International Journal of Parallel Programming, Vol. 16, No. 3, 1987

Deriving Parallel Computations from Functional Specifications: A Seismic Example on a Hypercube

Roger L. Wainwright[1]

Received July 1987; Revised December 1987

Three algorithms developed for a seismic model illustrate that, when the target hardware has many processing elements, functional programs can exhibit better performance than programs developed with conventional techniques. This stands in contrast to the widely held belief that functional programs necessarily pay in poor performance for their advantages in conciseness and likelihood of correctness. Two of the algorithms evolved from an analysis of the seismic model with the goal of finding parts of the computation that could proceed in parallel. The first algorithm has a low communication to computation ratio. The second and third algorithms have higher ratios trading communication time for computation time. The third algorithm was derived from a presentation of the input/output relationship of the model expressed as a composition of mathematical functions. This algorithm exhibited substantially better performance than either of the others. The algorithm achieves its good performance by setting up a producer-consumer pipeline between simultaneously operating portions of the computation. This pipeline balances computation and interprocessor communication more effectively than the other two algorithms.

KEY WORDS: Functional specification; data flow; load balancing; parallelism.

1. INTRODUCTION

Most existing parallel architectures are loosely classified into two groups: (1) shared memory architecture such as the Alliant, Encore, BBN Butterfly, and Cray-XMP where a common memory is shared by each processor and (2) nonshared memory architectures such as hypercubes, tree machines,

[1] Computer Science Department, The University of Tulsa, Tulsa, Oklahoma 74104.

0885-7458/87/0600-0243$05.00/0 © 1987 Plenum Publishing Corporation

and connection machines where each processor has its own local memory and data is passed from one processor to another by some sort of interconnection scheme. Hypercubes are multiprocessor arrays[1-4] with elaborate interconnection features connecting the nodes together. In general, a hypercube parallel computer is a collection of sequential computing nodes joined to their nearest neighbors in an n-dimensional cube. The nodes are completely independent machines with their own processor memory, I/O, and resident copy of the operating system. The Amoco hypercube system, where this work was supported, is currently an NCUBE/ten system[2] with 64 nodes each with half a million bytes of local memory. Each processor is capable of register to register operations at the rate of half a million floating point or two million fixed point operations per second. Section 2 of this paper discusses some fundamental issues concerning program development on a hypercube. Sections 3 and 4 describe the seismic model and the characteristics of the algorithm for this model. In Section 5, three algorithm implementations are presented for this model. Section 6 compares the performance of the three algorithms and in Section 7 summary and conclusions are presented.

2. HYPERCUBE PERFORMANCE ISSUES

The two most important considerations affecting the overall parallel performance of a hypercube are the computational speed of the nodes and the communication speed between the nodes. The ratio between these two quantities is the determining factor on how to implement a good parallel algorithm on a hypercube. A single precision floating point multiply can be performed on a node in our system at a rate of 0.094 megaflops. This was determined empirically by performing a single multiply operation within a loop, then subtracting out the loop overhead. If multiplications occur between array elements then the additional array reference overhead reduces the effective rate to 0.075 megaflops. Performing more complicated arithmetic expressions, such as 15 to 20 multiplication operations in one assignment statement, resulted in better utilization of register to register operations. However, this only increased the effective rate by a factor of two.

Vasicek and Beguelin[5] determined that the node interprocessor communication rate is given by the following formula where message-time is given in clicks (146.28 microseconds per click):

$$\begin{aligned}\text{Message-time} = 1.70 &+ (\text{number-of-hops} * 1.53) \\ &+ (\text{number-of-bytes} * 0.0128) \\ &+ (\text{number-of-hops} * \text{number-of-bytes} * 0.00876)\end{aligned} \tag{1}$$

It turns out that communication costs compared to computational costs are balanced for long messages and dominated by communication costs for short messages. For example, to send 40,000 bytes a distance of one hop takes approximately 0.127 seconds using this formula. Performing 10,000 single precision floating point array operations which corresponds to 40,000 bytes at a rate of 0.075 megaflops takes approximately 0.133 seconds. The communication time is approximately the same in this case. Sending only 40 bytes of data, however, takes approximately 6.0×10^{-4} seconds compared to computational time of 1.33×10^{-4} seconds for 10 floating point operations. In this case the communication time is approximately 4.5 times slower. When passing data among nodes it is generally more efficient to avoid passing extremely small amounts of data.

