Efficient Parallel Divide-and-Conquer for a Class of Interconnection Topologies. [1]

I-Chen Wu

School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

Abstract: In this paper, we propose an efficient scheduling algorithm for expanding any divide-and-conquer (D&C) computation tree on k-dimensional mesh, hypercube, and perfect shuffle networks with p processors. Assume that it takes t_n time steps to expand one node of the tree and t_c time steps to transmit one datum or convey one node. For any D&C computation tree with N nodes, height h, and degree d (maximal number of children of any node), our algorithm requires at most $(N/p + h)t_n + \varphi dht_c$ time steps, where φ is $O(\log_2 p)$ on a hypercube or perfect shuffle network and is $O(\sqrt[k]{p})$ on a $n_{k-1} \times \cdots \times n_0$ mesh network, where $n_{k-1} = \cdots = n_0 = \sqrt[k]{p}$. This algorithm is general in the sense that it does not know the values of N, h, and d, and the shape of the computation tree as well, *a priori*. Most importantly, we can easily obtain a linear speedup by nearly a factor of p, especially when $N \gg ph(1 + \varphi dt_c/t_n)$.

1. Introduction

Divide and conquer (D&C) is a common computation paradigm in which the solution to a problem is obtained by solving subproblems recursively. Examples of D&C computations include various sorting methods such as quick sort [8], computational geometry procedures such as convex hull calculation [14], AI search heuristics such as constraint satisfaction techniques [6], adaptive data classification procedures such as generation and maintenance of quadtrees [16], and numerical methods such as multigrid algorithms [12] for solving partial differential equations.

A D&C computation can be viewed as a process of expanding and shrinking a tree. In this paper, we only consider expanding a tree for simplicity of discussion. Each node in the tree corresponds to a problem instance, and expanded children of the node correspond to its subproblems. If a node expands no children, it is a leaf in the tree. At any time, nodes on a wavefront that cuts across paths from the root to leaves can be active in performing the node expansion operation.

At first glance, one might think that it should be straightforward to perform D&C in parallel, because nodes on the wavefront can all be processed independently. However, if one wants to achieve good load balancing between the processors, then parallelizing D&C becomes nontrivial. In fact, doing efficient D&C on any real parallel machines has been a major challenge to many researchers [4, 5, 9, 18] for many years.

The difficulties are due to the fact that many D&C computations are highly dynamic in the sense that these computations are data-dependent. During computation, a problem instance can be expanded into any number of subproblems depending on the data that have been computed so far. In fact, the trees of D&C computations can be expected to be sparse and totally irregular. As a result, load balancing must be adaptive to the tree structure and must be done dynamically at run

[1] This research was supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title *Research on Parallel Computing* issued by DARPA/CMO under Contract MDA972-90-C-0035, ARPA Order No. 7330, in part by the National Science Foundation and the Defense Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives, and in part by the Office of Naval Research under Contract N00014-90-J-1939.

time. This implies that computation loads need to be moved around between processors during computation. This paper presents efficient scheduling algorithms on several different architectures which can achieve good load balancing while minimizing the communication overhead of moving computations around.

1.1. Definitions and Notation

A D&C algorithm described above needs a *node expansion procedure* to *generate* subproblems recursively. Given a problem instance as the input, we can view the computation as a process of expanding a tree by applying the procedure.

A *tree of a computation* is defined as the tree whose nodes have all been expanded by the node expansion procedure. When a node is *expanded*, zero or more children may be *generated*. If a node does not generate any children, the node is a *leaf*; if a node generates one or more (up to d) children, the node is an *internal node*. Each newly generated node will in turn be expanded by some processor in the future. In the tree, the number of nodes, the *height*, and the *degree*, defined as the maximal number of children of any node, are denoted by N, h, and d respectively. A node is said to be at *tree level* i if it is the i-th node on the path from the root to the node. (For example, the root is at level 1.)

For the parallel system that will carry out the computation, we have the following assumptions.

- The parallel system has p processors. By enumerating these processors, we can let P_i denote the $(i+1)$-st processor in the system, where $0 \le i < p$. (We call i the ID of P_i.) For convenience, $P_{i:j}$ denotes the processors, P_i, P_{i+1}, $\cdots$, P_{j-1}, and P_j.

- It takes t_n *time steps* for a processor to expand a node. Note that for simplicity the time for storing/scheduling a node locally is assumed to be negligible (or is included in the t_n time steps).

- It takes t_c time steps for a sending/receiving processor to send/receive one datum or convey one node over a network connection. We also assume that handling incoming messages *preempts* the node expansion procedure.

  We distinguish between t_n and t_c because t_n can vary a lot depending on the node granularity and t_c can vary a lot depending on the parallel architecture. Also, for simplicity, it takes $O(1)$ time steps for other primitive operations, e.g., context switching and setting a specific time in the future to do some operation.

