A Strategy for Scheduling Partially Ordered Program Graphs onto Multicomputers

Ben Lee and Chae Shin

Oregon State University
Department of Electrical and Computer Engineering

A. R. Hurson

The Pennsylvania State University
Computer Science and Engineering Department

Abstract

The issue of scalability is key to the success of massively parallel processing. Due to their distributed nature, message-passing multicomputers are appropriate for achieving scalar performance. However, the message-passing programming model lacks programmability due to difficulties encountered by the programmers to partition and schedule the computation over the processors and to establish efficient inter-processor communication in the user code. Therefore, this paper presents a compile-time scheduling heuristic, called BLAS, that maps programs onto the processors of a message-passing multicomputer. In contrast to other methods proposed in the literature, BLAS takes a more global approach in attempt to balance the tradeoff between exploitation of parallelism and reducing communication overhead. To evaluate the effectiveness of BLAS, detailed simulation studies of scheduling SISAL programs are presented.

1 Introduction

Two types of parallel programming models have become prevalent for highly parallel architectures. In the *shared-memory model*, synchronization among processes is achieved using shared variables stored in memory with a global address space. Due to their natural extension to single-CPU systems, shared-memory multiprocessors are considered among the easiest parallel computers to program [8]. However, when it comes to building massively parallel processors, a major drawback of shared-memory multiprocessors is the lack of scalability. In contrast, multicomputers based on the *message-passing model* have distributed memories, and synchronization among processes is performed using inter-processor communication (IPC). These systems consist of a collection of multiple computers or processors interconnected by message-passing networks (e.g., mesh, hypercube, fat tree, etc.) that provide point-to-point static connections among the processors. Due to their distributed nature, multicomputers based on the message-passing model are better in achieving scalable performance.

A major drawback in programming multicomputers is the difficulty of properly coding message-passing primitives in the user program. This difficulty is due to the processes or tasks residing on different processors. They must use communication primitives, such as `send` and `receive`, and therefore the programmer is forced to be constantly aware of the data movements between processes. Another difficult problem is the job of partitioning and scheduling of tasks to maximize the inherent concurrency in a program while minimizing IPC costs. These are two conflicting objectives which are strongly influenced by program and architectural characteristics.

In light of the aforementioned discussions, a method called the *Balanced Layered Allocation Scheme* (BLAS) is proposed that automates the process of partitioning and scheduling program graphs onto the processors of multicomputers. BLAS is a compile-time method that has been implemented to process general program graphs represented as directed acyclic graphs. BLAS has also been modified to accept SISAL program graphs represented in the intermediate form IF2 [13]. Through simulation, the effectiveness of BLAS is compared against two scheduling algorithms—internalization [10] and dynamic level scheduling [12].

2 Partitioning and scheduling

The two major issues involved in mapping programs across the processors of a multicomputer are *partitioning* and *scheduling* [8, 12]. Partitioning refers to the decomposition of a program into tasks or processes. On the other hand, scheduling refers to the assignment and the ordering of these tasks to processors. These two problems are referred to as the *mapping problem*. An effective mapping scheme must consider the tradeoff between the amount of parallelism exposed and the amount of communication overhead incurred. The mapping problem is further complicated due to the existence of variety of architectural differences as well as interconnection topologies.

Partitioning programs to tasks is important for a number of reasons. A task defines the basic unit of work for scheduling and thus the granularity of a computation. Therefore, the main goal in partitioning programs to tasks is to maximize parallelism while minimize the overhead required to support the tasks [12].

Once a program has been decomposed into tasks, these tasks are assigned and scheduled onto the available processors. The scheduling process is driven by a number of target parameters, such as program and architectural characteristics. For example, program characteristics define the partial ordering among the tasks in a program and is therefore a constraint on how the tasks should be assigned to a processor. Architectural characteristics, such as latency of the network, also place a constraint on the assignment process. Therefore, the major objective in scheduling tasks onto processors is maximizing the inherent concurrency in a program while minimizing IPC.

2.1 Preliminaries

The input to our scheduler is a *partially ordered program graph* (POPG), which is a directed acyclic graph representation of a program $G \equiv G(N, A)$, where N represents the set of tasks or nodes and A represents the partial ordering < between nodes. Therefore, a directed path from node n_i to node n_j implies that n_i precedes n_j (i.e., $n_i < n_j$). Moreover, an execution time t_i associated with $n_i \in N$. A communication time C_{ij} is associated with all the arcs $a_{ij} \in A$, and each a_{ij} is assumed to carry a label D_{ij} that specifies the amount of data transferred from n_i to node n_j. We assume a POPG is partially ordered in the sense that some partitioning is done prior to the scheduling process. In other words, the granularity of the nodes in POPG is coarser than instruction-level parallelism (as in dataflow).

The scheduling problem consists of assigning the nodes N on the set of processor $P = \{ p_1, ..., p_m \}$ that satisfy the partial ordering and optimizes the overall performance. That is, the objective function is to maximize the speedup S, which is defined as the single processor execution time T_S divided by the parallel execution time on multiple processors T_P. The upper bound on speedup S is defined by the *average parallelism* S_∞ of a program, which can be characterized as the ratio of T_S to the length of the critical path in the program T_{CP}.

2.2 Related work

There have been a number of proposed methods for optimally scheduling tasks for limited cases [1]. However, if the execution times vary or if the number of processors is greater than two, the problem of optimally scheduling a program to processors is NP-complete [10]. The scheduling problem is further complicated by the fact that IPC exists between nodes assigned to different processors. Therefore, heuristics are used solve the scheduling problem suboptimally in polynomial time.

