Mapping Divide-and-Conquer Algorithms to Parallel Architectures

V. Lo, S. V. Rajopadhye,
S. Gupta,
D. Keldsen,
M. Mohamed,
J. Telle

CIS-TR-89-19
January 19, 1990

Abstract

In this paper, we identify the binomial tree as an ideal computation structure for parallel divide-and-conquer algorithms. We show its superiority to the classic full binary tree structure with respect to speedup and efficiency. We also present elegant and efficient algorithms for mapping the binomial tree to two interconnection networks commonly used in multicomputers, namely the hypercube and the two-dimensional mesh. Our mappings are optimal with respect to both average dilation and link contention. We discuss the practical implications of these results for message-passing architectures using store-and-forward routing vs. those using wormhole routing.

Department of Computer and Information Science
University of Oregon

Mapping Divide-and-Conquer Algorithms to Parallel Architectures*

V. Lo, S. V. Rajopadhye,
S. Gupta,
D. Keldsen,
M. Mohamed,
J. Telle
Dept. of Computer Science
University of Oregon
Eugene, OR 97403-1202
lo@cs.uoregon.edu

October 20, 1989

Abstract

In this paper, we identify the binomial tree as an ideal computation structure for parallel divide-and-conquer algorithms. We show its superiority to the classic full binary tree structure with respect to speedup and efficiency. We also present elegant and efficient algorithms for mapping the binomial tree to two interconnection networks commonly used in multicomputers, namely the hypercube and the two-dimensional mesh. Our mappings are optimal with respect to both average dilation and link contention. We discuss the practical implications of these results for message-passing architectures using store-and-forward routing vs. those using wormhole routing.

*This research was supported by a grant from the Oregon Advanced Computing Institute (OACIS). OACIS is a consortium of academic, industrial, and government agencies in the state of Oregon.

1 Introduction

The problem of mapping parallel algorithms to parallel architectures involves the assignment of tasks in the parallel computation to processors and the routing of messages through the underlying communication network. This problem has been found to be NP-hard for a variety of models of parallel computations and for a spectrum of optimality criteria, including minimization of total interprocessor communication, minimization of response time, and several load balancing metrics. As a result, research in the area of mapping algorithms has focused on the design of suboptimal heuristics and the development of efficient optimal solutions for restricted types of parallel computations.

In this paper, we identify the binomial tree as an ideal computation structure for parallel divide-and-conquer algorithms. We show its superiority to the classic full binary tree structure with respect to speedup and efficiency. We also present elegant and efficient algorithms for mapping the binomial tree to two interconnection networks commonly used in multicomputers, namely the hypercube and the two-dimensional mesh.

This work is part of a larger research project called OREGAMI [LRG+] whose purpose is the design of software tools for mapping parallel algorithms to parallel architectures. OREGAMI includes a description language for specifying the static and dynamic communication characteristics of regular parallel computations, a library of mapping algorithms for regular and arbitrary computations to a spectrum of interconnection networks, and an interactive graphics tool for visualization and evaluation of mappings. The mappings described in this paper form part of the OREGAMI library of mapping algorithms.

1.1 Our Model of Parallel Algorithms and Parallel Architectures

We view a parallel algorithm to be a network of communicating sequential processes; these processes are persistent throughout the lifetime of the computation, are *large grained*, and are executed on one processor of a multicomputer throughout the computation. Thus, we model a parallel computation as a graph $G_C = (V_C, E_C)$ where the nodes V_C represent tasks and an edge between nodes A and B represents a communication (possibly bi-directional) between A and B. Note that this is the static task graph model proposed by Stone [Sto77] and Bokhari [Bok87] and not the precedence-constrained (DAG) model. Many computation graphs have discernable structures or patterns — computation graphs in the shape of rings, chordal-rings, trees etc. can be found in the literature.

The parallel architectures that we consider are message-passing *multicomputers*. Multicomputers are a network of processors, each having a local memory and I/O facilities for sending and receiving messages to and from other processors in the network; each processor has access only to its local memory and there is no global memory. The topology of the interconnection network may be arbitrary, but usually is some well-known topology such as a mesh, hypercube, cube-connected-cycles etc. . Examples of commercially available multicomputers are the iPSC-2 and NCUBE hypercubes and Transputer networks from Inmos and Cogent. We also model the multicomputer architecture as a graph $G_A = (V_A, E_A)$, where the nodes V_A represent processors and the edges E_A correspond to the processor-to-processor connections of the underlying interconnection network.

1.2 The Mapping Problem

Mapping involves two decisions: how the tasks of the computation graph will be allocated to the processors of the multicomputer and how the communication edges will be laid out along the links of the processors. In this paper, we assume that the number of available processors is greater than or equal to the number of tasks so that each task can be assigned to a unique processor. (When this is not the case, the graph can be *contracted* so that a node of the contracted graph represents a number of nodes of the original graph. Techniques for contraction can be found in the literature ([FF82], [BS87], [Lo88], [LRG+] and will not be described here). Clearly, the mapping chosen will affect the overall execution time of the program and thus the speedup attainable by the parallel algorithm. A "good" mapping should achieve load balancing among the processors and should minimize the overhead of interprocessor communication.

