q_i \neq W_i$, then there is at least one communication between the subtree and its parent, which is the minimum number of communications. If $Q_i = W_i$, then there is no

communication between the subtree and its parent. Therefore, the total number of communications is minimized. \square

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

Lemma 5: The CWA and MWA algorithms minimize 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 [15]. 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 either CWA or MWA, the longest path has two edges. Therefore, there is no negative cycle. \square

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

TWA is an optimal algorithm. It minimizes communication and maximizes locality while balancing the load. The optimality of heuristic algorithms, CWA and MWA, 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 study CWA and compare its performance to DEM. CWA can fully balance the load but DEM cannot in most cases. Table I shows the percentage of fully-balanced cases of the DEM algorithm. We run the DEM algorithm with different numbers of processors and different weights. Each result is from 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

Table I: The Percentage of Fully-Balanced Cases of DEM

Number of Processors	Average weight					
	2	5	10	20	50	100
4	74.30%	75.30%	75.70%	75.10%	74.50%	74.60%
8	31.20%	30.60%	34.80%	33.20%	36.20%	32.00%
16	4.00%	5.80%	6.30%	7.60%	5.40%	7.70%
32	0.00%	0.20%	0.10%	0.30%	0.20%	0.70%
64	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

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 14 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.

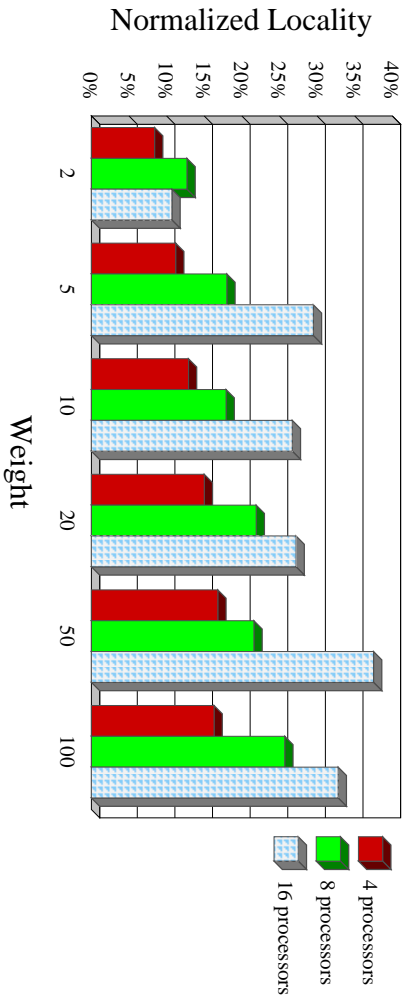
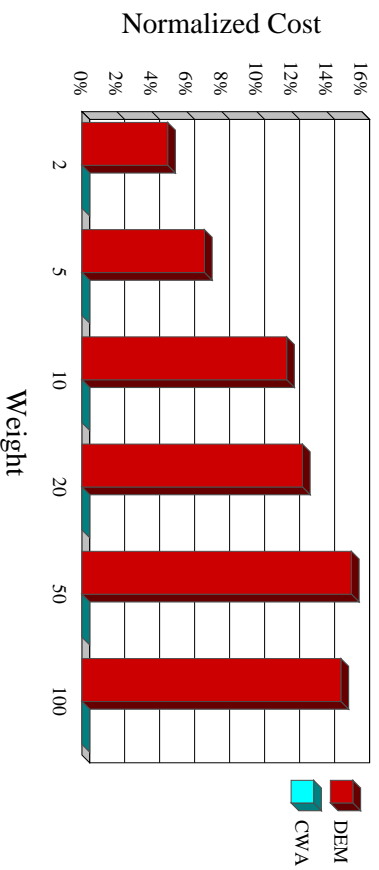


