A Balanced Layered Allocation Scheme for Hypercube Based Dataflow Systems

Vincent R. Freytag and Ben Lee

Department of Electrical and Computer Engineering
Oregon State University

A. R. Hurson

Department of Electrical and Computer Engineering
The Pennsylvania State University

Abstract

The dataflow model of computation allows the exploitation of fine-grain parallelism inherent to programs. However, one of the major issues yet to be resolved is the proper allocation of programs to processors. The problem lies in maximizing parallelism while minimizing interprocessor communication costs. This paper proposes a method called the Balanced Layered Allocation Scheme (BLAS) which utilizes heuristic rules to find a balance between computation and communication costs in hypercube based dataflow systems. Simulation studies indicate that the BLAS is effective in reducing communication overhead as well as overall execution times of dataflow programs.

1. Introduction

Since the early 1970s there has been much research in the field of dataflow computing [2, 3, 13, 17]. Through these research efforts, several parallel computers based on the dataflow concept have been developed [3, 4, 8, 15, 18, 21]. In contrast to the control-flow model of computation, the dataflow model of computation does not utilize a program counter (PC) to sequence instructions. Instead, the execution of instructions is based solely on the *availability* of their operands. An instruction will execute if and only if all of its operands are available. It is through this property that dataflow operations are asynchronous.

A dataflow program is represented as a directed graph $G \equiv G(N,A)$, where the nodes, N, represent instructions or functions (*i.e.*, macro-actors) and the arcs, A, represent the data dependencies between nodes. Operand *values* are passed along the arcs in the form of data packets, called *tokens*. A node *fires* (executes) when it has received all the necessary input tokens to perform its operation. When a node fires, the input tokens are consumed and one or more output tokens are generated. Instructions executed on dataflow computers do not have any set sequencing constraints except the data dependencies implicit in the program itself (i.e., self-scheduling). Since the data dependencies are encoded in a dataflow graph, explicit synchronization is not needed. Thus, a dataflow graph will exploit all possible parallelisms of a given program.

Although the dataflow model of computation offers many attractive properties for parallel processing, there are still several problems that need to be addressed [2, 3, 11]. One of these concerns is the allocation of programs to processors while maintaining control of interprocessor communication [2, 15]. Therefore, this paper addresses the issue of program allocation for hypercube based dataflow systems.

The organization of this paper is as follows: Section 2 discusses dataflow multiprocessors and the allocation problem. Section 3 proposes the Balanced Layered Allocation Scheme (BLAS). Section 4 analyzes the effectiveness of the BLAS through simulation studies. Finally, Section 5 provides a brief summary and concluding remarks.

2. Dataflow multiprocessors and the allocation problem

In recent years a number of improvements have been made to dataflow architectures. One of the most important developments to emerge is the simplified process of matching tokens called *direct matching* [8, 15, 16]. In a direct matching scheme, storage is dynamically allocated in an *activation frame* for all the tokens generated in a code-block. A token is composed of a pointer to an instruction (IP), a pointer to an activation frame (FP), and a value. The pair of pointers <FP.IP> comprise the token *continuation* (or *tag*). A presence bit associated with a frame location indicates if a token is present. When a new token arrives at a node, the presence bit is checked to determine if a token is present. If the presence bit is set, then the value held in the frame location is extracted and the instruction is executed. After firing the instruction, one or more new tokens are generated and the frame slot is returned to its original

0-8186-3200-3/92 $03.00 © 1992 IEEE

empty state. Thus, direct matching provides a simple and elegant method for matching tags by eliminating the need for expensive and sophisticated matching hardware.

Another recent architectural change is the emergence of a dataflow/von Neumann hybrid. In this approach, the sequential execution efficiency of von Neumann computing is combined with the fine-grain parallelism of pure dataflow computing. There are currently two major approaches used to incorporate control-flow sequencing in the dataflow model of computation —*multithreading* [12, 16] and *macro-actors* [8, 18, 21].

The first technique of incorporating control-flow sequencing into the dataflow model of computation is implemented using a simple *recirculation* scheduling paradigm. Recall that a computation is completely described by the pair of pointers <FP.IP>, where IP represents a pointer to the current instruction. Therefore, the successive instruction can be described by <FP.IP+1>. In addition to this simple manipulation of continuations, the hardware must also support the immediate re-insertion of tokens into the execution pipeline. This is achieved by bypassing the Token Queue using a Direct Recirculation Path [15].