In this paper, we will consider three different kinds of networks:

1. *k-Dimensional (k-D) Mesh Network:* In this network, there is a k-D array of $n_{k-1} \times n_{k-2} \times \cdots \times n_0 (= p)$ processors, where n_i is the size in dimension i. The processor at the coordinate $(i_{k-1}, i_{k-2}, \cdots, i_0)$ is $P_{i_{k-1}n_{k-2}\cdots n_1 n_0 + \cdots + i_1 n_0 + i_0}$, where $0 \le i_j < n_j$ for each j. In each dimension j, the processor is connected to processors at the coordinate $(i_{k-1}, \cdots, i_j \pm 1, \cdots i_0)$, if they exist. For simplicity, assume $n_{k-1} = \cdots = n_0 = \sqrt[k]{p}$. An example is the iWarp system [3] with an 8×8 mesh network. If in each dimension the last element is connected back to the first one the network is called a *k-D torus network*.

2. *Hypercube Network:* Let $p = 2^q$. This network can be viewed as a q-D mesh network with size two in each dimension. Examples are the hypercube systems described in [7, 17].

3. *Perfect Shuffle Network:* Let $p = 2^q$ and $i_{q-1}i_{q-2}\cdots i_0$ be the binary representation of i. Every processor P_i is connected to $P_{i_{q-2}\cdots i_0 i_{q-1}}$, $P_{i_{q-2}\cdots i_0 \bar{i}_{q-1}}$, $P_{i_0 i_{q-1}\cdots i_1}$, and $P_{\bar{i}_0 i_{q-1}\cdots i_1}$, where $\bar{i}_b$ is the complement of i_b.

In a parallel system, a *scheduling algorithm* for a D&C computation schedules frontier nodes on processors for expansion. A *frontier node* is a node which has been generated but has not yet been expanded. The scheduling algorithm proposed in this paper is general in the sense that it does not know the values of N, h and d, and the shape of the computation tree as well, *a priori*.

For the algorithm, the total time for a D&C computation is the *latency* (the total number of time steps) to complete the whole computation on a parallel system. Assume that one processor can do one *work unit* in one time step. The *aggregate overhead* of a communication operation or processor idling refers to the total number of work units which would have been done if the operation had not been done or the processors had not been idle. For example, if one processor executes one operation and all the other processors are idle in some time step, the aggregate overhead for processor idling is $p-1$ within this time step.

1.2. Previous Work

There have been several approaches in performing parallel D&C. A simple approach (e.g., in [2]) is to expand all the nodes above a fixed level on one processor and then distribute nodes at this level to other processors. Load balancing would be done poorly in this approach when the tree is irregular. Another approach [10, 15, 18] is to distribute generated nodes to balance the load. For this scheme, the communication overhead can be very high. For example, the execution times for the algorithms in [10, 15] are all $O(N(t_n + t_c)/p)$ with high probability.

Recently, some researchers have made efforts to reduce communication overhead. A popular approach [4, 5, 9, 19, 20] is based on the "donate-highest-subtree" strategy, in which an idle processor will be given frontier nodes near the root. Since a subtree rooted near the top usually has many descendants and these descendants may all be expanded locally, this strategy tends to reduce the amount of interprocessor communication. Without using global information, the scheme in [5] with round-robin scheduling requires at most $(N/p + h)t_n + \phi t_c$, where $\phi = O(p^2 h)$, according to our estimation. In contrast, by using global information, Wu and Kung [19] designed an algorithm whose computation time is at most $(N/p + h)t_n + C_{wk}t_c$, where $C_{wk} = O(pdh)$. The computation time for Wu and Kung's algorithm is still high because their algorithm uses one processor to balance the load (for simplicity of their discussion about communication complexity). Obviously, the processor may become a "hot spot". This paper will present an efficient D&C algorithm which uses the *concentration route* technique [11] to reduce the communication overhead on various networks.

1.3. Main Result

Theorem 1 *A scheduling algorithm can be devised such that the total time for a computation tree is*

$$T \leq (N/p + h)t_n + \varphi dht_c$$

where N, h, and d are the number of nodes, the height, and the degree of the tree respectively; $\varphi = O(\lg p)$ on a hypercube or perfect shuffle network[2] *and $\varphi = O(\sqrt[k]{p})$ on a k-D mesh network.*

[2] $\lg b$ denotes $\log_2 b$

For the upper bound of this theorem, the left item, $(N/p + h)t_n$, represents the time mainly spent for computation, while the right item, φdht_c, represents the time mainly spent for communication overhead. We will examine how close these items are to optimum.

In the left item, the two values Nt_n/p and ht_n are the inherent lower bounds of the computation tree because: (1) p processors require at least Nt_n/p time steps to expand N nodes; (2) it takes at least ht_n time steps to expand h nodes on the farthest path from the root to a leaf. Nt_n/p dominates the item, as N usually is an exponential function of h and therefore is much larger than ph.