A number of heuristic algorithms based on *critical path list schedules* have been developed [1, 3, 7]. However, these methods are not applicable in general because the communication overhead associated with the predecessor-successor nodes assigned to different processors is not considered. For this reason, there have been numerous heuristic solutions that consider communication overhead, which can be compared on the basis of their proximity to optimal solutions and complexities [11].

A related work proposed by Sarkar attempts to minimize the schedule length on an unbounded number of processors [10]. Sarkar's algorithm, called *internalization*, initially places each node in a separate block and then considers the arcs of the program graph in descending order of communication costs. In an iterative fashion, each C_{ij} associated with arc a_{ij} is set to zero by merging the nodes n_i and n_j into a single block as long as the schedule length does not increase (this method is also known as edge-zeroing [7]). This process completes when all the arcs have been scanned. The scheduling phase then merges the blocks until the number of blocks is equal to the number of processors. The complexity involved determining whether two blocks should be merged is $O(N + A)$. Since there are A such iterations, the complexity of the internalization algorithm is at least $O(A(N + A))$.

Sih and Lee have proposed a method called the *Dynamic Level Scheduling* (DLS) algorithm that also considers communication costs [12]. The algorithm first considers the static levels of the nodes, where the static level of a node is a weight that equals the maximum execution time from that node to the exit node. Basically, the scheduler considers the difference between the static level of a node and the maximum of the earliest time the node can start execution on a processor j (including communication costs) and the time the last node assigned to the processor j finishes execution. This difference is called the dynamic level of a node for a particular processor. The idea is to evaluate the dynamic levels of all combinations of ready nodes and available processors in an attempt to find the best node-processor combination for scheduling. Sih and Lee also propose to streamline the algorithm by selecting only a single ready node and limiting the number of processors for which dynamic levels are evaluated.

These methods indicate the two contrasting philosophies on how scheduling should be performed. Internalization algorithm attempts to find the best partition possible before blocks are merged onto the processors. On the other hand, the DLS algorithm is more of a variation of list scheduling where a ready node/processor combination with the higher precedence (i.e., dynamic level) is scheduled first. Therefore, in general, the internalization algorithm results in a better schedule while DLS algorithm is faster [12].

The key to partitioning a POPG into blocks for the internalization algorithm is knowing the IPC costs associated with the arcs. The IPC cost between a predecessor and a successor processor in a multicomputer with a static network is a function of the startup cost, the size of the message, the bandwidth, the hop time, and the number of hops between the processor. For commercial parallel machines the startup cost dominates the IPC latency. For example, The startup overhead for CM-5 can be as high as 86μs while the time to actually transport a single packet of a message is only 0.126μs [9]. Moreover, the inter-hop distances associated with the arcs are usually not known until nodes of a POPG is actually assigned to the processors. Thus, the internalization algorithm establishes a priority for merging nodes based on a parameter usually not known until the assignment takes place. The DLS algorithm, on the other hand, does not suffer from this problem. The earliest time a node can start execution on a processor is determined only after all its predecessor nodes have been assigned to processors; therefore, a good estimation of the IPC cost can be obtained. However, the drawback of the DLS algorithm is the scheduling policy tends to be greedy, i.e., the assignment process considers only the current ready nodes.

In this paper an alternative method called *Balanced Layered Allocation Scheme* (BLAS) is proposed, which eliminates deficiencies with the aforementioned scheduling policies. BLAS achieves this by taking a more global approach to scheduling nodes. The development of BLAS emanated from our initial work on scheduling dataflow programs on Multistage Interconnection Networks (MIN) based dataflow computers, called Vertically Layered (VL) allocation scheme [8].

3 The proposed method

3.1 Balanced Layered Allocation Scheme (BLAS)

BLAS consists of three phases: separation, assignment, and ordering. The objective of the *separation phase* is to identify as many sets of serially connected nodes as possible. Each set of serially connected node is called a *virtual thread.* Only the expected execution times t_i are considered in determining the virtual threads.[1] The separation phase starts by identifying the critical path of a POPG. These nodes are defined as belonging to the virtual thread N_{CP}. Since the critical path defines the longest path from the root node to the exit node, assignment of the critical path to a single processor minimizes IPC associated with the critical nodes. For simplicity, if no unique critical path exists, the algorithm arbitrary chooses one.

Once the critical path is identified, the set of nodes N_{CP} is marked and queued into a First-In-First-Out (FIFO) queue Q while maintaining to their precedence constraints. All other virtual threads are determined in an iterative manner as follows: Let N^{k-1}_{marked} represent the set of nodes that have already been arranged into virtual threads at step k-1 (initially we have $N^0_{marked} = N_{CP}$). A node n_i is removed from Q and the set of nodes comprising the *longest directed path* N^k_{LDP} emanating from n_i is formed such that

$N^{k-1}_{marked} \cap N^k_{LDP} = \{\emptyset\}$ and $N^{k-1}_{marked} \cup N^k_{LDP} = N^k_{marked}$.

The method used in finding the longest directed path emanating from n_i involves the same procedure as finding the critical path without considering the marked nodes. The separation phase is similar to the linear clustering algorithm proposed by Kim and Browne [6].

The *assignment phase* starts by arbitrary assigning N_{CP} to a processor. Then, each virtual thread N^k_{LDP}, for k= 1, 2, 3, ..., is dequeued from Q and assigned to processors in an iterative fashion. In this manner, the effects of execution times and communication costs can be weighed against the different processor assignments for N^k_{LDP}. After weighing the effects of N^k_{LDP} on each processor, the virtual thread N^k_{LDP} is assigned to the processor that provides the best tradeoff between exposing parallelism and minimizing IPC overhead. Currently, there are two processor selection criteria used by BLAS. The first method considers all the processors for assignment. The second method, which is faster, considers only adjacent processors.