One potential problem with the recirculation method is that successive tokens are not generated until the last stage of the pipeline [15]. Therefore, the execution of the next instruction in the computational thread will experience a delay that is equal to the number of stages in the pipeline. On the other hand, the recirculation method allows up to m independent threads to be interleaved in an m-stage pipeline. This method of thread interleaving is known as multithreading and is implemented in the Monsoon [15].

The second technique of implementing sequential scheduling is the macro-actor concept. In this scheme, instructions are grouped into larger grains where instructions within a grain can be scheduled in a control-flow fashion and the grains themselves are scheduled in a dataflow fashion. For example, the EM-4 dataflow multiprocessor implements macro-actors based on the *strongly connected arc model*. In this model, arcs of a dataflow graph are categorized into normal arcs or strongly connected arcs [18, 21]. Nodes that are connected by strongly connected arcs are grouped together in a macro-actor and executed by a single processing element.

Our proposed scheme assumes that the underlying architecture incorporates the macro-actor concept since performance is not limited by the number of independent computational threads in an application program. Although the proposed algorithm is applicable to any processor organization, we demonstrate our algorithm on a hypercube processor organization. We have selected the hypercube topology to discuss our algorithm due to their powerful interconnection features and wide use [17].

A k-dimensional hypercube is a network of 2^k nodes having addresses between 0 and $2^k - 1$. Nodes are adjacent if their addresses differ in only one bit position. Each node in a k-dimensional hypercube has k adjacent nodes, connected by an *edge*, and the number of *hops* (maximum distance) between any two nodes is k. It is easy to show that the number of edges in a hypercube with n nodes is $(n/2)\log_2 n$. A four-dimensional hypercube with sixteen nodes is illustrated in Figure 1.

Despite the architectural differences in multiprocessor systems, it is apparent that they all share a common goal in the allocation of programs: maximizing the inherent concurrency of a program while minimizing the contention for processing resources. Yet, if the execution times of instructions vary, or if the number of processors is larger than two, the problem of optimally allocating a program to processors is NP-complete [13, 17]. The allocation problem is further complicated by the fact that communication costs exist between instructions assigned to different processors. Therefore, heuristics are generally used to solve the allocation problem [13, 19, 20]. One such approach is the Vertically Layered (VL) allocation scheme proposed by Lee *et al.* for Multistage Interconnection Network (MIN) based dataflow systems [13].

The VL allocation scheme is a compile-time method based on two underlying philosophies: (1) assign concurrently executable instructions to separate PEs and (2) assign data dependent instructions to the same PE [13]. Thus, the goal of these two philosophies is to minimize execution times and communication costs. The VL allocation scheme is implemented with a separation phase and an optimization phase. During the separation phase, the dataflow graph representation of a program is separated into vertical layers which are then assigned to processors. The vertical layers are determined by first identifying the critical path (CP) of the program graph. The CP is then assigned to the most central processor.

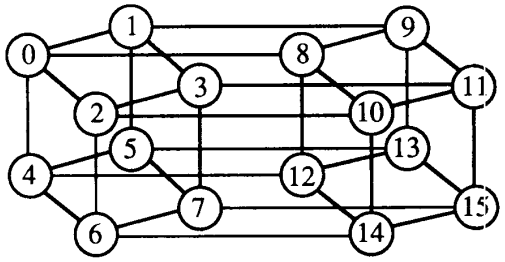

Figure 1. A four-dimensional hypercube.

Thus, the CP is given the highest priority for processor assignment. The remaining vertical layers are found by recursively determining the *Longest Directed Path* (LDP) from nodes that have already been allocated by using the same approach as when the CP was found. These LDPs are then assigned to processors based on the density of allocated nodes (*i.e.*, load balancing). Note that the separation phase is carried out without considering communication costs. Once the relative assignment of the nodes to processors is known, the optimization phase attempts to minimize inter-PE communication behaviors.