For the communication overhead, Wu and Kung [19] have proved that the communication cost is $\Omega(C_{wk})$ if the total computation time ignoring communication overhead is near the optimum Nt_n/p. The *communication cost* is defined as the number of cross nodes, where a *cross node* is a node that is generated by one processor but expanded by another processor. Let the right item of the upper bound in Theorem 1 be rewritten as $\varphi C_{wk}t_c/p$. In this new formula, the value $\Theta(C_{wk}t_c)$ is the optimal aggregate overhead for transmitting cross nodes; the value φ is dependent on the network. It appears that φ is related to a network's diameter, the maximum distance between any pair of processors. This diameter causes longer response time to move a node to an idle processor.

By Theorem 1, we can easily obtain a linear speedup by nearly a factor of p, especially when $N \gg ph(1 + \varphi dt_c/t_n)$. This is not like other theoretical results [10, 15, 20]: a linear speedup by a factor of cp can be obtained, where $c(< 1)$ may be a very small constant. Hence, our algorithm is practical for implementation. We will use this algorithm as a basis to develop a programming system on the 26-processor Nectar system [1] developed at Carnegie Mellon University.

In the rest of the paper, Wu and Kung's algorithm [19] on which our algorithm is based will be reviewed in Section 2. To avoid the "hot spot" problem, our algorithm will use a more sophisticated and efficient control mechanism based on the *concentration route* (see [11]), in Section 3. Finally, this algorithm is proved to satisfy Theorem 1 in Section 4.

2. Wu-Kung's Algorithm

In their algorithm, each processor maintains a *local stack* of frontier nodes for depth-first traversal. For each processor, nodes generated by the processor are pushed onto its local stack. Every processor schedules a node from the top of its local stack (if not empty) for expansion, so nodes at deeper levels of the tree will be expanded by the processor earlier.

In addition to local stacks, the algorithm also uses a data structure, called a *global pool* (abbr. *GP*), to keep track of nodes at a particular tree level which has not been scheduled by any processor for expansion. Note that the GP is maintained by some processor. This level, identified by a variable gl, has the following property: nodes at higher levels have all been scheduled by processors. Every processor will try to take a node from the GP to work on whenever it becomes idle (i.e. its local stack is empty).

Initially, the GP contains only the root and the value of gl is zero. When all the nodes at level gl in the GP have been scheduled by the processors, the GP becomes empty. Then, all the processors are requested to send in frontier nodes at level $gl + 1$ from their local stacks after t_n time steps when all the nodes at level $gl + 1$ have been generated. The GP is refilled with this set of new nodes, and gl is increased by one. This process repeats until all the nodes have been expanded.

For the algorithm, we will do a depth-first traversal on each processor by using its local stack while globally we use the donate-highest-subtree strategy by using the GP. Consequently, the processor will exhaust all possible work locally before asking for a new node (from the GP) which has high probability of having the most descendants. This is the basic reason why Wu and Kung prove that the communication cost can be as low as pdh, in spite of the fact that the computation time can be exponential in h. Another advantage of this local depth-first strategy is that it uses the minimum amount of memory.

3. Our Algorithm

Our algorithm based on Wu-Kung's algorithm distributes the GP over all the processors to avoid the hot spot problem. Each processor has a local GP, where the frontier nodes of the GP generated on the processor are stored. In order to refill the GP with frontier nodes at the next level, each processor simply moves these nodes from its local stack to its local GP. When a processor runs out of its local stack, the processor will send out a request for a new node from the GP. Such a node request operation can be viewed as a load balancing problem: move a node of the GP from a (high load) processor containing some nodes of the GP to a (low load) processor being idling.

To perform the load balancing operation, the whole system will synchronize once per *period* (whose value will be determined later). The synchronization for load balancing is "passive" in the sense that the operation will be invoked only when the high load status or the low load status on a processor has been changed. The high load status and the low load status are determined as follows.

1. If no node is being expanded and the local stack is empty, the processor is said to have the *low load* status.
2. If the local GP is not empty, the processor is said to have the *high load* status.

Note that a processor can have both low load status and high load status at the same time. If the status of the processor during some period has been changed, the processor will invoke an operation for load balancing at the beginning of the next period.

The main technique used for load balancing is called *token concentration*. Based on the technique, we concentrate nodes from high load processors to a contiguous block of processors starting at P_0. Then we deconcentrate these nodes to low load processors. In Section 3.1, we will describe how to embed on each network a binary tree which will be used by the technique of token concentration, described in Section 3.2. The details of load balancing are described in Section 3.3.

3.1. Embedded Binary Trees on Networks

For load balancing, we want to embed a binary tree of *processor nodes* (abbr. *p-nodes*) on each network. (Note that several p-nodes can be embedded on the same processor.) For each network, the embedded binary tree, called *an Inorder Leaves binary tree* (abbr. *IL-tree*), must satisfy the following properties:

P1 Each p-node above the lowest level has one or two children; each processor at the lowest level has no child. (If one p-node has only one child, we let it be the left child.)