For the iterative assignment of N^k_{LDP} to each processor P, the assignment phase considers the following properties:

(1) Let T^P_i represent the *completion time of processor i* when N^k_{LDP} is assigned to processor P. Then the *completion time*, T^P, is given by

$$T^P = max\{T^P_i \mid P = 0,\ 1,\ \ldots,\ p-1\},$$

where p is the number of available processors.

(2) Each set of nodes N^k_{LDP} is then assigned to the processor P that yields the *lowest completion time*, T_{min}, where

$$T_{min} = min\{T^P \mid P = 0,\ 1,\ \ldots,\ p-1\}.$$

If T_{min} yields more than one minimum, a processor is arbitrarily chosen. A completion time can be evaluated in a similar manner as determining the critical path in the separation phase by processing the POPG in a topological order that also includes IPC costs. The assignment phase is completed when Q is empty.

The final phase of our scheduler involves *ordering* of the nodes within the processors. This is necessitated by the fact that a number of virtual threads can be assigned to a processor; therefore, static ordering of the nodes is required. For dataflow computers, due to the self scheduling property of the dataflow model of computation, the ordering phase is not needed. However, multicomputers based on the control-flow model of computation lack this property and require the static ordering of the nodes.

[1] BLAS can be easily modified to consider POPGs with known IPC cost in the separation phase (see Section 6).

The ordering phase starts by considering the earliest time e_j a node n_j can start execution on processor P_j. This can be done by basically following the same procedure as in the separation phase with the inclusion of the IPC costs, i.e.,

$$e_j = \max_{i \in \langle PN_j \rangle} \{e_i + t_i\} \quad \text{if } P_i = P_j$$

$$e_j = \max_{i \in \langle PN_j \rangle} \{e_i + t_i + C_{ij}\} \quad \text{if } P_i \neq P_j,$$

where PN_j represents the set of all the predecessor nodes of n_j, and $\langle PN_j \rangle$ denotes the index set of PN_j. The nodes assigned to a processor is scheduled based on the topological order of e's.

3.2 An example

To illustrate the BLAS algorithm, consider the problem of scheduling the nodes of POPG shown in Figure 1 to a two-dimensional hypercube using all processor selection method. For illustrative purposes, we arbitrarily assume that communication cost between adjacent processors n_i and n_j is twice the average execution time of a node, *i.e.*, $C_{ij} = C_{ji} = 10$ units of time, and the average execution time of the nodes in the graph of Figure 1 is 5 units of time. Thus, based on the store-and-forward routing scheme, we have $C_{01} = 10$, $C_{02} = 10$, $C_{13} = 10$, $C_{23} = 10$, $C_{03} = 20$, and $C_{12} = 20$.

Applying the separation algorithm to the POPG in Figure 1 yields the following set of virtual threads: N^0_{marked} = N_{CP}={N1, N2, N5, N9, N11, N13}, N^1_{LDP} ={N3, N7, N10, N12}, N^2_{LDP} ={N4, N8}, and N^3_{LDP} ={N6}.

The assignment phase starts by dequeuing the virtual thread N_{CP}={N1, N2, N5, N9, N11, N13} from Q and arbitrarily assigning it to processor 2 (Figure 2). The set of nodes N^1_{LDP}={N3, N7, N10, N12} is then dequeued and iteratively assigned to every possible processor to determine which assignment yields the lowest completion time. For example, assigning the virtual thread N^1_{LDP} to processor 0 yields:

$T^0_0 = 32$, $T^0_1 = 0$, $T^0_2 = 38$, $T^0_3 = 0$, and $T^0 = 38$.

Assigning the virtual thread N^1_{LDP} to processor 1 yields:

$T^1_0 = 0$, $T^1_1 = 42$, $T^1_2 = 38$, $T^1_3 = 0$, and $T^1 = 42$.

Assigning the virtual thread N^1_{LDP} to processor 2 yields:

$T^2_0 = 0$, $T^2_1 = 0$, $T^2_2 = 57$, $T^2_3 = 0$, and $T^2 = 57$.

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

$T^3_0 = 0$, $T^3_1 = 0$, $T^3_2 = 38$, $T^3_3 = 32$, and $T^3 = 38$.

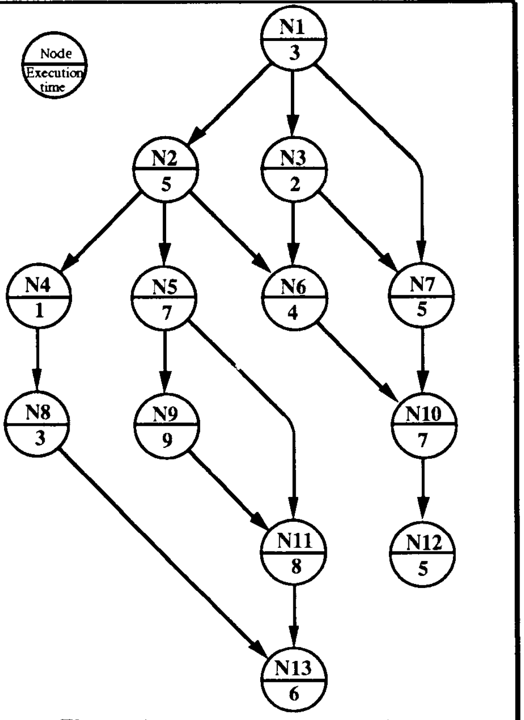

Figure 1: An example of a POPG.