Performance studies indicate that the VL allocation scheme is very effective in reducing the communication overhead; however, one shortcoming of the VL allocation scheme is its poor performance when the number of available PEs is much less than the maximum parallelism of the dataflow graph [13]. A reason for the poor performance in such a case is that the VL allocation scheme neglects the effects of communication delays when LDPs are assigned to layers. Furthermore, the VL allocation scheme assumes constant interprocessor communication delays among all pairs of PEs limiting its application to a small class of interconnection networks. In light of the above discussion, we extend the work done by Lee *et al.* and propose an alternative method for allocating programs on hypercube based dataflow computers.[1]

3. The proposed allocation scheme

The proposed allocation scheme is based on three general objectives: (1) assign concurrently executable nodes to separate processing elements, (2) assign data dependent nodes to the same processing element, and (3) assign nodes to PEs that lead to an earlier completion time of the program. Objective (1) encourages the exploitation of parallelism on dataflow processors, while objective (2) minimizes communication costs. However, these are two conflicting objectives and therefore objective (3) is used to provide a compromise by considering the effects of node execution times and interprocessor communication costs. In doing so, local communication and parallelism are encouraged while poor allocations that yield large communication costs and long program execution times are discouraged.

The proposed heuristic algorithm is based on two parameters: execution times and communication costs. For a directed graph G, an execution time, t_i, is associated with all $n_i \in N$. It is assumed that these execution times are known in advance at compile-time. In addition, a communication delay, C_{ij}, exists between adjacent processing elements PE_i and PE_j. These communication costs are a function of the target architecture. If α denotes the time required to initiate a packet, and if β denotes the time required to send the packet, then the total time required to send a single token to adjacent processors is $\alpha+\beta$. Thus, communication between adjacent processors, PE_i and PE_j, requires a constant time $C_{ij} = \alpha+\beta$. It is assumed that the communication delays among all adjacent processors, C_{ij}, in a hypercube based system are identical. It is also assumed that communication between non-adjacent processors in a hypercube requires an integral multiple of C_{ij}.

The proposed allocation scheme utilizes Critical Path (CP) and Longest Directed Path (LDP) heuristics to initially separate the dataflow graph representation of a program into modules. Ideally, the CP dictates the minimum execution time of a program and thus the CP is given the highest priority for processor assignment. All other program modules in G are found recursively by determining the LDP emanating from the nodes that have already been assigned to processing elements. Communication costs and execution times are then considered when modules are assigned to PEs. Program modules are allocated to processors only after iteratively considering these parameters on all PEs. We now present the Balanced Layered Allocation Scheme (BLAS) to demonstrate how a program can be distributed into separate program modules for hypercube based dataflow systems.

For the implementation of the BLAS, only acyclic dataflow graphs are considered.[2] Consider an arbitrary directed acyclic dataflow graph $G \equiv G(N,A)$ where N represents the set of instructions and A represents the dependencies between these instructions. Thus, a directed path between node n_i and node n_j implies that n_i precedes n_j (*i.e.*, $n_i \rightarrow n_j$). An example of a directed acyclic graph is shown in Figure 2 where each node, n_i, has an associated execution time, t_i. The proposed algorithm requires a dataflow graph to have a root node and a single exit node. If a root node or an exit node does not exist, a *dummy* root node or exit node is introduced with negligible execution times and negligible communication delays between PEs.

A dataflow graph is partitioned into program modules based on execution times and communication delays. Each program module, containing a set of serially connected nodes, is rearranged into separate layers such that each layer has a one-to-one correspondence with a

[1] Although our discussion is based on hypercubes, the application of the proposed scheme is by no means limited only to hypercube based systems.

[2] To facilitate the handling of cyclic graphs see [13].

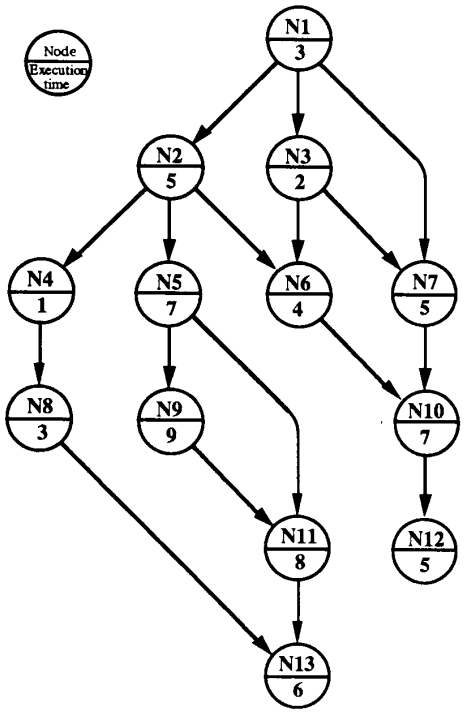

Figure 2. A directed acyclic dataflow graph.

processor in the hypercube. Thus, a k-dimensional hypercube with 2^k different processors has 2^k unique layers numbered 0, 1, ..., $2^k - 1$.

