On Runtime Parallel Scheduling

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 together to schedule work. Parallel scheduling is able to accurately balance the load by using global load information at compile-time or runtime. It provides a high-quality load balancing. This paper presents an overview of the parallel scheduling technique. Particular scheduling algorithms for tree, hypercube, and mesh networks are presented. These algorithms can fully balance the load and maximize locality at runtime. Communication costs are significantly reduced compared to other existing algorithms.

1. Introduction

One of the challenges in programming parallel machines is to schedule work to processors [18]. There are two types of application problem structures: problems with a predictable structure, also called *static problems*, and problems with an unpredictable structure, called *dynamic problems*. Solving dynamic problems is difficult since the number of tasks and the grain size of a task may not be known prior to execution.

There are two basic scheduling strategies: *static scheduling* and *dynamic scheduling*. Static scheduling distributes the workload before runtime, and can be applied to static problems. Most existing static scheduling algorithms are sequential, executed on a single processor system. Dynamic scheduling performs scheduling activities concurrently at runtime, which applies to dynamic problems. Although a dynamic scheduling can apply to static problems as well, we usually use static scheduling for static problems because static scheduling provides a more balanced load distribution than dynamic scheduling, as well as avoiding the runtime overhead.

Static scheduling utilizes the knowledge of problem characteristics to reach a global optimal, or nearly-optimal, solution with well-balanced load. It has recently attracted considerable attention among the research community [17, 34, 16, 36, 11, 22]. The quality of scheduling heavily relies on accuracy of weight estimation. The requirement of large memory space to store the task graph restricts the scalability of static scheduling. In addition, static scheduling is not able to balance the load for dynamic problems.

Dynamic scheduling has certain advantages. It is a general approach suitable for a wide range of applications. It can adjust load distribution based on runtime system load information [14, 15, 28]. However, most runtime scheduling algorithms utilize neither the characteristics information of application problems, nor global load information for load balancing decision. Efforts to collect load information for a scheduling decision certainly compete the resource with the underlying computation during runtime. System stability usually sacrifices both quality and quickness of load balancing.

It is possible to design a scheduling strategy that combines the advantages of static and dynamic scheduling. This scheduling strategy should be able to generate well-balanced load without incurring large overhead. With advanced parallel scheduling techniques, this ideal scheduling becomes feasible. In a parallel scheduling, all processors are cooperated together to schedule work. Some parallel scheduling algorithms have been introduced in [26, 13, 7, 12, 33, 1]. Parallel scheduling utilizes global load information stored in all processors and is able to accurately balance the load. As an alternative strategy to the other static and dynamic scheduling, it provides a high-quality, scalable scheduling. Parallel scheduling opens a new direction for compile-time and runtime load balancing.

Parallel scheduling can be applied to static problems. Most static scheduling algorithms running on a single processor are not scalable to massively parallel computers. To speed up the scheduling algorithm and to relax the demand of memory space, a scheduling algorithm can be parallelized. There are two possible schemes of parallel scheduling algorithms: the vertical scheme and the horizontal scheme. In the vertical scheme, each processor will be assigned a portion of work using a *space domain partitioning*. This scheme is able to provide a high-quality scheduling, but involves in a large communication overhead for frequent information exchange. In the horizontal scheme, each processor will be assigned a portion of work using a *time domain partitioning*. Each processor will schedule its own portion of the graph before exchanging information with each other to determine the final schedule. A low-overhead algorithm can be developed in the horizontal scheme. However, this kind of algorithm is not scalable to massively parallel systems, and usually provides a sub-optimal scheduling. Kwok and Ahmad have developed an algorithm in the horizontal scheme [1]. The scheduling length obtained by the parallel scheduling algorithm is about 10% longer than its sequential counterpart.

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 [29]. This parallel scheduling system is shown in Figure 1. 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. Each individual processor determines if it is ready to transfer to the next system phase based on its local condition. In the second system phase, the old tasks that have not been executed will be scheduled together with the newly generated tasks. This process will repeat iteratively until all the computations are completed.

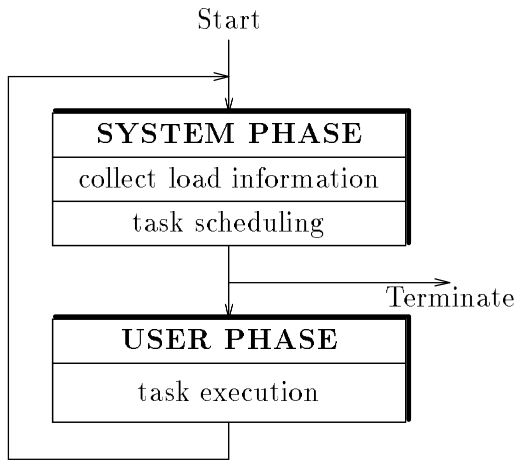


Figure 1: A parallel scheduling system.

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. They can also be used as static scheduling algorithms for independent jobs or tasks. These algorithms can fully balance the load and maximize locality. It significantly reduces 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 sections 3, 4, and 5, respectively. The properties of these algorithms are described in section 6. Previous works are discussed in section 7, and section 8 concludes the paper.

2. The Optimal Scheduling Problem

Our goal is to schedule works so that each processor has the same work load. To achieve this goal, an estimation of the task execution time is needed, which can be done either by a programmer or by a compiler. Sometimes, the estimation is application-specific, leading to a less general approach, and sometimes it is difficult to obtain such an estimation. Due to these difficulties, each task is presumed to require the equal execution time and the goal of the algorithm is to schedule tasks so that each processor has the same number of tasks. The inaccuracy due to the grain-size variation can be corrected in the next system phase.

The other goal is to minimize communication overhead for this load balancing process. Ideally, the tasks with less communication/computation ratios should have a higher priority to migrate. Although the communication/computation ratio is difficult to predict, it has been observed that the communication/computation ratio does not change substantially in a single application. Thus, we can use the number of task migrated instead of the actual communication cost as the objective function.

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. If $w_i < w_{avg}$, the node needs to know where to receive the tasks.

An *optimal scheduling* minimizes task communications. 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 [23] as follows. Each edge is 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. 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. Figure 2 shows a load distribution in an eight-node hypercube network. The graph constructed for Figure 2 is given in Figure 3, where $w_{avg} = 8$. The minimum cost algorithm [23] generates a solution as shown in Figure 4.

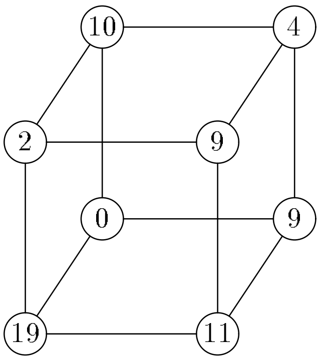


Figure 2: A load distribution in a 3-dimensional hypercube.

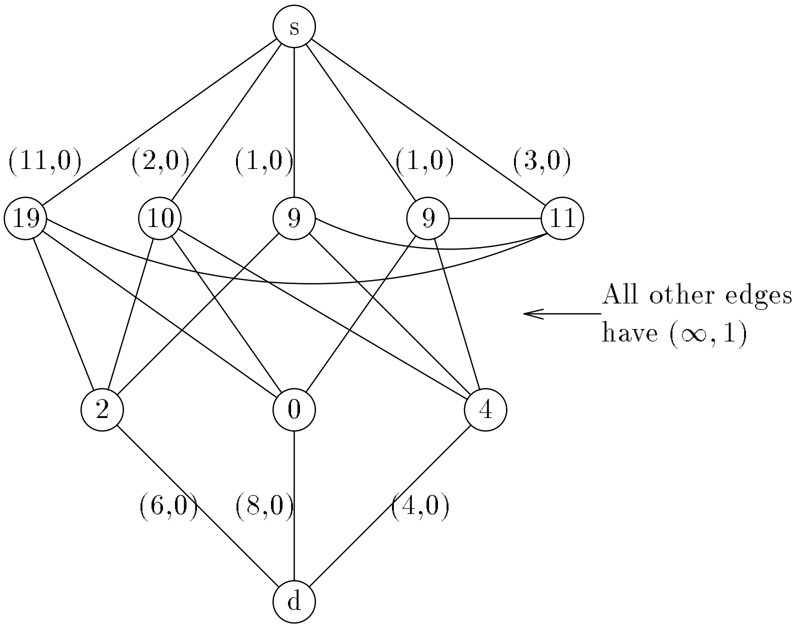


Figure 3: Graph for optimal scheduling problem (Figure 2).

The complexity of the minimum cost algorithm is $O(N^2 v)$, where N is the number of nodes and v is the desired flow value [23]. 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 processors. For a topology other than trees, we need to find a heuristic algorithm.