More formally, a mapping is specified by two functions *map-node* and *map-edge*, which can be described as follows

map-node: $V_C \rightarrow V_A$
map-edge: $E_C \rightarrow Paths_A$

under the constraint that

map-edge$(< a, b > \in E_C)$ is a path from *map-node(a)* to *map-node(b)* in G_A

where

computation graph $G_C =< V_C, E_C >$,
architecture graph $G_A =< V_A, E_A >$,
$Paths_A$ is the set of all possible paths in G_A

In Section 2 of this paper we describe the conventional computation graph for divide-and-conquer algorithms, the complete binary tree, and identify a more suitable computation graph, the binomial tree. In section 3 we describe mappings of this graph onto the hypercube and the

2-dimensional mesh. In section 4 we evaluate our mappings with respect to average dilation and link contention, and we discuss the practical implications of these results for message-passing architectures using store-and-forward routing vs. those using wormhole routing. Section 5 contains conclusions and areas of ongoing and future work.

2 Tree Structures for Parallel Divide-and-Conquer Algorithms

2.1 Conventional Divide-and-Conquer

The computation graph associated with many parallel divide-and-conquer algorithms is the complete *binary tree*. This is true when the algorithm is designed as follows:

Step 1 Give the root of the binary tree the problem to be solved

Step 2 Let the root divide the problem into two sub-problems and pass them on to its two children to solve

Step 3 Continue Step 2 recursively until the problem has been broken up sufficiently to exploit a desired degree of parallelism or to easily-solved based cases

Step 4 Let the *leaves* of the binary tree solve the subproblems they hold and pass on the results to their parents

Step 5 Let the parents (which are the interior nodes) combine the results received from their children and pass on this result to their parents

Step 6 Continue Step 5 recursively till the values reach the root

The example shown in Figure 1 illustrates this process.

Example: Mergesort the list (4 7 5 6 2 10 20 27 1 3 32 8 9 15 12 67) using a *complete binary tree* as the computation graph.

Although this implementation of a parallel divide-and-conquer algorithm is intuitive, it is naive and inefficient. The inefficiencies are both in terms of the number of processors used and processor utilization. (Because we assume one process per processor, the two words are used interchangeably.)

- If the input is of size n and the leaf computation in step 4 involves the trivial sorting of 2-element lists, then the total number of nodes in the binary tree must be $n - 1$. As we shall see, this is many more processors than necessary.

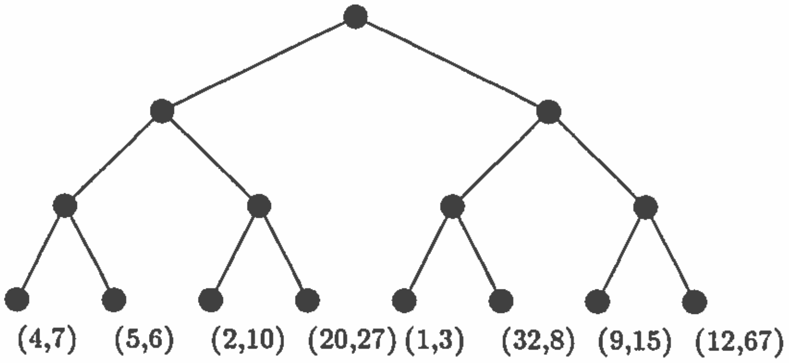


Figure 1: Divide-and-Conquer Using a Binary Tree

- The processor utilization in the binary tree is poor since the interior nodes are *idle* while the leaves do the computation — the interior nodes only take part in the dividing and merging of data.

In the following section we will describe a computation graph that is free from both the above disadvantages.

2.2 Binomial Trees for Divide-and-Conquer

The *binomial tree* is a combinatorial structure defined inductively as shown in Figure 2 [Knu73].

A *canonical* labeling of the binomial tree is to label the tree in post-order starting at zero and using the binary representation of integers. Some of the properties of the binomial tree (B_p) that are relevant to this paper are

- B_p has 2^p nodes
- B_p has $2^p - 1$ edges
- the depth (max. no. of edges from the root to a leaf) of B_p is p
- only the *root* B_p of has p children, and only *one* of its children has $p-1$ children

We propose the binomial tree as a better alternative to the complete binary tree as the computation graph for divide-and-conquer algorithms. To understand how this can be done, consider the

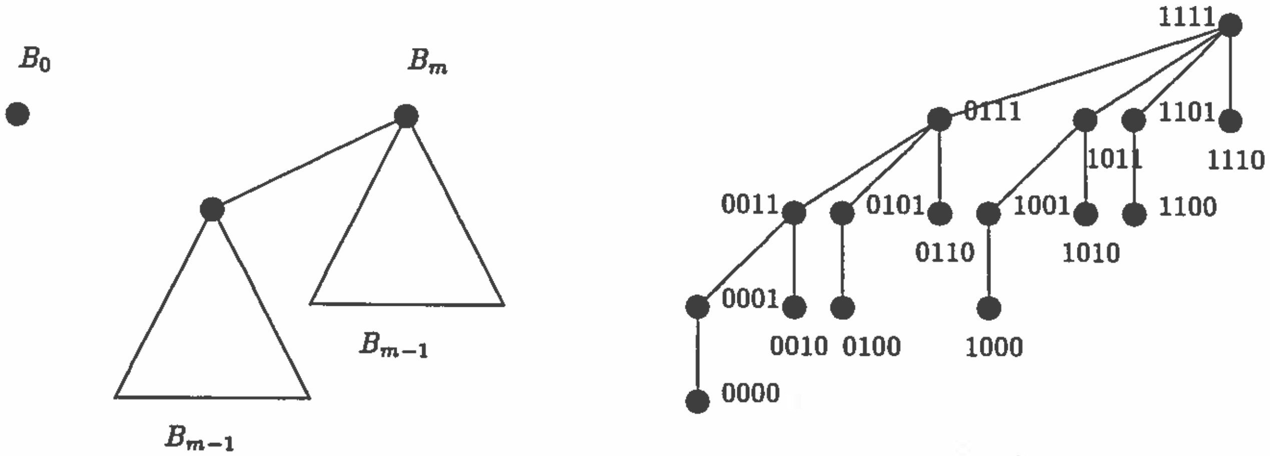


Figure 2: Binomial Tree and Its Canonical Labelling

following changes to Steps 2, 3, 4 and 5 of the six steps described in Section 2.