The allocation process starts by locating the critical path, CP, of a directed acyclic graph G. Since the critical path defines the longest path from the root node to the exit node, assignment of the critical path to a single layer minimizes interprocessor communication associated with the critical nodes. When determining the critical path, only the expected execution times, t_i, are considered. If a unique critical path does not exist, the algorithm arbitrarily chooses one.

Using the method outlined above, the critical path of the dataflow graph in Figure 2 is $N1 \rightarrow N2 \rightarrow N5 \rightarrow N9 \rightarrow N11 \rightarrow N13$. The set of nodes defining the critical path, N_{CP}, is assigned to an arbitrary layer. The nodes that comprise N_{CP} are then marked and queued into a First-In-First-Out (FIFO) queue Q while maintaining their precedence constraints. All remaining nodes are selected iteratively for allocation as follows: Let N^{S-1}_{marked} represent the set of nodes assigned to a layer at step S-1. Initially, $N^0_{marked} = N_{CP}$. A node, n_j, is removed from Q and the set of nodes N^S_{LDP} comprising the longest directed path emanating from n_j is formed such that:

$$N^{S-1}_{marked} \cap N^S_{LDP} = \{\varnothing\} \text{ and}$$
$$N^S_{marked} = N^{S-1}_{marked} \cup N^S_{LDP}.$$

The method used in finding the longest directed path emanating from n_j involves the same method as finding the critical path without considering the marked nodes.

To determine the placement of the remaining program modules, each set of nodes N^S_{LDP} is assigned in an iterative fashion to every possible layer. In this manner, the effects of execution times and communication costs can be weighed against the different layer assignments for N^S_{LDP}. Thus, after weighing the effects of N^S_{LDP} in each layer, the set of nodes comprising N^S_{LDP} is assigned to the layer favoring the objectives of the BLAS. For the iterative assignment of N^S_{LDP} to each layer L, the algorithm considers two properties:

(1) Let T^L_i represent the *completion time of layer i* when N^S_{LDP} is assigned to layer L. Then, T^L is the set of T^L_i, *i.e.*,

$$T^L = \left\{ T^L_i \mid i = 0, 1, \ldots, 2^k - 1 \right\}.$$

(2) Let T^L_{graph} represent the *completion time of graph G*, *i.e.*,

$$T^L_{graph} = \max\left\{ T^L \mid L = 0, 1, \ldots, 2^k - 1 \right\}.$$

Each set of nodes N^S_{LDP} is then assigned to the layer L that yields the *lowest completion time*, T_{min}, where

$$T_{min} = \min\left\{ T^L_{graph} \mid L = 0, 1, \ldots 2^k - 1 \right\}.$$

If T_{min} yields more than one minimum, then a layer is chosen arbitrarily. The set of nodes N^S_{LDP} is then marked and queued into Q while maintaining precedence constraints. The allocation phase is completed when the queue Q is empty.

To illustrate our algorithm, consider the allocation of the dataflow graph in Figure 2 to a two-dimensional hypercube processor organization as shown in Figure 3. For illustrative purposes, we arbitrarily assume that communication costs between adjacent layers are twice the average execution time of an instruction, *i.e.*, $C_{ij} = C_{ji} = 10$ units of time since the average execution time of the nodes in the graph of Figure 2 is 5 units of time. Thus, in our two dimensional hypercube, $C_{01} = 10$, $C_{02} = 10$, $C_{13} = 10$, $C_{23} = 10$, $C_{03} = 20$, and $C_{12} = 20$. The set of nodes comprising the critical path N_{CP}={N1, N2, N5, N9, N11, N13} is arbitrarily assigned to layer 2

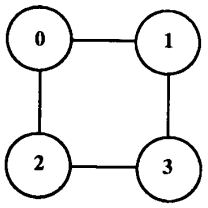

Figure 3. A two-dimensional hypercube organization.