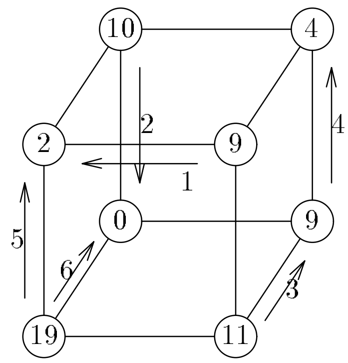


Figure 4: An optimal solution of Figure 2.

In the following sections, we will present three algorithms for the tree, hypercube, and mesh topologies. The tree algorithm is an optimal algorithm with parallel complexity of $O(log\ N)$. The hypercube and mesh algorithms are heuristic algorithms. The hypercube algorithm has a complexity of $O(log\ N)$. The complexity of the mesh algorithm is $O(n_1 + n_2)$ for a $n_1 \times n_2$ mesh. The common feature of these algorithms is that the total number of tasks is obtained by a global reduction operation so that the average number of tasks per node can be calculated. A node will not send its tasks to other nodes unless the number of tasks in the node exceeds the average. Therefore, only necessary tasks are migrated. Other algorithms, such as DEM [12], do not utilize this global information, leading to unnecessary communications.

3. Scheduling Algorithm for the Tree Topology

When the network topology is a tree, the complexity of optimal scheduling can be reduced. Here we present a parallel scheduling algorithm for tree structured interconnection network. Each node in the tree represents a processor. The algorithm, called *Tree Walking Algorithm (TWA)*, is shown in Figure 5. The first step will be executed only at the system setup time, but not in every system phase. Steps 2 and 3 collect the system load information. In step 2, the total number of tasks is counted with a global reduction operation. At the same time, each node records the number of tasks in its subtree and its children's subtrees, if any. In step 3, the root calculates the average number of tasks per node and then broadcasts the number to every node so that each node knows if its subtree is over-loaded or under-loaded in step 4. 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 others. In step 5, the workload is exchanged so that at the end of the system phase, each node has the almost same number of tasks.

In this algorithm, steps 2 and 3 spend $2m$ communication steps, where m is the depth of the

Tree Walking Algorithm (TWA)

1. Assign each node an *order* i according to the preorder traversal; and N_i, the number of nodes of its subtree, where $i = 0, 1, ..., N-1$, and $N = N_0$ is the number of nodes in the system. This step executes only at the system setup time.

2. Let w_i be the number of tasks in node i. Perform a *sum* reduction of w_i and let W_i denote the sum of w_i in the subtree of node i. Each node also keeps records of W_i of its children (if any).

3. The total number of tasks $T = W_0$. The root calculates $w_{avg} = \lfloor T/N \rfloor$ and $R = T \ mod \ N$, and broadcasts w_{avg} and R to all other nodes.

4. Each node calculates its quota q_i that indicates how many tasks are to be scheduled to the node:
$$q_i = \begin{cases} w_{avg} + 1 & if \ i < R \\ w_{avg} & if \ i \geq R \end{cases}$$
Each subtree, rooted at node i, calculates its quota
$$Q_i = \sum_{all \ nodes \ in \ subtree \ i} q_j$$
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
$$r_i = \begin{cases} 0 & if \ i \geq R \\ N_i & if \ i \leq R - N_i \\ R - i & if \ R - N_i < i < R \end{cases}$$
Each node keeps records of Q_i and Q_j, where node j is node i's child (if any).

5. Each node i waits for the incoming message from its parent if $W_i < Q_i$, and the incoming message from each of its children j if $W_j > Q_j$.

   Each node i, after it has received all incoming messages, sends $(W_i - Q_i)$ tasks to its parent if $W_i > Q_i$; and for each child j, if $W_j < Q_j$, sends $(Q_j - W_j)$ tasks to child j.

Figure 5: The Tree Walking Algorithm

tree. The communication steps in step 5 is that 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, $m = logN$, and the number of communication steps of this parallel algorithm on N processors is $O(logN)$.

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 at step 2. The root calculates the value of w_{avg} and R:

$$w_{avg} = 4, \; R = 5$$

Then, each node calculates the value of Q_i at step 4. The values of w_i, N_i, W_i, and Q_i are as shown below:

i	w_i	N_i	W_i	Q_i
0	1	9	41	41
1	4	3	20	15
2	5	1	5	5
3	11	1	11	5
4	7	2	9	9
5	2	1	2	4
6	3	3	11	12
7	3	1	3	4
8	5	1	5	4

The numbers of tasks to be exchanged between nodes are shown in Figure 6. The load exchange takes four communication steps to finish:

(1) node 3 to node 1, node 8 to node 6, node 4 to node 5
(2) node 1 to node 0
(3) node 0 to node 6
(4) node 6 to node 7

At the end of scheduling, nodes 0–4 have five tasks each and nodes 5–8 have four tasks each.

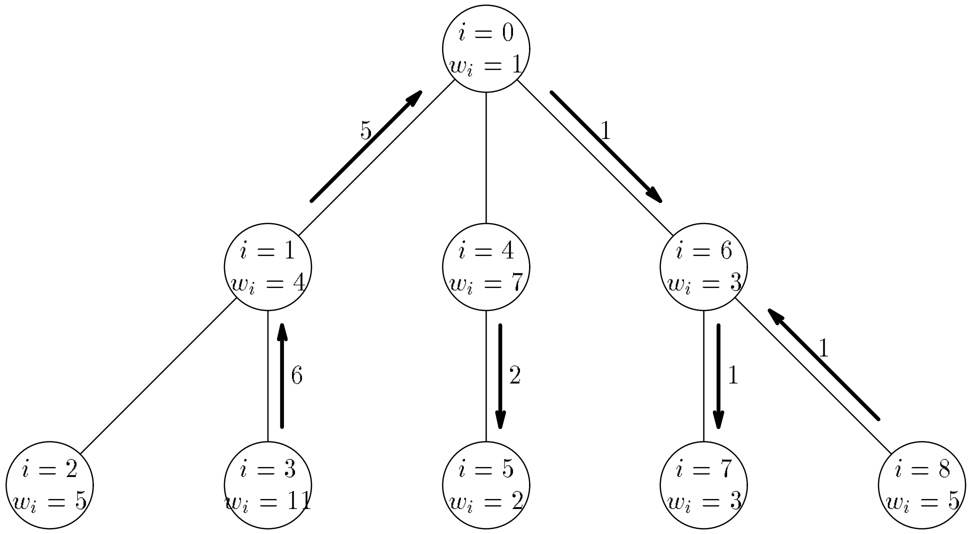


Figure 6: Example for the Tree Walking Algorithm.

4. Scheduling Algorithm for the Hypercube Topology

In this section, we study two algorithms designed for the hypercube topology. First, we study the DEM algorithm. Then, a new algorithm, Cube Walking Algorithm, is presented.

The DEM algorithm, proposed by Cybenko, is a synchronous approach [12]. In DEM, small domains are balanced first and these then combine to form larger domains until ultimately the entire system is balanced. The algorithm 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 nodes 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 [33].

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.