- **Step 2** Let the root divide the problem into two sub-problems and pass on *one of them* to a child that has not received any work yet, and *keep the other half to itself*
- **Step 3** Let *every* node perform Step 2 recursively until the problem has been broken up sufficiently to exploit the desired degree of parallelism
- **Step 4** *Every* node does the computation assigned to it
- **Step 5** The results are passed up the tree and merged *in the reverse of the order in which the sub-problems were passed down the tree*

Steps 1 and 6 are as before. The previous mergesort example is used to illustrate this process, as shown in Figure 3.

We must now convince ourselves that the procedure described above, indeed results in a binomial tree in the general case. In the above procedure we start with a single node and each application of **Step 2** adds a new (leaf) child to every node of the existing tree. Since we start with a single node, which is the binomial tree B_0, we now have to prove the following

> **Remark 1:** If a leaf node is appended to every node in a binomial tree B_n, the resulting graph is the binomial tree B_{n+1} with the root of B_n as the root of B_{n+1}, $\forall\ n \in N$.
>
> **Proof:** We will use an inductive proof. As our *base case* we take the binomial tree B_0 and add a leaf to every node. Let the graph thus produced be A.

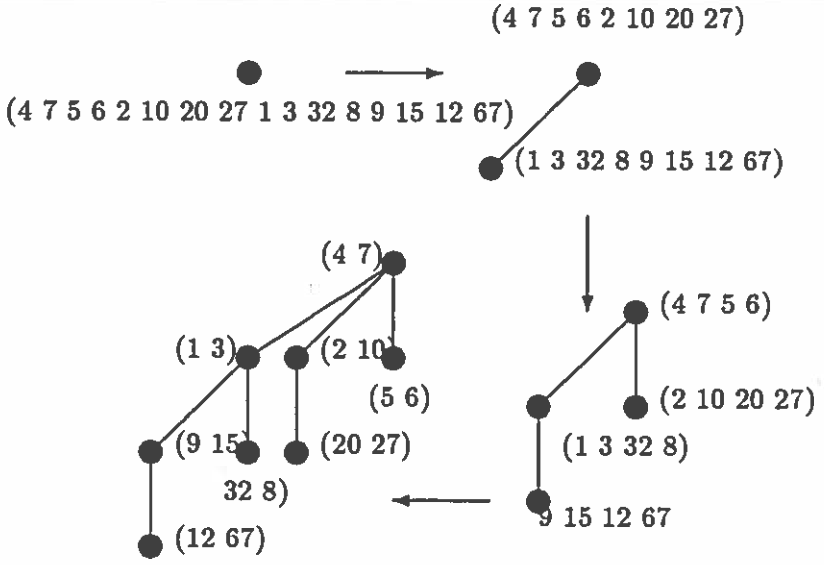


Figure 3: Mergesort Using the Binomial Tree

It is clear that A is the binomial tree B_1 and thus our assertion holds for the base case. Let us now suppose that the assertion holds for all binomial trees of degree $\leq m-1$, where $m \in N$. We now have to prove the assertion for the binomial tree B_m.

By the definition of a binomial tree, B_m consists of two binomial trees B_{m-1} connected root to root. Let the roots of the subtree be a and b. Let us now add a leaf to every node of B_m. By our inductive assumption, this will cause the two trees B_{m-1} to become binomial trees B_m, with roots a and b. Also, since no edges were removed from the original tree, the nodes a and b are still connected. Thus the new graph consists of two binomial trees B_m connected root to root and hence must be a binomial tree B_{m+1}. Thus the assertion holds for binomial trees of degree m. □

It is quite clear that the "keep half, give away half" approach that is followed by each process (in contrast to giving away both halves) results in *each* process doing the same amount of computation (sorting in case of the example). Of course, the leaves do not take part in the *conquer* (merge) part, but that was the case even for the binary tree. Thus it is obvious intuitively that the binomial tree is a more efficient computation graph. More precisely,

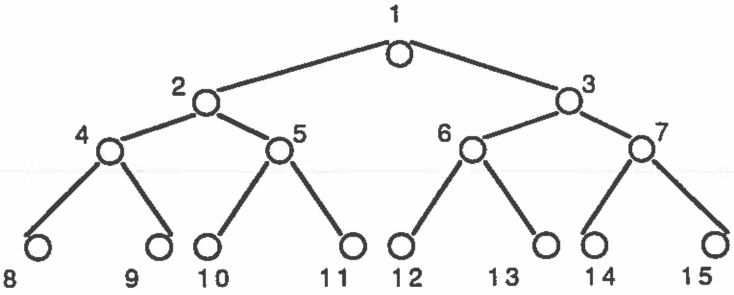
- If the input is of size n and we wish the leaf computation in step 4 to be the trivial sorting of 2-element lists, then the number of nodes (processes) in the binomial tree must be $n/2$. This is about half the size of the corresponding complete binary tree which is of size $n-1$.
- In the binomial tree, once a process receives work to do, it never idles until it is completely finished. The pattern of busy and idle times for the conventional divide-and-conquer vs. that for the binomial tree divide-and-conquer are shown in the Gantt charts in Figure 4.