as illustrated in Figure 4. After the arbitrary assignment of N_{CP} to layer 2, the nodes comprising N_{CP} are then marked and queued. Node N1 is then removed from Q and the set of nodes N^1_{LDP}={N3, N7, N10, N12} is formed. The set of nodes N^1_{LDP} is then iteratively assigned to every possible layer to determine which assignment yields the lowest completion time. For example, assigning the set of nodes N^1_{LDP} to layer 0 yields:

$$T^0_0 = 32,\ T^0_1 = 0,\ T^0_2 = 38,\ T^0_3 = 0,$$
$$T^0 = \{32,\ 0,\ 38,\ 0\},\ \text{and}\ T^0_{graph} = 38.$$

Assigning the set of nodes N^1_{LDP} to layer 1 yields:

$$T^1_0 = 0,\ T^1_1 = 42,\ T^1_2 = 38,\ T^1_3 = 0,$$
$$T^1 = \{0,\ 42,\ 38,\ 0\},\ \text{and}\ T^1_{graph} = 42.$$

Assigning the set of nodes N^1_{LDP} to layer 2 yields:

$$T^2_0 = 0,\ T^2_1 = 0,\ T^2_2 = 57,\ T^2_3 = 0,$$
$$T^2 = \{0,\ 0,\ 57,\ 0\},\ \text{and}\ T^2_{graph} = 57.$$

Assigning the set of nodes N^1_{LDP} to layer 3 yields:

$$T^3_0 = 0,\ T^3_1 = 0,\ T^3_2 = 38,\ T^3_3 = 32,$$
$$T^3 = \{0,\ 0,\ 38,\ 32\},\ \text{and}\ T^3_{graph} = 38.$$

According to these results, N^1_{LDP} is assigned to the layer that yields the lowest completion time, *i.e.*,

$$T_{\min} = \min\left\{T^0_{graph} = 38, T^1_{graph} = 42, T^2_{graph} = 57, T^3_{graph} = 38\right\}.$$

Therefore, N^1_{LDP} can be assigned to either layer 0 or layer 3. Layer 3 is chosen arbitrarily as illustrated in Figure 5. After all the nodes of Figure 2 have been allocated, a layered graph similar to Figure 6 is generated. In summary, we present the algorithm for the Balanced Layered Allocation Scheme.

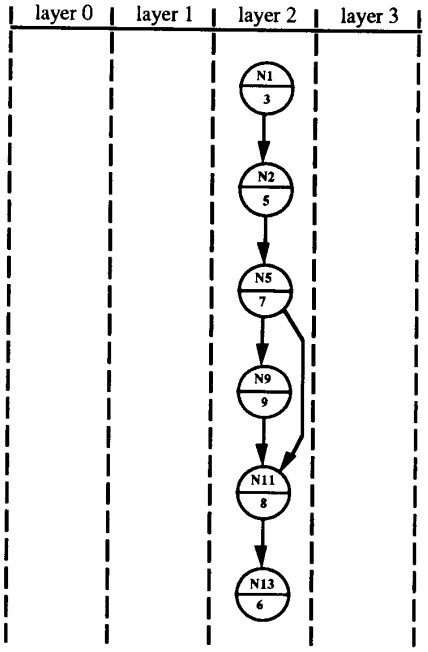

Figure 4. Arbitrary layer assignment of the critical path.

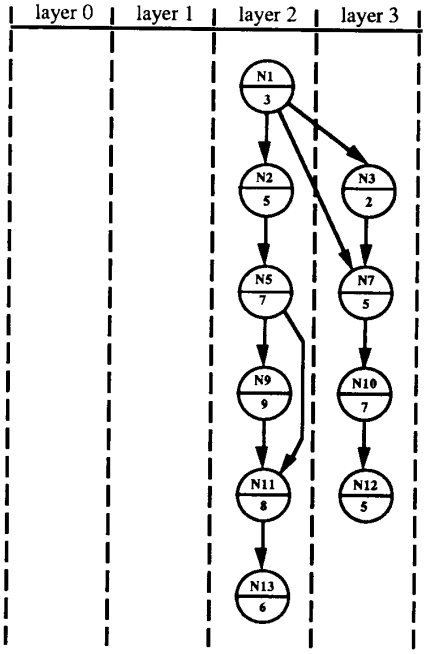

Figure 5. The state of the layers after the assignment of N^1_{LDP}={N3, N7, N10, N12}.