P2 Each edge of the tree requires at most one physical connection of the network.

P3 At each level, a processor cannot be present twice.

P4 All the processors must be present at the lowest level (once) in the order of processor sequence $(P_0, P_1, ..., P_{p-1})$ from left to right.

Property P1 implies that all the leaves are at the lowest tree level. Property P2 ensures that two p-nodes connected by an edge will need at most one hop to communicate with each other. Property P3 ensures that all the processors at the same level can perform in parallel. Property P4 implies inorder leaves.

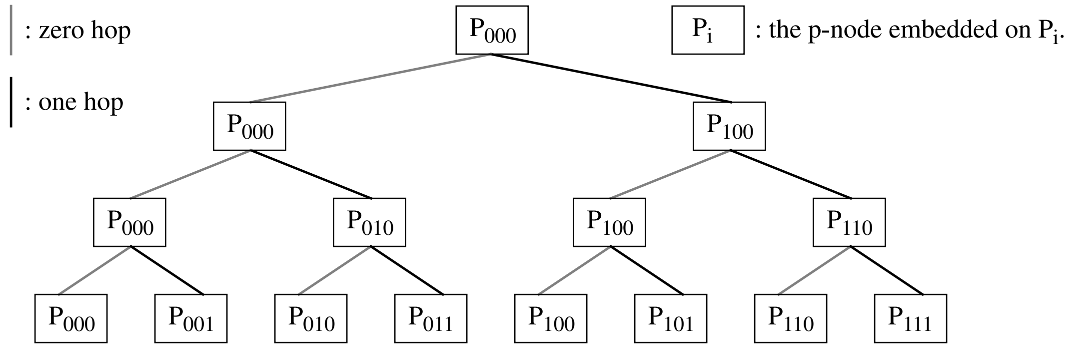

Figure 1: Embedding an IL-tree on a hypercube network for the case $p = 8$.

For the hypercube network, we use the following method to find an IL-tree. Let the root (at level one) be on P_0 and the tree height be $q + 1$. (Recall that $q = \lg p$.) By recurrence, we define that for a p-node on P_i at level $l(\leq q)$ its left child is on $P_{i_{q-1}\cdots i_{q-l+1}0i_{q-l-1}\cdots i_0}$ and its right child is on $P_{i_{q-1}\cdots i_{q-l+1}1i_{q-l-1}\cdots i_0}$. An example for an eight-processor hypercube is illustrated in Figure 1.

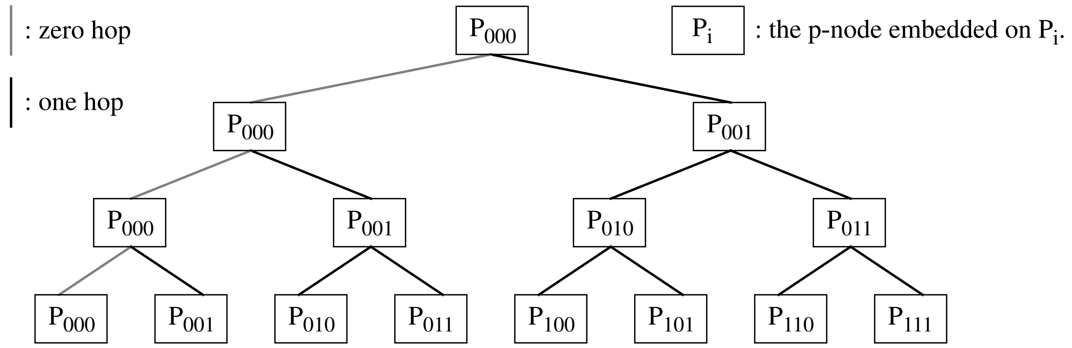

Figure 2: Embedding an IL-tree on a perfect shuffle network for the case $p = 8$.

For the perfect shuffle network, we use the following method to find an IL-tree. Let the root be P_0 and the tree height be $q + 1$. By recurrence, we define that for a p-node on P_i at level $l(\leq q)$ its left child is on $P_{i_{q-2}\cdots i_0 0}$ and its right child is on $P_{i_{q-2}\cdots i_0 1}$. An example for an eight-processor hypercube is illustrated in Figure 2.

For the k-D mesh network, we use the following method to find an IL-tree. Let the root be P_0 and the tree height be $k(n_0 - 1) + 1 = O(n_0)$. (Recall that $n_0 = \sqrt[k]{p}$.) By recurrence, we define that for an internal p-node on P_i at level l, $(m-1)(n_0-1) < l \leq m(n_0-1)$, it has the left child on P_i, but it has the right child on $P_{i+n_0^{k-m}}$ only if P_i is at the coordinate $(i_{k-1}, \cdots, i_{k-m}, \cdots, i_0)$ where $i_{k-m} = l - (m-1)(n_0-1) - 1$. An example for a 3×3 mesh network is illustrated in Figure 3.