From the above observations, we make the following claims about the use of the binomial tree for divide-and-conquer parallel algorithms. Let $Start_A$ denote the time an algorithm A starts execution and let $Finish_A$ denote the time an algorithm A finishes execution. The completion time of algorithm A is defined as $C_A = Finish_A - Start_A$. Also let *dnc-binary* and *dnc-binomial* represent parallel divide-and-conquer algorithms structured as the full binary tree and the binomial tree, respectively.

Result 1: Completion time $C_{dnc-binomial} \leq C_{dnc-binary}$.

Proof: We give here an informal visual proof using the Gantt chart representation of the execution sequence of the two divide-and-conquer algorithms shown in Figure 4. The Gantt charts simply lays out the sequence of events along the time line. The completion time of the algorithm is the time at which the last process completes execution and is marked on the charts. As can be seen by comparing the two charts, the binomial tree version of the parallel divide-and-conquer algorithm is faster because it only sends(receives) data to(from) one child(parent) after each divide(merge) stage

1	S2	S3												R2	M
2			S4	S5							R4	R5	M	S1	
3			S6	S7							R6	R7	M	S1	
4					S8	S9		R8	R9	M	S2				
5					S10	S11		R10	R11	M	S2				
6					S12	S13		R12	R13	M	S3				
7					S14	S15		R14	R15	M	S3				
8							C	S4							
9							C	S4							
10							C	S5							
11							C	S5							
12							C	S6							
13							C	S6							
14							C	S7							
15							C	S7							



0	S1	S2	S4	C	R4	M	R2	M	R0	M
1	R0	S3	S5	C	R5	M	R3	M	S0	
2		R0	S6	C	R6	M	S0			
3		R3	S7	C	R7	M	S1			
4			R0	C	S0					
5			R1	C	S1					
6			R2	C	S2					
7			R3	C	S3					

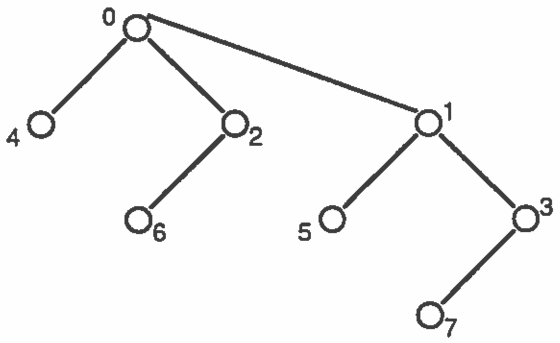


Figure 4: Gantt Charts for Divide and Conquer Algorithms

in the computation. We have assumed that when a parent node send portions of the computation to each of its two children in the full binary tree, the two sends must be serialized. This is true of current message-passing technologies and does not affect the correctness of the proof. If parallel multicast is possible, the completion times are at best equal. □

Result 2: The efficiency $E_{dnc-binomial} \geq \frac{2(n-1)}{n} \star E_{dnc-binary}$.
Proof: Let $A - sequential$ be the fastest serial algorithm for an algorithm A, and let $A - parallel$ be a parallel implementation of algorithm A.
Recall that speedup $S_A = \frac{C_{A-sequential}}{C_{A-parallel}}$ and that
efficiency $E_A = \frac{S_A}{\text{no. of processors used in}_{A-parallel}}$.

From Result 1, we can see that

$$S_{dnc-binomial} = \frac{C_{dnc-sequential}}{C_{dnc-binomial}} \geq \frac{C_{dnc-sequential}}{C_{dnc-binary}} = S_{dnc-binary}$$

Since

$$E_{dnc-binomial} = \frac{S_{dnc-binomial}}{n/2} \text{ and } E_{dnc-binary} = \frac{S_{dnc-binary}}{n-1},$$
$$E_{dnc-binomial} \star (n/2) \geq E_{dnc-binary} \star (n-1)$$

and we have the desired result. □

3 Mapping Algorithms

Below, we present algorithms for mapping the binomial tree onto the hypercube and the 2-dimensional mesh. We assume that the number of processors is equal to the number of processes and thus that we assign exactly one process to each processor. A point to note is that this requirement is not difficult to meet for divide-and-conquer algorithms since we can divide a problem only as far as we want to, and thus can restrict the size of the computation graph to match the size of the target architecture.

3.1 Mapping onto a Hypercube

The first mapping, Mapping H, maps B_m to a hypercube of degree m. In the previous section we showed how to obtain B_m by attaching a leaf node to every node of B_{m-1}. We will now show that growing a binomial tree in this manner provides a mapping onto the hypecube. Let us start with the binomial tree of one node B_0 and label it 0 in binary. As we grow B_0 to B_1, B_1 to B_2 and so on, we will use the following labeling scheme:

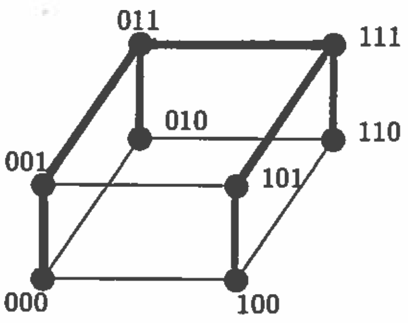


Figure 5: Mapping the Binomial Tree to the Hypercube

> *bt-label:* If a is a node in B_{m-1} with label $a_m a_{m-1} \ldots a_0$ and b is the leaf attached to a while growing B_{m-1} to B_m, then in B_m, a has the label $a_m a_{m-1} \ldots a_0 1$ while b has the label $a_m a_{m-1} \ldots a_0 0$

Obviously this scheme results in a labeling of B_m such that any two adjacent nodes have labels that differ in one bit. Also recall that B_m has 2^m nodes. The canonical labeling scheme for the hypercube is similar:

> *hc-label:* Nodes in the *m-dimensional hypercube* are labeled with the binary numbers from 0 to 2^m. Two nodes are adjacent if their labels differ in exactly one bit position.

