Automatic Source-Code Parallelization Using HICOR Objects

Mark R. Gilder
(518) 276-4857
gilderm@cs.rpi.edu

Mukkai S. Krishnamoorthy
(518) 276-6911
moorthy@cs.rpi.edu

FAX: (518) 276-4033

Rensselaer Polytechnic Institute
Department of Computer Science
Troy, New York 12180

Abstract

We show that by using an intermediate representation, which supports a formalized interface on which to construct parallelization tools, the mapping of the representation onto parallel architectures can be performed quickly and efficiently. An intermediate representation called HICOR (*Hierarchical Intermediate Code Object Representation*) is shown to facilitate the exploitation of parallel operations by providing an abstraction layer for performing high-level intermediate code analysis, scheduling, and code generation. An object-oriented design approach has been employed in the development of HICOR and associated tools. Source language constructs are transformed into specialized object classes. Inheritance properties provided by the object-oriented paradigm are utilized to provide a common interface to each object in the HICOR representation. It is this interface that provides the needed consistency and flexibility in which to construct tools that has since been lacking. In particular, a tool to perform *Compile-Time Scheduling* is presented. The scheduling algorithm employed differs from traditional scheduling problems in that merging of tasks is performed to reduce both task creation and communication costs in determining the final schedule. Architectural parameters are provided as input to the heuristic allowing the scheduler to produce near-optimal results for a wide variety of MIMD architectures. Once the final schedule is determined the *Target Code Generator*, also presented, is used to generate the corresponding target code. A prototype system has been implemented in C++ which incorporates the HICOR intermediate representation with the tools described. The target architectures include the Sun 630 MP/4, Sequent Symmetry S81, and Stardent Titan.

Keywords: Object-oriented intermediate representation; compile-time scheduling; code generation; automatic parallelization.

1 Introduction

Widely accepted intermediate code forms such as *quadruples*, *triples*, and *postfix* notations are not sufficient representations for targeting multiprocessor architectures. In particular, useful information that can be quickly attained from high-level source-code constructs are often lost during the translation to an intermediate representation thereby complicating the design of parallelization tools. More importantly, current compilers do not easily allow the incorporation of newly proposed tools [3, 4, 6, 13, 17, 21, 27, 32] for lack of a formal interface in terms of the intermediate representation. Hence, the development of general-purpose parallelization tools becomes an expensive proposition.

There are at least two other forms of intermediate representations that we are aware of. They are namely the PDG (Program Dependence Graph) and the HTG (Hierarchical Task Graph). In the PDG representation nodes are statements and predicate expressions. Edges incident to a node represent both data dependence and control dependence. The PDG allows different program transformations. Details of this representation and algorithms based on this form are presented in [8]. In HTG, there are simple nodes (representing single tasks), compound nodes (nodes consisting of other tasks in HTG) and a loop node representing a task that is a loop whose iteration body is an HTG. Details of this representation and algorithms are presented in [14]. The HTG representation has more structure than the PDG. In our hierarchical representation, the abstraction for individual statements, blocks of statements, conditional statements, procedural statements, and iterative constructs arise naturally because of the object-oriented specification of the intermediate code. Further our representation provides a formal interface (through objects, classes, methods, and inheritance) to obtain information pertinent for the detection of parallelism. Also, since the granularity required to achieve a speed-up varies with respect to a particular architecture, tools constructed using our representation need not be altered when applied to different architectures due to the formal interface mechanisms provided.

Current research efforts, such as ParaScope at Rice University [18], PTRAN at IBM T. J. Watson Research Center [31], and the Typhoon parallelizing compiler project at Cornell University [24], address this issue but are with respect to a single programming language and are not explicitly based on object-oriented principles.