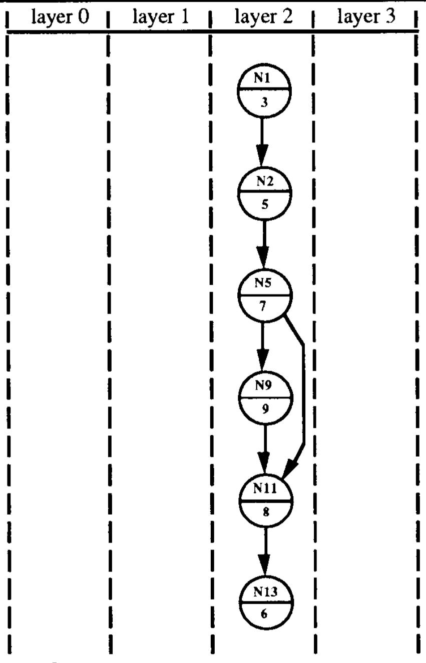

Figure 2: Arbitrary assignment of the critical path.

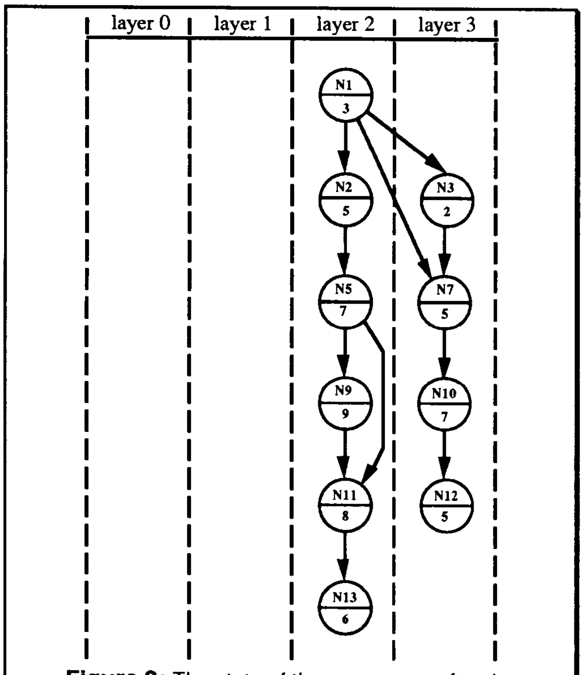

Figure 3: The state of the processors after the assignment of N^1_{LDP}={N3, N7, N10, N12}.

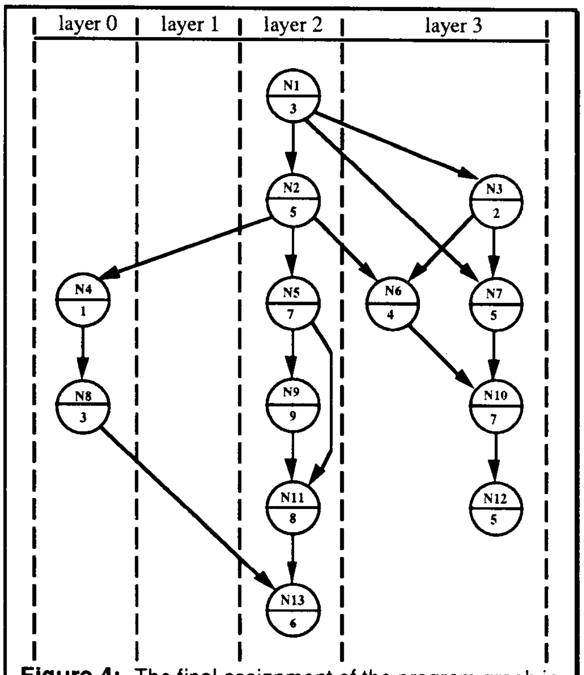

Figure 4: The final assignment of the program graph in Figure 1.

According to these results, N^1_{LDP} is assigned to the processor that yields the lowest completion time, i.e.,
$T_{min} = min\{T^0 = 38, T^1 = 42, T^2 = 57, T^3 = 38\}$.
Therefore, N^1_{LDP} can be assigned to either processor 0 or 3. Processor 3 is chosen arbitrarily as illustrated in Figure 3. After all the nodes in Figure 1 have been assigned, a layered graph shown in Figure 4 is generated.

There are a number of important observations that can be made about the characteristics of our scheduling algorithm. First, the particular assignment in Figure 4 results in the optimum schedule. Second, the heuristics for determining virtual threads employ a priority by giving the highest priority to the critical path and then to the longest directed paths emanating from the nodes that have been already converted to virtual threads. Finally, the scheduling process provides an ideal tradeoff between parallelism and IPC overhead. For example, consider the assignment of the virtual thread N^2_{LDP} ={N4, N8}. Any other assignment of N^2_{LDP} other than processor 0 results in an increased completion time. After the static ordering of the nodes for each processor, the final schedule is shown in Figure 5. Note that the same schedule would have been generated by using the adjacent processor selection method .

The absolute worst-case time complexity of our scheduling algorithm is $O(N(N+A))$, where N is the number of nodes and A is the number of arcs. This is based on the assumption that the time complexity of determining LDPs decreases incrementally with each iteration. In reality, the time complexity decreases at a much faster rate with each iteration and therefore results in a time complexity of $O((p+V)(N+A))$, for $V << N$, where p is the number of processors and V is the number of virtual threads.

4 Transformation of SISAL programs to POPGs

Major motivation of Streams and Iterations in a Single Assignment Language (SISAL) is to provide a parallel language that promotes the development of correct parallel programs by isolating the programmer from the overwhelming complexities of parallel computing [2]. This is achieved in SISAL by exposing implicit parallelism through data independence and guaranteeing determinate results with their side-effect free semantics.