Thus, our mapping consists of placing node a of B_m onto processor p of the hypercube iff a and p have the same label. Every edge of the binomial tree is mapped along the corresponding edge of the hypercube. Figure 5 illustrates the mapping for an 8-node binomial tree to the $3 - cube$

Mapping H:

$$map - node(a) = p \text{ iff } bt - label(a) = hc - label(p)$$

and

$$\textit{map-edge}(< a, b >) =< p_1, p_2 > \text{ iff } \textit{map-node}(a) = p_1 \text{ and } \textit{map-node}(b) = p_2$$

We note in passing that the above labeling scheme results in a labeling that is identical to the canonical binomial tree labeling described earlier. Also, this mapping shows that the binomial tree

B_m can be a spanning tree of an *m-cube*. This mapping can be found in [Athas & Seitz] although it is not presented as a mapping, but as part of a program code.

3.2 Mapping onto a 2-dimensional Square Mesh

The second mapping algorithm, Mapping M, maps B_{2m} to a $2^m \times 2^m$ mesh. The mapping algorithm is recursive and is intuitively described as "flip" $B_{2(m-1)}$ mapping *right* and then flip the result *down* to achieve the B_{2m} mapping". A more precise description follows. See Figure 6.

- The base case is the (trivial) mapping of B_0 onto a 1×1 mesh
- The mapping of B_{2m} for $m \geq 0$ is obtained in two-steps. First, the mapping of $B_{2(m-1)}$ to the $2^{m-1} \times 2^{m-1}$ mesh is reflected about a vertical axis to the right of the mesh, thus placing them in opposite halves of the $2^{m-1} \times 2^m$ mesh. The roots are then connected along the shortest (straight-line) path, achieving an intermediate mapping of B_{2m-1} to a $2^{m-1} \times 2^m$ mesh. Next, the procedure is repeated by reflecting the intermediate mapping about a horizontal axis below the intermediate mesh, yielding the desired B_{2m} mapping. The original $B_{2(m-1)}$ mapping thus remains in the upper left-hand corner of the mesh, the root of the intermediate mapping is in the upper right-hand corner and the root of B_{2m} is in the lower right-hand corner.

Algorithm M is describe more precisely below: Let the nodes in the computation graph G_C be labeled according to the canonical labeling scheme for the binomial tree. Nodes in the square mesh are labeled (x, y) where x is the row number and y is the column number, assuming the node in the upper lefthand corner is labeled $(1, 1)$.

Mapping M:

- For $G_C = B_0$ and $G_A = 1 \times 1$ mesh, $map - node_{B_0}(0) = (1, 1)$

- For $G_C = B_m$ and $G_A = h_m \times w_m$ *mesh*,

$$\begin{aligned} map - node_{B_m}(a) &= map - node_{B_{m-1}}(a') \text{ if } a = 0a' \\ map - node_{B_m}(a) &= (x, w_{m-1} - y + 1) \text{ if } a = 1a' \text{and m is odd} \\ &= (h_{m-1} - x + 1, y) \text{ if } a = 1a' \text{ and m is even} \end{aligned}$$

where

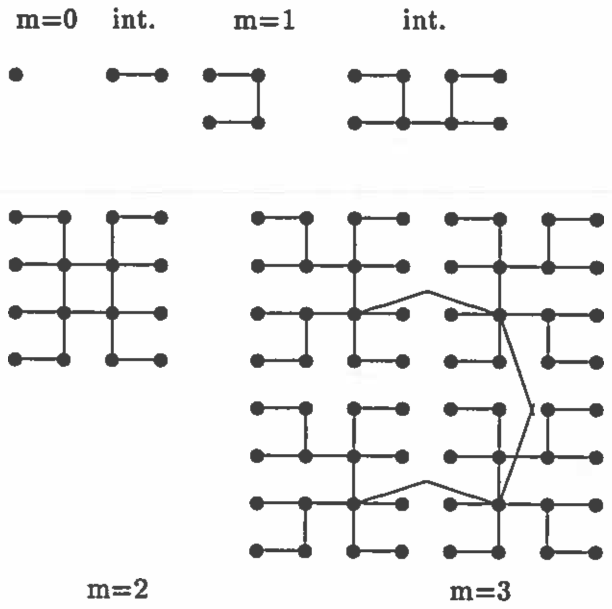


Figure 6: Mapping the Binomial Tree to the Mesh

$$(x, y) = map - node_{B_{m-1}}(a')$$

Also, for all edges $< a, b > \in V_C$, if

$$\begin{aligned} &\textit{map-node}_{B_m}(a) = (x1, y1), \text{ and} \\ &\textit{map-node}_{B_m}(b) = (x2, y2), \text{ and} \\ &\qquad x1 \geq x2, y1 \geq y2 \end{aligned}$$

then

$$\begin{aligned} \textit{map-edge}_{B_m}(< a, b >) &= \{(x2, y2), (x2, y2+1), (x2, y2+2), \\ &\quad \ldots, (x1, y1-1), (x1, y1)\} \text{ if x1=x2} \\ &= \{(x2, y2), (x2+1, y2), (x2+2, y2), \\ &\quad \ldots (x1-1, y1), (x1, y1)\} \text{ if y1=y2} \end{aligned}$$