Consider the following simple problem illustrating this point. Suppose a single node must multiply two 50,000 element vectors together to form a single vector. Also suppose an adjacent node (distance one away) is available to do some of the calculations. How do you partition the work? One way is to have the first node do all of the work and not use the adjacent node at all. In this case the calculations take approximately 0.67 seconds. An extreme solution is to send all 100,000 values to the adjacent node to calculate the products and return 50,000 values. This takes the same 0.67 seconds to perform the calculations plus an additional 1.89 seconds for communication time for a total of 2.56 seconds. An optimal solution is somewhere between these extremes. Suppose X bytes are sent to the adjacent node such that both nodes are kept busy for the same amount of time performing the calculations. Communication time must also be taken into consideration. Using formula (1) and 0.075 megaflops, the value for X can be estimated *a priori* using formula (2) given as:

$$(50000 - X/8)\text{ flops} * \frac{1\text{ second}}{0.075 * 10^6\text{ flops}} \tag{2a}$$

$$= (1.7 + 1.53 + 0.022\,X)\text{ clicks} * \frac{(146.28 * 10^{-6}\text{ second})}{1\text{ click}} \tag{2b}$$

$$+ X/8\text{ flops} * \frac{1\text{ second}}{0.075 * 10^6\text{ flops}} \tag{2c}$$

$$+ (1.7 + 1.53 + 0.022 * X/2)\text{ clicks} * \frac{(146.28 * 10^{-6}\text{ second})}{1\text{ click}} \tag{2d}$$

In this formula, (2a) represents the time in seconds for the original node to perform its numerical computations (8 bytes represent two operands and

thus one arithmetic operation), (2b) represents the time to communicate X bytes to the adjacent node, (2c) represents the time for the adjacent node to perform its numerical computations and (2d) represents the time to communicate $X/2$ bytes back to the original node. In this case the optimal value for X is approximately 80,000 bytes. This means that approximately 10,000 of the 50,000 pairs of values to be multiplied should be sent to the adjacent node yielding a total time of 0.53 seconds. This represents a speedup of 1.26 using two nodes versus one node. This is a simple example, but it serves to point out the importance of considering both computational and communication speeds when developing algorithms to run on a hypercube.

3. DESCRIPTION OF THE SEISMIC MODEL

This paper reports my experiences on implementing several algorithms for the seismic model shown in Fig. 1. In this model, seismic data has been collected as a function of time over several seismic lines. Each line is a collection of numerous seismic traces, where each trace begins on the surface of the earth and proceeds to depth d in the model. The desired results of the model is a seismic image. The size of the problem depends on the number of seismic lines and traces to analyze. The input parameters, intermediate calculations and final results for this model are described by:

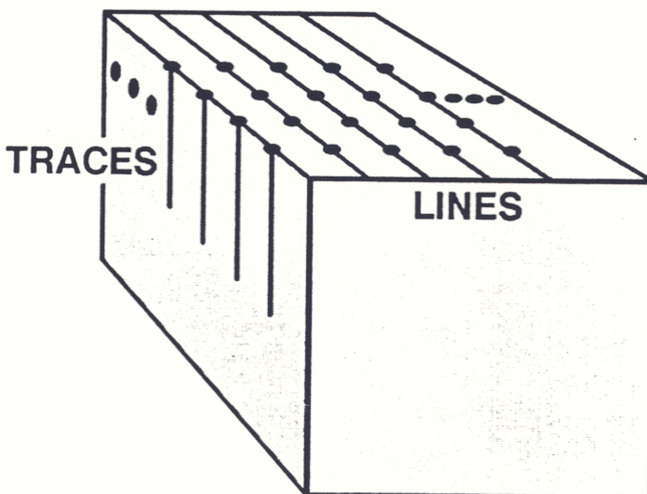

Fig. 1. Seismic model.

Input Parameters:

- $U(t)$ is a two dimensional matrix of seismic data collected in the field as a function of time, t.
- $G(v, d)$ represents the geophysical parameters of the model (velocity and depth profile).

Intermediate Calculations:

- $T(z)$ is a one dimensional vector of traversal times calculated for $z = 1..d$, where d is the depth of the model.

Final Results:

- $B(z)$ is a two dimensional matrix of the resultant seismic image for $z = 1..d$.

The functional dependencies between U, G, T, and B are shown in formula 3. The same information is given in Fig. 2 in the form of a data flow diagram.

$$
\begin{aligned}
&T(1) = f(G) \\
&T(z) = g(T(z-1), z, G) \qquad \text{for} \quad z = 2..d \\
&B(z) = h(U, T(z), z) \qquad \text{for} \quad z = 1..d
\end{aligned}
\tag{3}
$$

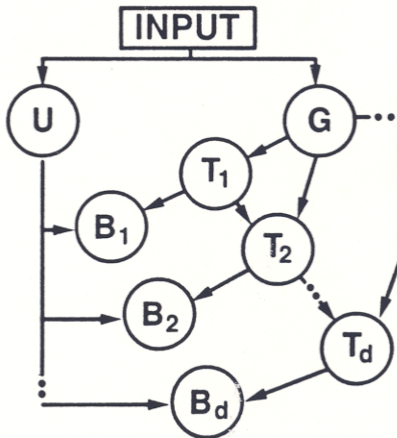

Fig. 2. Data flow diagram.

4. CHARACTERISTICS OF THE ALGORITHM

Figure 3 depicts the sequential algorithm for this model. The characteristics of the algorithm are listed here. These are the same characteristics presented in formula 3 and Fig. 2.