SISAL front-end compiles programs into intermediate forms, called IF1 and IF2. These intermediate forms are based on acyclic graphs that define dependences among the instructions in a program. Therefore, they are an appropriate mean to develop a conceptual framework for understanding program behavior and thus scheduling programs. IF1 (and its optimized counterpart IF2) consists four components [13]: Nodes represent operations; edges

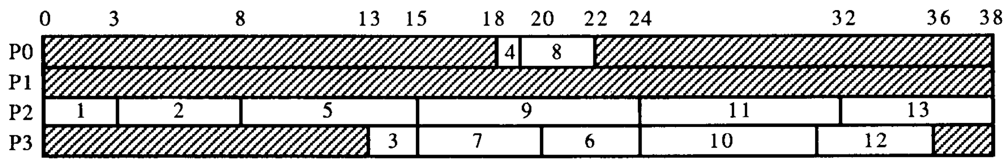

Figure 5: The final schedule for the POPG in Figure 1.

correspond to values that are passed from one node to another, types can be attached to each edge or function; and graph boundaries define subgraphs, each consisting of a group of nodes and edges. The graph boundaries and nodes have numbered ports to distinguish multiple inputs and outputs. Values that cross graph boundaries are refer to as imports and exports. IF1 nodes are hierarchical and can be of two types: simple or compound. The outputs of a *simple node* is a direct function of its inputs (there are over fifty simple nodes). A *compound node* contains subgraphs and its output depends on the interaction among these subgraphs. There are five compound nodes in IF1: `Select`, `TagCase`, `Forall`,`While`, and `Until`.

Before SISAL programs can be scheduled using BLAS, they must be converted to equivalent POPGs. There are two main parts to the conversion process: (1) Transformation of implicit dependences of IF2 to explicit dependences and (2) transformation of loops. The first transformation is required because inputs and outputs of graph boundaries are not connected as in the dataflow sense. Therefore, implicit dependences are converted to data dependences by identifying matching export and import port numbers as well as considering the semantics of the compound node. This is done by using an *association list* that defines a mapping between the subgraphs and their intended use by the compound node. In order for the transformation to take place, three types of dependencies must be considered within a compound node [13]:

- Data dependence between the compound node and its subgraphs - This occurs when a compound node passes some of its input values to its subgraphs, or when a subgraph returns values that become the results of the compound node.
- Data dependence among subgraphs - This occurs in loops, where the loop values calculated in the loop body are passed to the result part and back into the loop body.
- Control dependence - This occurs when a predicate determines which expression to evaluate, or when an iterative loop is to terminate.

The transformation process uses the information provided by the association lists to construct an equivalent POPG. The conversion process only maintains the acyclic dependences.

The transformation of loops is performed in order to unravel parallelism in loop constructs. In SISAL, the main source of parallelism comes from `Forall` loops. Therefore, body subgraphs of `Forall` compound nodes are duplicated according to the number of parallel iterations. It is important to note that instruction-level parallelism within a subgraph is not exposed. Our studies indicate exploiting parallelism within a subgraph result in no real benefit due to their fine-grain nature. Thus, the nodes within a subgraph are serialized to form a single macro node. Nested `Forall` loops are also handled in a similar manner.

Another type of transformation is performed on `While` and `Until` loops. The `While` and `Until` compound nodes repeatedly apply the loop body to a set of values, stopping and returning results when the test becomes false. The difference between the two constructs is that `While` performs the test for termination before the body is executed while `Until` performs the termination test after the body has executed. The transformation process basically serializes the execution of the loop bodies according to the number of iterations. This parameter can be defined explicitly by the programmer, derived by the compiler, or refined by monitoring previous executions of SISAL program. Currently, we rely on data obtained from the execution of SISAL programs.

The final transformation occurs with conditionals, i.e., `Select` compound nodes. The `Select` is used to handle two-way selection (i.e., "if-then-else") and contains the Selector subgraph representing a predicate and two Alternative subgraphs corresponding to true or false branches. Since the result of the predicate is known only at run-time, we assign a probability to each of the Alternative subgraphs such that $p_{true} + p_{false} = 1$. By default, all subgraphs other than the `Select` node has a probability of 1. The conversion process to POPGs is done by multiplying the true and false subgraph execution times with its respective probabilities to form a single node.

5 Simulation studies

In this section, the performance of BLAS is compared with the internalization and the DLS algorithms. Instead of streamlined version, DLS algorithm was implemented to evaluate dynamic levels over all combinations of ready nodes and available processors to provide a fair comparison.

There were a number of input program graphs used for this comparison. The first set consisted of several hypothetical graphs generated by hand containing between 13 to 193 nodes. The second set of graphs was obtained by transforming IF2 graphs generated from SISAL programs.

Our target architectures for the simulation studies were hypercube and mesh with the following assumptions:

(1) For the hypothetical graphs, the execution times for the nodes were uniformly distributed over $[t_{min}, t_{max}]$. For IF2 graphs generated from SISAL, the execution time t_i for each node is assigned from IF2 Execution Cost Profile. This is a machine dependent parameter obtained from SISAL, where each simple node (i.e., instruction) is associated with number of clock cycles and is representative of modern processor execution costs.

(2) The IPC costs are based on nCUBE 3 using cut-through routing [4]:

$$C_{ij} = t_s + ht_h + \frac{D_{ij}}{BW}, \qquad \text{if } i \neq j$$

where t_s is the startup latency, t_h is the per-hop time, h represents the number of hops the message travels, D_{ij} is the length of the message, and BW is the channel bandwidth. The startup latency t_s consists of the time to prepare the message, the time to execute the routing algorithm, and the time to establish an interface between the local processor and the router. Based on the nCUBE 3, t_s is assumed to be around 5 μs or 250 processor cycles. The terms $ht_h + D_{ij} / BW$ is called the transport time, and it represents the time spent from when the head of the message leaves processor i to when the tail of the message arrives at processor j. nCUBE 3 is claimed to have a network bandwidth of nearly 50Mbytes/s; therefore, for small messages we assume the transport time is dominated by the term ht_h. With intermediate per-hop latency of 200 ns (or 10 cycles), C_{ij} is given as

$$C_{ij} = 250 + h \cdot 10 \text{ cycles} \quad \text{if } i \neq j \text{, and}$$
$$C_{ij} = 0 \qquad \text{if } i = j.$$

Based on these assumptions, the ratio of startup time to per-hop time t_s/t_h was kept constant at 25. In order to take into effect the granularity of computation, the IPC costs were varied based on a ratio of communication to execution time, C/T. This was done by keeping the average execution time T fixed while the communication cost C was varied. This allowed us to study the performance of the scheduling algorithm under varying granularity of computation.