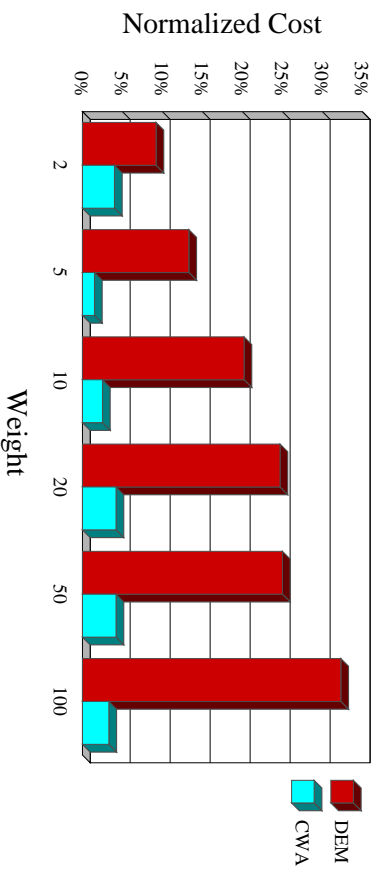
Figure 14: Normalized locality of DEM

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

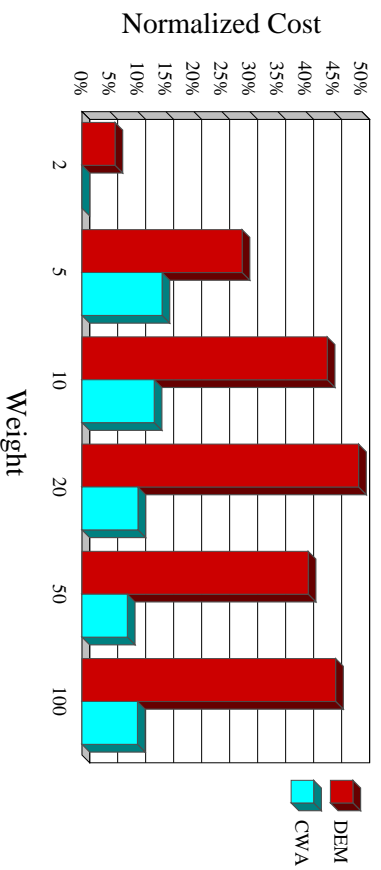
$$\frac{C_{DEM} - C_{OPT}}{C_{OPT}}$$



(a) 4 processors

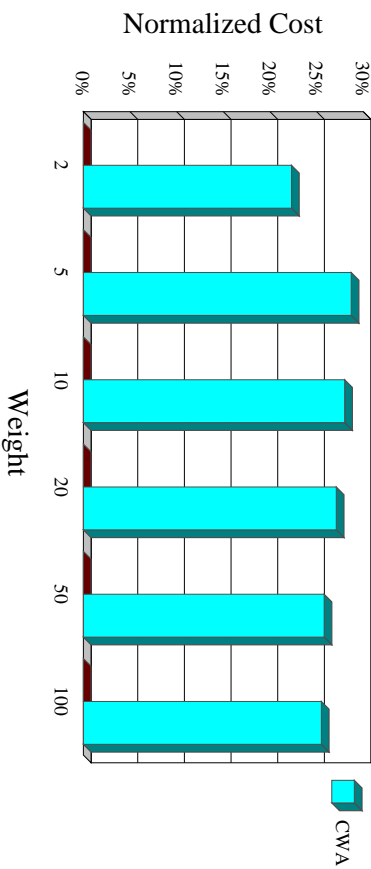


(b) 8 processors

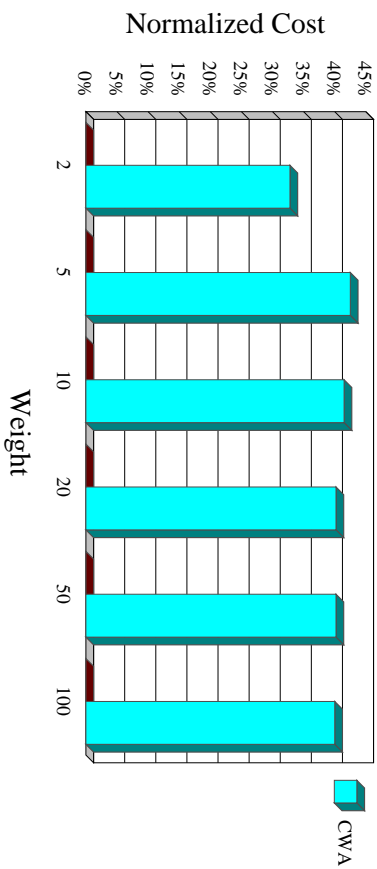


(c) 16 processors

Figure 15: Normalized communication costs of DEM and CWA



(a) 64 processors



(b) 256 processors

Figure 16: Normalized communication costs of CWA

and

$$\frac{C_{CWA} - C_{OPT}}{C_{OPT}},$$

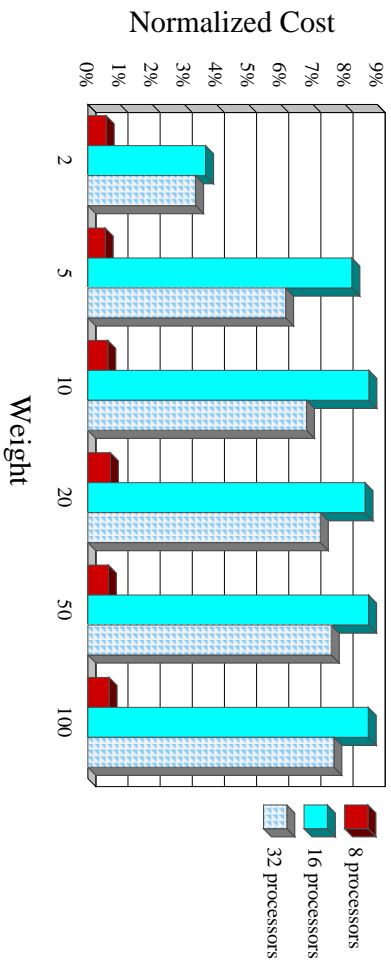
where C_{DEM} , C_{CWA} , and C_{OPT} are the number of task-hops of the DEM, CWA, and optimal algorithms, respectively. Figure 15 compares the normalized communication costs on 4, 8, and 16 processors. Each result is the average of the DEM fully-balanced cases in 1,000 test cases. The number of task-hops 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 16 shows the normalized communication costs of CWA on 64 and 256 processors. Each data presented here is the average of 100 different test cases.