1. *BLOCK A* represents a fixed amount of one time calculations for the input and preprocessing of U and G.
2. *BLOCKS T* and *B* are executed d times in a loop which does not depend on the size of the problem to be solved.
3. *BLOCK T* is not easily divided into parallel partitions. The amount of calculations performed in this block is fixed and not dependent on the size of the problem.
4. *BLOCK B* is easily partitioned into separate parallel parts which can be performed in parallel on several nodes. The amount of work performed in *BLOCK B* increases directly as the size of the problem increases.
5. *BLOCK B* depends on the calculations performed in *BLOCK T* during each cycle in the loop.
6. *BLOCK T* does not depend on the calculations performed in *BLOCK B*, but does depend on calculations from *BLOCK T* in the previous cycle.

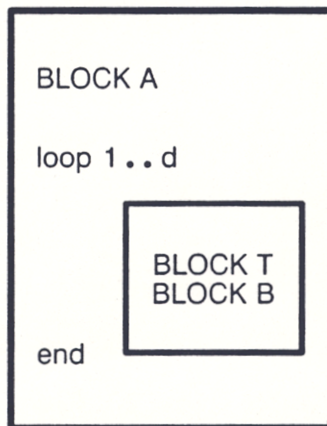

Fig. 3. Sequential algorithm in block form.

5. ALGORITHM IMPLEMENTATIONS

Three different parallel strategies were developed and implemented for the generic algorithm previously stated. The strategy for each of the algorithms is later described and shown in Figs. 4–6. The algorithms are depicted in general terms ignoring any topological properties of the hypercube.

Algorithm I was designed to minimize the interprocessor communication between nodes. In this algorithm a single dedicated node (node zero) performs the one time *BLOCK A* calculations, determines how to partition the workload for *BLOCK B* evenly among the available nodes and passes this information one time to each working node, then quits. Nodes one through n (called working nodes) work independently of each other. Each working node redundantly calculates every *T BLOCK* and for each *T BLOCK* calculates only its part of the associated *BLOCK B* sending the results back for collection. As expected, this algorithm has a low communication to computation ratio. This algorithm is also called the "*X-Y* parallel local *T* algorithm." Every node calculates *BLOCK T* locally. The calculations for *BLOCK B*, which are performed horizontally over the *x-y* plane for a given depth, are done in parallel. See Figs. 1 and 4.

Algorithm II is a classical producer/consumer algorithm. In this algorithm, node zero performs the one time *BLOCK A* calculations, partitions the workload for each *BLOCK B* evenly among the available nodes and passes this information one time to each working node. This is exactly

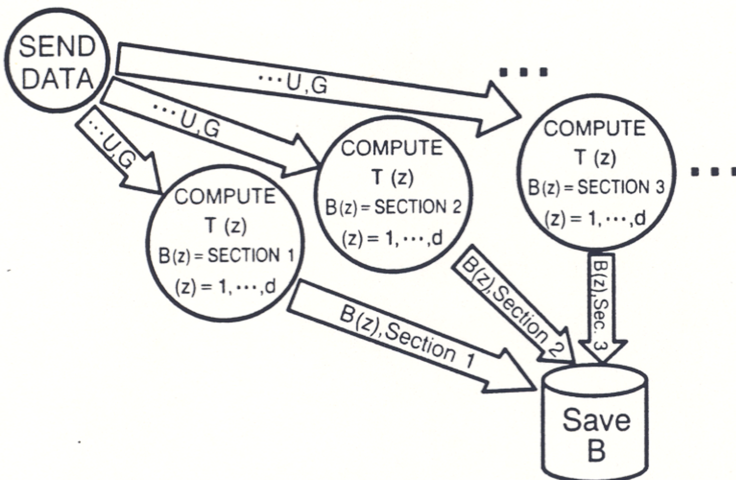

Fig. 4. Algorithm I: (*X-Y*) parallel with local *T*.

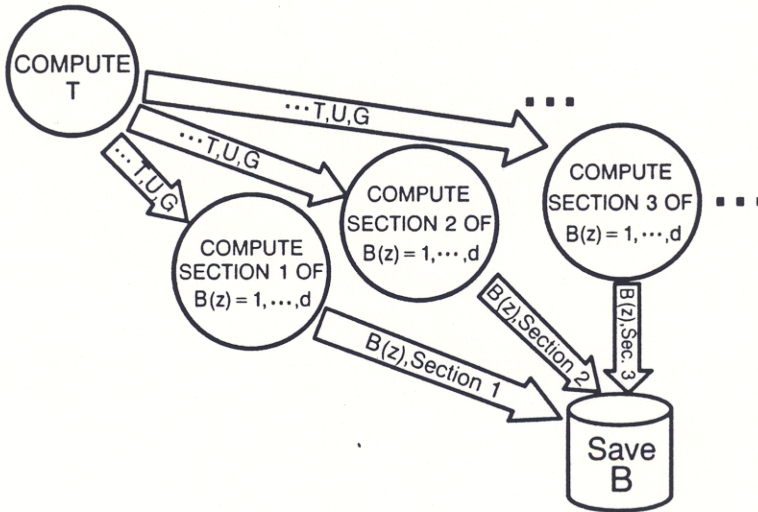

Fig. 5. Algorithm II: (X-Y) parallel with global T.