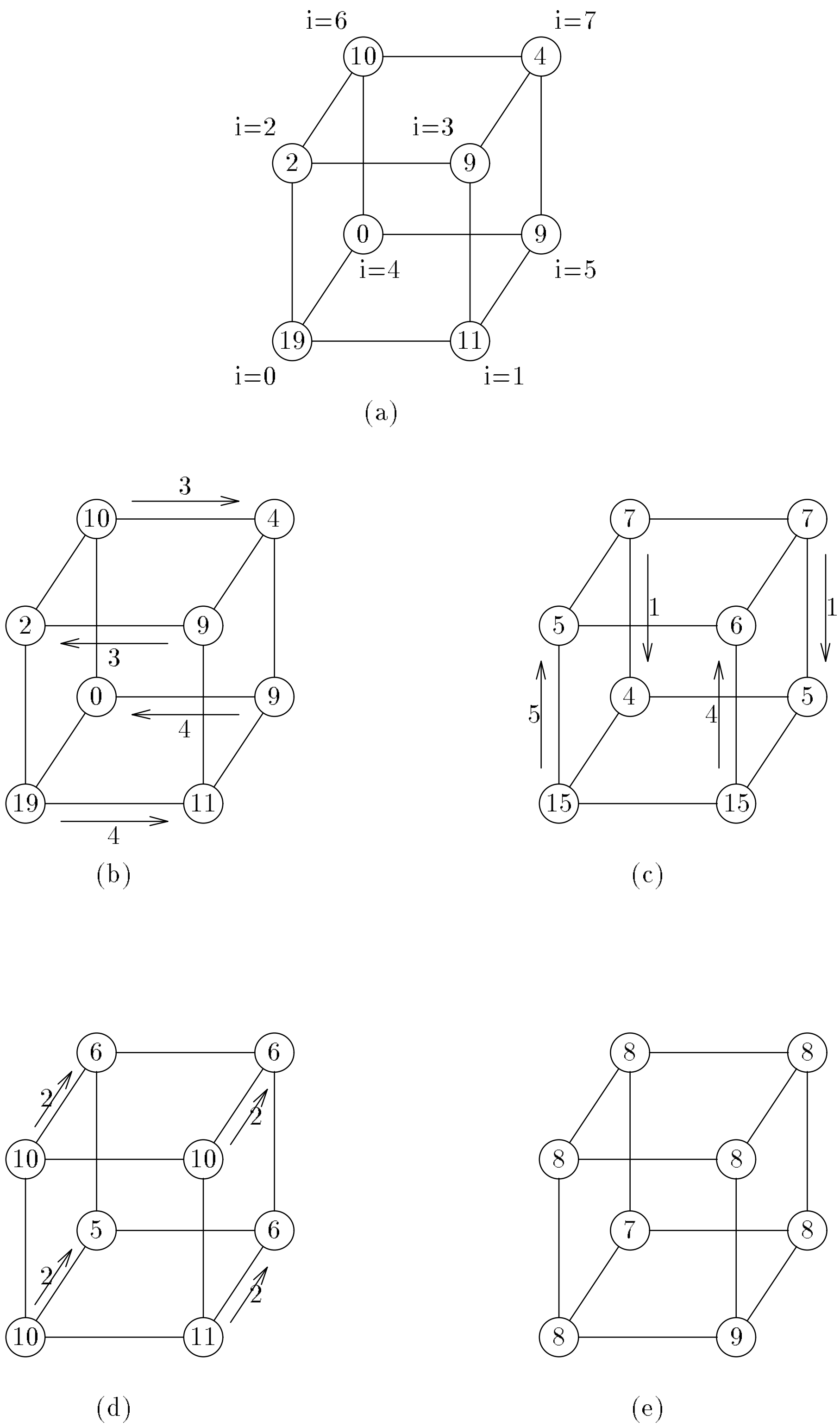


Figure 8: An example for the DEM algorithm.

Example 2:

The DEM algorithm is illustrated in Figure 8 for an 8-processor hypercube. The load distribution before executing the 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, compared to the optimal scheduling shown in Figure 4, which has only 21 task-hops.

The following lemma shows that the maximum difference of the number of tasks after executing the DEM algorithm is at most d, the dimension of the hypercube.

Lemma 1: After executing the DEM algorithm, the maximum difference $D = w_{high} - w_{low}$ is less than or equal to d, where w_{high} is the largest number of tasks and w_{low} is the smallest number of tasks in a processor.

Proof: After executing the DEM algorithm, the number of tasks in two neighbor processors differs at most by 1. In a d-dimensional hypercube, the maximum distance between two processors is d. Therefore, the number of tasks in any two processors differs at most by d. □

Figure 9 shows an example where $D = 4$ for a 4-dimensional hypercube.

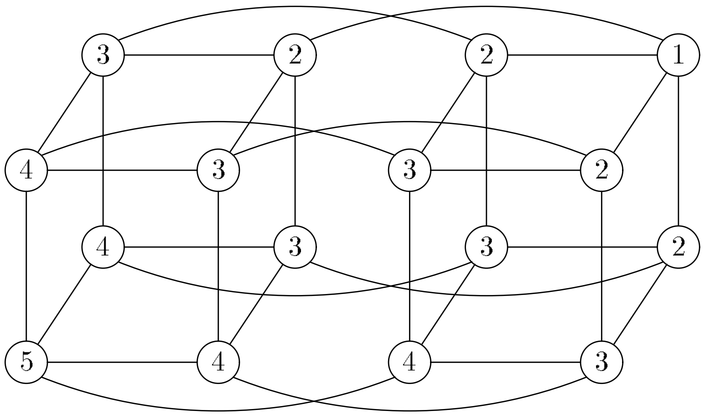


Figure 9: An example which shows that the number of tasks differs by 4 resulting from DEM.

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 on how many tasks should be sent. Consequently, a node may send excessive tasks to its neighbor. Therefore, 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 the same and equal to the total number of tasks in the entire cube. In step 2, each node calculates the average number of tasks per node. 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. A quota vector q is calculated at step 3 so that each node knows if its k-dimensional subcubes are over-loaded or under-loaded.

In step 4, task exchanges are repeated d times. 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 $i = 0, 1, ..., N/2-1$. And, we send tasks only in one direction, from the over-loaded subcube to the other. In this way, an over-loaded 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 δ vector for every node in the over-loaded subcube.

Utilizing global information. Although CWA conducts task movement dimension by dimension like DEM, it relies not only on information from the nodes in pair, but also on partial global information from its k-dimensional subcube, where $k = 1, ..., d-1$. Such a decision is more conservative compared to DEM. In the DEM algorithm, two paired nodes attempt to average their number of tasks anyway. However, the attempt often leads to a non-optimal solution, since the final destination of extra tasks depends on global demands instead of local demands.

At the beginning of step 4, a δ vector in every node records the number of excessive tasks in each dimension. An aggressive strategy is simply to let $\delta_i^j = w_i^j - Q_i^j$. Take an example shown in Figure 11(a). We have $\delta_3^0 = 1$. As we can see in dimension 3, even though the 2-dimensional subcube of node n_3, including nodes n_3, n_2, n_1, n_0, is over-loaded with nine tasks and node n_3 has one extra task, the 1-dimensional subcube of n_3, including nodes n_2 and n_3, still lacks five tasks. A better strategy is for node n_3 to reserve the extra task for its own 1-dimensional subcube, instead of contributing it to node n_7. Therefore, if at any dimension l, the total number of tasks is less than its total quota, the extra tasks possessed by individual nodes should not be sent out,

Cube Walking Algorithm (CWA)

Assume the cube dimension is d, and the number of nodes is $N = 2^d$.

Let $\oplus$ denote the bitwise exclusive OR, $\wedge$ the bitwise AND, and $\vee$ the bitwise OR.

1. Exchange the value of w_i^k along each dimension $k = 0, 1, ..., d-1$.
   Each node computes its w vector:
$$w_i^{k+1} = w_i^k + w_{i\oplus 2^k}^k$$

2. $T = w_i^d, \quad w_{avg} = \lfloor T/N \rfloor, \quad R = T \ mod \ N$
   T is the total number of tasks.

3. Each node computes its quota vector $Q_i^k, k = 0, ..., d-1$
$$Q_i^k = w_{avg} * 2^k + r_i^k$$
   where
$$r_i^k = \begin{cases} 0 & if \ i \wedge (N - 2^k) \geq R \\ 2^k & if \ i \vee (2^k - 1) < R \\ R - i \wedge (N - 2^k) & otherwise \end{cases}$$

4. For $k = d-1 \ to \ 0$ do
   All nodes i do
   For $j = k \ to \ 0$
$$\delta_i^j = \begin{cases} 0 & if \ \exists l, \ k \geq l \geq j, \ w_i^l \leq Q_i^l \\ w_i^j - Q_i^j & otherwise \end{cases}$$
   Adjust δ vector: initialize $\eta_i^k = 0$
   For $j = k-1 \ to \ 0$
$$\gamma_i^j = \begin{cases} \eta_i^{j+1} & if \ \delta_i^j \geq \eta_i^{j+1} \ \ and \ \ i \wedge 2^j = 0 \\ \delta_i^j & if \ \eta_i^{j+1} > \delta_i^j \geq 0 \ \ and \ \ i \wedge 2^j = 0 \\ 0 & if \ \delta_{i\oplus 2^j}^j \geq \eta_i^{j+1} \ \ and \ \ i \wedge 2^j \neq 0 \\ \eta_i^{j+1} - \delta_{i\oplus 2^j}^j & if \ \eta_i^{j+1} \geq \delta_{i\oplus 2^j}^j \geq 0 \ \ and \ \ i \wedge 2^j \neq 0 \end{cases}$$
   where $\delta_{i\oplus 2^j}^j = w_i^{j+1} - w_i^j - Q_{i\oplus 2^j}^j$.
$$\eta_i^j = \begin{cases} \delta_i^j - \delta_i^{j+1} & if \ \ \delta_i^j - \gamma_i^j > \delta_i^{j+1} \\ \gamma_i^j & otherwise \end{cases}$$
$$\delta_i^j = \delta_i^j - \eta_i^j$$
   If $w_i^k > Q_i^k$, send δ_i^0 tasks as well as its δ vector to node $i \oplus 2^k$, and update its own w vector for each dimension $j = 0, 1, ..., k-1$: $w_i^j = w_i^j - \delta_i^j$.

   If $w_i^k < Q_i^k$, receive tasks as well as the δ vector from node $i \oplus 2^k$, and update its own w vector for each dimension $j = 0, 1, ..., k-1$: $w_i^j = w_i^j + \delta_{i\oplus 2^k}^j$.