The method and assumptions used for performance study of the MWA algorithm are the same as those for the CWA algorithm. MWA is able to fully balance the load and maximize locality. However, its communication is not minimized in most cases. The normalized communication cost

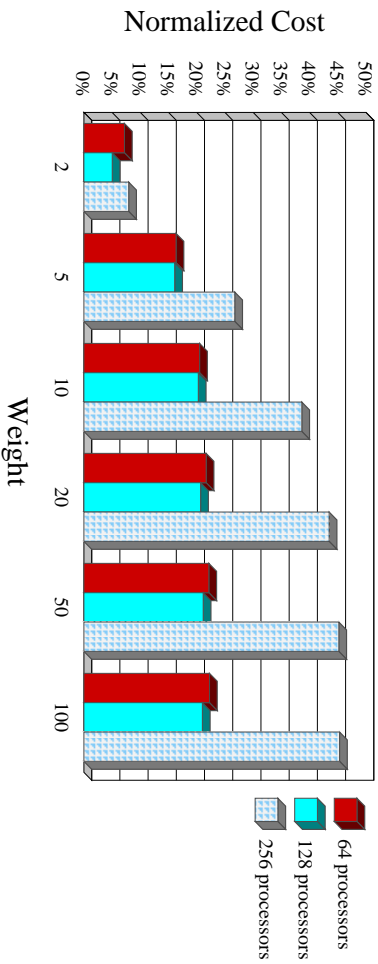
of MWA with respect to the optimal algorithm is measured by

$$\frac{C_{MWA} - C_{OPT}}{C_{OPT}},$$

where C_{MWA} and C_{OPT} are the numbers of task-hops of the MWA and optimal algorithms, respectively. As mentioned in Lemma 3, the number of task-hops of MWA on two or four processors is the minimum. Figure 17 shows the normalized communication costs on 8 to 256 processors. The mesh organization is either $M \times M$ or $M \times M/2$. Each data presented here is the average of 100 different test cases. For small meshes, MWA provides a nearly optimal result. The cost increases with the number of processors.



(a) 8, 16, and 32 processors



(b) 64, 128, and 256 processors

Figure 17: Normalized communication cost of MWA.