In this paper, we present another approach to accomplishing this for multiple source languages and multiple target architectures. The design employed provides a consistent view of all high-level constructs available at the source code level throughout all phases of the compilation. In particular, the intermediate code generated by the compiler front-end is explicitly represented by objects instantiated from a specialized class hierarchy called HICOR

(Hierarchical Intermediate Code Object Representation) [12]. Each object supports a formal interface mechanism through which internal object information is made available. This added level of abstraction allows flexible parallelization tools to be developed without regard to the underlying structure of the internal representation. Hence, the tool becomes immune to any internal changes made in the intermediate representation. Tools to perform *Compile-Time Scheduling* and *Target Code Generation* are presented to demonstrate the flexibility in using this approach.

The remainder of this paper is organized as follows: Section 2 provides a general overview of the prototype system developed; Section 3 provides a brief overview of the HICOR intermediate representation used. Section 4 describes the implementation of the *Compile-Time Scheduler*; Section 5 illustrates the *Target Code Generation* scheme employed in mapping the output of the *Compile-Time Scheduler* onto a particular architecture, and provides an evaluation of the overall system.

2 System Overview

Figure 1 depicts the major components of the HICOR parallelization system developed [10, 12]. In particular, source languages are mapped into an architecturally independent intermediate language called CICL (*Common Intermediate Code Language*) through the use of specific front-end *Language Translators*. Once performed, the equivalent HICOR instance is generated. The HICOR instance may be viewed as the memory-resident instantiation of the CICL instance. The initial HICOR instance is independent of any target architecture. However, once the target is specified, the HICOR instance is mapped into a suitable view through the use of the *Architectural Filter* object. The filter provides a common interface for manipulating the architectural constraints specified by the *Target Architecture Parameters*. Each object in the HICOR instance encapsulates the internal intermediate code corresponding to the original source code construct it represents. Hence, all data manipulation and extraction must occur through a set of common access routines or *methods*. It is these methods that provide the formal *Tool Interface Layer* on which parallelization tools may be constructed. The output of the system is a near-optimal execution DAG (directed acyclic graph) rendered by the *Compile-Time Scheduler* for a specified target architecture. Once computed the *Target Code Generator* is used to generate suitable target code for one of three currently supported architectures, i.e., the Sun 630 MP/4, Sequent Symmetry S81, and Stardent Titan. The system was implemented on a SPARCstation 2 and was written entirely in C++. The system is open-ended and interfaces nicely to other packages which either produce C, Pascal, or Fortran as output; or accept C as input. Furthermore, a GUI has been developed for the system which provides the user with a visual representation of the parallelizations performed [26].