4 Evaluation of Our Mapping Algorithms

In this section, we evaluate our mappings with respect to the cost of inter-processor communication (IPC). The overhead of IPC can be minimized by mapping processes that communicate as close to each other as possible and by avoiding contention on the links of the interconnection network. Our discussion addresses both mathematical metrics and the practical performance implications for message-passing technologies such as store-and-forward and wormhole routing.

4.1 Definition of Communication Metrics

Below we define and discuss the metrics.

Dilation: Dilation of an edge $< a, b > \in E_C$ is defined as the number of edges in the path P where *map-edge*$(< a, b >) = P\}$.

Ideally the dilation of every edge in V_C should be 1, but this is impossible to achieve in most cases. Two metrics that are commonly used to evaluate mappings in terms of IPC cost are *maximum dilation*[*] and *average dilation.*

[*] Some researchers call this simply the dilation of the mapping

Maximum Dilation: Maximum Dilation of a mapping of G_C to G_A is defined by $max\{d{:}d{=}dilation\ of\ < a, b >, < a, b > \in E_C\}\}$

This metric is limited since it does not measure a mapping by the dilation of all edges but by only the worst one. A more realistic metric is *average dilation.*

Average Dilation: Average Dilation of a mapping of G_C to G_A is defined by $\sum_{e \in E_C} dilation\ of\ e/|(E_C|)$

When a mapping has exactly one process per processor, the average dilation is at best 1.

A mathematical definition for *contention* would be cumbersome for our discussion, so we use the intuitive definition below.

Link Contention: Link contention occurs when two or more messages must be transmitted on the same link simultaneously.

4.2 Evaluating the Hypercube Mapping

Result 3: The *maximum dilation* and the *average dilation* for Mapping H is 1 and therefore optimal with respect to average dilation.

Result 4: Mapping H has no contention and is therefore optimal with respect to contention.

Proofs: The mapping onto the hypercube described in Section 4.1 maps every edge of the binomial tree onto a distinct edge of the hypercube. Thus the dilation of *every* edge is 1 and thus the *average dilation* is also 1 and there is no contention. □

4.3 Evaluating the Mesh Mapping

Result 5: The *maximum dilation* for Mapping M which maps B_{2m} onto the $2^m \times 2^m$ mesh is given by

$$\begin{aligned} Max(m) &= (2^m - 1)/3 \text{ if m is even} \\ &= (2^m + 1)/3 \text{ if m is odd} \end{aligned}$$

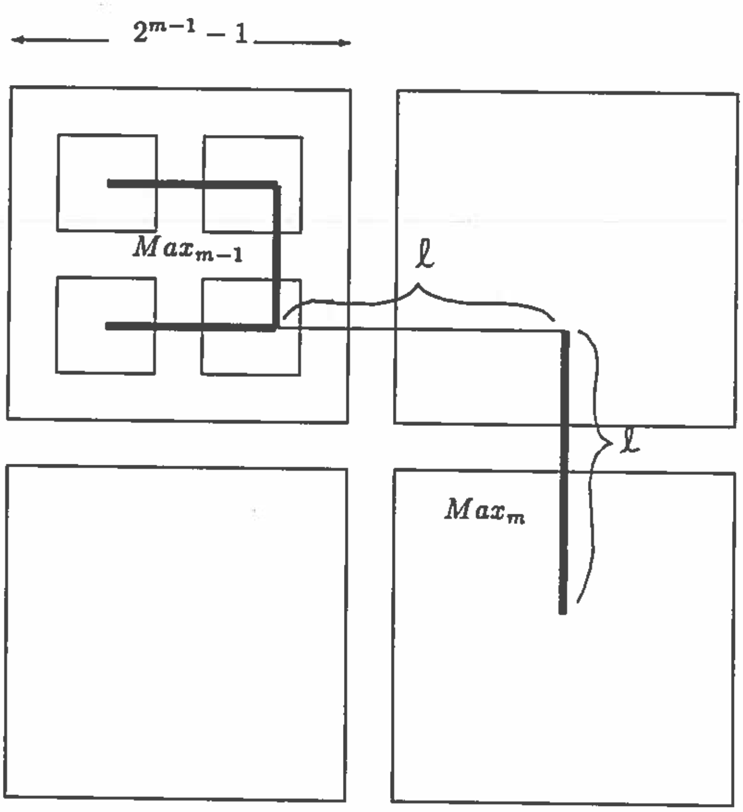


Figure 7: Mapping of B_{2m} *to a* $2^m \times 2^m$ *mesh*

Proof: From Figure 7 it is evident that the *maximum dilation* of the mapping of B_{2m} onto an $h_{2m} \times w_{2m}$ mesh is given by

$$\begin{aligned} Max(m) &= (2^{m-1} - 1) - Max(m-1) + 1 \qquad (1) \\ &= 2^{m-1} - Max(m-1) \qquad (2) \end{aligned}$$

The result is obtained by solving the recurrence equation (2) See Appendix A. □

Result 6: $Avg(m) \leq 1.2$

Proof: From Figure 7, it is also obvious that the *total dilation* of the mapping (which we will denote by *Tot(m)*) is given by

$$Tot(m) = 4Tot(m-1) + 3Max(m) \qquad (3)$$

Solving the recurrence equaton (3) (see Appendix B) we get

$$Tot(m) = (1.2)2^{2m} - (-1)^m 0.2 - 2^m$$

Thus the *average dilation* is given by

$$Avg(m) = ((1.2)2^{2m} - (-1)^m 0.2 - 2^m)/2^{2m} - 1$$

which can be shown to asymptotically approach 1.2. □

Result 7: Mapping M has zero contention and is therefore optimal with respect to contention.