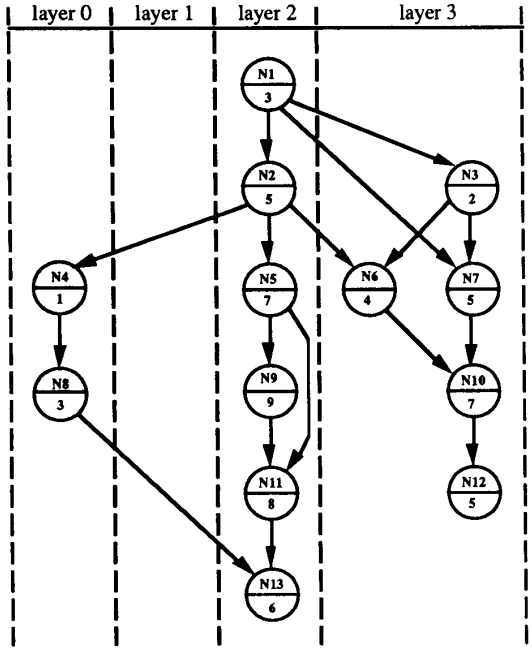

Figure 6. The balanced layered graph of Figure 2.

Algorithm1: Balanced Layered Allocation Scheme

Input: An acyclic dataflow graph $G(N,A)$, where all $n_i \in N$ have an expected execution time t_i.

Output: A set of 2^k layers having a one-to-one correspondence with processors in a k-dimensional hypercube.

Allocation

- Determine the set of nodes belonging to the critical path and assign these nodes to an arbitrary layer.
- Queue the set of nodes comprising the critical path into a FIFO queue Q while maintaining precedence constraints.

WHILE (Q is not empty) DO
- CALL ALLOCATE

END

PROCEDURE ALLOCATE
BEGIN
- Remove the node n_i from the front of the queue Q.

FOR (All the arcs emanating from n_i) DO
- Determine the longest directed path N^S_{LDP} emanating from n_i such that:

$$N^{S-1}_{marked} \cap N^S_{LDP} = \{\varnothing\} \text{ and}$$
$$N^S_{marked} = N^{S-1}_{marked} \cup N^S_{LDP}.$$

ENDFOR

- Insert the set of nodes N^S_{LDP} into the queue Q.

FOR L=0 TO $2^k - 1$ DO

FOR i=0 TO $2^k - 1$ DO
- Find the completion time of layer i, T^L_i, if N^S_{LDP} is assigned to layer L.

ENDFOR
- Let T^L represent the set of T^L_i, *i.e.*,

$$T^L = \left\{ T^L_i \mid i = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

- Find the completion time of graph G, T^L_{graph}, *i.e.*,

$$T^L_{graph} = \max\left\{ T^L \mid L = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

ENDFOR
- Assign the set of nodes N^S_{LDP} to the layer that yields the lowest completion time, T_{min}, where

$$T_{min} = \min\left\{ T^L_{graph} \mid L = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

- Queue the set of nodes N^S_{LDP} into the queue Q while maintaining precedence constraints.
- Mark each node in N^S_{LDP} to indicate that these nodes have been allocated.

END

4. Simulation results

To analyze the effectiveness of the BLAS, six dataflow graphs were chosen for our simulation studies. The first dataflow graph, GRAPH1, is the example used throughout this paper. The second dataflow graph entitled EX1 is a 12-node graph utilized in [13] to illustrate the VL allocation scheme. The final four dataflow graphs were also obtained from [13]. These graphs consist of: (1) a 16-node dataflow graph entitled QUICKSORT for the implementation of a quicksort algorithm, (2) a 32-node dataflow graph entitled NWP32 representing a numerical weather prediction program, (3) an 82-node graph entitled 82V that performs assignment and sequencing operations, and (4) a 146-node graph entitled NWP147 which is a more complex version of the numerical weather prediction problem.

Our simulation studies are based on the following assumptions:

(1) The underlying dataflow architecture is a hypercube.
(2) The execution time t_i of each node is assigned randomly from a uniform distribution with an average of five time units.
(3) The inter-PE communication delays are varied based on a ratio of *communication to execution time, C/E.*
(4) The inter-PE communication delays associated with non-adjacent processors is an integral multiple of the fundamental communication delay C_{ij}.
(5) The intra-PE communication delay is negligible when compared to the inter-PE communication delay.

