

Copyright © 1990, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

U.C.B./ERL
M 90/119
Dec 17, 1990
44 pages

**A MULTIPROCESSOR SCHEDULING
STRATEGY**

by

Gilbert C. Sih and Edward A. Lee

Memorandum No. UCB/ERL M90/119

17 December 1990

(Revised 13 June 1991)

COVER PAGE

**A MULTIPROCESSOR SCHEDULING
STRATEGY**

by

Gilbert C. Sih and Edward A. Lee

Memorandum No. UCB/ERL M90/119

17 December 1990
(Revised 13 June 1991)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

**A MULTIPROCESSOR SCHEDULING
STRATEGY**

by

Gilbert C. Sih and Edward A. Lee

Memorandum No. UCB/ERL M90/119

17 December 1990
(Revised 13 June 1991)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A MULTIPROCESSOR SCHEDULING STRATEGY

**Gilbert C. Sih
Edward A. Lee
Department of EECS
University of California
Berkeley, CA 94720**

ABSTRACT

This paper presents a new compile-time scheduling heuristic called **declustering**, which schedules acyclic precedence graphs that fit the synchronous data flow (SDF) model [1] onto multiprocessor architectures. This technique accounts for interprocessor communication (IPC) overheads and considers interconnection constraints in the architecture so that shared resource contention can be avoided. The algorithm initially invokes a new clustering method that uses graph-analysis techniques to isolate parallelism instances. When constructing an initial set of clusters, this procedure explicitly addresses the tradeoff between exploiting parallelism and incurring communication cost. By hierarchically combining these clusters and then systematically decomposing this hierarchy, the declustering method exposes parallelism instances in order of importance and attains a cluster granularity that fits the characteristics of the architecture. We show that declustering retains the clustering advantage of avoiding IPC, yet overcomes the inflexibility associated with traditional clustering approaches.

Index Terms: clustering, declustering, interprocessor communication, multiprocessor scheduling, parallelism detection, parallel processing.

The authors gratefully acknowledge the support of SRC, Cygnet, Dolby Laboratories, and the State of California Micro program.

1. INTRODUCTION

Hardware advances in parallel processing machines have far exceeded the capabilities of current software to exploit them effectively. The task of efficiently coordinating parallel processors is formidable, requiring a program partitioning that matches the available hardware, and a mapping strategy that considers the interprocessor communication (IPC) and synchronization overheads created by data exchanges.

This paper introduces the **declustering algorithm**, a nonpreemptive, compile-time scheduling technique that maps precedence graphs onto multiprocessor architectures. A program is described as an acyclic precedence expansion graph (APEG) $G = \{N, A\}$, where $N = \{N_i : i = 1, \dots, n\}$ is a set of nodes (tasks) which represent program computations, and A is a set of directed arcs $\{A_{ij}\}$ which represent both precedence constraints and data paths. Each arc A_{ij} carries label D_{ij} which specifies the number of data units passed from N_i to N_j on each invocation of the program. We assume without loss of generality that each APEG has exactly one terminal node. This condition can be enforced by connecting multiple endnodes to a dummy terminal node N_t with dummy arcs that pass no data.

These graphs may be expansions of data flow graph algorithmic descriptions that fit the Synchronous Data Flow (SDF) model [1]. This description naturally exposes inherent internode parallelism in the algorithm and allows partitioning, scheduling, and insertion of synchronization primitives to be performed at compile-time, thereby avoiding much run-time overhead. Under a *fully-static* scheduling paradigm, the class of applications effectively expressed using this programming model is limited. Constructs such as conditionals and data-dependent iteration must be excluded to provide deterministic program behavior, making the application domain most suitable for signal processing algorithms and some scientific computations. In this environment, a fairly accurate estimate of the execution time of node N_i ($E(N_i) > 0$) can be obtained at compile-time. An example APEG is shown in figure 1, where the number below each node shows its execution time. Under a *self-timed* scheduling paradigm, several of these constraints can be relaxed,

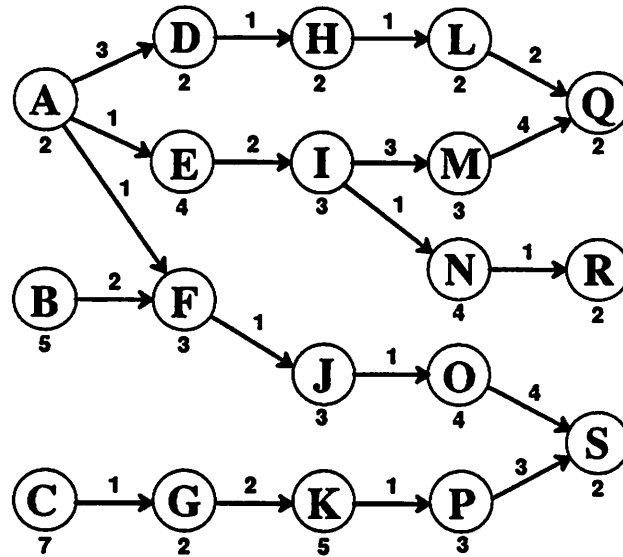


Figure 1. An example APEG

permitting the class of relevant applications to be somewhat broadened. See [2] for a discussion of these scheduling applicability issues.

The declustering algorithm targets architectures of the form shown in figure 2, in which a set of homogeneous processors $P = \{P_k : k = 1, \dots, p\}$ with local memory, communicate through global shared memory modules via some type of interconnection network, such as a shared-bus, banyan, or crossbar network.

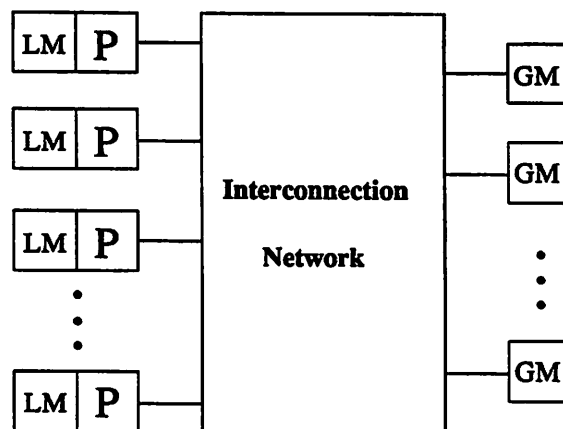


Figure 2. The class of relevant architectures

Our scheduling objective is to minimize the **schedule length, or makespan**, where all inter-processor communication overheads are included. This goal equivalently maximizes the **speedup**, defined as the shortest time required for sequential execution of the APEG on a single processor, divided by the time required for parallel execution on multiple processors. This scheduling problem is NP-complete in the strong sense, even if there are an infinite number of processors available [3, 4]. Hence we will rely upon heuristics.

Many related works have concentrated on task allocation, attempting to assign tasks to processors in order to achieve some objective, such as load balancing, minimization of IPC, or some combination of the two [5, 6, 7]. Although these works are innovative, they either ignore precedence constraints or attempt to minimize an objective other than schedule length, and thus are not applicable in this context.

Several other scheduling approaches have been proposed that attempt to minimize makespan when interprocessor communication costs are included. The dynamic-level scheduling algorithm [8, 9] is a single-pass technique intended for environments requiring fast scheduling. It modifies the classical HLFET algorithm [10] by using dynamically changing priorities to match nodes and processors for scheduling at each step. Greenblatt and Linn propose a method using branch and bound heuristics [11] to prune the search space of possible schedules. Clustering techniques, which were first applied to solve task allocation problems [12], have been adopted as a popular scheduling approach. These schemes divide the graph into several sets of nodes, or clusters, which are then mapped onto the processors, usually through some heuristic or graph theoretic technique. Since clustering is intrinsically related to declustering, we examine two important clustering methods in section 2 [3, 13].

The declustering scheduler has been implemented as part of an interactive software design system for digital signal processing (DSP) that permits rapid prototyping of new DSP algorithms and architectures [14]. The DSP algorithms are specified as block diagrams, where the computation blocks range in granularity from simple operators such as adders or multipliers, to higher

level functions such as FIR filters or FFT's, to complicated subsystems such as filter banks or speech coders. The design system translates these block descriptions into APEGs before scheduling them onto the specified architecture. This environment requires that the scheduling technique be fast (to support interactive prototyping), flexible (to handle widely-varying node granularities), and adaptable (to permit applicability to the wide range of DSP architectures). To enable wide retargetability, the declustering algorithm is split into two sections: an architecture-independent section containing the scheduling routines, and an architecture-dependent section containing the routing and communication resource reservation routines that have been specifically designed for the targeted architecture. The architecture-independent section calls the architecture-dependent component to perform such tasks as routing a path between two processors, or computing the time needed to communicate a given amount of data between specified processors. If the designer switches architectures, the appropriate architecture-dependent section is loaded from an existing library of such routines.

This paper is organized as follows: section 2 discusses clustering schemes and uses their limitations to motivate the declustering approach. Sections 3, 4, 5, and 6 describe the various phases of the declustering algorithm. Section 7 compares the scheduling performance of declustering with two prominent clustering schemes, and section 8 summarizes this discussion and offers suggestions for future research.