6. Previous Works

Parallel scheduling and static scheduling share some common ideas [1, 2, 3, 4, 18]. Both of them utilize global information to achieve high quality load balancing. But, parallel scheduling is different from static scheduling in three aspects. First, the scheduling activity is performed at runtime. Therefore, it can deal with the dynamic problems. Second, the possible load imbalance caused by inaccurate grain size estimation can be corrected by the next turn of scheduling. Third, it eliminates the requirement of large memory space to store task graphs, as scheduling is conducted in an incremental fashion. It then leads to better scalability for massively parallel machines and large size applications.

Large research efforts have been directed towards process allocation in distributed systems [7, 5, 6, 19, 20, 21, 22, 23]. A recent comparison study of dynamic load balancing strategies on highly parallel computers is given by Willebeek-LeMair and Reeves [16]. Eager *et al.* compared the sender-initiated algorithm with the receiver-initiated algorithm [6]. Work with a similar assumption to ours includes the Gradient Model developed by Lin and Keller [24]. The randomized allocation algorithms developed by different authors are quite simple and effective [25, 5, 26, 27]. The adaptive contracting within neighborhood (ACWN) [22] and receiver-initiated diffusion (RID) [16] are other effective algorithms.

Runtime parallel scheduling is similar to dynamic scheduling to a certain degree. Both methods schedule tasks at runtime instead of compile-time. Their scheduling decisions, in principle, depend on and adapt to the runtime system information. However, substantial differences make them appear as two separate categories. First, the system functions and user computation are mixed together in dynamic scheduling, but there is a clear cutoff between system and user phases in runtime parallel scheduling, which potentially offers easy management and low overhead. Second, placement of a task in dynamic scheduling is basically an individual action by a processor based on partial system information; whereas in parallel scheduling, the scheduling activity is always an aggregate operation based on global system information.

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 [28, 29]. Some other works also deal with this type of problem [30, 31, 32]. The project PARTI automates prescheduling for nonuniform problems [33]. The dimension exchange method (DEM) is applied to application problems without geometric structure [9]. It was conceptually designed for a hypercube system but may be applied to other topologies, such as k -ary

n -cubes [34]. It balances load for independent tasks with an equal grain size. The method has been extended by Willebeek-LeMair and Reeves [16] so that the algorithm can run incrementally to correct the unbalanced load caused by varied grain sizes. Nicol has proposed a direct mapping algorithm which computes the total number of tasks by using sum-reduction [10]. However, it does not minimize the communication cost, nor eliminate communication conflict. An incremental scheduling for N-body simulation is presented in [35]. The task graph is rescheduled periodically to correct the load imbalance. However, its runtime scheduling has not yet been parallelized.

7. Conclusion

Recent research has demonstrated that runtime parallel scheduling can provide a low-overhead load balancing with global load information. In parallel scheduling, a synchronous approach removes the stability problem and is able to balance the load quickly and accurately. Parallel scheduling combines the advantages of static scheduling and dynamic scheduling. All processors cooperate to collect load information and to exchange workload in parallel. With parallel scheduling, it is possible to obtain high quality load balancing with a fully-balanced load and maximized locality. Communication costs can be reduced significantly. Three algorithms for tree, hypercube, and mesh networks have been presented in this paper. It is not difficult to develop an algorithm for the k -ary n -cube by combining the CWA and MWA algorithms.

Acknowledgments

The author would like to thank Xin He for his helpful discussion.

References

- [1] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, June 1990.
- [2] M. Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, pp. 330–343, July 1990.
- [3] T. Yang and A. Gerasoulis, "PYRROS: Static task scheduling and code generation for message-passing multiprocessors," *The 6th ACM Int'l Conf. on Supercomputing*, July 1992.
- [4] Y. Chung and S. Ranka, "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors," in *Supercomputer '92*, Nov. 1992.
- [5] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 662–674, May 1986.
- [6] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Performance Eval.*, vol. 6, pp. 53–68, Mar. 1986.