the same as Algorithm I. However, unlike Algorithm I, node zero continues to operate. It performs the *BLOCK T* calculations, sending the results to every working node. The rate of the production of the *BLOCK T* results by node zero is independent of the consumption of the *BLOCK T* data by the working nodes. Thus in this algorithm all of the *BLOCK T* calculations are

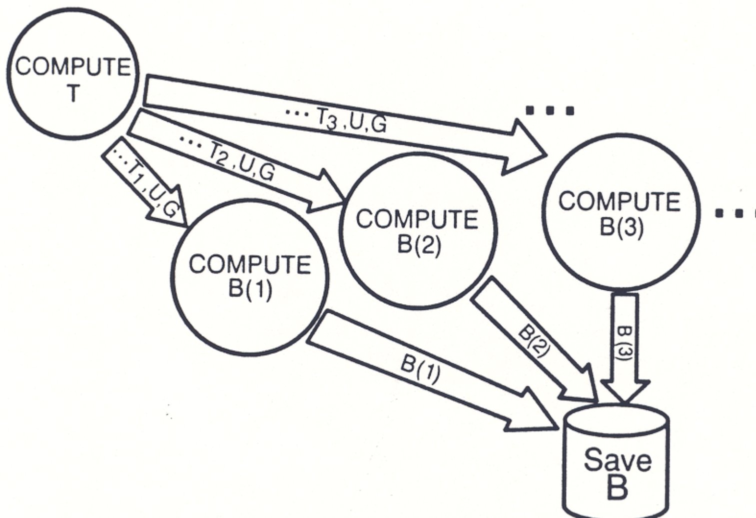

Fig. 6. Algorithm III: Z parallel with global T.

performed in one node and broadcasted to each of the working nodes rather than each working node calculating *BLOCK T* itself. Notice that each working node still receives all of the *BLOCK T* calculations. This algorithm reduces the total amount of calculations required in each working node in comparison to Algorithm I; however, it increases the interprocessor communication between nodes. In Algorithm II, nodes one through n work independently of each other consuming *BLOCK T* data, performing only their portion of *BLOCK B* calculations, and sending results back for collection. Algorithm II requires a careful balance between the production and consumption of the *BLOCK T* data to optimize performance. This algorithm is also called the "*X-Y* parallel global *T* algorithm." *BLOCK T* is calculated by one node and the *BLOCK B* calculations, which are performed horizontally over the x-y plane for a given depth, are done in parallel. See Figs. 1 and 5.

Algorithm III is a minor but very effective variation of Algorithm II. Node zero performs the one time *BLOCK A* calculations as always and all of the *BLOCK T* calculations are performed by node zero as well. This is the same as Algorithm II. However, in Algorithm III unlike the previous algorithms, *BLOCK B* work is *not* partitioned among the working nodes. Instead each working node receives only a selected few *BLOCK T* results (not every *BLOCK T* as in the previous algorithms) and calculates *all* of the associated *BLOCK B*, not a fraction of the *BLOCK B* work as in the previous algorithms. For example, if there are n working nodes, then node i, $i = 1,\ldots n$, will receive *BLOCK* T_j and calculate all of *BLOCK* B_j for $j = i + mn$, where $m = 0, 1, 2,\ldots$ such that $j \leqslant d$. This algorithm like Algorithm II is a producer/consumer algorithm requiring a delicate balance between the production and the consumption of *BLOCK T* data. This algorithm is also called the "*Z*-parallel global *T* algorithm." *BLOCK T* is calculated by one node and the parallel calculations of *BLOCK B* are performed vertically across the planes in the *Z* direction. It would seem that after a certain point when all working nodes are keeping up with the production of the *T BLOCK*s from node zero that adding additional working nodes would be ineffective. This is indeed the case. Algorithm III is the direct result of executing the problem as stated functionally in Formula 3 and depicted in Fig. 2. Compare the similarity between the flow of data shown in Fig. 2 and Algorithm III depicted in Fig. 6. In fact I did not consider this solution until I wrote the generic problem in functional form. In this case depicting the problem in its functional form yielded significant insight into the interdependencies within the problem and suggested an efficient algorithm for solving the problem. It is interesting that a graph reducing parallel evaluator[6,7] could automatically derive this computation from a functional program specifying the algorithm.

6. ALGORITHM COMPARISONS

All of the algorithms were implemented using from 2 to 64 nodes for various problem sizes ranging from 8 to 360 units of work. Results are reported for the case of 360 units of work which corresponds to a medium to large size problem. In terms of the physical model shown in Fig. 1 this corresponds to 360 traces over several lines. Due to memory limitations, 360 units of work is the largest possible problem size without using external storage. Since 60 nodes evenly divides 360 units of work, using all 64 nodes resulted in insignificant improvement over using 60 nodes. Therefore, all tables report results for a maximum of 60 working nodes (61 nodes total including node zero). Tables I, II, and III report the results of Algorithms I, II, and III respectively.