Note that each of the above trees satisfies properties P1-P4. This can be easily verified by the help of Figures 1, 2, and 3. In addition, for convenience, the height of the IL-tree for each network is denoted by h_{IL}, which is $q + 1$ for the hypercube or perfect shuffle network and is $O(n_0)$ for a k-D mesh network.

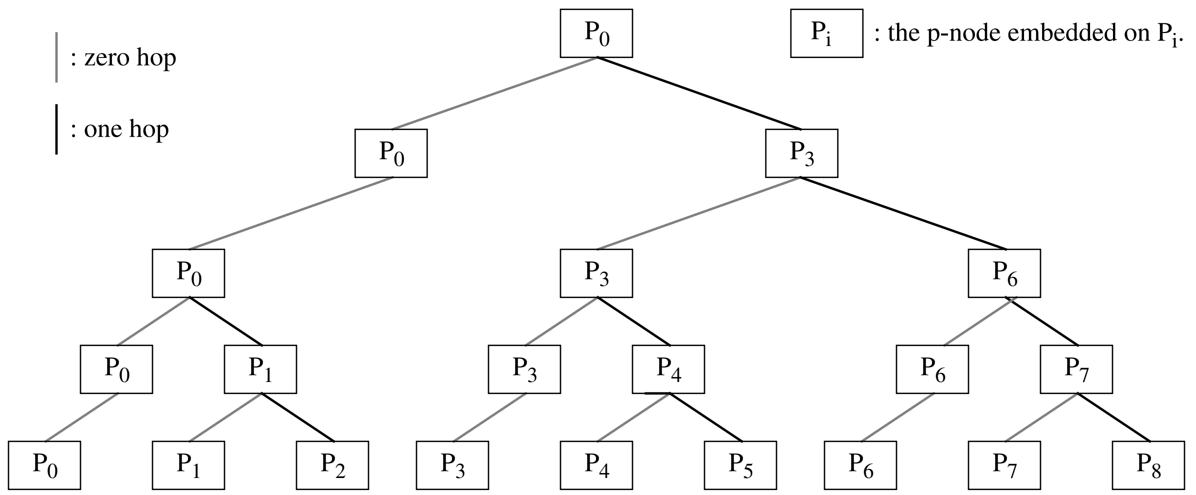

Figure 3: Embedding an IL-tree on a 3 $\times$ 3 network.

3.2. Token Concentration

Token concentration is an important technique used for load balancing (see Leighton's algorithm in Chapter 4 of [13]). Before the operation of token concentration, each processor may create one token. (Note that one processor has no more than one token at any time.) The problem of token concentration is defined as follows (an example is illustrated in Figure 4).

> **m-Token Concentration:** Assume that there are m' tokens each in a distinct processor P_i, $i < p$. Move these tokens such that each processor P_i, $i < \min(m, m')$, has one token.

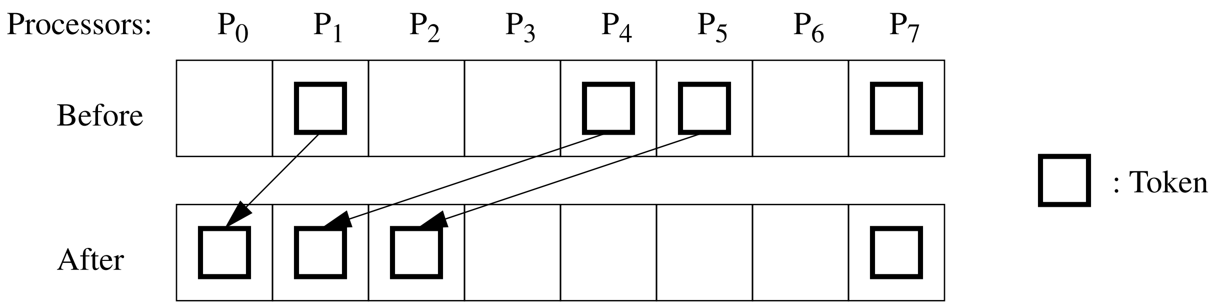

Figure 4: 3-token concentration.

Token concentration consists of two stages: (1) select a destination processor for each token; (2) send the token to the processor. The simplest method for the first stage is that for each token on P_i its destination processor is P_j where j is the number of tokens on processors $P_{0:i-1}$, as illustrated in Figure 4. This method requires a wave-up procedure COUNT and a wave-down procedure LABEL. The second stage uses a route procedure CONCENTRATE. The details of these procedures are described as follows.

COUNT

For each p-node v in the IL-tree, procedure COUNT counts the number S_v of tokens on all the leaves in the subtree rooted at v. For a leaf p-node v, $S_v = 0$ if v has no token; $S_v = 1$ if v has one token. For each internal p-node v, $S_v = S_{lc} + S_{rc}$, where p-node lc (rc) is the left (right) child of v. We can simply let $S_{rc} = 0$ if the right child does not exist. These operations can be done level by level from leaves up to the root in the IL-tree.