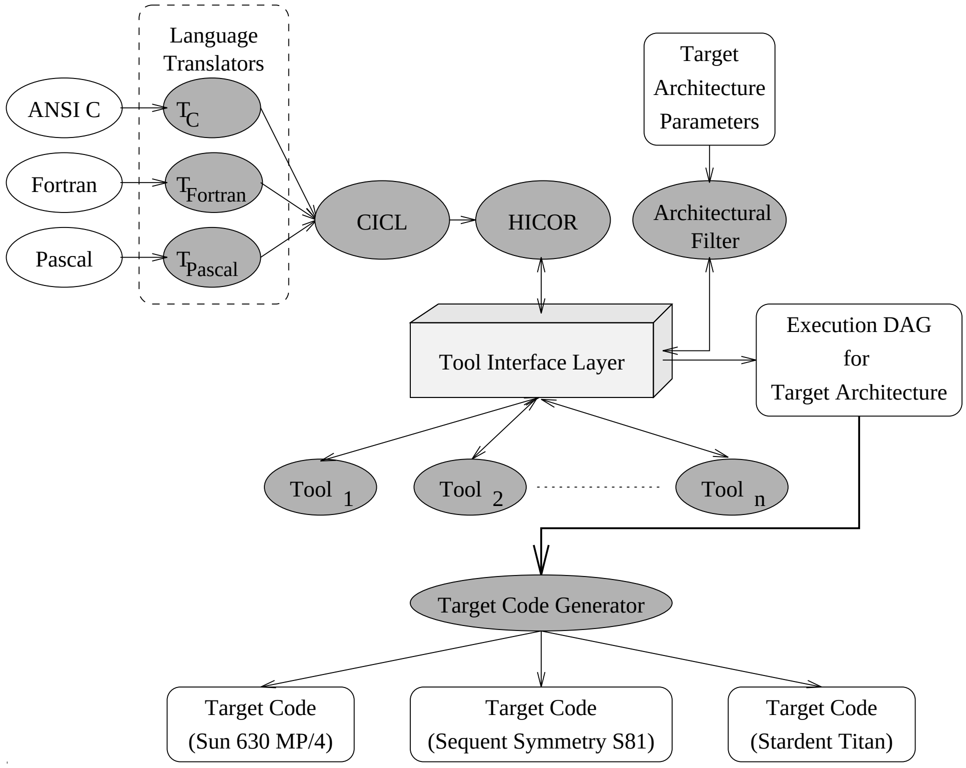

Figure 1: System Components

3 HICOR as an Intermediate Representation

HICOR is an object oriented intermediate representation which provides a hierarchical view of program constructs. Each object in the representation encapsulates the internal intermediate code corresponding to the source construct it represents. Internal object information is provided through a set of common access routines or *methods*. It is these methods that provide the *Tool Interface Layer* on which parallelization tools may be constructed. In most instances a single HICOR object may be decomposed into an ordered collection of smaller, more detailed, objects; for example, a HICOR object which represents a *Function* in some source language, may be viewed as a collection of objects, each representing the immediate basic blocks it consists of. In this respect, the grain size represented by a particular HICOR object is related directly to its position in the hierarchy. Objects occurring at higher levels are more computationally complex than those occurring at lower levels. Therefore, as we descend through the hierarchy, the grain-size represented decreases. For example, Figure 2 contains a C-function called MinMax which computes the minimum and maximum values of the input array a. The first few levels of the HICOR representation are provided in Figure 3. The numbers to the left of each object indicate the implicit execution order of the original serial version; hence the HICOR representation is constructed such that the control-flow graph (CFG) for a particular object may be extracted by performing a topmost DFS from any object in question. A detailed description of all HICOR objects and available methods may be found in [10, 12].

From these available methods dependency information for any portion of the HICOR representation may be rendered. For example, Figure 4 illustrates the dependency graph generated by invoking the GenerateDependencies method on object 94 of the MinMax function. In particular, dependency information can be generated very efficiently since only the first level of basic blocks for the corresponding function are analyzed, i.e., objects 11, 12, 51, 53, 54, 69, 71, 72, 87, 90, and 93; hence for this example a dependency graph was produced by analyzing approximately 1/8 of the associated objects. The results clearly indicate that the loops to find both the minimum and maximum values of the array a, i.e., objects 87 and 69, respectively, can be executed in parallel once the loop represented by object 51 has finished. Also, the loop which initializes a, i.e., object 11, can be executed in parallel with the assignment statement represented by object 12. We should mention that the system also determines loop-level dependencies using techniques described in [9, 13, 23]. Hence, the ability to parallelize each iteration of the loop represented by object 11 would also be extracted during the code generation phase.

4 Scheduling for Parallel Execution

The *Compile-Time Scheduling* tool is used to map a dependency graph (DAG) derived from the HICOR representation into a view suitable for execution on the specified target architecture.

As previously illustrated the DAG produced consists of tasks which may vary greatly with respect to computational grain size each task comprises. It is therefore imperative that the DAG be manipulated to guarantee that the generated parallel code, along with the incurred overhead, produces a better result then the sequential version. This problem differs from the *General Scheduling Problem* in that we allow tasks in the DAG to be merged in an effort to reduce the task creation overhead as well as reducing the task communications overhead. For example, Figure 5 illustrates the optimal schedule attainable given that tasks 1 and 2 must be executed before task 3.