All three algorithms require a minimum of two nodes; node zero and at least one working node. Each entry in Tables I, II, and III counts node zero as one of the nodes. In order to have a basis for comparison, a separate algorithm was developed to run on a single node. Results from this algorithm are recorded in each Table in the first entry. Notice that each algorithm exhibits poor performance while using a small number of nodes. This is because node zero is an "overhead" node. For example, each algorithm is approximately 50% efficient using two nodes (one working node and node zero). When using two nodes each algorithm is basically a sequential operation. This is the reason why each algorithm initially exhibits a sharp decline in efficiency and then a rapid rise in efficiency until the overhead of node zero is compensated for (see Fig. 8).

Table I. Algorithm I: (*X*-*Y*) Parallel with Local *T*[a]

Number of nodes	% Time spent in *BLOCK T*	Total time	Speed up	Cost	Efficiency
1	1.60	11326.80	1.00	11326.8	1.00
2	1.60	11331.97	0.99	22663.9	0.49
3	3.15	5762.60	1.96	17287.8	0.65
4	4.05	4488.53	2.52	17954.1	0.63
5	5.36	3384.47	3.34	16922.4	0.67
6	6.56	2768.35	4.09	16610.1	0.68
11	12.28	1479.01	7.66	16269.1	0.69
21	21.69	837.10	13.53	17579.1	0.64
31	29.03	625.48	18.11	19389.9	0.58
41	34.82	521.43	21.72	21378.6	0.53
61	43.14	420.88	26.91	25673.7	0.44

[a] Problem size = 360. All times are in seconds.

Table II. Algorithm II: (X-Y) Parallel with Global T[a]

Number of nodes	Wait BLOCK T	Send BLOCK T	Total time	Speed up	Cost	Efficiency
1	—	—	11326.80	1.00	11326.8	1.00
2	0.0	10741.0	11155.52	1.01	22311.1	0.51
3	0.0	5367.0	5579.81	2.03	16739.4	0.67
5	0.0	3002.0	3204.04	3.53	16020.2	0.71
7	0.0	1962.0	2158.84	5.25	15111.9	0.75
11	169.0*	1106.0	1300.62	8.71	14306.8	0.79
16	121.3*	680.0	874.23	12.95	13987.7	0.81
21	98.6*	472.0	667.13	16.98	14009.7	0.81
31	410.1*	374.0	570.85	19.84	17696.4	0.64
33	256.3*	396.0	593.56	18.08	19587.5	0.58
37	310.8*	447.0	645.56	17.54	23885.7	0.47
61	720.3*	751.0	957.69	11.83	58419.1	0.19

[a] Problem size = 360. All times are in seconds.

Figures 7 and 8 compare the performance of all three algorithms. Figure 7 compares the speedup of the algorithms versus the number of nodes used. Figure 8 compares the efficiency of each algorithm versus the number of nodes. The data used to produce these figures came from Tables I, II, and III with additional sample points.

Table III. Algorithm III: Z-Parallel with Global T[a]

Number of nodes	Wait BLOCK T	Send BLOCK T	Total time	Speed up	Cost	Efficiency
1	—	—	11326.80	1.00	11326.8	1.00
2	0.0	10762.0	11143.84	1.02	22287.7	0.51
3	0.0	5198.0	5584.37	2.03	16753.1	0.67
5	0.0	2440.0	2794.97	4.05	13974.8	0.81
7	0.0	1530.0	1830.11	6.19	12810.7	0.88
11	0.0	802.0	1136.62	9.96	12502.8	0.90
16	0.0	447.0	769.77	14.71	12316.3	0.92
21	0.0	290.0	584.89	19.36	12282.9	0.92
31	0.0	120.0	398.08	28.45	12340.5	0.91
37	0.0	82.0	340.41	33.27	12595.2	0.90
41	10.0	62.0	315.32	35.92	12928.1	0.87
52	12.0	23.0	255.97	44.25	13310.4	0.85
61	12.0	7.0	239.71	47.25	14622.3	0.77

[a] Problem size = 360. All times are in seconds.