To illustrate the results of the simulation studies, L2 and 10×10 `matmult` were chosen as representative POPGs. L2 is a 193-node hand generated hypothetical graph that performs assignment and sequencing [8]. The execution times for the nodes were uniformly distributed over [10, 40] with an average execution time of 25. 10×10 `matmult` is a 134-node matrix multiplication program generated from IF2. The range of node execution time is [0, 250] with an average node execution time of 99. In order to increase the granularity, the inner-loop containing multiplication and reduction for each i and j iterations was lumped into a single node. We have experimented with parallelizing of the inner-loop; however, due to its fine-grained nature (compared to the IPC costs) the expansion provided no real benefit. Figures 6 and 7 depict the total execution times versus number of processors for L2 for varying C/T ratios on a hypercube and a $\sqrt{P} \times \sqrt{P}$ mesh, respectively. Figures 8 and 9 show the performance for 10×10 `matmult` for varying C/T ratios on a hypercube and a $\sqrt{P} \times \sqrt{P}$ mesh, respectively.

As expected, the execution time for the BLAS decreases as the number of processors increases for both POPGs. Moreover, as the IPC overhead increased, so did the execution times illustrating the important effect communication overhead has on the overall performance. L2 and 10×10 `matmult` have maximum parallelism of 17 and 25, respectively, therefore no real benefit is gained by adding additional processors beyond maximum parallelism. In addition, as C/T ratio increases, less number of processors is required before the performance begins to saturate.

In general, BLAS (both all processor and adjacent processor selection methods) outperformed both internalization and DLS algorithms. For the internalization algorithm, the two step process of minimization of the execution time on an unbounded number of processors followed by an assignment to a fixed number of processors does not necessarily provide a good mapping. This is because the order in which the nodes are clustered is usually very crucial to partitioning step of the internalization algorithm. However, when the IPC overhead is almost constant or unknown, due to small number of messages, the clustering the nodes based on this parameter becomes less effective. Another problem with the internalization is that after the nodes are partitioned into blocks, the blocks are assigned onto the processors using a modified priority list scheduling, which tends to be greedy.

The DLS algorithm also has similar problems due to its greedy nature. In fact, the DLS algorithm exhibits an interesting behavior when the number of available processors is small. Whenever, dynamic levels are evaluated for two nodes generated from a fork, i.e., nodes which have the same parent node, the node with the higher static level will be scheduled onto the same processor as the parent node. Depending on the dynamic level, the second child node will either be scheduled onto the same processor as the parent node or to another processor. Often the node will be scheduled onto another processor despite the fact that the overall performance would have been better if the node was scheduled onto the same processor as the parent node. Thus, when the number of processors is small, thrashing of IPC occurs and the performance degrades. This problem does not occur when the number of processor is sufficiently large because there are more processors to choose from. This problem is more severe for the `matmult` because of its symmetrical nature containing mainly of joins and forks.

Figure 10 shows a comparison between the two processor selection methods for `matmult`. This is a plot that

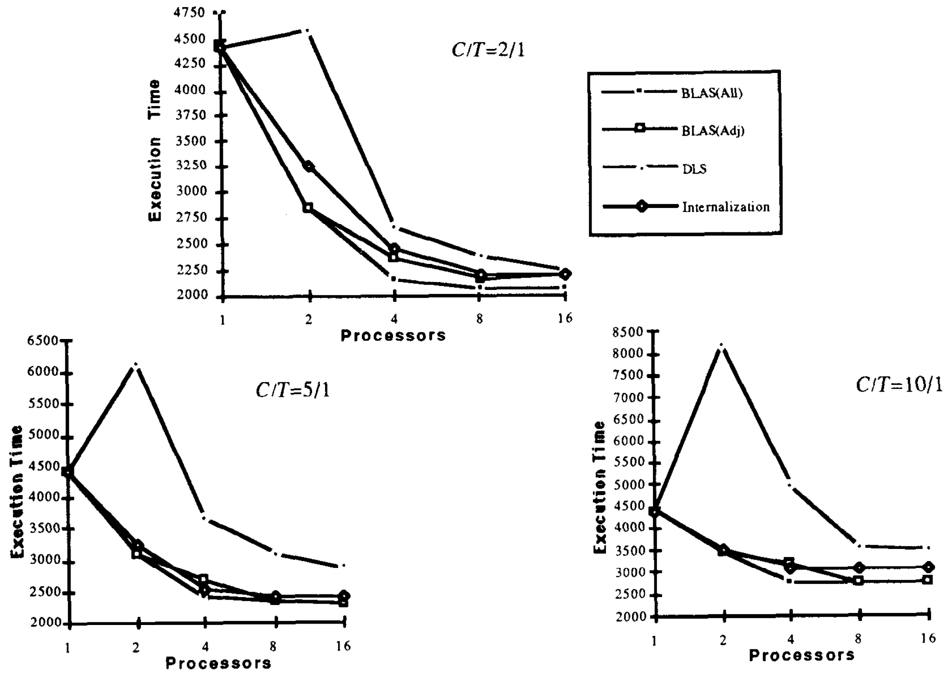

Figure 6: Execution time vs. number of processors for L2 on a hypercube.