However, if we consider both task creation costs and communication overhead costs, as

```
Obj_94          void MinMax() { /* N is even */
                    int i, j, a[N], b[N/2], c[N/2], min, max;

Obj_11              for(i = 0; i < N; i++)
Obj_6                   a[i] = random()%10000;

Obj_12              j = 0;
Obj_51              for(i = 0; i < N; i = i + 2)
Obj_47                  if(a[i] > a[i + 1]) {
Obj_33                      b[j] = a[i];
                            c[j++] = a[i + 1];
                        }
                        else {
Obj_46                      b[j] = a[i + 1];
                            c[j++] = a[i];
                        }

Obj_53,54           max= b[0];
Obj_69              for(i = 1; i < N/2; i++)
Obj_64                  if(b[i] > max)
Obj_63                      max = b[i];

Obj_71,72           min= c[0];
Obj_87              for(i = 1; i < N/2; i++)
Obj_82                  if(c[i] < min)
Obj_81                      min = c[i];

Obj_90              printf("Minimum value is: %d\n",min);
Obj_93              printf("Maximum value is: %d\n",max);
                }
```

Figure 2: MinMax: function to find the minimum and maximum values of array a

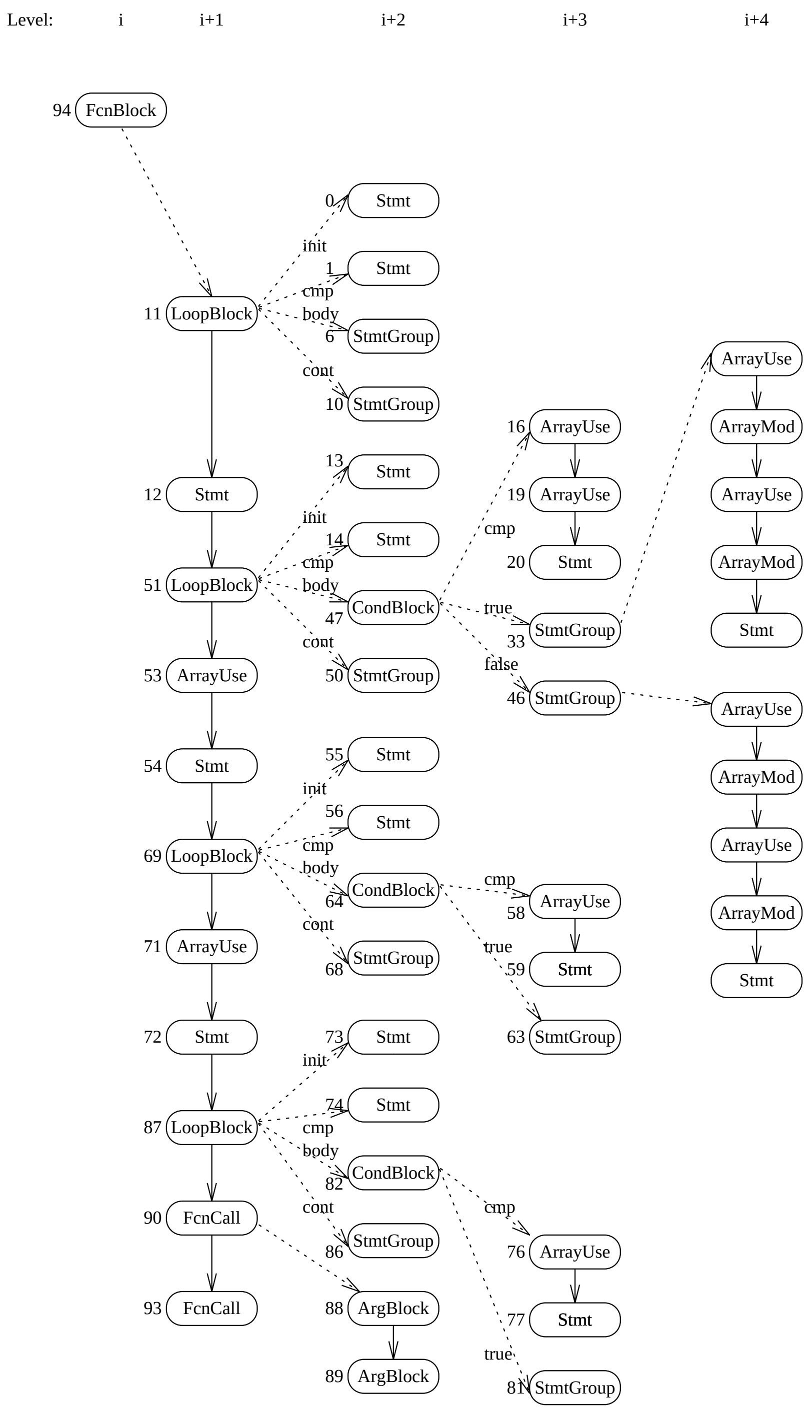

Figure 3: HICOR Representation

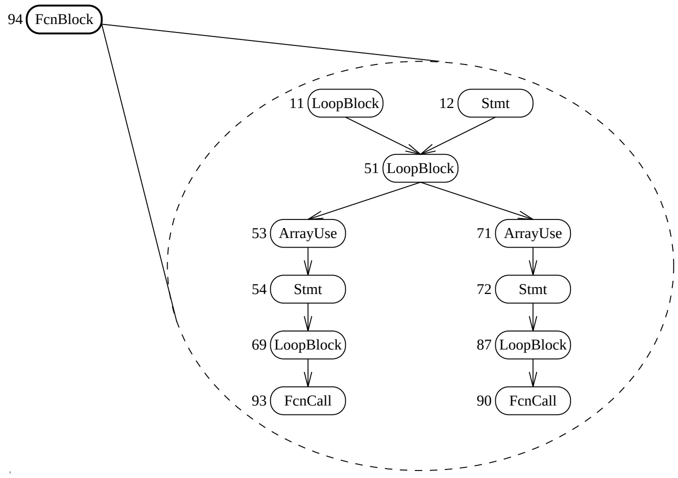

Figure 4: Dependency DAG generated for HICOR object 94 of the MinMax function

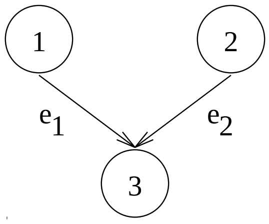

Figure 5: Optimal schedule without merging