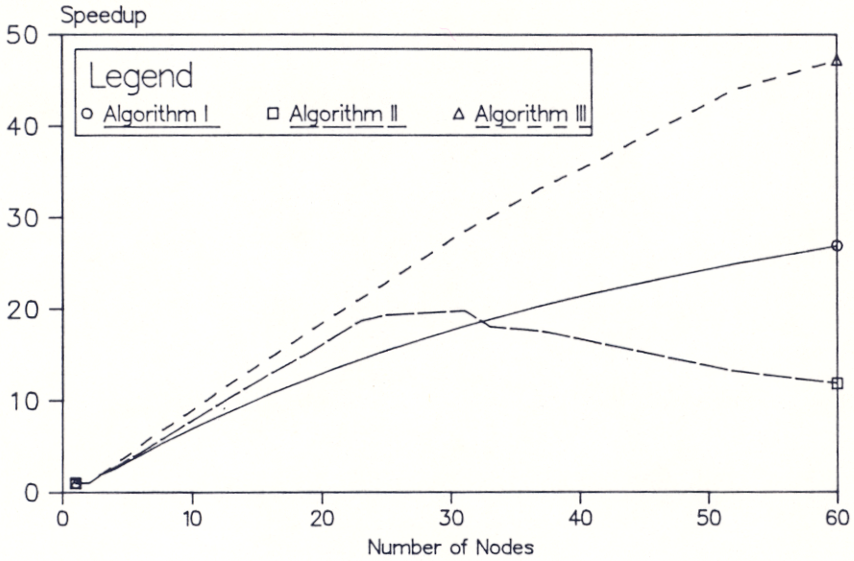

Fig. 7. Speedup versus number of nodes.

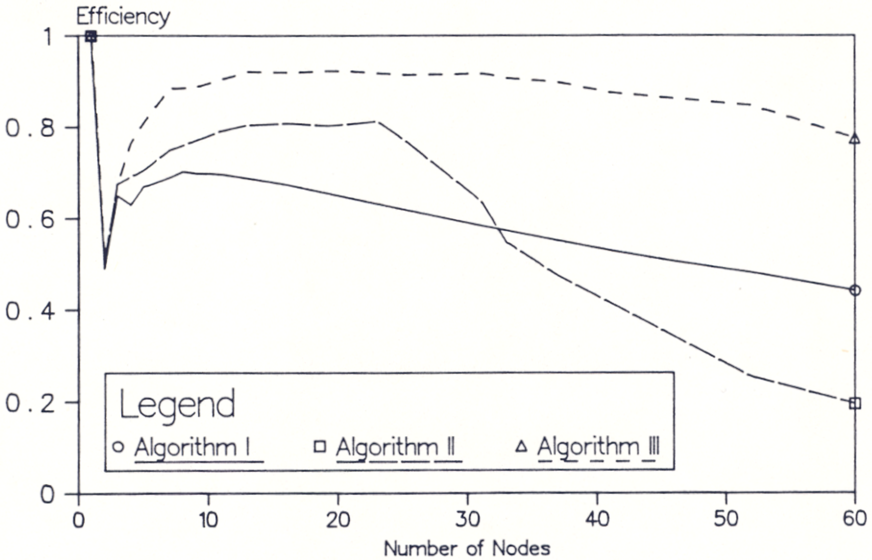

Fig. 8. Efficiency versus number of nodes.

Figures 7 and 8 and Table I show some of the properties of Algorithm I. Some observations concerning Algorithm I include:

1. Adding more nodes reduces the execution time and increases speedup (see Figs. 7 and 8).
2. Increasing the number of nodes reduces the amount of work each node performs on *BLOCK B* thus increasing the percent of time spent in *BLOCK T*. This greatly reduces the efficiency. In Table I using 61 nodes, each working node performs 6 units of work, (360 units of work/60 working nodes). The algorithm spent only 43% of its time in *BLOCK T* for a speed up of almost 27 and an efficiency of 0.44.
3. The larger the problem size the better Algorithm I performs. The work performed in *BLOCK T* is independent of the problem size. However, the work in *BLOCK B* increases the percentage of the work that can be parallelized (*BLOCK B*) increases.

In Algorithm II node zero generates all of the *BLOCK T* data sending the results to each of the working nodes independent of the progress of the working nodes. If the work load between node zero producing *BLOCK T* data and the consumption of *BLOCK T* data by the working nodes is out of balance then some interesting results occur (see Table II):

1. When the Wait *BLOCK T* values are zero node zero is producing *BLOCK T* data faster than the working nodes can consume it. In other words there was no wait time for *BLOCK T* data when nodes 1 through n required it.
2. Notice the amount of time required by node zero to send *BLOCK T*. The amount of time to calculate *BLOCK T* is fixed. The communication time for the data varies with number of nodes used. The results of *BLOCK T* calculations are always available (no wait time) in the case of seven nodes or less. This means node zero is producing *BLOCK T* data faster than it is being consumed. In the case of two nodes (one working node), the sending time is over 10,000 seconds, but when three nodes are used (two working nodes), it decreases to a little over 5,000. With additional nodes it becomes even less. This is because the buffer size for queued messages is fixed. If the limit is reached then the sending node (node zero in this instance) is temporarily suspended (blocked) until some of the messages are consumed. Hence nearly all of the 10,000 seconds to send *BLOCK T* data is due to wait time caused by the message buffer limit and not actual communication time. As additional nodes are added to the problem the production and

consumption become balanced where the buffer limit is never reached. Hence after a point this delay does not occur. When more than seven nodes are used there exists some wait time for the data. The * entries for wait times for *BLOCK T* in Table II indicate varying wait times (all positive) among the working nodes. An average value is recorded.