Figure 10: The Cube Walking Algorithm.

and therefore the elements of the δ vector are set to zero. In this example, we set $\delta_3^1 = 0$ because $w_3^1 = 11$ and $Q_3^1 = 16$. Also, because $w_3^1 < Q_3^1$, δ_3^0 is set to be 0. In the under-loaded subcube, including nodes n_4, n_5, n_6, n_7, because $w_i^{d-1} < Q_i^{d-1}$ for node i in the subcube, the δ_i vector will be reset to zero. By utilizing subcube information, not every over-loaded node in the over-loaded subcube will send extra tasks out.

Handling of inconsistent δ vectors. Sometimes $\sum_i \delta_i^0$ is not equal to the number of extra tasks in the subcube, which leads to an *inconsistent decision* to be discussed as follows. An inconsistent state is shown in Figure 11(a): the 2-dimensional subcube is over-loaded by 9 tasks while the 1-dimensional subcube claims 14 extra tasks. In this situation, there exists a j such that $\delta_0^j > \delta_0^{j+1}$. It is inconsistent in a sense that any smaller subcube should not send more tasks out than its larger subcube does. Instead, the δ vector needs to be further adjusted to eliminate any inconsistency. We want to ensure that $\delta_i^j \leq \delta_i^{j+1}$ for all i and j, and $\delta_i^j + \delta_{i \oplus 2^j}^j = \delta_i^{j+1}$. This implies that the two j-dimensional subcubes agree with each other to set the total number of tasks to be sent from both subcubes to be equal to that of the $(j+1)$-dimensional subcube. The adjustment is carried out by calculating a γ vector and an η vector. When $\delta_i^j > \delta_i^{j+1}$, the corresponding element η_i^j records the difference, that is, the total number of tasks to be adjusted in subsequent smaller subcubes. The value of η_i^j can be heuristically distributed down to subcubes by splitting it into elements of γ of the smaller subcube. The current heuristics is to adjust the δ in the lower part of the subcube before the higher part. This adjustment needs to be applied repeatedly if necessary until each individual node's δ value is modified. Finally, δ_i at node n_i is adjusted to the value that indicates exactly how many tasks should be sent to its paired node n_i'. Then the nodes in the over-loaded subcube send tasks to the under-loaded subcube, as well as their δ vectors.

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 calculated:

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

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

$$\{8, 16, 32\}.$$

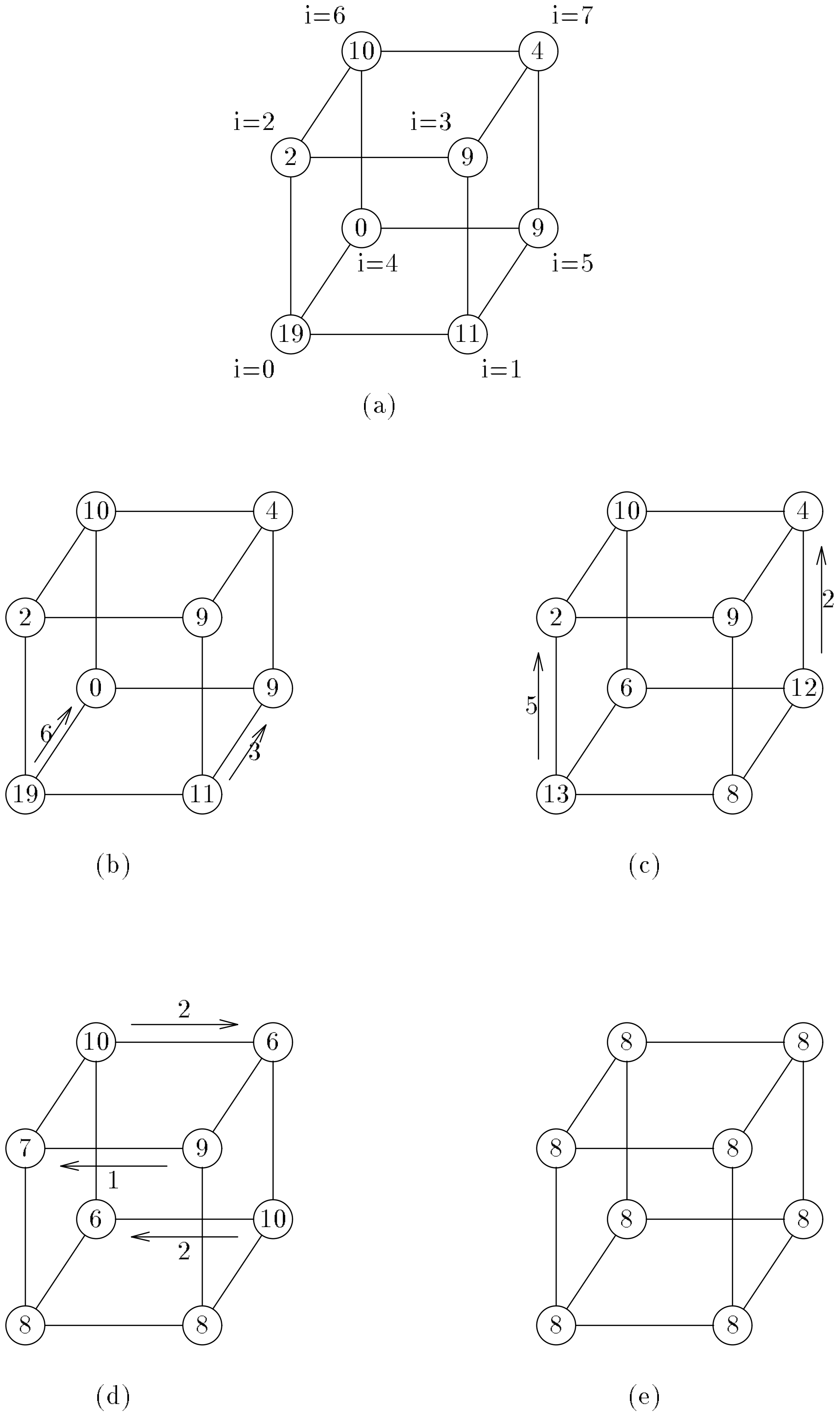


Figure 11: A running example of CWA.

At step 4, when $k = 2$, the values of w_i^k and δ_i^j before adjustment are as shown below:

Node	d0		d1		d2	
i	w_i^0	δ_i^0	w_i^1	δ_i^1	w_i^2	δ_i^2
0	19	11	30	14	41	9
1	11	3	30	14	41	9
2	2	0	11	0	41	9
3	9	0	11	0	41	9
4	0	0	9	0	23	0
5	9	0	9	0	23	0
6	10	0	14	0	23	0
7	4	0	14	0	23	0

Because $\gamma_0^1 > \gamma_0^2$ and $\gamma_1^1 > \gamma_1^2$, the values of $\gamma_0^1, \gamma_1^1, \gamma_0^0, \gamma_1^0$ are adjusted as follows:

$$\gamma_0^1 = 0, \quad \eta_0^1 = 14 - 9 = 5, \quad \delta_0^1 = 14 - 5 = 9$$

$$\gamma_0^0 = 5, \quad \eta_0^0 = 5, \quad \delta_0^0 = 11 - 5 = 6$$

$$\gamma_1^1 = 0, \quad \eta_1^1 = 14 - 9 = 5, \quad \delta_1^1 = 14 - 5 = 9$$

$$\gamma_1^0 = 0, \quad \eta_1^0 = 0, \quad \delta_1^0 = 3 - 0 = 3$$

and the table is updated to:

Node	d0		d1		d2	
i	w_i^0	δ_i^0	w_i^1	δ_i^1	w_i^2	δ_i^2
0	19	6	30	9	41	9
1	11	3	30	9	41	9
2	2	0	11	0	41	9
3	9	0	11	0	41	9
4	0	0	9	0	23	0
5	9	0	9	0	23	0
6	10	0	14	0	23	0
7	4	0	14	0	23	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 $k = 1$, there is no adjustment performed. The values of w_i^k and δ_i^j are as shown below:

Node	d0		d1	
i	w_i^0	δ_i^0	w_i^1	δ_i^1
0	13	5	21	5
1	8	0	21	5
2	2	0	11	0
3	9	0	11	0
4	6	0	18	2
5	12	2	18	2
6	10	0	14	0
7	4	0	14	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}, {6, 7} have been balanced. Each subcube has 16 tasks.

When $k = 0$, the values of w_i^k and δ_i^j are as shown below:

Node	d0	
i	w_i^0	δ_i^0
0	8	0
1	8	0
2	7	0
3	9	1
4	6	0
5	10	2
6	10	2
7	6	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.

Now, we compare performance of the DEM and CWA algorithms. For this purpose, we consider a test set of load distributions. In this test set, the load at each processor is randomly selected from a normal distribution, with the mean equal to the specified average number of tasks. The number of dimensions varies from 2 to 8 (4 to 256 processors). 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. Comparisons are carried out based on results obtained by running each algorithm with the test set.