2. CLUSTERING ALGORITHMS

Clustering algorithms have gained wide popularity for scheduling in the presence of inter-processor communication costs [15]. By forcing nodes that communicate heavily onto the same processor, these strategies produce mappings that avoid excessive IPC cost. Since processor assignments need only be assigned for each cluster, rather than for each node, clustering also reduces the time needed for scheduling.

The *linear clustering* technique, proposed by Kim and Browne [13], iteratively applies a critical path algorithm to transform the graph into a virtual architecture graph (VAG), which

consists of a set of linear clusters and the interconnections between them. A linear cluster is a degenerate tree in which every node has at most one immediate predecessor and one immediate successor. At each step, the algorithm groups together the most expensive directed path (in computation and communication) into a single linear cluster. The clustered nodes are removed from the graph and this process is iteratively repeated until the entire graph has been divided into clusters. After some refinement procedures, the algorithm applies graph-theoretic techniques to map the virtual architecture graph onto the physical processor architecture.

The *internalization* clustering method, suggested by Sarkar [3], clusters nodes together in an attempt to minimize the schedule length on an unbounded number of processors. The algorithm initially places each node in a separate cluster and considers the APEG arcs in descending order according to the amount of data transferred over each arc. Given arc A_{ij} (which connects nodes N_i and N_j), the algorithm merges the clusters containing these nodes ($C(N_i)$ and $C(N_j)$) to "internalize" any communications between nodes in these respective clusters. The algorithm accepts this cluster merging step if it does not increase the estimate of the parallel execution time of this clustered graph on an infinite number of processors. Otherwise, the clusters are unmerged and the next arc is considered.

The parallel execution time estimate for the given set of clusters is computed in a manner resembling classical CPM (critical path method) methodology. Forward and backward passes through the graph are used to find the earliest and latest start times for each node, where the communication costs between nodes in separate clusters are included. However, this procedure differs from the CPM techniques in that nodes in the same cluster are constrained to be executed sequentially on the same processor. To enforce this constraint, the algorithm sorts the nodes in each cluster in increasing order of their latest starting times and appends additional precedence constraints to ensure this ordering. When the list of APEG arcs is exhausted, a processor assignment phase uses a modified list scheduling approach to map the finished clusters to the physical processors. The procedure temporarily shifts each unassigned cluster onto each of the processors

in turn, estimates the parallel execution time in each case, and maps the cluster onto the processor that yields the minimum execution time estimate.

In addition to the advantages mentioned earlier, clustering procedures also bring several disadvantages. By grouping nodes into higher-granularity units, these techniques constrain the possible mappings of nodes to processors, limiting their ability to balance processor loads. Clustering algorithms also face significant difficulties when mapping clusters to the physical processor architecture. For example, consider the graph shown in figure 3, which naturally decomposes into the three clusters ABCDE, FGHI, and JKLM. If the architecture has three processors, this clustering method schedules the graph effectively, producing the schedule shown in figure 4 with makespan 21. Here, we have assumed that the targeted architecture is a shared-bus multiprocessor. For interprocessor communication, the source processor executes a send communication node (write to shared memory) and the destination processor executes a receive communication node (read from shared memory). We assume that the send and receive communication nodes

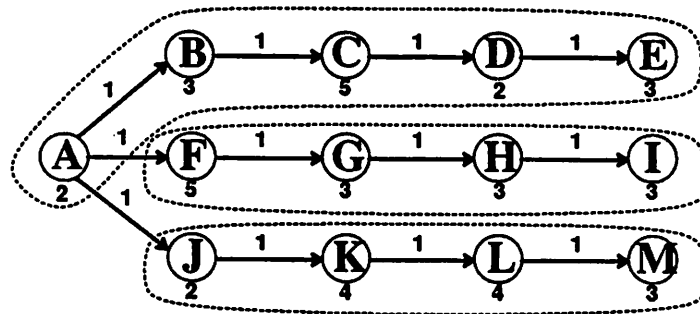


Figure 3. An APEG broken into three clusters

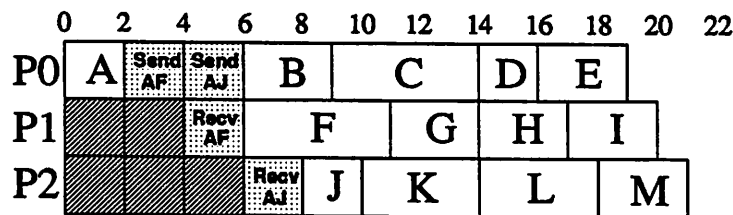


Figure 4. The clusters scheduled onto three processors

each take two time units per data sample, and we allow a send node for one communication to be overlapped with a receive node of another communication. These assumptions are made solely for purposes of illustration and are not intrinsic to the proposed algorithm itself. Now, if the graph in figure 3 is to be scheduled onto a two-processor architecture, the clustering scheme performs poorly, producing the schedule shown in figure 5 with makespan 30.

Both of the clustering methods described earlier will produce similar schedules, in which two clusters are scheduled on one processor and one cluster is scheduled onto the other processor. The load imbalance, which results in low processor utilization, is caused by the failure of these traditional clustering schemes to account for the number of processors in the architecture when choosing the granularity of the clusters. This difficulty in accounting for architectural considerations during scheduling is a primary motivation for the declustering approach. The question of whether a given instance of parallelism should be exploited requires consideration of the graph structure, the node granularity, the number of processors, the cost of IPC, and the structure of the processor interconnection topology. Furthermore, when processing resources are limited, one faces the additional problem of exploiting some parallelism instances at the expense of others.

General Algorithm Description

The declustering algorithm is specifically designed to address these scheduling considerations. The splitting of the algorithm into topology dependent and independent components accounts for the interconnection structure, and the scheduling of all communications as well as all computations eliminates contention for communication resources. The declustering algorithm is divided into four main sections, as shown in figure 6.

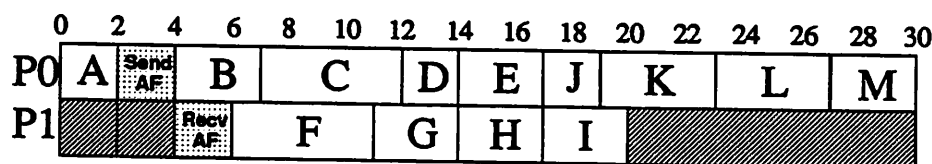


Figure 5. The clusters scheduled onto two processors

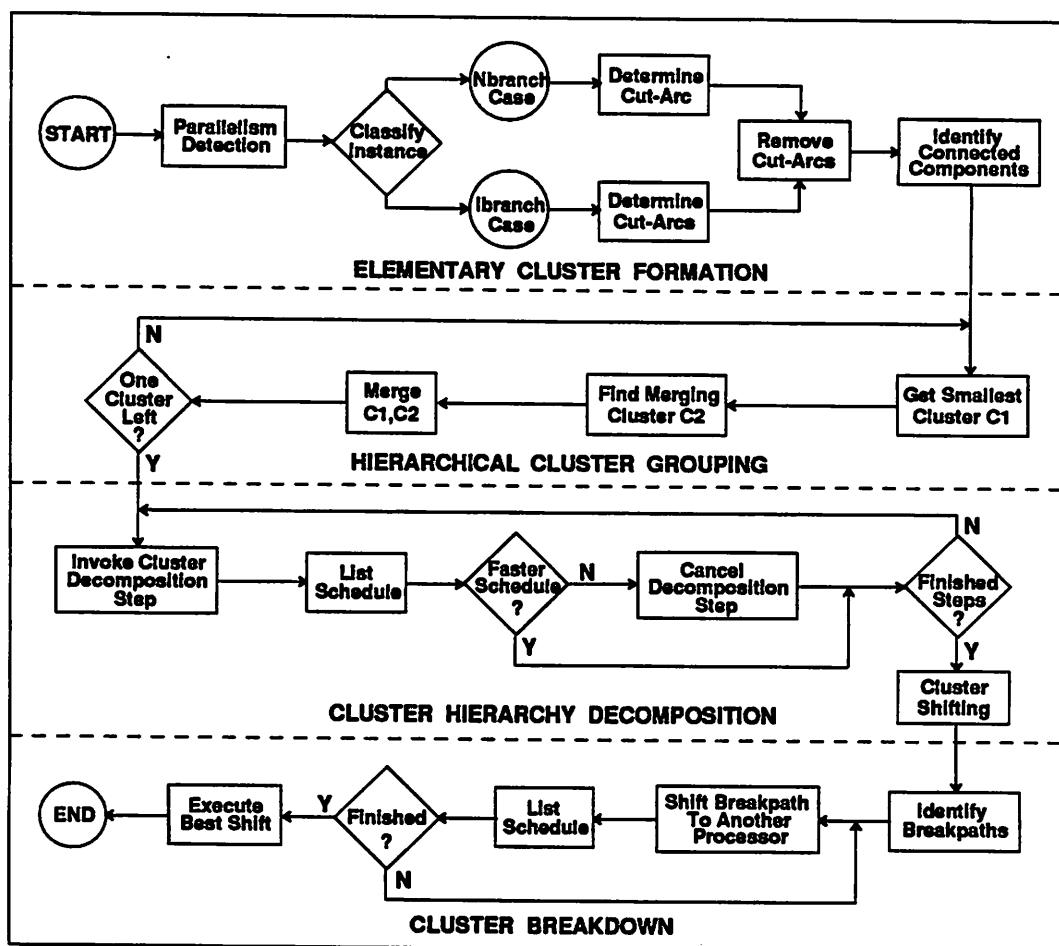


Figure 6. A flowchart description of the declustering algorithm

The first stage is a new clustering approach that divides the graph into elementary clusters using a novel set of parallelism analysis techniques. This phase considers the graph structure while explicitly addressing the tradeoff between exploiting parallelism and incurring IPC. The second stage combines the clusters in a hierarchical fashion by considering the intercluster IPC and parallelism relationships. It effectively ranks the instances of graph parallelism in preparation for declustering. The third stage decomposes the hierarchy by systematically breaking higher level clusters into their component sub-clusters and mapping one of the sub-clusters onto a different processor. This phase insures effective use of the available processors by exploiting the parallelism instances in order of importance. The fourth stage analyzes the best schedule obtained so far and breaks down the elementary clusters if additional flexibility is needed to

achieve an effective load balancing. By traversing the cluster hierarchy from top to bottom (large-grain to small-grain), this "declustering" process matches the level of cluster granularity to the characteristics of the specified architecture.