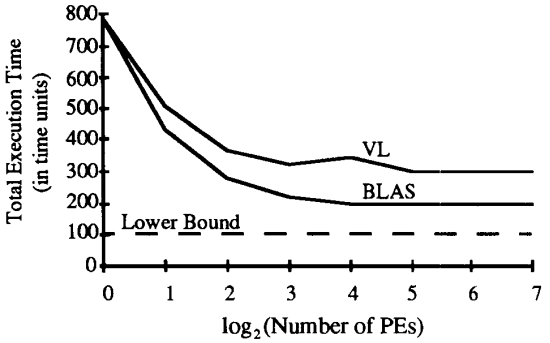

Figure 7. A plot of total execution times versus $\log_2$(Number of PEs) for C/E = 10/5.

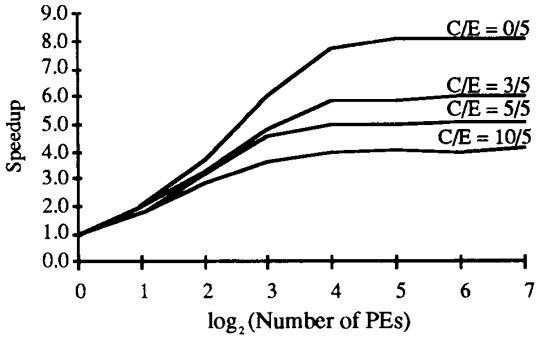

Figure 8. A plot of speedup for varying C/E ratios.

To illustrate the results of the simulation studies, NWP147 was chosen as a representative dataflow graph. Figure 7 depicts the total execution time versus the number of PEs when the *C/E* ratio is 10/5. As expected, the total execution time for the BLAS decreases as the number of PEs increase. The lower bound in Figure 7 represents the critical path of NWP147 with negligible inter-PE communication delays.

Also included in Figure 7 are the execution times of NWP147 when it is allocated using the VL allocation scheme. For this simulation, the VL allocation scheme was modified for hypercube processor organizations. This comparison was made to see how well the BLAS faired against the VL allocation scheme which has shown promising results [13]. As shown in Figure 7, the BLAS yields lower execution times than the VL allocation scheme. The *speedup* of NWP147 as a function of *C/E* ratios is illustrated in Figure 8. When the *C/E* ratio increases, the speedup decreases as would be expected. In general, the BLAS is very successful in minimizing communication overhead in the initial allocation of nodes to layers. Therefore, simulation results have indicated that an optimization phase similar to the one used in the VL allocation scheme is not necessary for the BLAS.

Table I presents the overall performance improvements of the BLAS relative to the VL allocation scheme. When inter-PE communication delays are negligible the BLAS performs equally as well as the VL allocation scheme. Additionally, the BLAS always performs at least as well as the VL allocation scheme and, in general, the performance improvement of the BLAS increases as the *C/E* ratio increases. More importantly, the trend for large dataflow graphs is a monotonic increase in the performance improvement for increasing *C/E* ratios.

5. Summary and conclusion

The problem of allocating dataflow graphs to a hypercube based dataflow computer has been discussed. The concept of the Balanced Layered Allocation Scheme, which utilizes CP and LDP heuristics to handle the allocation problem, was introduced. The central idea of this method is to arrange the nodes of a dataflow graph into layers that have a one-to-one correspondence with processors in a hypercube. During the allocation, CP and LDP heuristics determine the set of nodes which are to be assigned to processors. Each set of nodes is assigned in an iterative fashion to every possible layer. In this manner the effects of execution times and communication

Table I. Performance improvement of the proposed algorithm relative to the VL allocation scheme.

C/E	GRAPH1	EX1	QUICKSORT	NWP32	82V	NWP147
0/5	0.0%	0.0%	0.0%	0.0%	0.0%	2.0%
1/5	3.1%	4.2%	4.0%	14.0%	0.6%	2.3%
3/5	4.4%	30.2%	0.0%	17.0%	2.5%	22.4%
5/5	15.9%	15.0%	29.5%	16.1%	11.0%	29.6%
10/5	4.1%	1.9%	16.1%	48.5%	26.4%	34.2%
15/5	0.0%	25.0%	4.5%	62.8%	53.8%	44.9%

costs can be weighed against each other for every possible layer assignment. Sets of nodes are then assigned to the layer that yields the earliest completion time of the program.