Proof: The proof is by induction on the size of the binomial tree.

- The base cases are the cases $0 \leq k \leq 2$. In these binomial trees Mapping M assigns each edge of B_{2k} to a distinct link in the $2^k \times 2^k$ square mesh and therefore has zero contention.

- Assume that our induction hypothesis holds for binomial trees of size B_{2k}, $0 \leq k \leq (m-1)$. We must prove that there is no contention in Mapping M for B_{2m} to the $2^m \times 2^m$ square mesh. By the construction procedure for Mapping M, the mapping of B_{2m} consists of mapping four binomial trees $B^1_{2(m-1)}$ through $B^4_{2(m-1)}$ to disjoint quadrants in the $2^m \times 2^m$ mesh and connecting the roots of these four smaller trees along the shortest paths between them. Referring to the Gantt charts in Figure 4, we can see that the first message in the divide-and-conquer algorithm involves a single send from the root of B_{2m} (also the root of $B^1_{2(m-1)}$) to the root of $B^2_{2(m-1)}$. Because there is no other message-passing incurred at the same time, there is no contention involved. The second message in the divide-and-conquer algorithm involves simultaneous communication from $B^1_{2(m-1)}$ to $B^3_{2(m-1)}$ and from $B^2_{2(m-1)}$ to $B^4_{2(m-1)}$. However, it can be seen from Figure 6, that these messages are routed on disjoint paths in the mesh and also incur no contention. By definition of the binomial tree divide-and-conquer algorithm, all subsequent communication occurs within $B^1_{2(m-1)}$ through $B^4_{2(m-1)}$ until messages are passed up the tree. A similar argument holds for the merge stage message-passing. By induction, there is no contention within these subtrees. Since these four subtrees are mapped to separate quadrants of the mesh, there is no contention among the subtrees. □

Conjecture 8: Mapping M is optimal with respect to average dilation. The proof is currently under development.

4.4 Evaluating the Mapping with respect to Real Communication Performance

Whenever there is network dilation, message transit times become dependent on the routing algorithm as well as the channel bandwidth. There are two main methods of routing messages in a network: store-and-forward and wormhole. In store-and-forward routing, the message is copied (in its entirety) to each node (one at a time) along the path from source to destination. Wormhole routing is a pipelining technique where a portion of the message is sent one hop, then that portion is sent a further hop while another portion is sent where the first portion was, etc., until the whole message reaches the destination. With store-and-forward routing, transit time is proportional to the number of hops, whereas with wormhole routing, number of hops really doesn't matter (so long as the messages are big enough to allow for pipelining). The drawback to a wormhole routing scheme is that it requires the use of all the links along the entire path from source to destination to be free during the message transfer,* whereas store-and-forward routing only requires the use of one link at a time. In general, network dilation becomes an important factor when using store-and-forward routing, while contention is important for wormhole routing.

Our mapping minimizes average dilation (making it good for store-and-forward networks) while still having no contention (making it ideal for wormhole routing). The only drawback to our mapping is that in traditional divide-and-conquer algorithms, large amounts of data must be transferred during the first few phases. In our mapping, this corresponds to sending the most data over the links with the largest dilation (in the mesh). In this case, minimal average dilation (which our mapping provides) may not correspond to minimal execution time in a network that uses store-and-forward routing. However, for variants of divide-and-conquer where a constant amount of

*A slight modification to wormhole routing doesn't suffer from this problem.

information is passed at every step, or for networks with wormhole routing, our mapping gives superior performance.

5 Conclusions and Ongoing and Future Work

Since divide-and-conquer is a widely used paradigm, implementing it efficiently is important. Our contributions are listed below with a table summarizing our results:

- identification of the binomial tree as the ideal computation graph for divide-and-conquer.

	Comparison of Tree Structures for Divide and Conquer Algorithms	
	binomial tree	full binary tree
no. of nodes	$n/2$	$n-1$
efficiency	$\geq 2e$	e

- optimal mappings for the binomial tree to the hypercube and to the mesh.

	Summary of Mapping Performance	
	Mapping H	Mapping M
avg. dilation	1 (optimal)	$\leq$ 1.2 (optimal??)
contention	none (optimal)	none (optimal)
store-and-forward	excellent	good
wormhole	excellent	excellent

Our continuing work in this area focuses on other forms of divide-and-conquer algorithms: ones whose structure can be represented as an n-ary tree; dynamically evolving and potentially unbalanced trees; and non-tree structured divide-and-conquer algorithms. In addition, we would like to find efficient mappings of the binomial tree to other networks such as the *deBruijn* network and the *butterfly*. Finally, we are currently developing mapping algorithms for the common situation in which the communication volume on each edge in G_C is not necessarily uniform but some function of the depth of the sender node in the binomial tree.