The algorithm is an iterative scheme that performs schedule analysis between scheduling iterations. We have found single-pass scheduling techniques to be unsuitable, due to the difficulty in evaluating the impact of a scheduling decision at any intermediate point in the scheduling process. Node placements that appear logical when viewed locally may produce a poor end result. To complicate matters further, scheduling exhibits a phenomenon known in the artificial intelligence community as the "horizon effect". This property asserts that no matter how far one looks ahead before making a local decision, there may exist something "just over the horizon" which can render the decision detrimental to the objective. To effectively decide node placements during the scheduling process involves performing computations that essentially amount to scheduling the graph several times. Rather than perform this type of calculation for each node placement, we advocate an approach that focuses computational energy in analyzing a schedule and finding the most promising alternative placements.

3. ELEMENTARY CLUSTER FORMATION

The elementary cluster formation phase consists of a new clustering procedure that decomposes the precedence graph into groups of nodes by isolating a collection of arcs that are likely candidates for separating the nodes at both ends onto different processors. These cut-arcs are temporarily cut, or removed from the graph and the algorithm designates each remaining connected component as an elementary cluster. These cut-arcs should be distinguished from the cutsets used by the network flow scheduling techniques pioneered by Stone [7]. Whereas each cutset in Stone's technique corresponds in a one-to-one fashion with a task assignment, the cut-arcs in this case represent promising locations for potential exploitation of parallelism; they do not determine task assignments.

The problem of finding an effective set of cut-arcs is complex. An insufficient number of cut-arcs constrains the possible processor assignments severely, causing reduced scheduling performance. Conversely, an overabundance of cut-arcs leads to an excessive time required for scheduling. The key to effective cut-arc selection lies in skillfully trading off the amount of parallelism exploited with the interprocessor communication cost incurred. Obtaining such a tradeoff requires both a method for detecting parallelism within the graph, and an effective means of comparing parallelism with communication cost. To clarify some terminology used in this section, a branch node is a node that has two or more immediate successors, while a merge node is a node possessing two or more immediate predecessors. The static level of node N_i , denoted $SL(N_i)$, is the sum of the execution times of all the nodes along the longest directed path from N_i to the terminal node N_t . This quantity is also commonly called the critical path length from node N_i to terminal node N_t .

3.1. Parallelism Detection

To enable parallelism detection, we find for each node N_i , its reachability set $RS(N_i)$, defined as the set of nodes (excluding dummy terminal node N_t) reachable through a directed path from N_i . If node N_i lies in the reachability set of node N_j , this implies that a precedence constraint exists from N_j to N_i . Stated another way, nodes N_i and N_j can be executed in parallel if and only if each node is not contained in the other's reachability set. To reveal parallelism between paths of nodes, we use a "divide and conquer" approach that considers two paths at a time. Recognizing that parallelism is created at branch nodes, we focus attention on paths that diverge from these nodes. The branch nodes in the graph are sorted smallest static level first, so that the branch nodes near the end of the graph are initially considered. For each branch node N_i , the algorithm obtains a list of its immediate successors $IS(N_i)$ and sorts them largest static level first. At each step, the procedure considers the first two successors in the list and computes the intersection of their reachability sets to categorize this instance into one of two classes, called the

Nbranch and *Ibranch* cases respectively. We illustrate these cases using an example in which the algorithm is considering branch node A and its immediate successors B and C.

Nonintersecting Branch Case

If $RS(B) \cap RS(C)$ is empty, the paths stemming from B and C never combine. We classify this instance into the *Nbranch* (Nonintersecting branch) case and isolate the longest paths from nodes B and C to dummy terminal node N_t (but not including N_t) for parallelism consideration. In the example *Nbranch* case shown in figure 7, the available parallelism between the two paths is $\min \{SL(B), SL(C)\}$. This is the maximum overlap in execution time possible if the longest paths starting from B and C are separated onto different processors.

Intersecting Branch Case

If $RS(B) \cap RS(C)$ is not empty, this instance fits into the *Ibranch* (Intersecting branch) case because the paths stemming from nodes B and C combine at some point. The node in the intersection with largest static level is the first merge node at which paths starting from B and C combine. The algorithm isolates the longest path from B to the merge node and the longest path from C to the merge node for parallelism consideration. This is illustrated in figure 8, where node Z is the merge node. If we designate the string of nodes from B to X as path1 and the string of nodes from C to Y as path2, the available parallelism is the minimum of the sum of node exe-

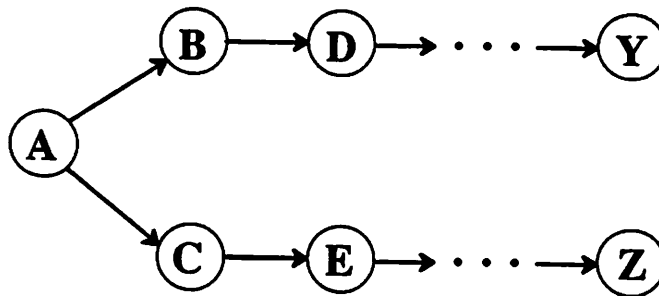


Figure 7. The nonintersecting branch case

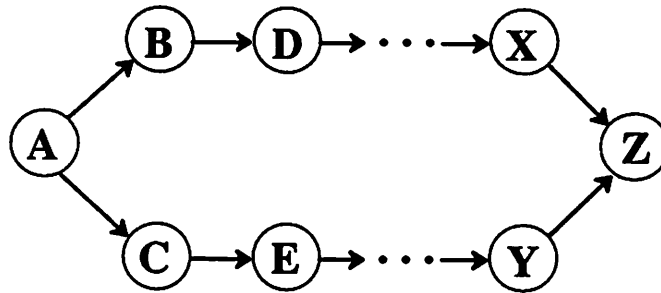


Figure 8. The intersecting branch case

cution times in path1 and the sum of node execution times in path2.

3.2. Cut-arc Determination

This method of categorizing two-path parallelism instances into the Nbranch and Ibranch classes provides a tractable means of determining cut-arcs, because finding the minimum makespan solution for each isolated parallelism instance is a straightforward calculation in each case. If the parallelism instance can execute more rapidly on two processors than on one processor (large communication costs may preclude this possibility) the algorithm finds the single arc that will be cut in the Nbranch case, or the two arcs that will be cut in the Ibranch case.

3.3. Algorithm Description

To make the elementary cluster formation algorithm as lucid as possible, we first demonstrate the clustering process on the example graph in figure 1. We then close this section by displaying the steps of this procedure in an algorithmic C-like syntax in figure 14.

To break the graph in figure 1 into two-path parallelism instances, the algorithm first identifies nodes A and I as branch nodes. Node I, which is examined first, is found to have two immediate successors M and N, where $RS(M) \cap RS(N) = \phi$. This two-path parallelism instance, shown in figure 9, therefore fits into the Nbranch case. Again, assuming the same shared-bus architecture mentioned earlier, the algorithm quickly ascertains that separating paths {I-M-Q} and {N-R} onto separate processors leads to the minimum makespan solution. It therefore adds

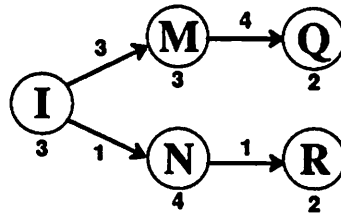


Figure 9. A nonintersecting branch case

arc IN to the list of cut-arcs and moves on to examine branch node A. Since node A has more than two immediate successors, the procedure sorts them into a list largest static level first : $IS(A) = [E F D]$. At each step, it considers the two remaining successors with largest static level, which are initially E and F in this case. After finding that $RS(E) \cap RS(F) = \phi$, the algorithm classifies this instance into the Nbranch case and isolates the two longest paths stemming from nodes E and F for parallelism consideration, as shown in figure 10. Here, the optimum solution is to cut arc AF. Since this cut-arc lies in the path containing successor node F, the algorithm removes node F from the list of successors, so that $IS(A) = [E D]$.