3. In Table II a balance is obtained between the sending and waiting of *BLOCK T* results. This occurs at 31 nodes. See Figs. 7 and 8 and note the change that occurs to Algorithm II at 31 nodes. When the number of nodes used is below the balance point the delay in sending *BLOCK T* data is primarily due to buffer limits, and when the number of nodes used is above the balance point then the delay in sending *BLOCK T* data is primarily due to the increased number of nodes to which data must be sent.
4. The balance point for the sending and receiving of *BLOCK T* data is also the point where best (minimal) time was achieved. Unlike Algorithm I, Algorithm II actually gets worse after a certain point when additional nodes are added to do the work! See Figs. 7 and 8 and note total time recorded in Table II. The larger the problem, the higher the balance point, however. When I ran Algorithm II for problem size of four the balance point occurred using one working node. In this case the use of additional nodes made the problem worse. At the other end of the spectrum a problem of size 1000 will have a balance point beyond the available 64 nodes that we have.

In Algorithm III node zero generates all of the *BLOCK T* data, sending the results to each of the working nodes independent of the progress of the working nodes. This is identical to Algorithm II except each working node performs all of the associated *BLOCK B* work for each *BLOCK T* that it receives. The work load balance between node zero producing *BLOCK T* data and the consumption of *BLOCK T* data by the working nodes is a critical issue in this algorithm compared to the others. When a node is sending a message to another node in the *NCUBE* system and the input message buffers of the receiving node are full, then the program in the sending node is temporarily halted or blocked until buffer space is available for the message to be sent. When the receiving node is able to receive more messages, then the sending node is awakened and execution is resumed. I mentioned this earlier as the reason for the unusually long communication times for sending *BLOCK T* data in Table II for Algorithm II.

Unfortunately, there is a side effect to the blocking of a node in the

NCUBE hypercube system which can be critical: while a node is suspended it will not allow any message traffic to go through it! This is a deficiency in the hypercube system from our point of view. In the case of Algorithm III, node zero is sending *BLOCK T* data and also receiving traffic (*BLOCK B* results) from the working nodes on the way back to the cube manager for collection. In the case of small problem sizes this did not cause any problems. However, for problem size 360 a deadlock occurred in every instance regardless of the number of nodes used.

There are at least two possible solutions in order to get Algorithm III to work for large problem sizes. First, allow node zero to initially send a fixed number of *BLOCK T* data messages to each working node and then send additional data only upon request from each working node. This will ensure that the buffer limit will never be exceeded and thus prevent a possible deadlock. Second, rather than controlling the buffer limit which requires additional message handling and coordination, do not allow any traffic to go through node zero. In our *NCUBE* system four of the 64 nodes have a direct connection link to the cube manager. Node zero is one of these connections, and as a result many of the messages from other working nodes go through node zero to the cube manager. The second solution is the one that was implemented. Because of the physical arrangement of the hypercube it is not possible to actually remove node zero from the burden of handling traffic from other nodes to the cube manager. By the convention established for message traffic in the hypercube system, node 63 will never handle any messages from other nodes to the cube manager. As a result of this simple observation the program normally executed on node zero was moved to node 63, for example for a cube of size 6. This solved the hypercube deadlock problems in Algorithm III.

Other noteworthy points concerning Algorithm III (see Table III) include:

1. When the Wait *BLOCK T* values are zero in Table III "node zero" is producing *BLOCK T* data faster than the working nodes can consume it. That is, there was no wait time for *BLOCK T* data when nodes 1 through *n* required it. However, buffer limits are exceeded as discussed earlier and, as a result, large send *BLOCK T* times are recorded. This is the same as Algorithm II.

2. A balance is obtained between the sending and waiting of *BLOCK T* results. This occurs at 61 nodes. When the number of nodes used is below the balance point the delay in sending *BLOCK T* data is primarily due to buffer limits, the same as in Algorithm II. However, when the number of nodes used is above the balance point, then the delay in sending *BLOCK T* data levels

off to a constant value unlike Algorithm II. This is because *BLOCK T* data goes to only one node, not to all working nodes. Burton and Huntbach[8] suggest several things to consider when deciding when to move some pending process to another processor in order to improve the balance in workloads and to avoid communication bottlenecks. Keller and Lin[9] discuss in detail the factors influencing the choice of granularity of a problem. Trade-offs include communication overhead and the flexibility of dynamic load balancing. A future topic of research is to consider the benfits of dynamic load balancing in this model.

3. The balance point for the sending and receiving of *BLOCK T* data is also the point where the CPU time begins to level off. That is, after a certain point adding additional working nodes will not alter the execution time. After the balance point has been reached where the working nodes are keeping up with the production of *BLOCK T* data, adding additional nodes will simply ensure idle time for some nodes. As expected, the larger the problem the higher the balance point. Unlike Algorithm II, Algorithm III does not get worse with the addition of more working nodes with respect to CPU time. The efficiency goes down, however.