First, we test the DEM algorithm for its ability to balance the load. We run the DEM algorithm with different numbers of processors and different weights. Table I shows the load imbalance of the DEM algorithm, which is measured by

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

Each result is the average of 1,000 test cases. The load imbalance increases, but not directly proportional, with the number of dimensions. It is inversely proportional to the average weight.

Table I: The Load Imbalance of DEM

Number of dimensions	Number of processors	Average weight					
		2	5	10	20	50	100
2	4	6.43%	2.47%	1.22%	0.62%	0.26%	0.13%
3	8	9.49%	3.82%	1.81%	0.94%	0.37%	0.19%
4	16	10.88%	4.20%	2.12%	1.05%	0.42%	0.21%
5	32	11.62%	4.49%	2.26%	1.12%	0.44%	0.22%
6	64	12.20%	4.62%	2.29%	1.15%	0.46%	0.23%

Table II shows the percentage of fully-balanced cases of the DEM algorithm. 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.

Table II: 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%

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

$$\frac{T_{DEM} - T_{OPT}}{T_{OPT}},$$

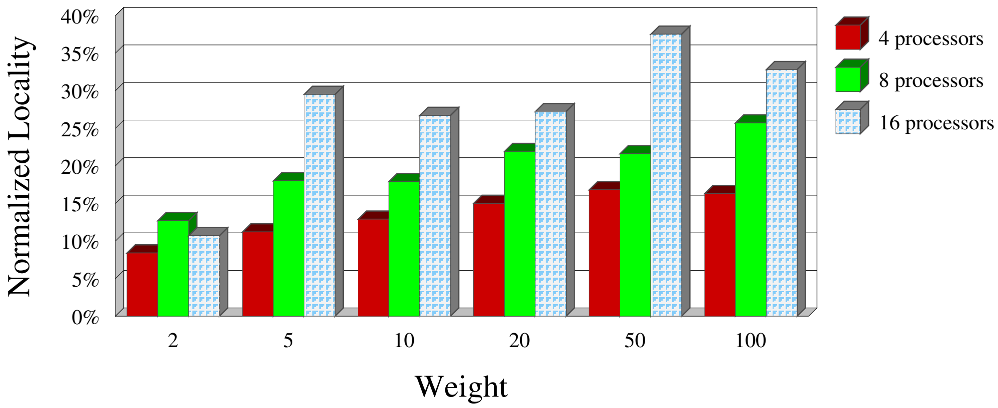


Figure 12: Normalized locality of DEM

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 12 shows the normalized locality on 4, 8, and 16 processors. Too few fully-balanced cases exist on more than 16 processors, so they are not reported here.

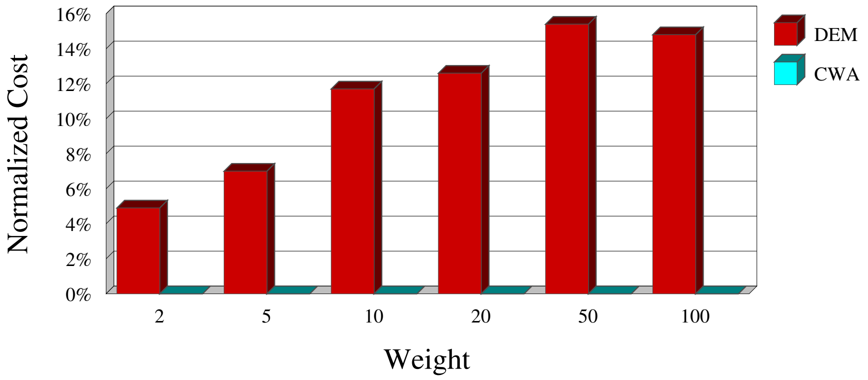
Finally, we compare the load balancing overhead. DEM is very simple so that the runtime overhead for load balancing decision is small. However, unnecessary task migration lead to a large communication overhead. Compared to the time spent on the load balancing decision, communication time is a dominate 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

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

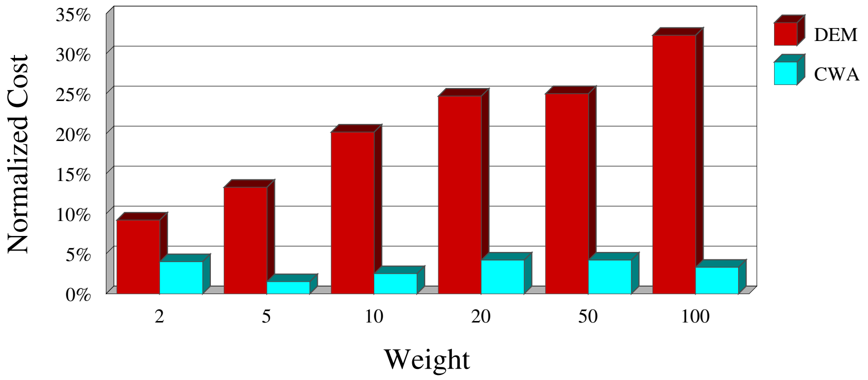
and

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

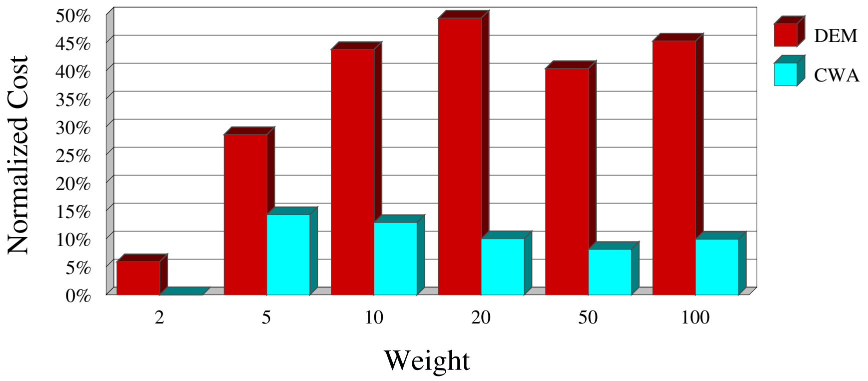
where C_{DEM}, C_{CWA}, and C_{OPT} are the number of task-hops of the DEM, CWA, and optimal algorithms, respectively. Figure 13 compares the normalized communication costs on 4, 8, and 16 processors. Each result is the average of the DEM fully-balanced cases in 1,000 test cases. The number of task-hops of CWA on four processors is minimum. It can also be seen that the communication costs of DEM are much larger than those of CWA. Figure 14 shows the normalized communication costs of CWA on 64 and 256 processors. Each data presented here is the average of 100 different test cases.