The algorithm next examines the two successor nodes, D and E, and determines that node Q is the first merge node in $RS(D) \cap RS(E)$, at which the paths starting from D and E combine. This Ibranch case, shown in figure 11, requires two cut-arcs to split paths onto separate processors. The algorithm first calculates the makespans for cutting pairs of arcs in the four "corner" cases (AD, LQ), (AD, MQ), (AE, LQ), and (AE, MQ), which are pairs of arcs directly connected to the branch node or merge node. The procedure uses the shortest makespan obtained over the

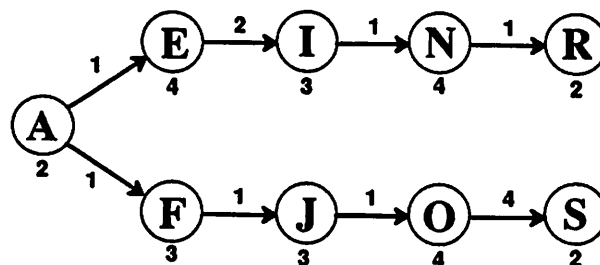


Figure 10. Another nonintersecting branch case

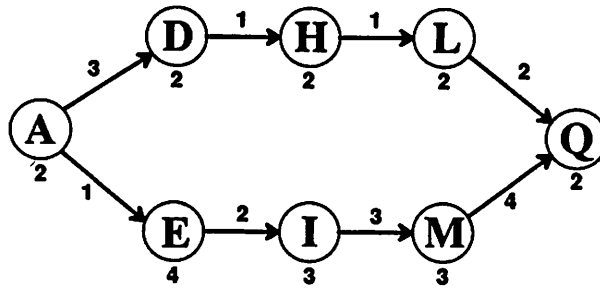


Figure 11. An intersecting branch case

corner cases as a bound to quickly eliminate other eligible pairs of cut-arcs. Due to the excessive communication costs, the minimum makespan solution is to leave all arcs uncut; that is, schedule the entire structure on a single processor. Since no path is cut, the algorithm deletes the successor with smaller static level from $IS(A)$ to prevent the same two paths from being analyzed again, and moves on to examine the next two nodes in $IS(A)$. If only one successor remains, the routine passes to the next branch node.

After exhausting the branch nodes, the algorithm connects a dummy start node N_s to all the initial nodes in the graph, and computes static levels and reachability sets for each node in the reverse direction (right to left), denoting these quantities as $SLR(N_i)$ and $RSR(N_i)$ respectively. The procedure identifies all the merge nodes in the graph and repeats the procedure applied to the branch nodes, except that path analysis proceeds in the opposite direction (This is equivalent to inverting the direction of every directed arc in the graph, recomputing the static level and reachability set for each node, and repeating the branch analysis procedure on this inverted graph). In addition to treating cases in which the parallelism is initially extant, this consideration of merge nodes handles situations in which two paths combine in several places. Since the parallelism detection strategy only considers the first merge node (with largest static level) when isolating an lbranch case, the parallel paths combining at the later merge nodes would be ignored if all the merge nodes were not identified and analyzed.

Continuing our example, the algorithm examines merge node F and its two immediate predecessors A and B. After classifying this instance into the nonintersecting merge (Nmerge) case, the procedure determines that the best solution is not to cut any arcs. When considering merge node Q, the algorithm recognizes that this Imerge case has already been examined as an Ibranch case for node A and skips on to consider merge node S and its two immediate predecessors O and P. Since $RSR(O) \cap RSR(P) = \phi$, the algorithm classifies this instance into the Nmerge case, traces the two longest paths to the dummy start node as illustrated in figure 12, and isolates KP as the optimal cut-arc. Notice that arc KP is not directly connected to a branch or merge node. The significance of this point will become apparent when we compare clustering schemes in section 7.

After all the cut-arcs have been determined, the algorithm temporarily removes them from the graph and invokes a depth-first search to isolate the connected components remaining in the graph. Each remaining component is designated as an elementary cluster. After cutting arcs IN, AF, and KP, the elementary clusters in our example are shown below in figure 13. Selecting an arc to be a cut-arc does not force the nodes at each end onto separate processors; rather, the cut-arcs represent promising locations for splitting paths onto different processors. The actual mapping decisions are made in later phases of the algorithm.

We summarize the steps taken by the elementary cluster formation phase in figure 14.

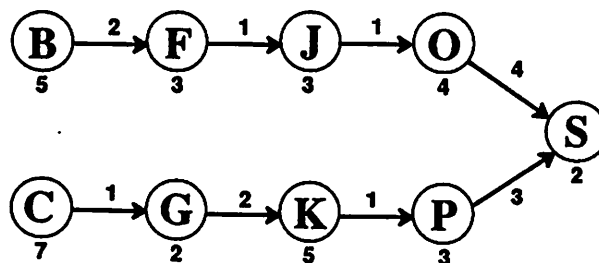


Figure 12. A nonintersecting merge case

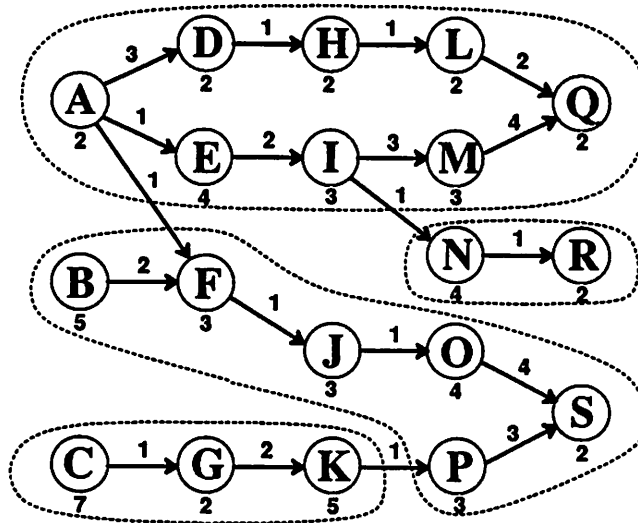


Figure 13. The elementary clusters in the graph

INPUT: The acyclic precedence expansion graph
OUTPUT: A set of elementary clusters

0) Find the static level (SL) and reachability set (RS) for each node

1) Sort graph branch nodes smallest SL first

2) For each branch node BN {

Sort immediate successors of BN into list IS(BN), largest SL first

while ($|IS(BN)| \geq 2$) {

IS1 = first node in IS(BN), IS2 = second node in IS(BN)

if ($INTSET = RS(IS1) \cap RS(IS2) = \emptyset$) /* NBranch case */

Trace paths from IS1 and IS2 to terminal node N_i

Find the optimal cut-arc for this 2-path parallelism instance

if (cut-arc is null) /* Excessive communication cost */

Remove the successor (IS1 or IS2) with smaller SL from IS(BN)

else

Remove the successor (IS1 or IS2) closest to the cut-arc from IS(BN)

}

else { /* IBranch case */

Identify merge node M as the node with highest SL in INTSET

Trace paths from IS1 to M and from IS2 to M

Find the 2 optimal cut-arcs for this 2-path parallelism instance

if (cut-arcs are null) /* Excessive communication cost */

Remove the successor (IS1 or IS2) with smaller SL from IS(BN)

else

Remove the successor (IS1 or IS2) closest to a cut-arc from IS(BN)

}

}

3) Invert the direction of every graph arc and repeat steps 0 - 2

4) Temporarily remove the cut-arcs from the graph

5) Find the connected components remaining in the graph

(Each remaining connected component is an elementary cluster)

Figure 14. The elementary cluster formation algorithm

4. HIERARCHICAL CLUSTER GROUPING

The hierarchical cluster grouping stage combines existing clusters in a pairwise fashion until a single cluster remains that contains every node in the graph. This procedure establishes a parallelism hierarchy, effectively sorting the graph parallelism by importance in preparation for declustering.

This procedure initially sorts the elementary clusters by the sum of the execution times of the nodes they contain (smallest first). This allows the smallest cluster, say C_1 , to be considered for merging at each step. The algorithm determines the cluster that communicates most heavily with C_1 by examining the intercluster cut-arcs. Ties are broken by arbitrarily selecting the cluster with smaller total execution time. The algorithm proceeds to merge C_1 and its selected cluster into a new larger cluster, and records this cluster combination step (e.g. $[C_1 + C_2 = C_3]$) in order of execution for future reference. After a cluster combination step is invoked, the algorithm removes the two component clusters from the list of current clusters, adds the new larger cluster to the list in sorted order, and considers the next smallest cluster for merging. When only a single cluster remains, the algorithm schedules this cluster onto a processor (we arbitrarily pick P_0) to obtain an initial makespan. These steps are summarized below in figure 15.

```

INPUT : The elementary clusters
OUTPUT : An ordered list of cluster combination steps
Let  $C(i)$  denote the set of clusters at time step  $i$ 
0)  $i = 0$ ;  $C(0)$  = set of elementary clusters
1) while ( $|C(i)| > 1$ ) {
     $C1(i)$  = Smallest cluster (smallest sum of node execution times) in  $C(i)$ 
     $C2(i)$  = Cluster in  $C(i)$  that communicates the most data with  $C1(i)$ 
              (Break ties by selecting smaller cluster)
    Merge clusters  $C1(i)$  and  $C2(i)$  to get new cluster  $C3(i)$ 
    Store cluster combination step  $[C1(i) + C2(i) = C3(i)]$ 
    Remove  $C1(i)$  and  $C2(i)$  from  $C(i)$ 
    Add  $C3(i)$  to cluster set  $C(i)$ 
     $i = i + 1$ 
}

```

Figure 15. The hierarchical cluster grouping algorithm

The order in which the clusters are combined is important. The procedure selects the smallest cluster for combination at each step because combining this cluster with another does not suppress much parallelism. While this technique incorporates interprocessor communication considerations, it builds clusters in such a fashion to maintain as much parallelism as possible for as long as possible. The result is that the combination steps near the end merge clusters that have the largest amounts of parallelism between them, subject to the communication pattern in the graph. This effectively imposes a ranking of the instances of parallelism present in the graph, where the combination steps near the beginning suppress the less important parallelism instances, and the combination steps near the end suppress the most prominent parallelism instances.