defined by the target architecture, this solution is far from optimal. In fact, if the task creation cost is close to the cost of executing the task, then the serial schedule depicted in Figure 6 may be better. Note that merging two tasks decreases the overhead by: $(2 * \alpha) + e_1 + e_2$ where α is the task creation overhead and e_1 and e_2 represent the communication overhead from tasks 1 to 2 and tasks 1 to 3 respectively.

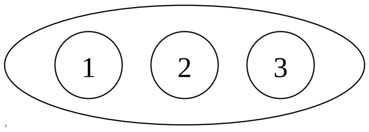

Figure 6: Optimal schedule with merging

In the following section we formally define the scheduling problem for which we wish to find a near optimal solution.

4.1 Associated Cost Scheduling Problem

The *Associated Cost Scheduling Problem* (ACSP) is directed at finding the optimal schedule for a given target architecture which maintains all precedence relations but also allows for merging of tasks.

Before formally defining the ACSP problem, we first present the *General Scheduling Problem* (GSP) which has been proven to be NP-complete by Karp [22]. This result also appears in [34]. We later use this result to prove that ACSP is NP-complete.

The GSP problem may be stated as follows: Given a set S of n tasks, a partial order $\prec$ on S, a weighting function W, an number of processes k and a time limit t, does there exist a total function F from S to $\{0, 1, \ldots, t-1\}$ such that

(i) for j and $j' \in S$, if $j \prec j'$, then $F(j) + W(j) \leq F(j')$.