(a) 4 processors

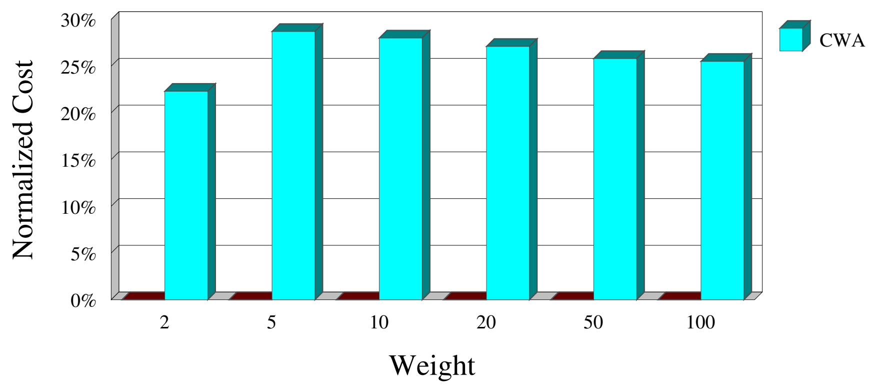


(b) 8 processors

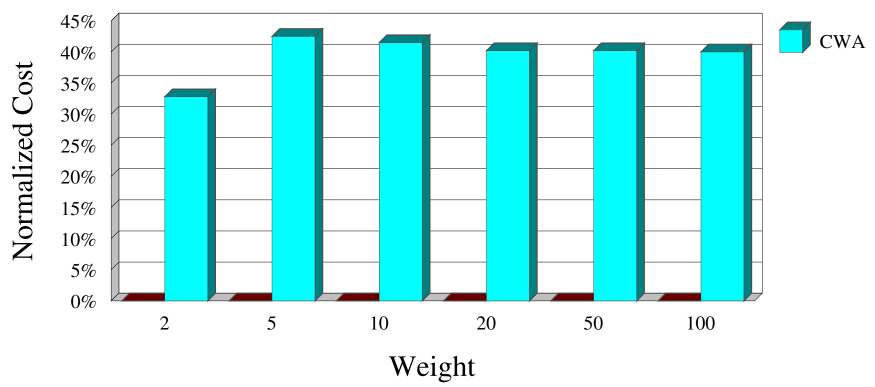


(c) 16 processors

Figure 13: Normalized communication costs of DEM and CWA



(a) 64 processors



(b) 256 processors

Figure 14: Normalized communication costs of CWA

5. Scheduling Algorithm for the Mesh Topology

A parallel scheduling algorithm for the mesh topology, named *Mesh Walking Algorithm (MWA)*, is shown in Figure 15. Let $w_{i,j}$ be the number of tasks in node (i,j) before the algorithm is applied. The first step scans the partial vector w along each row i, and each node (i,j) records a w vector $w_{i,k}$, $k = 0, ..., j$. Thus, each node stores the weight information of nodes that are in the same row with a column number that is less than or equal to j. At step 2, each node in column $n_2 - 1$ calculates the sum $s_i = \sum_{k=0}^{n_2-1} w_{i,k}$, and $t_i = \sum_{k=0}^{i} s_k$ is calculated by a *scan-with-sum* operation along these nodes. The value $T = t_{n_1-1}$ at node $(n_1 - 1, n_2 - 1)$ is equal to the total number of tasks in the mesh. Then node $(n_1 - 1, n_2 - 1)$ computes the average number of tasks per node, w_{avg}. 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 broadcast to all nodes. Also, the values of s_i, t_i and t_{i-1} are spread along each row i. In step 3, a quota vector q is calculated in each node so that each node knows if it is overloaded or underloaded. Each node (i, j) also calculates a row accumulation quota Q_i, which is the quota of the submesh consisting of all nodes from row 0 to row i.

Step 4 balances the load among rows. All nodes calculate the values of y and x. If $y_i < 0$, row i will receive $|y_i|$ tasks from row $i+1$. If $x_i > 0$, row i will receive x_i tasks from row $i-1$. If $y_i > 0$, the submesh from row 0 to row i is overloaded, and y_i tasks need to be sent to row $i+1$. If $x_i < 0$, the submesh below row i is underloaded, and $|x_i|$ tasks need to be sent to row $i-1$. Then, a δ vector is calculated, which is the number of extra tasks in each node (i, k), $k = 0, ..., j$. When $y_i > 0$, an η vector is calculated to determine how many tasks $d_{i,j}$ at each node need to be sent. The total number of extra tasks in row i, $\sum_{k=0}^{n_2-1} \delta_{i,k}$ is larger than or equal to y_i. In this algorithm, the first m nodes (where $\sum_{k=0}^{m} \delta_{i,k} \geq y_i$ and $\sum_{k=0}^{m-1} \delta_{i,k} < y_i$) send y_i tasks to row $i+1$. The γ vector indicates how many tasks are needed by previous nodes. When $x_i < 0$, a similar way is used to calculate the u vector, which indicates how many tasks at node (i, j) are to be sent to row $i-1$.

In step 5, the load is balanced in each row. The z and v vector are calculated to determine the task exchange between adjacent nodes in a row.

In this algorithm, step 1 spends n_2 communication steps to collect load information along each row. Step 2 spends n_1 communication steps to collect load information across rows. Broadcasting and spreading operations spend $n_1 + n_2$ communication steps. Steps 4 and 5 also spend n_1 and n_2 communication steps at most for load balancing, respectively. 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 16. At the beginning of scheduling, each node has $w_{i,j}$ tasks ready to be scheduled. The vectors $w_{i,k}$ listed below are obtained by a scan operation at step 1.

Mesh Walking Algorithm (MWA)

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

1. Let $w_{i,j}$ be the number of tasks in node (i,j). Scan the partial vector w along each row. Each node (i,j) records a vector $w_{i,k}, \;\; k = 0, ..., j$.
2. Nodes $(i, n_2 - 1)$ perform $s_i = \sum_{k=0}^{n_2-1} w_{i,k}$.
   Perform a *scan-with-sum* $t_i = \sum_{k=0}^{i} s_k$. $\;\; T = t_{n_1-1}$ is the total number of tasks.
   Node $(n_1 - 1, n_2 - 1)$ computes $w_{avg} = \lfloor T/N \rfloor, \;\; R = T \; mod \; N$.
   The values of w_{avg} and R are broadcast to all nodes.
   The values of s_i, t_i and t_{i-1} are spread along each row i.
3. Each node computes its quota $q_{i,j}$, and row accumulation quota Q_i
   $$q_{i,j} = \begin{cases} w_{avg} & if \;\; (i*d+j) \geq R \\ w_{avg}+1 & otherwise \end{cases} \qquad and \;\; Q_i = w_{avg} * d * (i+1) + r_i$$
   where
   $$r_i = \begin{cases} (i+1)*d & if \; (i+1)*d \leq R \\ R & otherwise \end{cases}$$
   Each node that is not in row 0 also computes the value of Q_{i-1}.
4. All nodes (i,j) do
   $$y_i = t_i - Q_i, \quad x_0 = 0, \quad x_i = t_{i-1} - Q_{i-1} \;\text{ for }\; i > 0$$
   If $x_i > 0$, receive tasks and its d vector from node $(i-1, j)$, $w_{i,j} = w_{i,j} + d_{i-1,j}$.
   If $y_i < 0$, receive tasks and its u vector from node $(i+1, j)$, $w_{i,j} = w_{i,j} + u_{i+1,j}$.
   $\delta_{i,k} = w_{i,k} - q_{i,k}, \;\; k = 0, ..., j$.
   If $y_i > 0$, initialize $\eta_{i,0} = y_i$, $\gamma_{i,0} = 0$, and compute the d vector for $k = 0, 1, ..., j$
   $$d_{i,k} = \begin{cases} \eta_{i,k} & if \;\; \delta_{i,k} > \eta_{i,k} + \gamma_{i,k} > 0 \\ \delta_{i,k} - \gamma_{i,k} & if \;\; \eta_{i,k} + \gamma_{i,k} \geq \delta_{i,k} > \gamma_{i,k} \\ 0 & otherwise \end{cases}$$
   $$and \quad \gamma_{i,k+1} = \gamma_{i,k} - (\delta_{i,k} - d_{i,k}), \quad \delta_{i,k} = \delta_{i,k} - d_{i,k}, \quad \eta_{i,k+1} = \eta_{i,k} - d_{i,k}$$
   Send $d_{i,j}$ tasks and its d vector to node $(i+1, j)$, $w_{i,j} = w_{i,j} - d_{i,j}$.
   If $x_i < 0$, initialize $\eta_{i,0} = |x_i|$, $\gamma_{i,0} = 0$, and compute the u vector for $k = 0, 1, ..., j$
   $$u_{i,k} = \begin{cases} \eta_{i,k} & if \;\; \delta_{i,k} > \eta_{i,k} + \gamma_{i,k} > 0 \\ \delta_{i,k} - \gamma_{i,k} & if \;\; \eta_{i,k} + \gamma_{i,k} \geq \delta_{i,k} > \gamma_{i,k} \\ 0 & otherwise \end{cases}$$
   $$and \quad \gamma_{i,k+1} = \gamma_{i,k} - (\delta_{i,k} - u_{i,k}), \quad \eta_{i,k+1} = \eta_{i,k} - u_{i,k}$$
   Send $u_{i,j}$ tasks and its u vector to node $(i-1, j)$, $w_{i,j} = w_{i,j} - u_{i,j}$.
5. All nodes (i,j) do
   $$z_{i,0} = 0, \quad z_{i,j} = \textstyle\sum_{k=0}^{j-1}(w_{i,k} - q_{i,k}) \text{ for } i > 0, \quad v_{i,j} = z_{i,j} + w_{i,j} - q_{i,j}$$
   If $z_{i,j} > 0$, receive tasks from node $(i, j-1)$.
   If $v_{i,j} < 0$, receive tasks from node $(i, j+1)$.
   If $v_{i,j} > 0$, send $v_{i,j}$ tasks to node $(i, j+1)$.
   If $z_{i,j} < 0$, send $|z_{i,j}|$ tasks to node $(i, j-1)$.

Figure 15: The Mesh Walking Algorithm.

Node	w vector
(0,0)	(7)
(0,1)	(7, 12)
(0,2)	(7, 12, 6)
(0,3)	(7, 12, 6, 16)
(1,0)	(17)
(1,1)	(17, 3)
(1,2)	(17, 3, 0)
(1,3)	(17, 3, 0, 15)
(2,0)	(2)
(2,1)	(2, 13)
(2,2)	(2, 13, 5)
(2,3)	(2, 13, 5, 5)
(3,0)	(5)
(3,1)	(5, 6)
(3,2)	(5, 6, 4)
(3,3)	(5, 6, 4, 12)

During step 2, nodes in the last column calculate s_i, and the values of t_i are obtained by a *scan-with-sum* operation along the column:

Node	s_i	t_i	t_{i-1}
(0,3)	41	41	–
(1,3)	35	76	41
(2,3)	25	101	76
(3,3)	27	128	101

Note that these nodes record the value of t_{i-1}, too. The total number of tasks $T = t_3 = 128$. The values of w_{avg} and R are calculated and broadcast to all nodes:

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

The values of s_i, t_i, and t_{i-1} are spread along each row i, so that every node in the same row has the load information.