Appendix

A Solving the Recurrence Equation for Max(m)

$$\begin{aligned}
Max(m) &= 2^{m-1} - Max(m-1) \\
&= 2^{m-1} - 2^{m-2} + \ldots - (-1)^i 2^{m-i} + (-1)^i Max(m-i) \\
&= 2^{m-1} - 2^{m-2} + \ldots - (-1)^{m-1} 2^1 + (-1)^{m-1} Max(1) \\
&= (2^{m-1} - 2^{m-2}) + (2^{m-3} - 2^{m-4}) \ldots - (-1)^{m-1} 2^1) + (-1)^{m-1} 2^0
\end{aligned}$$

Case 1. m is even

$$\begin{aligned}
Max(m) &= (2^{m-1} - 2^{m-2}) + \ldots + (2-1) \\
&= (2^m - 1)/3
\end{aligned}$$

Case 2. m is odd

$$\begin{aligned}
Max(m) &= (2^{m-1} - 2^{m-2}) + \ldots + (4-2) + 1 \\
&= 2((2^{m-1} - 1)/3) + 1 \\
&= (2^m + 1)/3
\end{aligned}$$

B Solving the Recurrence Equation for Tot(m)

Tot(m) = 4Tot(m-1) + 3Max(m)

Case 1: m is even

$$\begin{aligned}
Tot(m) &= 4Tot(m-1) + 2^m - 1 \\
&= 4[4Tot(m-2) + 2^{m-1} + 1] + 2^m - 1 \\
&= 16Tot(m-2) + 3(2^m + 1)
\end{aligned}$$

Case 2: m is odd

$$Tot(m) = 4Tot(m-1) + 2^m + 1$$

$$\begin{aligned} &= 4[4Tot(m-2) + 2^{m-1} - 1] + 2^m + 1 \\ &= 16Tot(m-2) + 3(2^m - 1) \end{aligned}$$

So, $\forall\ n \in N$,

$$\begin{aligned} Tot(2n) &= 16Tot(2n-2) + 3(2^{2n} + 1) && (4) \\ Tot(2n+1) &= 16Tot(2n-1) + 3(2^{2n+1} - 1) && (5) \\ Tot(2n-1) &= 16Tot(2n-3) + 3(2^{2n-1} - 1) && (6) \end{aligned}$$

Thus we have from (4) and (5)

$$Tot(2n+1) + Tot(2n) = 16Tot(2n-1) + 16Tot(2n-2) + 9 \cdot 2^{2n}$$

If m = 2n+1 (i.e., m odd),

$$Tot(m) + Tot(m-1) - 16Tot(m-2) - 16Tot(m-3) = 9 \cdot 2^{m-1} \qquad (7)$$

We also have from (4) and (6)

$$Tot(2n) + Tot(2n-1) = 16Tot(2n-2) + 16Tot(2n-3) + 9 \cdot 2^{2n-1}$$

If m = 2n (i.e., m even),

$$Tot(m) + Tot(m-1) - 16Tot(m-2) - 16Tot(m-3) = 9 \cdot 2^{m-1} \qquad (8)$$

Thus, from (7) and (8), $\forall\ m \in N$

$$Tot(m) + Tot(m-1) - 16Tot(m-2) - 16Tot(m-3) = 9 \cdot 2^{m-1} = (4.5)2^m$$

Thus the *characteristic equation* is

$$a^4 + a^3 - 16a^2 - 16a = 0 \qquad (9)$$

Solving (7), the characterictic roots are +4, -4, and -1.

So,

$$Tot(m) = A_1 4^m + A_2(-4)^m + A_3(-1)^m \tag{10}$$

The *Particular Solution*

The Particular Solution has the form $P2^m$. Substituting in (8), we have

$$\begin{aligned} P2^m + P2^{m-1} - 16P2^{m-2} - 16P2^{m-3} &= (4.5)2^m, \text{ or} \\ 8P + 4P - 32P - 16P &= 36, \text{ or} \\ P &= -1 \end{aligned}$$

Thus the *Particular Solution* is -2^m.

Combining the Particular Solution with (10) we have

$$Tot(m) = A_1 4^m + A_2(-4)^m + A_3(-1)^m - 2^m$$

So, using the values of *Tot(1), Tot(2) and Tot(3)* we have

$$\begin{aligned} 4A_1 - 4A_2 - A_3 - 2 &= 3 \quad (11) \\ 16A_1 + 16A_2 + A_3 - 4 &= 15 \quad (12) \\ 64A_1 - 64A_2 - A_3 - 8 &= 69 \quad (13) \end{aligned}$$

Solving (11), (12), and (13) we get

$$A_1 = 1.2,\ A_2 = 0,\ A_3 = -0.2$$

Thus $Tot(m) = (1.2)2^{2m} - (-1)^m 0.2 - 2^m$.

References

[Bok87] S. H. Bokhari. *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publishers, 1987.

[BS87] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, 4(5):439–458, October 1987.

[FF82] J. P. Fishburn and R. A. Finkel. Quotient networks. *IEEE Transactions on Computers*, C-31(4):288–295, April 1982.

[Knu73] D. L. Knuth. *The Fundamental Algorithms.* Addison-Wesley, 1973.

[Lo88] V. M. Lo. Algorithms for static task assignment and symmetric contraction in distributed computing systems. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 239–244, August 1988.

[LRG+] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. Moataz, and J. Telle. Oregami: Software tools for mapping parallel algorithms to parallel architectures. submitted to ACM PPOPP 1990.

[Sto77] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.