The total latency for procedure COUNT is $T_{count} = O(h_{IL}t_c)$ because all processors at the same level can work in parallel by property P3. Hence, the aggregate overhead of the procedure is at most pT_{count}. To reduce the upper bound of the overhead, we let the operation for the procedure be "passive" as described before. Let $t_p(= O(t_c))$ be the maximal propagation time for a processor to receive messages from two children and T_{start} be the starting time of the procedure. Initially, the leaf p-node v will report S_v to his parents only if v has created (or deleted) a token since the previous COUNT for the same type of tokens. (Note we will use two different types of tokens in the next section.) For each internal p-node v at level l, if it receives some values from its children by $T_{start} + (q - l)t_p$, it (except for the root) will report the new S_v to its parent at $T_{start} + (q - l)t_p$. Otherwise, v will do nothing. If for all processors their tokens have not been changed, no operation will be invoked by the procedure. If the number of tokens on a processor has been changed, it will cause at most h_{IL} processors to update their values Ss. Thus, the aggregate overhead of the procedure becomes at most δT_{count}, where δ is the number of processors changing their tokens.

LABEL(m)

After finishing procedure COUNT, we use procedure LABEL to label $m(> 0)$ tokens, each on P_i with $L_i(< m)$, where L_i is the number of tokens in $P_{0:i-1}$. (P_{L_i} is the processor that the token will be sent to.) For each p-node v, let D_v be the number of tokens on $P_{0:j-1}$, by assuming that P_j is the leftmost leaf in the subtree rooted at v. For the root r, let $D_r = 0$. For each p-node v, let $D_{lc} = D_v$ and let $D_{rc} = D_v + S_{lc}$, where lc (rc) is the left (right) child, if any. Obviously, for a leaf v on P_i with a token, $L_i = D_v$. To derive these values D_v, we can do these operations level by level from root down to the leaves in the IL-tree.

The latency for procedure LABEL is $T_{label} = O(h_{IL}t_c)$ because all processors at the same level can work in parallel. Then, the aggregate overhead of the procedure is at most pT_{label}. Again, to reduce the aggregate overhead, we let the parent of each p-node v not send D_v to v if $S_v = 0$ or $D_v \geq m$. Since only the ancestors of the first m tokens need to send D_v down, the aggregate overhead becomes at most mT_{label}.

CONCENTRATE

Procedure CONCENTRATE moves the $m(> 0)$ labeled tokens to a contiguous block of processors $P_{0:m-1}$. More precisely, each token associated with a label $L_i(< m)$ by procedure LABEL will be moved to processor P_{L_i}.

Nassimi and Sahni [11] have designed the collision-free concentration route in $O((\lg p)t_c)$ on hypercube and perfect shuffle networks. For a hypercube network, their method is very simple: at the i-th stage starting at the $(i \cdot 2t_c)$-th time step in this procedure, each processor with a token compares the i-th bits of the current processor's ID and the token's label L_i; if both are not the same, the token will be sent to the other processor in dimension i, otherwise no move is required. They proved that this kind of routing is collision-free (see [11]). So, the latency is $T_{conc} = O((\lg p)t_c)$ and the aggregate overhead for the procedure is at most mT_{conc}. This collision-free concentrate route can also be implemented on a perfect shuffle in the same manner. As for a k-D mesh (or torus) network, we will use the same idea: at the i-th stage, each token is moved to the right position in the i-th dimension. For a k-D torus network, tokens can be sent in the same direction without collision. For a k-D mesh network, since tokens may need to be moved in the opposite directions, collisions may happen. To solve the problem, we divide a stage into two substages and we move tokens in one direction at each substage so that it becomes collision-free. Since each substage takes at most $O(n_0 t_c)$ time steps, for the whole operation the latency is $T_{conc} = O(n_0 t_c)$ and the aggregate overhead is at most mT_{conc}. Interestingly, for these three networks, the latencies T_{conc} are all $O(h_{IL} t_c)$ and the aggregate overheads are at most mT_{conc}.

We will also need one more procedure, DECONCENTRATE, which just reverses the concentration route. For this procedure, each processor in $P_{0:m-1}$ must have one token associated with the ID of a distinct destination processor. This procedure will move these tokens to the corresponding destination processors. Since this procedure reverses the CONCENTRATE procedure, it is collision free and the time complexity is the same as that for procedure CONCENTRATE.

3.3. Load Balancing

For each period, we will do load balancing between high load processors and low load processors. At the beginning of one period, a high load processor, called a *busy processor* in the period, creates a *B-token* associated with a node selected from the GP unless there has been a B-token. Note that the processor which creates a B-token during the previous period may still has the B-token if the number of high load processors are more than that of low load processors. A low load processor, say P_i, called an *idle processor* in the period, creates a *I-token* with the value i unless there has been an I-token. (Note that we need two different token concentration operations for B-tokens and for I-tokens.) Let p_B (p_I) be the number of busy (idle) processors and $p_{min} = \min(p_B, p_I)$.