In step 3, each node calculates the values of $q_{i,j}$. Because $R = 0$, every node has the same quota, which is 8. Each node (i,j) calculates also the values of Q_i and Q_{i-1}:

$$Q_0 = 32, \ \ Q_1 = 64, \ \ Q_2 = 96, \ \ Q_3 = 128.$$

The values of y and x vectors at step 4 are shown below:

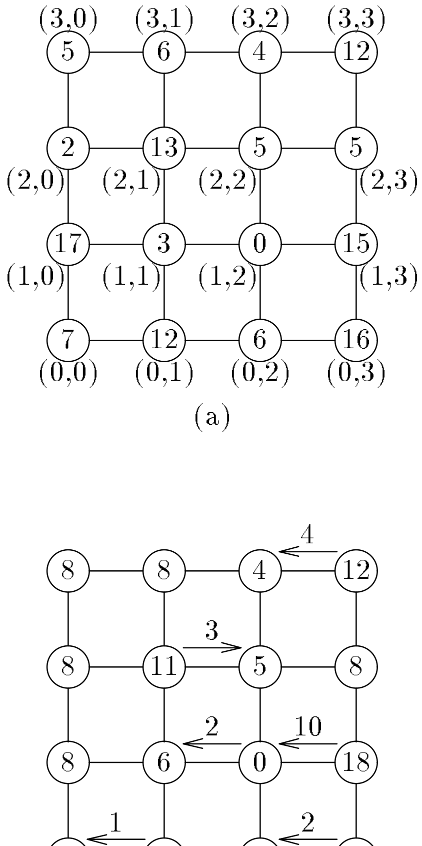


Figure 16: A running example of MWA.

Node	y	x
(0,j)	9	0
(1,j)	12	9
(2,j)	5	12
(3,j)	0	5

Because $y_0 > 0$, $\eta_{0,0}$ is set to be $y_0 = 9$. Then $d_{0,0} = 0$, $d_{0,1} = 4$, $d_{0,2} = 0$, and $d_{0,3} = 5$. Node (0,1) sends four tasks to node (1,1), and Node (0,3) sends five tasks to node (1,3). When nodes in the row 1 receive the tasks and d vectors, they update their weights: $w_{1,1} = 3 + 4 = 7$ and $w_{1,3} = 15 + 5 = 20$. Then, they calculate the d vectors and node (1,0) sends nine tasks to node (2,0), and node (1,3) sends three tasks to node (2,3). Finally, nodes in row 2 calculate the d vectors. Node (2,0) sends three tasks to node (3,0), and Node (2,1) sends two tasks to node (3,1). The values of vectors δ, η, and d are shown below:

Node	δ vector	η vector	γ vector	d vector
(0,0)	-1	9	0	0
(0,1)	-1, 4	9, 9	0, 1	0, 3
(0,2)	-1, 4, -2	9, 9, 6	0, 1, 0	0, 3, 0
(0,3)	-1, 4, -2, 8	9, 9, 6, 6	0, 1, 0, 0	0, 3, 0, 6
(1,0)	9	12	0	9
(1,1)	9, -2	12, 3	0, 0	9, 0
(1,2)	9, -2, -8	12, 3, 3	0, 0, 2	9, 0, 0
(1,3)	9, -2, -8, 13	12, 3, 3, 3	0, 0, 2, 10	9, 0, 0, 3
(2,0)	3	5	0	3
(2,1)	3, 5	5, 2	0, 0	3, 2
(2,2)	3, 5, -3	5, 2, 0	0, 0, -3	3, 2, 0
(2,3)	3, 5, -3, 0	5, 2, 0, 0	o, o, -3, 0	3, 2, 0, 0

At step 5, each node calculates the values of z and v as shown below:

Node	z	v
(0,0)	0	-1
(0,1)	-1	0
(0,2)	0	-2
(0,3)	-2	0
(1,0)	0	0
(1,1)	0	-2
(1,2)	-2	-10
(1,3)	-10	0
(2,0)	0	0
(2,1)	0	3
(2,2)	3	0
(2,3)	0	0
(3,0)	0	0
(3,1)	0	0
(3,2)	0	-4
(3,3)	-4	0

Then, processors exchanges tasks according to the values of z and v. This results in a balanced load, and each node has eight tasks. The total number of task-hops is 48.

A performance study has been conducted for the MWA algorithm. The method and assumptions are the same as that for the CWA algorithm. The normalized communication cost of MWA with respect to the optimal algorithm is measured by

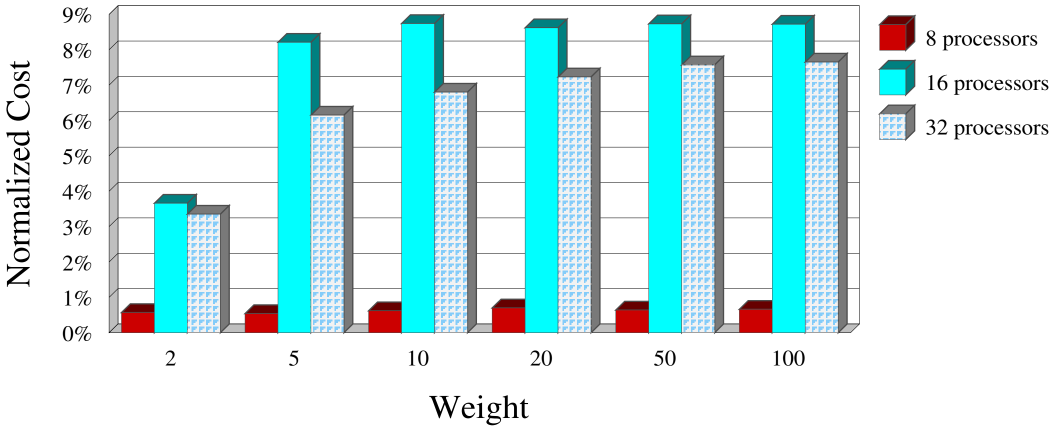
$$\frac{C_{MWA} - C_{OPT}}{C_{OPT}},$$

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 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.

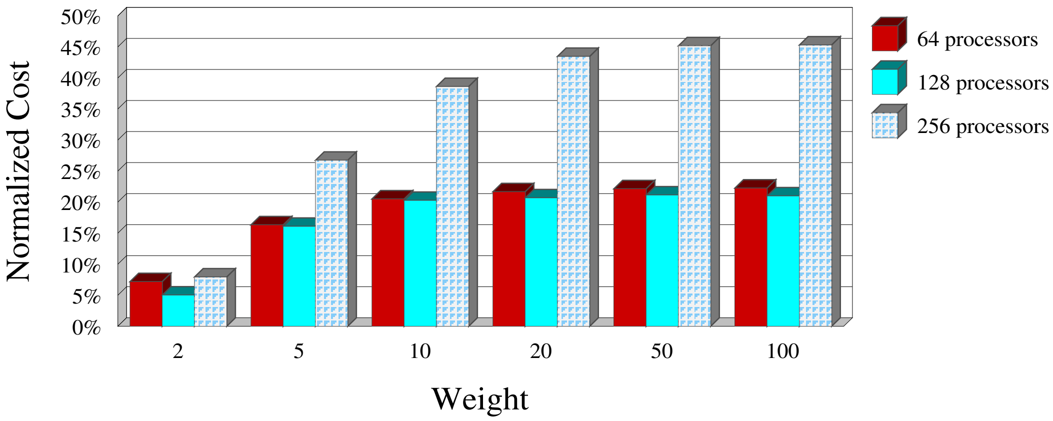
The normalized communication cost of MWA is much lower than using other algorithms such as DEM. For small meshes, MWA provides a nearly optimal result. The cost increases when the number of processors is large.

6. 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 three theorems show that the three algorithms are



(a) 8, 16, and 32 processors



(b) 64, 128, and 256 processors

Figure 17: Normalized communication cost of MWA.

able to fully balance the load. If the number of tasks can be equally divided by the number of processors, then each processor will have the equal number of tasks; otherwise, the number of tasks in each processor differs by one.

Theorem 1: The difference in the number of tasks in each processor is at most one after executing the TWA algorithm.

Proof: Executing the algorithm, the number of tasks in processor i is

$$w_i^{'} = w_i + (Q_i - W_i) + \sum_{j\ is\ child\ of\ i} (W_j - Q_j).$$

Because

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

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

Since q_i is either w_{avg} or $w_{avg} + 1$, the difference in the number of tasks in each processor is at most one. □

Theorem 2: The difference in the number of tasks in each processor is at most one after executing the CWA algorithm.