(ii) for each $j \in S$, $F(j) + W(j) \leq t$, and

(iii) for each i, $0 \leq i < t$, there are at most k values of j for which $F(j) \leq i < F(j) + W(j)$?

The *Associated Cost Scheduling Problem* may be formulated as follows: Given a set S of n tasks, a partial order $\prec$ on S, a weighting function W, a communication cost function β, a task creation cost α, a number of processors k and a time limit t, does there exist a total function f from S to $\{0, 1, \ldots, t-1\}$ such that

(i) for j and $j' \in S$, if $j \prec j'$, then

$$f(j) + W(j) + \alpha + \beta(j, j') \leq f(j')$$

(ii) for each $j \in S$,

$$f(j) + W(j) + \alpha + \beta(j, j') \leq t$$

where j' is any task s.t. $j \prec j'$.

(iii) for each i, $0 \leq i < t$, there are at most k values of j for which

$$f(j) \leq i < f(j) + W(j) + \alpha + \beta(j, j') \leq f(j')$$

for some j'.

where:

- $\beta(j, j')$: is the processing time needed to send the required data from task j to task j'.
- $W(j)$: is the processing time for task j.

Lemma 4.1 *GSP is polynomial-time transformable to ACSP.*

Proof: Given an instance of GSP we can construct the following instance of ACSP as follows:

(i) let $S_{\text{ACSP}} = S_{\text{GSP}}$, i.e. for each task in GSP create an equivalent task in ACSP maintaining all precedence relations.

(ii) let $W_{\text{ACSP}} = W_{\text{GSP}}$.

(iii) let $\beta(j, j') = 0$ for all tasks $j \in S_{\text{ACSP}}$ and all tasks j' s.t. $j \prec j'$.

(iv) let $\alpha = 0$.

(v) let $k_{\text{ACSP}} = k_{\text{GSP}}$ and $t_{\text{ACSP}} = t_{\text{GSP}}$.

By defining β and α as described above, the constraints of ACSP are identical to those of GSP. Hence, a schedule that satisfies GSP must also satisfy ACSP. It follows that if the instance of GSP has a solution, then so must the ACSP instance. Also, if the constructed instance of ACSP has a solution then so must the original GSP instance. $\square$

It is clear that ACSP is in NP. We may simply guess a possible schedule and then check whether each of the conditions are satisfied in polynomial time. Hence we have the following theorem.

Theorem 4.2 *The ACSP problem is NP-complete.*

4.2 Ω-Heuristic

In this section we present the Ω-Heuristic; a polynomial-time algorithm to find a near optimal solution to ACSP. The notation used throughout the remainder of this Section is may be found in Appendix A. Before presenting the algorithm we make the following observations (similar observations are made in [28]):

Proposition 4.3 *Given a DAG D produced by the* Hierarchical Dependency Analyzer *the optimal execution time for any schedule, ω_{opt}, is bounded by*

$$\omega_{opt} \geq \max\{\omega, T_1/k\}$$

where ω is the cost of the critical path of D, T_1 is the sequential execution time of D, and k is the number of processors for the given target architecture.

This is due to the fact that all tasks along any path of D must be executed sequentially, otherwise the precedence constraints would be violated, and since ω is the highest cost path in D in terms of cost, no produced schedule can finish in a time less than ω, even for an arbitrary number of processors. Also, if k is specified, then the optimal schedule produced can be no better than T_1/k.

Since no schedule can finish in time less than ω_{opt} we have:

Proposition 4.4 *The maximum speedup attainable for a given DAG D, denoted $S_{\max}$, is bounded by*

$$S_{\max} \leq \frac{T_1}{\omega_{opt}}$$

where T_1 is the sequential execution time of D.