The basic idea of load balancing is that during each period we will concentrate B-tokens and I-tokens and then deconcentrate B-tokens by using the values of I-tokens as the IDs of the destination processors. Thus, some p_{min} idle processors will each obtain a node (or a B-token) for expansion.

Before presenting the details of load balancing, we introduce one more relevant procedure, called REFILL, to refill the GP when the GP becomes empty. Initially, the procedure broadcasts a message from the root to each processor. This message causes each processor to increase gl by one. After t_n time steps (the time taken to finish expanding a node), all the nodes above the new level gl must have been expanded, that is, all the nodes at level gl must have been generated. At that time, each processor can refill its local GP with nodes at level gl from its local stack. Since each processor does depth-first search locally, there are at most d frontier nodes at level $l(\leq gl)$. So, each local GP can be refilled with at most d frontier nodes. Then, restart a new period and reset all the tokens.

For a D&C computation tree, initially, we let the GP have the root (on some local GP), gl be 1, and all the local stacks be empty. Then, we create B-tokens and I-tokens to begin the first load balancing period. During each period of time, the load balancing operation uses the following steps.

Step 1: For B-tokens, call procedure COUNT. Then the root in the IL-tree knows p_B.

Step 2: For I-tokens, call procedure COUNT. Then the root knows p_I.

For all processors, if their tokens have been the same as those during the previous period, nothing has been done in these two steps and then all the following time steps can be skipped.

Step 3: If $p_I = p$ and $p_B = 0$ at this time, finish the whole computation. If $p_B = 0$, call procedure REFILL and then restart a new period. (Go back to step 1.)

Step 4: For B-tokens, call procedure LABEL(p_{min}).

Step 5: For I-tokens, call procedure LABEL(p_{min}).

Step 6: For B-tokens, call procedure CONCENTRATE.

Step 7: For I-tokens, call procedure CONCENTRATE.

Step 8: For each of processors $P_{0:p_{min}-1}$, use the value of I-token as the ID of the destination processor and call procedure DECONCENTRATE to deconcentrate B-tokens. Then, each of the first p_{min} idle processors obtains one node (in a B-token) and then removes both I-token and B-token.

Steps 1, 4, and 6 concentrate B-tokens to processors $P_{0:p_{min}-1}$; steps 2, 5, and 7 concentrate I-tokens to processors $P_{0:p_{min}-1}$. Then step 8 deconcentrates B-tokens by I-tokens so that each idle processor obtains one B-token, a node from the GP. As for step 3, we check if the computation is complete or the GP is empty. If the GP is empty, the GP will be refilled with nodes at $gl + 1$ by procedure REFILL.

Note that if all the tokens on processors have been the same as those during the previous period, steps 3-8 can be skipped (see step 2). This is because $p_I = 0$, i.e., $p_{min} = 0$. The reason why $p_I = 0$ is explained as follows. Assume that (1) for all processors their tokens have been the same as those during the previous period (but not for the period before the previous one) and (2) the $p_I \neq 0$ for the previous and current periods. If $p_B = 0$ in the previous period, the computation have been finished or the procedure REFILL has been called by step 3 during the previous period. If the procedure REFILL has been called, I-tokens have been reset. If $p_B \neq 0$ during the previous period, some busy processor must have sent a node to one idle processor. Therefore, the token with the node has been moved out during the previous period. These contradict the first assumption. So, if all the tokens on processors have been the same as those during the previous period, $p_I = 0$.

For the following discussion, we call this period a *refill period* if $p_B = 0$ (see step 3), a *heavy load period* if $p_I \leq p_B(\neq 0)$, or a *light load period* if $p_I > p_B(\neq 0)$. In addition, the time between two consecutive REFILL procedures is called *one round*.

One property is that one round has at most d light load periods. For each light load period, each busy processor must send one node from its local GP to a distinct idle processor; therefore the number of nodes in the local GP will decrease by one. Since each local GP will be refilled with at most d nodes at the beginning of one round, the GP will become empty again (i.e. the round is over) after at most d light load periods.

Now, we can define the time $\mathcal{P}$ for one period as the maximal latency to finish the above operations in steps 1-8 except for procedure REFILL. Since the latency for each step except for procedure REFILL is $O(h_{IL}t_c)$, we know that $\mathcal{P} = O(h_{IL}t_c)$.

4. Proof of Theorem 1

A D&C computation tree of height h requires h rounds. To obtain the total time, we will derive the aggregate overhead for load balancing or processor idling during each round. Note that if a processor is not expanding a node, the processor must get idle or do the load balancing operation. Since each round has only one refill period (including procedure REFILL) and at most d light load periods, the aggregate overhead for these periods is at most $p(t_n + (d + 1)\mathcal{P})$.