Proof: After each iteration in step 4, the subcube with $w_i^k > Q_i^k$ sends $w_i^k - Q_i^k$ tasks to the other subcube with $w_i^k < Q_i^k$. Thus, each subcube has Q_i^k tasks. Finally, each processor has Q_i^0 tasks. Since Q_i^0 is either w_{avg} or $w_{avg} + 1$, the difference in the number of tasks in each processor is at most one. □

Theorem 3: The difference in the number of tasks in each processor is at most one after executing the MWA algorithm.

Proof: After step 4, the number of tasks in each row is

$$t_i - t_{i-1} - y_i + x_i = t_i - t_{i-1} - (t_i - Q_i) + (t_{i-1} - Q_{i-1}) = Q_i - Q_{i-1} = \sum_{k=0}^{n_2-1} q_{i,k}.$$

After step 5, the number of tasks in processor is

$$w_{i,j}^{'} = w_{i,j} + z_{i,j} - v_{i,j} = w_{i,j} + z_{i,j} - (z_{i,j} + w_{i,j} - q_{i,j}) = q_{i,j}.$$

Since $q_{i,j}$ is either w_{avg} or $w_{avg} + 1$, the difference in the number of tasks in each processor is at most one. □

These algorithms also maximize locality. *Local tasks* are the tasks that are not migrated to other processors, and *non-local tasks* are those that are migrated to other processors. Maximum locality implies the maximum number of local tasks and the minimum number of non-local tasks. In Lemmas 2 and 3 and Theorem 4 and 5, we assume that the number of tasks T is evenly divided by N, the number of processors. 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 processor with $w_i < w_{avg}$ must receive $(w_{avg} - w_i)$ tasks from other processors for a balanced load. Therefore, a total of $\sum_i \max(w_{avg} - w_i, 0)$ tasks must be migrated between processors. □

The next theorem proves that these three algorithms maximize locality.

Theorem 4: 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 processor is not less than $\min(w, w_{avg})$. In TWA, each processor receives tasks before sending tasks. In CWA and MWA, each processor sends tasks only when its weight is larger than w_{avg}. Thus, in all processors, 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. □

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

Theorem 5: The TWA algorithm minimizes $\sum_k e_k$, the total number of task-hops, and the total number of communications.

Proof: In each subtree, if $Q_i \geq W_i$, then it needs to receive from its parent $Q_i - W_i$ tasks, which is the minimum number of tasks to be transmitted to the subtree. Similarly, if $Q_i < W_i$, then it needs to send to its parent $W_i - Q_i$ tasks, which is the minimum number of tasks to be transmitted from the subtree.

In each subtree, 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. □

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 processors, the algorithms minimize the communication cost.

Lemma 3: The CWA and MWA algorithms minimize the communication cost in a system with two or four processors.

Proof: The communication cost in a system is minimized if there is no negative cycle [23]. In a system of two processors, there is no cycle. In a system of four processors, 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. □

7. Previous Works

Parallel scheduling and the static scheduling share some common ideas [17, 34, 16, 36, 11]. Both of them utilize the systemwise information and perform scheduling globally to achieve high-quality of load balancing. They also clearly separate the time to conduct scheduling and the time to perform computation. But, parallel scheduling is different from the 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 a better scalability for massively parallel machines and large size applications.

Large research efforts have been directed towards the process allocation in distributed systems [28, 14, 15, 31, 8, 9, 20, 30, 32, 5, 25]. A recent comparison study of dynamic load balancing strategies on highly parallel computers is given by Willebeek-LeMair and Reeves [33]. Eager *et al.* compared the sender-initiated algorithm and receiver-initiated algorithm [15]. Work with a similar assumption as ours includes the Gradient Model developed by Lin and Keller [24]. The randomized allocation algorithms developed by different authors are quite simple and effective [2, 3, 14, 21, 10]. The receiver-initiated diffusion (RID) is another effective algorithm [33].

Runtime parallel scheduling is similar to the dynamic scheduling in 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, there exist substantial differences, making 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. Fox *et al.* first adapted prescheduling to application problems with geographical structures [19, 26]. Some other works also deal with geographically structured problems [13, 6, 4]. The project PARTI automates prescheduling for nonuniform problems [27]. The dimension exchange method (DEM) is applied to application problems without geographical structure [12]. It was conceptually designed for a hypercube system but may be applied to other topologies, such as k-ary n-cubes [35]. It balances load for independent tasks with an equal grain size. The method has been extended by Willebeek-LeMair and Reeves [33] so that the algorithm can run incrementally to correct the unbalanced load due to varied grain sizes.

8. Concluding Remarks

Parallel scheduling combines the advantages of static scheduling and dynamic scheduling. It balances the load *in parallel.* 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.

It has been widely believed that a scheduling method that collects load information from all processors in the system is neither practical, nor scalable. Recent work has demonstrated that with runtime parallel scheduling it is possible to use the global load information to fully balance the load. Runtime parallel scheduling can provide a low-overhead task scheduling because tasks are packed to be sent to other processors, which reduces significantly the number of messages. It is scalable to at least 512 processors [29]. Furthermore, in a dynamic scheduling system, when it intends to quickly and accurately balance the load, the system could become unstable. In parallel scheduling, a synchronous approach removes the stability problem and is able to balance the load quickly and accurately.

Acknowledgments

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

References

[1] I. Ahmad and Y.K. Kwok. A parallel approach to multiprocessor scheduling. In *Int'l Parallel Processing Symposium*, 1995.

[2] W. C. Athas. *Fine Grain Concurrent Computations.* PhD thesis, Dept. of Computer Science, California Institute of Technology, May 1987.

[3] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–24, August 1988.

[4] S. B. Baden. Dynamic load balancing of a vortex calculation running on multiprocessors. Technical Report Vol. 22584, Lawrence Berkeley Lab., 1986.

[5] A. Barak and A. Shiloh. A distributed load-balancing policy for a multicomputer. *Software-Practice and Experience*, 15(9):901–913, September 1985.

[6] M.J. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Trans. Computers*, C-26:570–580, 1987.

[7] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience, accepted for publication*, 1991.

[8] T. L. Casavant and J. G. Kuhl. A formal model of distributed decision-making and its application to distributed load balancing. In *Int'l Conf. on Distributed Computing System*, pages 232–239, May 1986.

[9] 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*, pages 185–192, May 1987.

[10] S. Chakrabarti, A. Ranade, and K. Yelick. Randomized load balancing for tree structured computation. In *IEEE Scalable High Performance Computing Conference*, pages 666–673, 1994.

[11] Y.C. 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*, November 1992.

[12] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. of Parallel Distrib. Comput.*, 7:279–301, 1989.

[13] 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*, pages 583–590, 1989.

[14] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Eng.*, SE-12(5):662–674, May 1986.

[15] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Eval.*, 6(1):53–68, March 1986.

[16] H. El-Rewini and H. H. Ali. Scheduling conditional branching using representative task graphs. *The Journal of Combinatorial Mathematics and Combinatorial Computing*, 1991.

[17] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, June 1990.

[18] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, 1994.

[19] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice-Hall, 1988.

[20] A. Hac and X. Jin. Dynamic load balancing in a distributed system using a decentralized algorithm. In *Int'l Conf. on Distributed Computing System*, pages 170–177, May 1987.

[21] R. M. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. *Journal of ACM*, 40:765–789, 1993.

[22] Y.K. Kwok and I.Ahmad. A static scheduling algorithm using dynamic critical path for assigning parallel algorithms onto multiprocessors. In *Proc. of Int'l Conference on Parallel Processing*, pages II–155 — II–159, 1994.

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

[24] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Trans. Software Engineering*, 13:32–38, January 1987.

[25] Z. Lin. A distributed fair polling scheme applied to parallel logic programming. *International Journal of Parallel Programming*, 20, August 1991.

[26] J.K. Salmon. Parallel hierarchical N-body methods. Technical report, Tech. Report, CRPC-90-14, Center for Research in Parallel Computing, Caltech, 1990., 1990.

[27] J. Saltz, R. Mirchandaney, R. Smith, D. Nicol, and K. Crowley. The PARTY parallel run-time system. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing.* SIAM, 1987.

[28] Niranjan G. Shivaratri, Phillip Krieger, and Mukesh Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, December 1992.

[29] W. Shu and M. Y. Wu. Runtime Incremental Parallel Scheduling (RIPS) for large-scale parallel computers. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 456–463, 1995.

[30] V. Singh and M. R. Genesereth. A variable supply model for distributing deductions. In *9th Intel. Joint Conf. Artificial Intelligence*, pages 39–45, August 1985.

[31] J. A. Stankovic. Simulations of three adaptive, decentralized controlled, job scheduling algorithms. *Computer Networks*, 8(3):199–217, June 1984.

[32] Y.-T. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Trans. Comput.*, C-34(3):204–217, March 1985.

[33] Marc Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel and Distributed System*, 9(4):979–993, September 1993.

[34] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[35] 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*, 24(1):72–85, January 1995.

[36] 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.