Simulation studies indicate that the proposed allocation scheme is effective in reducing communication overhead and thus the overall execution time of a program distributed on a hypercube dataflow computer. Overall, the BLAS showed promising improvements over the VL allocation scheme. In addition, the proposed algorithm need not be restricted to the allocation of dataflow graphs to hypercube based dataflow multiprocessors. Since the method of assigning nodes to processors considers execution times as well as communication delays, the Balanced Layered Allocation Scheme is general enough to be applied to any multiprocessor system.

References

[1] Ackerman, W. B., "Data Flow Languages," *Computer*, Feb. 1982, pg. 15-25.

[2] Arvind and Agerwala, T., "Data Flow Systems," *Computer*, Feb. 1982, pg. 10-13.

[3] Arvind and Culler, D. E., "Dataflow Architectures," *Annual Review of Computer Science*, Vol. 1, 1986, pg. 225-253.

[4] Arvind and Nikhil, R. S., "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Transactions on Computers*, Vol. 39, No. 3, March 1990, pg. 300-318.

[5] Buehrer, R. and Ekanadham, K., "Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution," *IEEE Transactions on Computers*, Vol. C-36, No. 12, December 1987, pg. 1515-1522.

[6] Dinning, A., "A Survey of Synchronization Methods for Parallel Computers," *Computer*, July 1989, pg. 66-77.

[7] Dubois, M., Scheurich, C., and Briggs, F. A., "Synchronization, Coherence, and Event Ordering in Multiprocessors," *Computer*, Feb. 1988, pg. 9-21.

[8] Grafe, V. G. and Hoch, J. E., "The Epsilon-2 Multiprocessor System," *Journal of Parallel and Distributed Computing*, Vol. 10, 1990, pg. 309-318.

[9] Grafe, V. G. and Hoch, J. E., "The Epsilon-2 Hybrid Dataflow Architecture," *Proceedings of Compcon90*, March 1990, pg. 88-93.

[10] Hennesay, J. L. and Patterson, D. A., *Computer Architecture: A Quantitative Approach.*, San Mateo, California: Morgan Kaufmann Publishers, Inc., 1990.

[11] Hwang, K. and Briggs, F. A., *Computer Architecture and Parallel Processing*, New York: McGraw-Hill, Inc., 1984.

[12] Iannucci, R. A. "Toward a Dataflow / von Neumann Hybrid Architecture," *Proceedings of the 15th International Symposium on Computer Architecture*, 1988, pg. 131-140.

[13] Lee, B., Hurson, A. R., and Feng, T. Y., "A Vertically Layered Allocation Scheme for Data Flow Systems," *Journal of Parallel and Distributed Computing*, Vol. 11, 1991, pg. 175-187.

[14] Lubeck, O.M., "A User's View of Dataflow Architectures," *Proceedings of Compcon90*, March 1990, pg. 84-87.

[15] Papadopoulos, G. M. and Culler, D. E., "Monsoon: An Explicit Token Store Architecture," *1990 IEEE 17th Annual International Symposium on Computer Architecture*, May 1990, pg. 82-91.

[16] Papadopoulos, G. M. and Traub, K. R., "Multithreading: A Revisionist View of Dataflow Architectures," *18th Annual International Symposium on Computer Architecture*, 1991, pg. 342-351.

[17] Quinn, M. J., *Designing Efficient Algorithms for Parallel Computers*, 2nd ed., New York: McGraw-Hill Book Company, 1991.

[18] Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y., and Yuba, T., "An Architecture of a Dataflow Single Chip Processor," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pg. 46-53.

[19] Sarkar, V. and Hennessy, J., "Partitioning Parallel Programs for Macro-Dataflow," *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 1986, pg. 202-211.

[20] Sarkar, V. and Hennessy, J., "Compile-time Partitioning and Scheduling of Parallel Programs," *Proceedings of the Symposium on Compiler Construction*, 1986, pg. 17-26.

[21] Yamaguchi, Y., Sakai, S., Hiraki, K., Kodama, Y., and Yuba, T., "An Architectural Design of a Highly Parallel Dataflow machine," *Information Processing*, 1989, pg. 1155-1160.