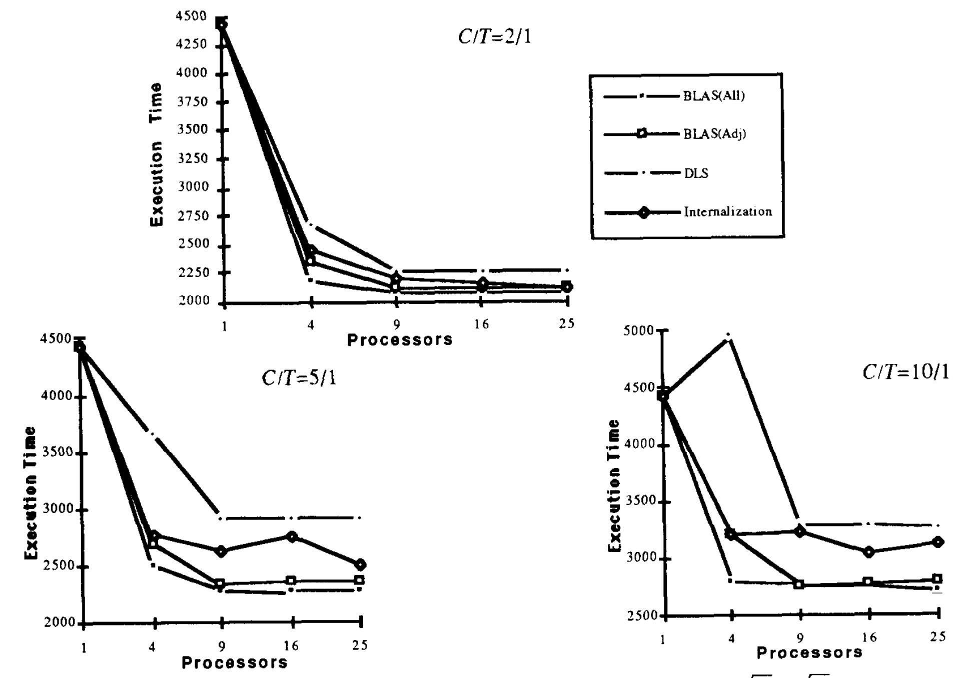

Figure 7: Execution time vs. number of processors for L2 on a $\sqrt{P} \times \sqrt{P}$ mesh.

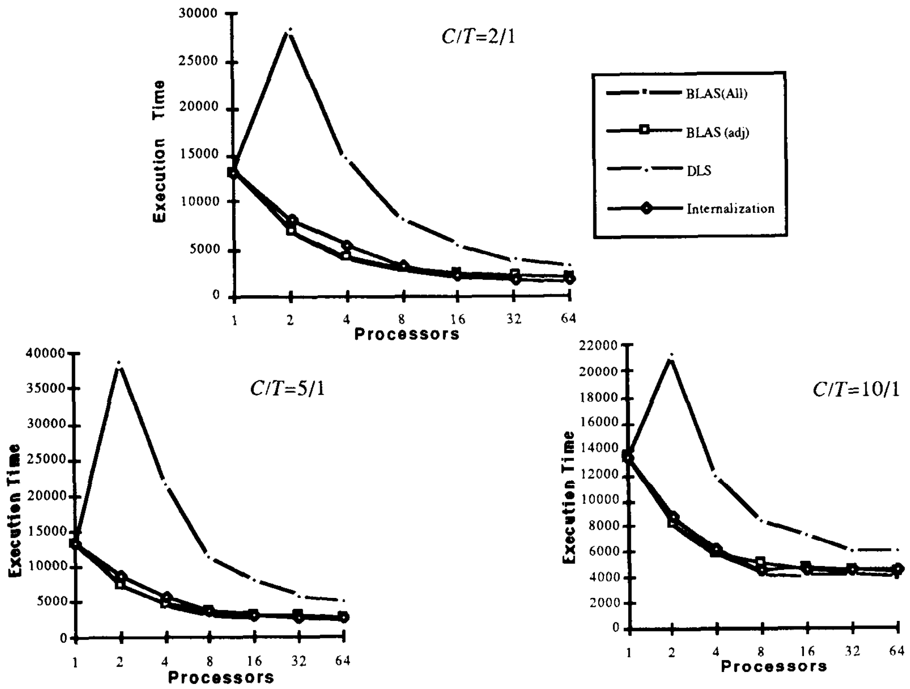

Figure 8: Execution time vs. number of processors for 10x10 `matmult` on a hypercube.