5. CLUSTER HIERARCHY DECOMPOSITION

The cluster hierarchy decomposition phase begins the declustering process, in which the parallelism hierarchy constructed in the first two stages is decomposed into successively smaller levels. To avoid the inflexibility induced by the clustering process, the algorithm traverses cluster granularity levels from large to small to find the level that is roughly consistent with the characteristics of the target architecture.

This procedure examines the list of cluster combination steps in reverse order, so that the last consolidation step (say $[C_{18} + C_{19} = C_{20}]$) is considered first. It then invokes a decomposition step on C_{20} by shifting either subcluster C_{18} or C_{19} onto a selected group of candidate processors, which are selected by the topology-dependent section by considering the cut arcs of the chosen subcluster and the finishing times of the processors. The algorithm shifts the chosen subcluster onto each of the candidate processors in turn, and list schedules [10] the graph to determine the makespan for each of the subcluster placements.

In contrast with the normal list scheduling approach, our list scheduling method schedules all communications as well as all computations. The topology-dependent section of the scheduler contains a routing algorithm tailored for the particular architecture, which routes a path between source and destination processors. The scheduler eliminates the possibility of communication

resource contention by reserving all the necessary resources for the duration of the data transmission. Our list scheduling method is also unique in that it has no global timeclock to distinguish nodes as being runnable or processors as being available. A node is runnable if all its immediate predecessors have already been scheduled, and the processor assignments for each node are already fixed by the declustering algorithm before scheduling is invoked. Since there are many possible orderings of nodes on each processor, a given processor assignment can lead to many possible schedules. Static levels are used as priorities to determine the ordering, so that at each step, the node with highest priority is scheduled on its assigned processor. Dynamic levels, which were used to select node-processor assignments in [8], are not necessary here because of the fixed processor assignments.

After decomposing a cluster by splitting one of its component clusters onto another processor, the algorithm accepts the decomposition step if a faster schedule is obtained. If a tie in makespan occurs, the first tiebreak criterion selects the schedule with less IPC, and the second criterion selects the schedule with a smaller sum of processor finishing times. If the step is accepted, the procedure saves the new schedule and records a new processor location for the shifted subcluster. Otherwise, it ignores the step and considers the next cluster combination step (in reverse order) for decomposition. This process is repeated until all the cluster combination steps have been considered. This procedure constrains the number of processor placement permutations that are examined, because each cluster remains in its default position if its decomposition step is discarded.

When the list of cluster combination steps is exhausted, the algorithm invokes the two cluster shifting procedures described in Appendix I, which search for better processor placements at the current level of cluster granularity. These techniques are repeated on successively lower levels of granularity until reaching the elementary clusters. This strategy is consistent with the overall approach in examining placements at the higher levels of cluster granularity before examining placements at the lower granularity levels. This top-down strategy helps avoid

overlooking the situation in which shifting two lower granularity component clusters individually does not produce a better result, but shifting these components together results in a better placement. If scheduling time is a critical factor, these cluster shifting techniques can be bypassed without loss of continuity; however, this omission incurs some loss of scheduling performance.

Rather than examine alternative placements in a haphazard fashion, the cluster shifting techniques attack the **schedule limiting progression (SLP)**, the progression of nodes and communications that inhibits attainment of a faster schedule. This progression may span several processors, but it cannot contain any idle time. The SLP is a function of the particular schedule, and depends on the effectiveness of the scheduling procedure as well as the characteristics of the targeted architecture. In contrast with the critical path, the SLP is sensitive to changes in the number of processing and communication resources. An example schedule is shown below in figure 16, with the SLP marked by the sequence of arrows.

Although it may seem unusual to build and then tear down a parallelism hierarchy, this procedure provides several benefits. The sorting of graph parallelism that occurs during hierarchy construction allows effective use of the available processors. As each new processor is pressed into service, the algorithm assigns it the largest section of unrealized parallelism remaining in the graph that has the smallest IPC cost. Faced with the inevitable problem of having to exploit some parallelism instances at the expense of others, the declustering algorithm insures that the most significant sections of graph parallelism are assigned first. The less important parallelism instances will be exposed later if there are sufficient resources remaining to warrant exploitation.

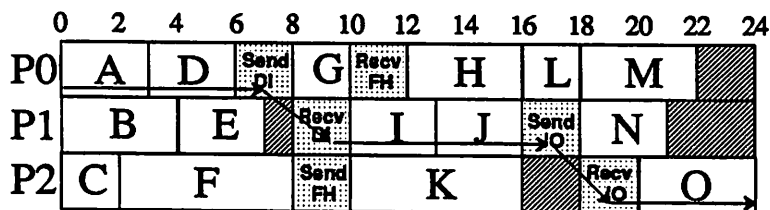


Figure 16. An example schedule with the SLP marked using arrows

By sweeping through a range of cluster granularities during cluster decomposition, this approach also performs natural partitioning that automatically adapts to the grain size of the nodes. When the cluster granularity reaches the point where IPC costs negate any gains realized in exploiting parallelism, the algorithm discards any further decomposition steps. So if the nodes are too fine-grained for the architecture to effectively use the internode parallelism, this process keeps the hierarchy at a higher level that is conducive to parallelism exploitation. The cluster hierarchy decomposition phase of the algorithm is outlined below in figure 17.

6. CLUSTER BREAKDOWN

The cluster breakdown phase supplies the capability necessary for effective load balancing when the appropriate cluster granularity is smaller than the level of the elementary clusters. Since breaking clusters incurs IPC cost, the cluster breakdown phase only decomposes clusters if this action produces a faster schedule; that is, if the gain from load balancing exceeds the additional communication cost.

The procedure first identifies the SLP nodes that are connected to at most one other SLP node on the same processor. Each of these SLP edge nodes denotes a starting point for a

```

INPUT1 : A single cluster containing every node, mapped onto P0
INPUT2: A list of cluster combination steps, ordered most recent first
OUTPUT : The best (shortest) schedule seen so far
0) Initial BestSchedule = Schedule every node on P0
1) For each cluster combination step (C1 + C2 = C3) {
    Let Cshift be the smaller of clusters C1 or C2 in execution time
    BestProcessor = processor that Cshift is currently on
    Ask topology-dependent section for candidate processors for Cshift
    For each candidate processor P(i) {
        Shift every node in Cshift onto P(i)
        List schedule the graph to get NewSchedule
        If ((NewSchedule period < BestSchedule period) ||
            (NewSchedule period = BestSchedule period) &&
            (NewSchedule has less IPC than BestSchedule))) {
            BestSchedule = NewSchedule
            BestProcessor = P(i)
        }
    }
    Shift every node in Cshift onto BestProcessor
}

```

Figure 17. The cluster hierarchy decomposition algorithm

breakpath, where a breakpath refers to a portion of the SLP that is a good candidate for being shifted onto another processor. Starting from an edge node, the procedure extends a path one node further into the SLP on the same processor. The current path of nodes is considered a breakpath if the sum of execution times in the path falls between derived lower and upper bounds. The lower bound insures that any gain created by shifting nodes onto a different processor exceeds the net IPC cost. The upper bound, set by considering the processor loads, insures that potential shifts have a possibility of obtaining a faster schedule. For each edge node, the algorithm finds the sequences of nodes that satisfy the bounding requirements. It individually switches each breakpath onto another processor, obtains the makespan, and executes the split that produces the fastest schedule.

To illustrate this procedure, consider once again the APEG in figure 3, which was split into three elementary clusters. Although this example maps onto three processors effectively by scheduling one cluster on each processor, the fastest two-processor schedule obtained after cluster hierarchy decomposition has makespan 30, as shown in figure 18. The load imbalance occurs because the granularity of the elementary clusters is too large for the two processor case; a smaller cluster granularity is required for effective processor utilization. The SLP, which lies entirely on processor 0, consists of nodes {A SendAF J B K C L D E M}. The algorithm analyzes the two edge nodes in this group, E and M, and uses the bounding techniques to obtain the possible breakpaths {E, DE, CDE, M, LM, and KLM}. After shifting processor assignments and list scheduling the graph in each of these cases, the procedure determines that splitting DE

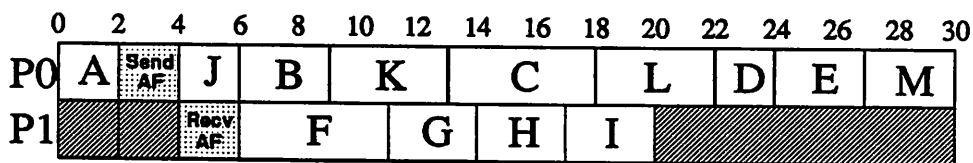


Figure 18. The fastest schedule obtained after cluster shifting

onto processor 1 gives an optimal schedule, as shown in figure 19. This ability to break down cluster granularity is essential, because as illustrated in the preceding example, the most effective cluster granularity is determined by the characteristics of the architecture. The steps taken by the cluster breakdown phase are shown below in figure 20.