- [7] N. G. Shivaratri, P. Krieger, and M. Singhal, "Load distributing for locally distributed systems," *IEEE Computer*, vol. 25, pp. 33–44, Dec. 1992.
- [8] S. Ranka, Y. Won, and S. Sahni, "Programming a hypercube multicomputer," *IEEE Software*, pp. 69–77, Sept. 1988.
- [9] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. of Parallel Distrib. Comput.*, vol. 7, pp. 279–301, 1989.
- [10] D. Nicol, "Communication efficient global load balancing," in *The Scalable High Performance Computing Conference*, pp. 292–299, Apr. 1992.
- [11] W. Shu and M. Wu, "Runtime Incremental Parallel Scheduling (RIPS) on distributed memory computers," *IEEE Trans. Parallel and Distributed System*, vol. 7, pp. 637–649, June 1996.
- [12] I. Ahmad and Y. Kwok, "A parallel approach to multiprocessor scheduling," in *Int'l Parallel Processing Symposium*, pp. 289–293, Apr. 1995.
- [13] M. Wu, "An efficient parallel scheduling algorithm," in *IEEE Symposium on Parallel and Distributed Processing*, Oct. 1996.
- [14] G. E. Blelloch, *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.
- [15] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.
- [16] M. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Trans. Parallel and Distributed System*, vol. 9, pp. 979–993, Sept. 1993.
- [17] S. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan, "Analysis of a graph coloring based distributed load balancing algorithm," *Journal of Parallel and Distributed Computing*, vol. 10, pp. 160–166, 1990.
- [18] Y. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel and Distributed System*, vol. 7, pp. 506–521, May 1996.
- [19] J. A. Stankovic, "Simulations of three adaptive, decentralized controlled, job scheduling algorithms," *Computer Networks*, vol. 8, pp. 199–217, June 1984.
- [20] T. L. Casavant and J. G. Kuhl, "Analysis of three dynamic distributed load-balancing strategies with varying global information requirements," in *Int'l Conf. on Distributed Computing System*, pp. 185–192, May 1987.
- [21] Y.-T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Trans. Comput.*, vol. C-34, pp. 204–217, Mar. 1985.
- [22] W. Shu, "Adaptive dynamic process scheduling on distributed memory parallel computers," *Scientific Programming*, vol. 3, pp. 341–352, 1994.
- [23] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," in *The Thirty-fifth Annual Symposium on Foundation of Computer Science*, pp. 356–368, Nov. 1994.
- [24] F. C. H. Lin and R. M. Keller, "The gradient model load balancing method," *IEEE Trans. Software Engineering*, vol. 13, pp. 32–38, Jan. 1987.

- [25] W. C. Athas, *Fine Grain Concurrent Computations*. PhD thesis, Dept. of Computer Science, California Institute of Technology, May 1987.
- [26] R. M. Karp and Y. Zhang, “A randomized parallel branch-and-bound procedure,” *Journal of ACM*, vol. 40, pp. 765–789, 1993.
- [27] S. Chakrabarti, A. Ranade, and K. Yelick, “Randomized load balancing for tree structured computation,” in *IEEE Scalable High Performance Computing Conference*, pp. 666–673, 1994.
- [28] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors*, vol. I. Prentice-Hall, 1988.
- [29] J. Salmon, “Parallel hierarchical N-body methods,” tech. rep., CRPC-90-14, Center for Research in Parallel Computing, Caltech, 1990.
- [30] K. M. Dragon and J. L. Gustafson, “A low-cost hypercube load balance algorithm,” in *Proc. of the 4th Conf. on Hypercube Concurrent Computers and Applications*, pp. 583–590, 1989.
- [31] M. Berger and S. Bokhari, “A partitioning strategy for non-uniform problems on multiprocessors,” *IEEE Trans. Computers*, vol. C-26, pp. 570–580, 1987.
- [32] S. B. Baden, “Dynamic load balancing of a vortex calculation running on multiprocessors,” Tech. Rep. Vol. 22584, Lawrence Berkeley Lab., 1986.
- [33] J. Saltz, R. Mirchandaney, R. Smith, D. Nicol, and K. Crowley, “The PARTY parallel runtime system,” in *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 1987.
- [34] C. Z. Xu and F. C. M. Lau, “The generalized dimension exchange method for load balancing in k-ary n-cubes and variants,” *Journal of Parallel and Distributed Computing*, vol. 24, pp. 72–85, Jan. 1995.
- [35] A. Gerasoulis, J. Jiao, and T. Yang, “Experience with graph scheduling for mapping irregular scientific computation,” in *First Workshop on Solving Irregular Problems on Distributed Memory Machines in conjunction with International Parallel Processing Symposium*, pp. 1–8, Apr. 1995.