Now we want to derive the aggregate overhead for processor idling and load balancing during all the heavy load periods in one round. The processor becomes idle when the local stack is empty and a requested node has not come up yet. When the local stack becomes empty, the processor will wait at most $\mathcal{P}$ time before creating a new I-token in the next load balancing period. In addition, during each heavy load period, each idle processor must be able to receive a new node. Thus, it takes at most $2\mathcal{P}$ time to receive a requested node within heavy load periods. Since we can refill the GP with at most dp nodes at the beginning of one round, at most dp nodes will be requested. Therefore, during all the heavy load periods of one round, the aggregate overhead of processor idling is at most $2dp\mathcal{P}$.

Now, we will consider the aggregate overhead for load balancing. For the ith heavy load period in the round, let p_B^i, p_I^i, and p_{min}^i denote p_B, p_I, and p_{min} in this period respectively. In addition, let δ_B^i (δ_I^i) denote the number of B-tokens (I-tokens) which have been changed. Then, the aggregate overhead for steps 1 and 2 is at most $(\delta_B^i + \delta_I^i)T_{count}$. The aggregate overhead for step 3 is zero if both p_B and p_I are zeros, and is $O(1)$ otherwise. The aggregate overhead for steps 4-8 is at most $p_{min}^i(2T_{label} + 3T_{conc})$. Since $p_{min}^i = p_I^i$ for a heavy load period, the aggregate overhead of load balancing in the period is $(\delta_B^i + \delta_I^i + p_I^i)T_{heavy}$, where $T_{heavy} = O(h_{IL}t_c)$. In total, the aggregate overhead for all the heavy load periods in one round is $T_p \leq \sum_i(\delta_I^i + \delta_B^i + p_I^i)T_{heavy}$. Since the GP has at most dp nodes at the beginning of one round, $\sum_i(\delta_B^i + \delta_I^i) \leq 4dp$. Since during each heavy load period each idle processor will receive a node, $\sum_i p_I^i \leq dp$. Thus, $T_p \leq 5dpT_{heavy}$.

Thus, for each round, the aggregate overhead for load balancing and processor idling is, in total, $pt_n + \varphi dpt_c$, where $\varphi = O(h_{IL})$. Since there are h rounds for the computation, the entire aggregate overhead is at most $pht_n + \varphi pdht_c$. Thus, the total latency for the computation is at most $(Nt_n + (pht_n + \varphi pdht_c))/p = ((N/p + h)t_n + \varphi dht_c)$. $\Box$

Acknowledgement

I would like to thank H.T. Kung for his encouragement and discussion for this work. I would also like to thank Greg C. Plaxton for a helpful discussion about the concentration route.

References

[1] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste. The design of Nectar: a network backplane for heterogeneous multicomputers. In *Third Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, Boston, Massachusetts, April 1989.

[2] H.E. Bal. *The shared data-object model as a paradigm for programming distributed systems*. PhD thesis, Vrije Universiteit, Amsterdam, Netherlands, 1989.

[3] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[4] C. Ferguson and R.E. Korf. Distributed tree search and its application to alpha-beta prunning. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI 1988)*, pages 128–132, Saint Paul, August 1988.

[5] R. Finkel and U. Manber. DIB – a Distributed Implementation of Backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[6] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, October 1980.

[7] J. P. Hayes, T. Mudge, Q. F. Stout, S. Colley, and J. Palmer. A microprocessor-based hypercube supercomputer. *IEEE MICRO*, 6(5):6–17, October 1986.

[8] C.A.R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[9] V. Kumar and V. N. Rao. Parallel depth-first search, part I: implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987.

[10] T. Leighton, M. Newman, A. G. Ranade, and E. Schwabe. Dynamic tree embeddings in butterflies and hypercubes. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 224–234, June 1989.

[11] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29(3):642–667, July 1982.

[12] J.M. Ortega and R.G. Voigt. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, 27(2):149–240, 1985.

[13] C.G. Plaxton. *Efficient Computation on Sparse Interconnection Networks*. PhD thesis, Dept of CS, Stanford University, September 1989.

[14] F.P. Preparata and M.I. Shamos. *Computational Geometry: an Introduction*. Springer-Verlag, New York, 1985.

[15] A. G. Ranade. Optimal speedup for backtrack search on a butterfly network. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 40–48, July 1991.

[16] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image processing, and GIS*. Addison-Wesley, Reading, MA., 1990.

[17] C. L. Seitz. The cosmic cube. *Communications ACM*, 28(1):22–33, January 1985.

[18] W. Shu and L. V. Kale. A dynamic scheduling strategy for Chare-Kernel system. In *Proceedings of Supercomputing '89*, pages 389–398, New York, NY, November 1989.

[19] I.-C. Wu and H.T. Kung. Communication complexity for parallel divide-and-conquer. To appear in *1991 Symposium on Foundations of Computer Science*, October 1991.

[20] Y. Zhang. *Parallel Algorithms for Combinatorial search problems*. PhD thesis, U.C. Berkeley, November 1989.