This cluster breakdown technique, although intrinsic to the declustering process, is applicable to other clustering methods. After clustering nodes to handle IPC cost, it is often useful to decompose clusters to gain flexibility for load balancing. The cluster shifting and breakdown techniques can be iteratively repeated upon successively lower levels of cluster granularity.

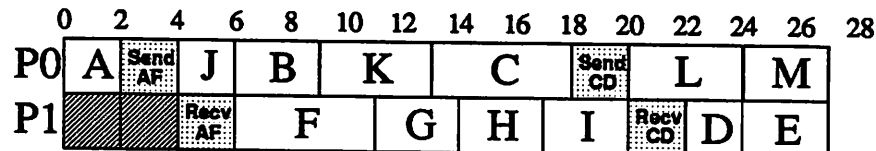


Figure 19. The schedule after cluster breakdown

```

INPUT: The elementary clusters and the current BestSchedule
OUTPUT : The final BestSchedule
1) Compute schedule limiting progression (SLP)
2) Identify SLP edge nodes
3) For each SLP edge node {
    Initialize current path (Cpath) to contain just the edge node
    CpathProc is the processor that Cpath is on
    CONTINUE = TRUE
    while (CONTINUE == TRUE) { /* Look at SLP on CpathProc */
        LBound = Estimated IPC cost to shift CPath to a different processor
        if (Cpath execution time sum > LBound) {
            candProcs = candidate processors to shift CPath onto
            For each processor P(i) in candProcs {
                Set UBound[P(i)] to insure that shifting CPath to P(i) has
                the possibility of obtaining a faster schedule
                if (Cpath execution time < UBound) {
                    Shift Cpath to P(i) and list schedule the graph
                    If (faster schedule) { Save schedule in BestSchedule }
                }
                else { CONTINUE = FALSE
            }
        }
    }
    else if (there are SLP nodes on CpathProc that are not in Cpath) {
        Extend Cpath by 1 node further into the SLP on CpathProc
    }
    else { CONTINUE = FALSE
}
}
}

```

Figure 20. The cluster breakdown algorithm

7. SCHEDULING RESULTS

To test the scheduling effectiveness of the declustering technique, we scheduled several digital signal processing algorithms onto a shared bus multiprocessor containing four DSP56001 processors. Since there is no dedicated communication hardware in this architecture, each processor must execute the send (write shared memory) and receive (read shared memory) communication nodes for interprocessor communication. We invoked the declustering algorithm, Sarkar's internalization algorithm, and a modified version of Kim and Browne's linear clustering algorithm on each of these DSP applications to compare scheduling performance. The modification to the linear clustering algorithm was necessary because the method used to assign clusters to processors was not described clearly enough for implementation. Our modified version, which we refer to as the critical-path clustering algorithm, uses the linear clustering method to determine a set of clusters, but substitutes phases 2 and 3 of the declustering algorithm for processor assignment. We tested these scheduling techniques using four different signal processing algorithms: two sound synthesis programs, a telephone channel simulator, and a 16-QAM (quadrature amplitude modulation) transmitter. The schedule makespans (in processor cycles) are shown in table 1 for each case. The declustering technique produced the best result (shortest schedule length) in each instance.

By comparing the times required to construct each schedule, as shown in table 2, we see that the critical-path clustering technique is the quickest and the internalization approach the slowest of these algorithms. If n represents the number of nodes and p represents the number of

SCHEDULE LENGTHS IN PROCESSOR CYCLES			
DSP Algorithm(#nodes)	Declustering	C-P Clustering	Internalization
Sound Synthesis I (26)	173	179	218
Sound Synthesis II (27)	170	214	210
Telephone Channel Simulator (67)	297	297	488
QAM Transmitter (411)	4661	4881	5024

Table 1. Schedule lengths in processor cycles for 4 DSP algorithms

processors, the critical path algorithm has complexity $O[n^3p]$, while the internalization algorithm has complexity $O[n^3(n+p)]$. The declustering algorithm also has complexity $O[n^3(n+p)]$, because the elementary cluster formation phase is $O[n^4]$, the hierarchical cluster grouping phase is $O[n^2]$, and the cluster hierarchy decomposition and cluster breakdown phases are both $O[n^3p]$.

These signal processing algorithms were all homogeneous graphs, meaning that each arc passes the same number of data units. For such graphs, the declustering algorithm usually produces the same set of clusters as the critical-path clustering algorithm. Since the communication penalty in breaking any arc in the graph is exactly the same, the graph will almost always be broken in locations that expose the greatest amount of parallelism, namely the arcs output from a branch node or input to a merge node. The ability of the declustering algorithm to break arcs not directly connected to a branch or merge node lies unused. To investigate the more interesting nonhomogeneous case, we randomly generated a set of 100 APEGs containing between 70 and 140 nodes, where the node execution times were uniformly distributed over $[4, 20]$, and the number of data units assigned to each arc were uniformly distributed over $[1, 5]$. These graphs were scheduled onto shared-bus architectures containing 4, 6, 8, 10, and 12 processors respectively, where the same communication protocol used in this paper was assumed. The declustering algorithm again demonstrated the best scheduling performance, and the average percentage speedup improvements (analogous to average percentage improvements in makespan) of the declustering algorithm over the other two clustering techniques are shown in figure 21.

TIME REQUIRED FOR SCHEDULING IN SECONDS			
DSP Algorithm(#nodes)	Declustering	C-P Clustering	Internalization
Sound Synthesis I (26)	5.65	2.35	5.60
Sound Synthesis II (27)	3.16	1.72	4.34
Telephone Channel Simulator (67)	34.16	19.50	48.32
QAM Transmitter (411)	1523.20	265.72	5024.46

Table 2. Time required to construct a schedule (in seconds) for each scheduling technique

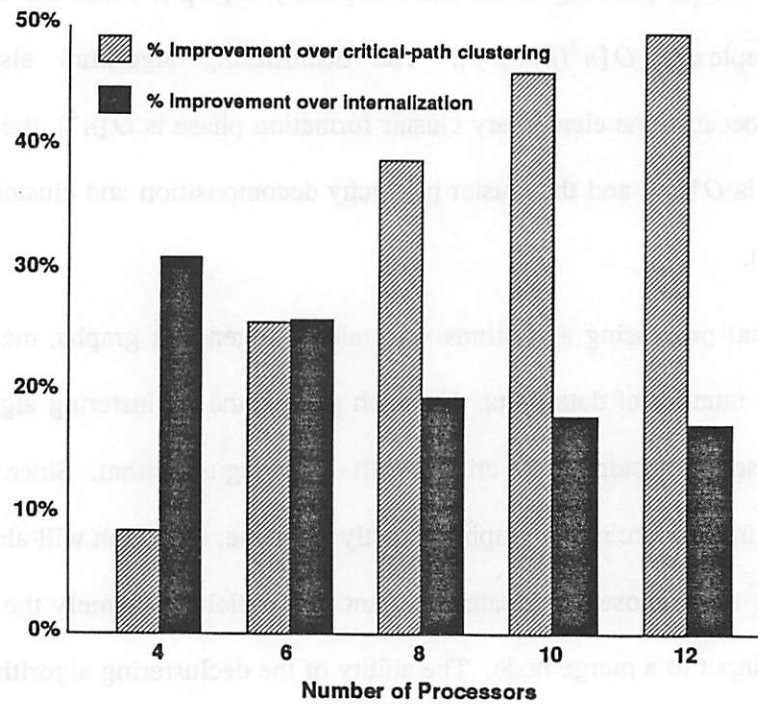


Figure 21. Percentage improvement in speedup of declustering over critical-path clustering and internalization

To supplement these results, we generated another set of APEGs containing between 170 and 260 nodes, where the node execution times were uniformly distributed over $[4, 40]$, and the number of data units on each arc were uniformly distributed over $[1, 10]$. We scheduled these graphs onto a simulated banyan switching network multiprocessor connecting eight processors with eight memory modules. This architecture required four processor cycles to communicate each data unit via shared memory, making the average node execution time equal to the mean communication time. Here, we found the average speedup improvement of declustering over critical-path clustering and internalization to be 23.3% and 16.1% respectively.