7. CONCLUSIONS

Three different algorithms arising from a seismic model used in oil exploration were implemented on a hypercube system (64 nodes). The difference between the three algorithms is the way in which parallelism is realized in the seismic model (Fig. 1). Algorithms I and II parallelized across the x-y plane horizontally. In Algorithm I there is a redundancy of computation to avoid communication costs and in Algorithm II there is additional communication, but it is overlapped with computation. Algorithm III however, parallelizes vertically along the Z axis. Algorithm III was derived from the functional specification of the problem. Notice the similiarity between Figs. 2 and 6. There are other ways to partition the seismic model in order to obtain different algorithms. I have considered only these three at this time.

Algorithm III, derived from the functional specification of the problem, exhibited faster execution times regardless of the number of nodes used or the size of the problem. In general, adding additional working nodes to the problem in Algorithm I decreased the overall execution time. This leveled off after a certain point. Adding working nodes in Algorithm II decreased execution time up to a point after which additional nodes

increased execution time. Adding working nodes to Algorithm III decreased execution time up to a point, after which additional nodes had no effect. The speedup for Algorithms I, II, and III are respectively 26.91, 11.83, and 47.25 using 61 nodes for a problem size of 360 traces over several seismic lines. (Algorithm II exhibited best speedup, 19.84, using 31 nodes). The efficiency of Algorithms I, II, and III are respectively 44%, 19%, and 77% when using 61 nodes. In each algorithm the efficiency was approximately 50% using two nodes. This is because the process of using node zero with only one working node is basically a sequential process. Using additional working nodes increases the efficiency rapidly up to a certain point before it begins to decrease again (see Fig. 8). The efficiency of Algorithm III drops off very slowly as the number of nodes increases. The efficiency did not drop below 90% until more than 40 nodes were used. The other algorithms are significantly more inefficient.

Each of the algorithms can be slightly improved. First, several nodes could be used for a tree structure fan-out of the data messages sent from node zero to the working nodes. This could reduce some of the communication delays. Secondly, the partitioning of the workload of *BLOCK B* in Algorithms I and II could be divided unevenly among the working nodes. It was observed in problems with a large number of working nodes that nodes farther from node zero finished their work after nodes closer to node zero. This difference was sometimes as much as 20% of the total execution time. This was expected since messages are delayed depending on the distance a receiving node is from the sending node. An improvement from 10 to 20% is expected if nodes closer to node zero are given slightly more work than nodes further away. Of course, the first suggested improvement will solve some of this.

Comparing Algorithm I versus Algorithm II in Figs. 7 and 8, there is a point where the lines cross. Thus, under certain conditions Algorithm I performs better than Algorithm II in execution time, speedup and efficiency, and under certain conditions Algorithm II performs better. However, my findings show Algorithm III, which was derived from the functional form of the problem, to be the best algorithm. This algorithm was not apparent until the functional specification was developed. It was derived by hand, but could have been derived automatically through graph reduction processing of the functional program. This algorithm exhibited superior performance in every instance in execution time, speedup and efficiency. Furthermore, this was independent of the problem size and the number of nodes used.

ACKNOWLEDGMENTS

The work reported in this paper was supported by Amoco Research Center, Tulsa, Oklahoma. I am grateful to my colleagues Tony Cox and Dan Vasicek for their helpful comments. I am also grateful to Rex Page for his thorough readings of the manuscript, which resulted in numerous improvements and clarifications. My thanks to the referees, especially referee E, for their constructive comments. This contributed greatly to the final form and content of the paper.

REFERENCES

1. S. Lakshivarahan and S. K. Dhall, A New Hierarchy of Hypercube Interconnection Scheme for Parallel Computers: Theory and Practice Research Report OU-PPI-TR-86-02, Parallel Processing Institute University of Oklahoma, Norman, Oklahoma (August 1986).
2. NCUBE Corporation, *NCUBE System Manual* (1985).
3. W. J. Ouchark, J. A. Davis, S. Lakshivarahan, and S. K. Dhall, Experiences with Intel Hypercube, Proceedings of the Workshop on Applied Computing, Oklahoma State University, Stillwater, Oklahoma, October 10–11 (1986).
4. C. L. Seitz, The Cosmic Cube, *Commun. ACM* **28**(1):22–33.
5. D. J. Vasicek and A. Beguelin, Communication Between Nodes of a Hypercube, Second Conference on Hypercube Multiprocessors, Knoxville, Tennessee (September 1986).
6. D. H. Grit and R. L. Page, Deleting Irrelevant Tasks in an Expression-Oriented Multiprocessor System, *ACM TOPLAS* **3**(1):49–59 (January 1981).
7. D. H. Grit and R. L. Page, A Multiprocessor Computer System for Parallel Evaluation of Applicative Programs, *J. Digital Systems*, Vol. 4, No. 2 (1980).
8. F. W. Burton and M. M. Huntbach, Virtual Tree Machines, *IEEE Trans. Comput.* **C-33**(3):278–280 (March 1984).
9. R. M. Keller and F. C. H. Lin, Simulated Performance of a Reduced-Based Multiprocessor, *IEEE Computer*, pp. 70–82 (July 1984).