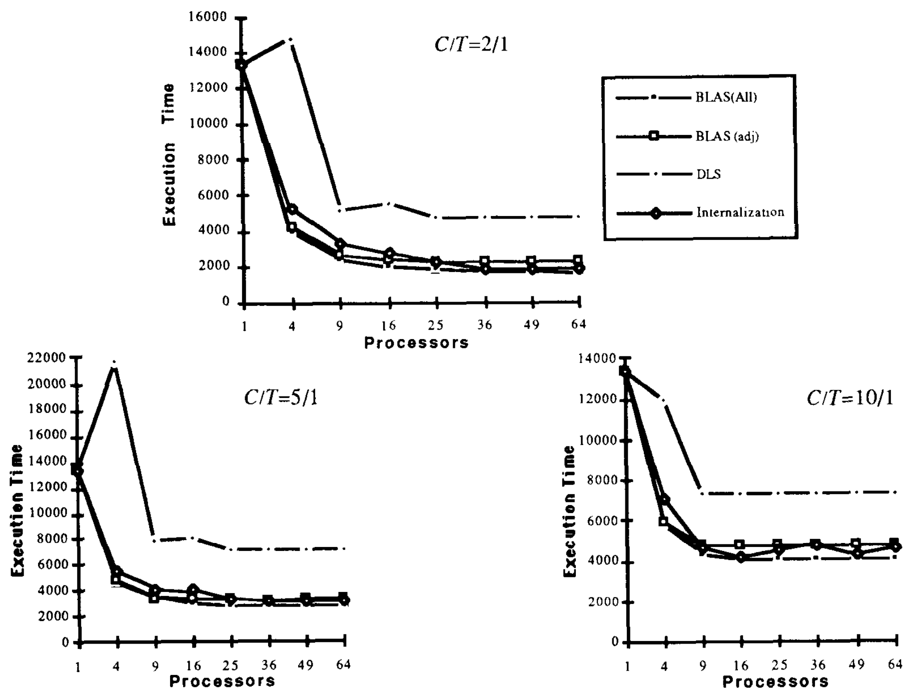

Figure 9: Execution time vs. number of processors for 10x10 `matmult` on a $\sqrt{P} \times \sqrt{P}$ mesh.

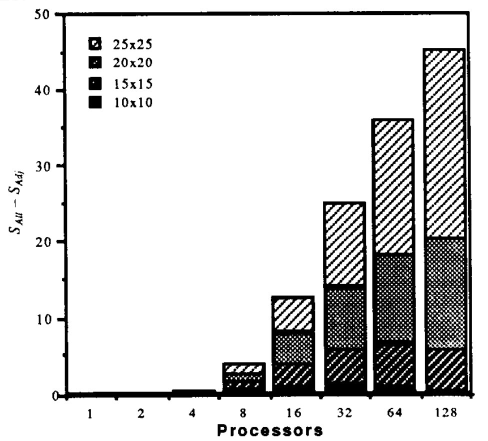

Figure 10: A comparison between the two processor selection methods for `matmult`

shows the difference between the two speedups (i.e., $S_{All} - S_{Adj}$) for various matrix sizes. The average execution time varied from 99 (for 10×10) to 250 (for 25×25), and again the startup time and the inter-hop time were 250 and 10, respectively. Due to the greedy nature of the adjacent processor selection method, its performance is always worse than the all processor selection method. The difference is even more profound as the number of processors available for the selection increases. For extreme cases, the differences in speedup can be as much as 25 for 25-by-25 matrix on 128 processors. The main reason for such a large difference is the ability of the any processor selection method to spread out the load over a large number of processors. However, the improvement in performance has a cost in terms of added complexity.

6 Summary and Future Research

In this paper, a compile time method, called BLAS, for scheduling POPGs onto multicomputers was presented. A method for transforming IF2 graph from SISAL to POPGs was also described. In contrast to list scheduling algorithms, BLAS takes a more global approach by first partitioning a program graph into virtual threads, and then assigning the virtual threads to processors. In general, the BLAS is very successful in providing a balance between exploiting parallelism and reducing communication overhead due to (1) the concept of virtual threads that serializes the nodes when no parallelism exists, and (2) the assignment of a virtual thread is always optimum relative to virtual threads that have already been assigned to processors.

Although our simulation studies indicate promising improvement over internalization and DLS algorithms, some optimization can be made to BLAS. For larger messages, including the cost of messages on the arcs would be an additional benefit in determining the virtual threads in the separation phase. This would lead to a reduction in IPC overhead not only due to startup costs, but also due to relatively large message sizes. Another improvement would be to consider combining different messages destined for the same processor. As discussed in Section 2.2, the startup costs dominates the IPC overhead; therefore, concatenating messages destined for the same processor will eliminate the additional startup costs incurred by sending different packets of shorter messages.

References

[1] Adam, T. L., Chandy, K. M., and Dickson, J. R., "A Comparison of List Schedules for Parallel Processing Systems," *Commun. ACM*, Vol. 17, No. 12, Dec. 1974, pp. 685-690.

[2] Cann, D. C., "The Optimizing SISAL Compiler Version 12.0" Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, CA, April 2, 1992.

[3] Coffman, E. G., Editor, Computer and Job Shop Scheduling Theory, John Wiley and Sons, New York, NY, 1976.

[4] Duzett, B. and Buck, R., "An Overview of the nCUBE 3 Supercomputer," *Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992, pp. 474-483.

[5] Gerasoulis, A. and Yang, T., "A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 16, 1992, pp. 276-291.

[6] Kim, S. J. and Browne, J. C., "A General Approach to Mapping of Parallel Computations Upon Multiprocessor Architectures," *Proceeding of International Conference on Parallel Processing*, Vol. 3, Aug. 1988, pp. 1-7.

[7] Kohler, W. H., "A Preliminary Evaluation of Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Transactions on Computers*, Dec. 1975, pp. 1235-1238.

[8] Lee, B., Hurson, A. R., and Feng, T. Y. "A Vertically Layered Allocation Scheme for Dataflow Computers," *Journal of Parallel and Distributed Computing*, Vol. 11, 1991. pp. 175-187.

[9] Papadopoulos, G. M. *et al.*, "*T: Integrated Building Block for Parallel Computing," *Proceedings of Supercomputing Conference*, Nov. 1993, pp. 624-635.

[10] Sarkar, V., Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors, The MIT Press, Cambridge, MA, 1989.

[11] Shirazi, B., Wang, M., and Pathak, G., "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *Jounal of Parallel and Distributed Computing*, 10, 1990, pp. 222-232.

[12] Sih, G. C. and Lee, E. A., "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Hetrogeneous Processor Architectures," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 2, February 1993, pp. 175-187.

[13] Skedzielewski, S. K. and Glaurert, J., "IF1 - An Intermediate Form for Applicative Langauages," Lawrence Livermore National Laboratory Manual M-170, 1985.