One reason for the performance advantage of declustering over critical-path clustering is the difference in clustering approaches. The graph parallelism analysis techniques allow the declustering algorithm to break arcs not directly connected to branch or merge nodes, a capability that critical-path clustering lacks. The declustering algorithm gains additional improvement by directly attacking the scheduling limiting progression rather than the critical path, because the

two are often different. The increasing speedup improvement as additional processors are added is primarily due to the enhanced load-balancing capability obtained through declustering. The graph-analysis clustering technique also surpasses the clustering performance of the internalization method. Although the internalization procedure is more flexible than the critical-path clustering technique, it occasionally "overclusters" the graph by combining nodes that should remain separate. This effect is caused by the algorithm's inherent ambiguity in deciding the order in which arcs that pass the same number of data units should be considered. Lacking a global view of the graph, the internalization procedure often merges arcs in a suboptimal order.

To illustrate why the parallelism-analysis clustering technique in the first stage of the declustering algorithm outperforms the critical-path clustering and internalization approaches, consider the graph example shown below in figure 22, which will be scheduled onto a two-processor shared-bus configuration. The declustering algorithm invokes the graph analysis techniques presented in section 3. After identifying nodes B and F as the immediate successors of branch node A, the procedure finds that cut-arc AB gives the minimum makespan solution for this Nbranch case. The algorithm proceeds to merge node I, identifies its immediate predecessors as nodes F and H, and determines that cutting arc HI yields the optimum solution in this Nmerge case. The elementary clusters are shown in figure 23, and the resulting schedule with makespan 24 is shown in figure 24.

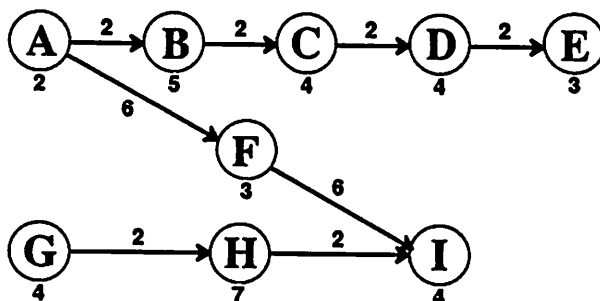


Figure 22. An example graph

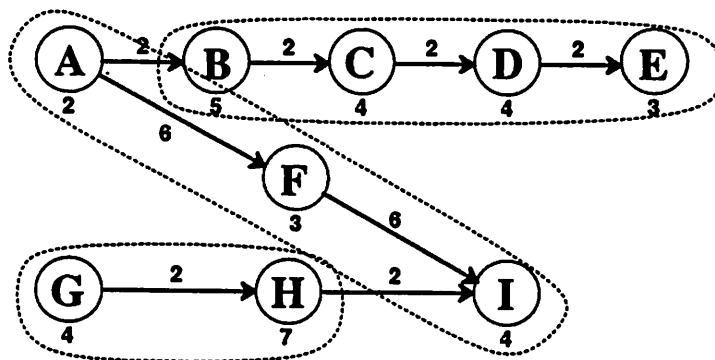


Figure 23. The declustering algorithm's elementary clusters

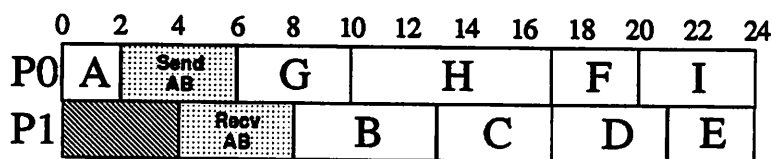


Figure 24. The declustering schedule

The critical-path clustering algorithm successively clusters paths {A-B-C-D-E}, {G-H-I}, and {F} to produce the set of clusters shown below in figure 25. The best placement of these clusters results in the schedule with makespan 30 shown in figure 26. It is immediately apparent that clustering the critical path can force a large communication to occur. Regardless of whether cluster F is grouped with cluster ABCDE or cluster GHI, this cluster composition forces a communication of 6 data units either between nodes A and F or nodes F and I. This occurs because

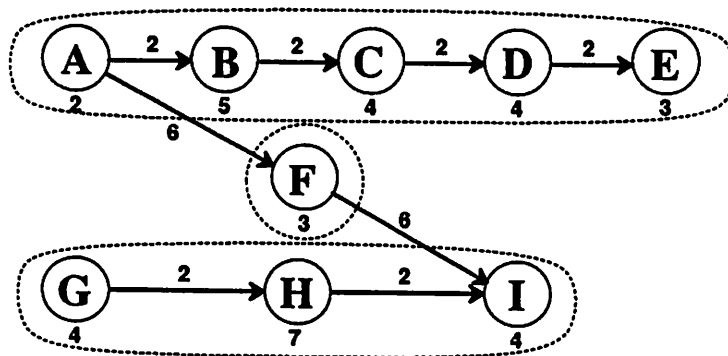


Figure 25. The clusters obtained through critical-path clustering

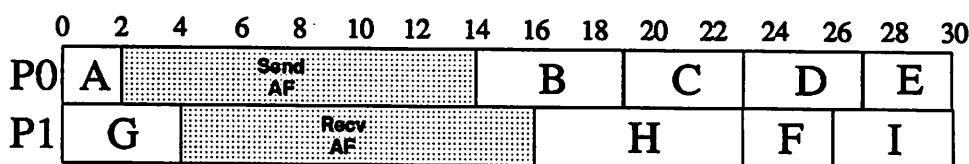


Figure 26. The critical-path clustering schedule

the algorithm treats each path as a complete entity, ignoring the fact that communication should weigh more heavily than computation for clustering purposes. Clustering the entire critical path together may be unnecessary. Since the critical path is often different from the schedule limiting path, clustering only part of the critical path (e.g. BCDE) may lead to a more effective solution.

The internalization algorithm initially invokes clustering steps $[A + F = AF]$ and $[AF + I = AFI]$, which avoids the transfers of six data units. It then executes $[AFI + B = ABFI]$, $[ABFI + C = ABCFI]$, $[ABCFI + D = ABCDFI]$, $[ABCDFI + E = ABCDEFI]$, $[G + H = GH]$, $[ABFI + H = ABFHI]$, and $[CDE + G = CDEG]$, to produce the clusters shown in figure 27. The cluster placement procedure leads to the schedule with makespan 29 shown in figure 28. The ordering of arcs

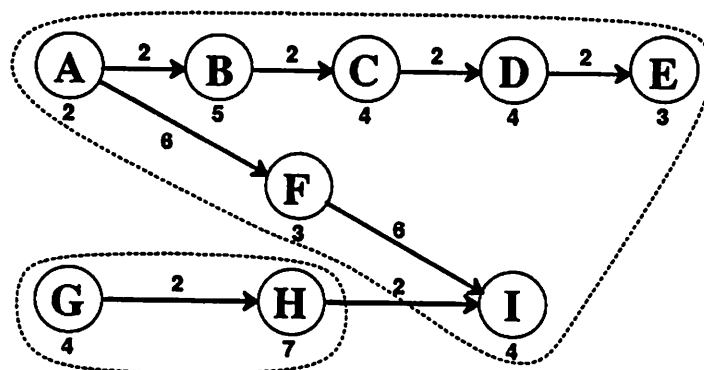


Figure 27. The clusters obtained through internalization

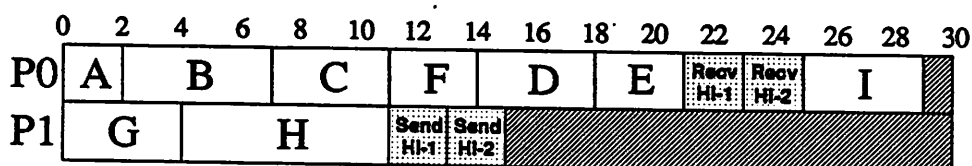


Figure 28. The internalization schedule

plays a major role in determining the clusters. In this example, there are 6 arcs that pass 2 data units each, and the order in which these arcs are considered determines the cluster composition. The ordering {DE CD BC GH HI AB} produces the same clusters as the declustering technique, while the ordering {AB HI BC GH CD DE} produces clusters ABCFGHI and DE. This ordering dependence reflects the local view exhibited by the internalization algorithm. By considering each arc individually, the algorithm loses its global perspective on how each merge affects the total cluster structure.

A quick comparison of these schedules illustrates the global scheduling perspective taken by the declustering algorithm. Instead of immediately invoking the two initially executable nodes A and G on processors P0 and P1 respectively, the declustering algorithm maps both nodes onto P0 and idles P1 for four time units to obtain the optimal schedule.

Next, consider the graph shown in figure 29, which will be scheduled onto a 3 processor shared-bus topology. The declustering algorithm identifies nodes C and D as the immediate successors of branch node A and determines that arc AC is the optimal cut-arc for this Nbranch case. Arcs FJ and IL are subsequently identified as the two best cut-arcs for the Ibranch case involving nodes B, E, and F. The algorithm next turns to merge node L, which has three immediate prede-

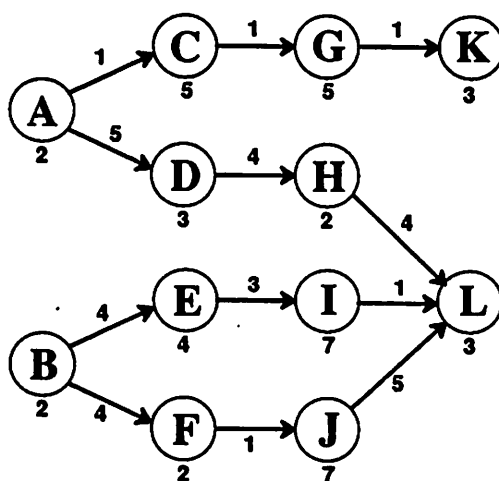


Figure 29. Another example graph

cessors H, I, and J. Since I and J have the highest values of $SLR(N_i)$, the procedure initially considers the Imerge parallelism instance involving nodes I, J, and L. Recognizing that this Imerge case was already examined when considering branch node B, the algorithm deletes node I from the list of immediate predecessors and proceeds to the Nmerge case involving nodes H, J, and L. This instance gives cut-arc FJ, which was already selected earlier. The elementary clusters, shown below in figure 30, result in a schedule with makespan 24. Notice the irregular structure of clusters BEFI and ADHJL. To avoid excessive IPC cost, the algorithm suppresses some of the available parallelism in the graph by clustering together nodes that can be executed in parallel.

The critical-path clustering method iteratively clusters paths {A-D-H-L}, {B-E-I}, {C-G-K}, and {F-J} to obtain the set of clusters shown below in figure 31, which results in a schedule with makespan 32. Unlike the declustering algorithm, the technique of iteratively clustering linear paths cannot break arc FJ because it is not directly connected to a branch or merge node. It is therefore forced to break arcs BF and JL that transfer four and five units of data respectively. This inability to break arcs in the middle of a sequential string can cause large scheduling inefficiencies for nonhomogeneous graphs.

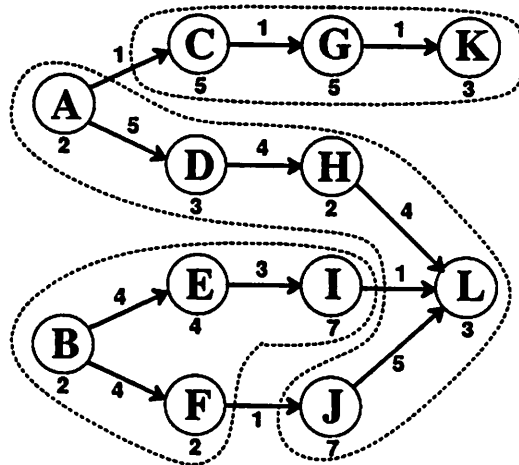


Figure 30. The declustering clusters

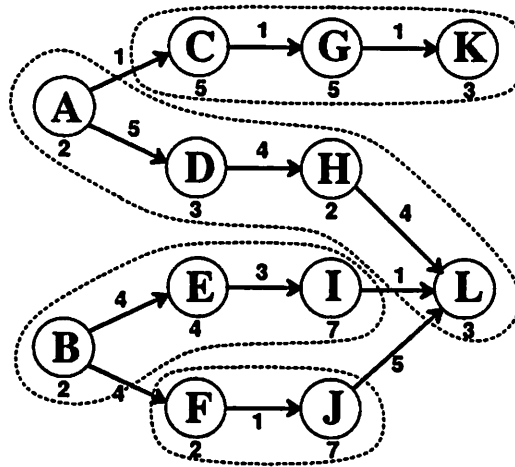


Figure 31. The critical-path clustering clusters

The internalization algorithm constructs the set of clusters shown below in figure 32, which produces a schedule with makespan 28. Although the internalization technique can construct irregular cluster shapes, it again illustrates a lack of global scheduling perspective by merging nodes A and C into the same cluster. Lacking the parallelism analysis techniques of the declustering method, the internalization algorithm does not see that merging arc CG instead of AC will expose a greater amount of graph parallelism.

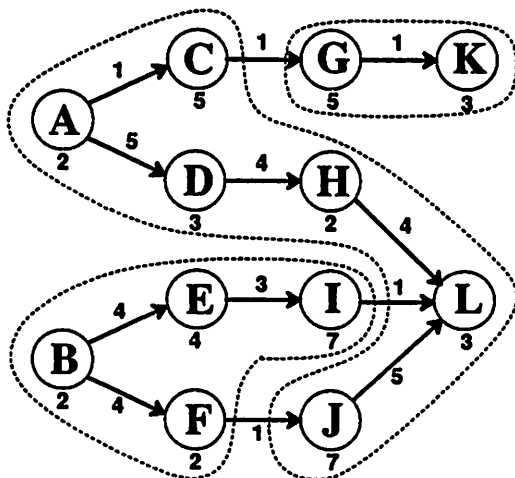


Figure 32. The internalization clusters

8. SUMMARY AND CONCLUSIONS

We have introduced a new compile-time multiprocessor scheduling heuristic called declustering, which accounts for interprocessor communication costs. The first stage of this algorithm is a new clustering strategy that outperforms traditional clustering schemes by using parallelism analysis techniques to explicitly compare the tradeoff between parallelism exploitation and IPC cost. By systematically establishing and then decomposing a parallelism hierarchy, declustering exposes graph parallelism instances in order of importance and adjusts the level of cluster granularity to suit the characteristics of the specified architecture. So while it retains the ability to account for IPC, it also displays the flexibility necessary for effective load balancing to ensure efficient processor utilization. The algorithm gains additional performance by methodically attacking the schedule limiting progression, rather than the critical path. Because of the hierarchical nature of this approach, the algorithm scales well to handle larger problems, especially for architectures that possess a hierarchical structure.

Although declustering is intended to target multiprocessors, we expect that message-passing multicomputers are also valid targets. The major difference is that whereas processors in a shared memory architecture are essentially equidistant for purposes of IPC, the distance between processors in a multicomputer varies according to the number of hops between processor locations. A technique that reassigns placements according to processor proximities will no doubt be beneficial in the multicomputer case.

APPENDIX I Cluster Shifting Techniques

The *communication reduction* routine, which attempts to remove instances of IPC from the SLP, first isolates the SLP clusters at the current level of granularity. For each SLP cluster, the

routine evaluates the difference between the number of data units passed to clusters on a different processor and the number of data units passed to other clusters on the same processor. Expressing this mathematically, we define $D[C_i, P(C_i)]$ to be the number of data units passed from cluster C_i to other clusters on the same processor $P(C_i)$, $D[C_i, \overline{P(C_i)}]$ to be the number of data units passed from C_i to clusters on any other processor, and $\Delta(C_i) = D[C_i, \overline{P(C_i)}] - D[C_i, P(C_i)]$ as the difference between these quantities. We arbitrarily designate the three SLP clusters with highest values of $\Delta(C_i)$ as cluster switch candidates, because there is a good chance that switching these clusters onto a different processor can reduce the amount of IPC incurred. For each cluster switch candidate, the $\Delta(C_i)$ criterion is used to identify other clusters as candidates for exchanging processor locations. It invokes each cluster location switch individually and executes the switch that produces the greatest improvement in makespan.

The *load shift* routine moves clusters onto different processors to balance processor loads more effectively. It categorizes each processor as being heavily loaded or lightly loaded by using a load threshold that is set to half of the largest sum of the computation and communication costs on any processor. The procedure invokes a sequence of cluster shifts from heavily to lightly loaded processors, obtains the new makespan in each case, and implements the shift that provides the greatest improvement in makespan.

9. REFERENCES

1. E.A. Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE T-Com C-36(2)*(January, 1987).
2. E.A. Lee and S. Ha, "Scheduling Strategies for Multiprocessor Real-Time DSP," *Globecom*, (November, 1989).
3. V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA (1989).

4. C. Papadimitriou and M. Yannakakis, *SIAM J. Comput.* 19(2) pp. 322-328 (April, 1990).
5. W.W. Chu, L.J. Holloway, M.T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *Computer*, pp. 57-69 (November 1980).
6. K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, pp. 50-56 (June 1982).
7. H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE T-Com SE-3(1)* pp. 85-93 (January, 1977).
8. G.C. Sih and E.A. Lee, "Scheduling to Account for Interprocessor Communication Within Interconnection-Constrained Processor Networks," *ICPP Proceedings*, pp. 9-16 (August, 1990).
9. G.C. Sih and E.A. Lee, "Dynamic-Level Scheduling for Heterogeneous Processor Networks," *2nd IEEE Symposium on Parallel and Distributed Processing*, pp. 42-49 (December, 1990).
10. T.L. Adam, K.M. Chandy, and J.R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM* 17(12) pp. 685-690 (December 1974).
11. B. Greenblatt and C.J. Linn, "Branch and Bound Algorithms for Scheduling Communicating Tasks in a Distributed System," *Compcon 1987*, pp. 12-16 ()
12. V.B. Gylys and J.A. Edwards, "Optimal Partitioning of Workload for Distributed Systems," *Proc. Compcon*, pp. 353-357 (Fall, 1976).
13. S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures," *ICPP Proceedings* 3 pp. 1-8 (August, 1988).
14. J.C. Bier, E.E. Goei, W.H. Ho, P.D. Lapsley, M.P. O'Reilly, G.C. Sih, and E.A. Lee, "Gabriel: A Design Environment for DSP," *IEEE Micro* 10(5) pp. 28-45 (October, 1990).
15. A. Gerasoulis, S. Venugopal, and T. Yang, "Clustering Task Graphs for Message Passing Architectures," *ICS Proceedings*, pp. 447-456 (June, 1990).