In order to achieve a speedup of $S_{\max}$ the computation must be broken in threads containing no more than ω_{opt} time. Since each of these threads must then be assigned to a specific processor we have the following lower bound on the number of processors:

Proposition 4.5 *The lower bound on the number of processors k required to achieve a speedup of $S_{\max}$ for a given DAG D is bounded by*

$$k \geq \left\lceil \frac{T_1 - \omega_{opt}}{\omega_{opt}} \right\rceil + 1$$

where T_1 is the sequential execution time of D and ω_{opt} is the optimal execution time.

Given a DAG D we can easily compute the bound for the maximum speedup attainable, $S_{\max}$, since the values ω, T_1, and k are known. By computing $S_{\max}$ valuable insight into algorithm performance is attained. Hence, $S_{\max}$ may be used to measure the parallelizability of a particular problem instance. Therefore we have some estimate of algorithm performance a priori.

In implementing the scheduler we allow the user to specify the minimum speedup required ($s_{\min}$) for generating parallel code. If $S_{\max} < s_{\min}$, then serial code is produced.

4.3 Algorithm

In this section we present our polynomial-time scheduling algorithm for constructing a near-optimal execution-time schedule. The main algorithm generates a *Layered Representation Graph L*, defined below, which represents all tasks to be scheduled on the specified target architecture. It differs from the DAG generated by the *Hierarchical Dependency Analyzer*, in that a single task in L may consist of several tasks from D, i.e., several tasks from D may be merged into a single task in L as long as no precedence constraints are violated.

Definition 1 *A* **Layered Representation Graph**, *or LR-graph, L may be viewed as a partitioning of tasks from a DAG D into layers: $L_0, L_1, \ldots, L_k$ such that the following properties hold:*

1. *each layer $L_i, 0 \leq i \leq k$, consists of at least one task from D, i.e., no layers of L may be empty.*
2. *all tasks in any layer L_i may be executed concurrently, i.e., there are no precedence constraints between any two tasks in L_i.*
3. *if t_i and t_j are two tasks such that $t_i \prec t_j$, then if $t_i \in L_k$ then $t_j \notin \{L_i | i = 0, \ldots, k\}$*

For example, in Figure 7 a sample DAG generated by the *Hierarchical Dependency Analyzer* is provided. The quantities to the left of each task provide the estimated cost of executing that task on a previously selected target architecture. A valid *LR*-graph for this DAG is provided in Figure 8. Although this example illustrates a one-to-one mapping from tasks in D to tasks in L, this is not a requirement; for example tasks 4, 6, and 7 could have easily been merged into a single task as illustrated in Figure 9.

The Ω-Heuristic consists of three major components as illustrated by the driver routine listed in Figure 10. In particular, the Ω-*Cut-Merge* is responsible for partitioning the tasks of D

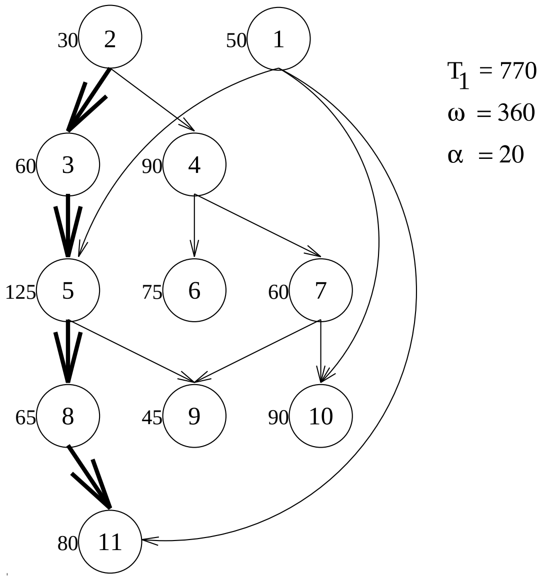

Figure 7: Sample DAG to be scheduled

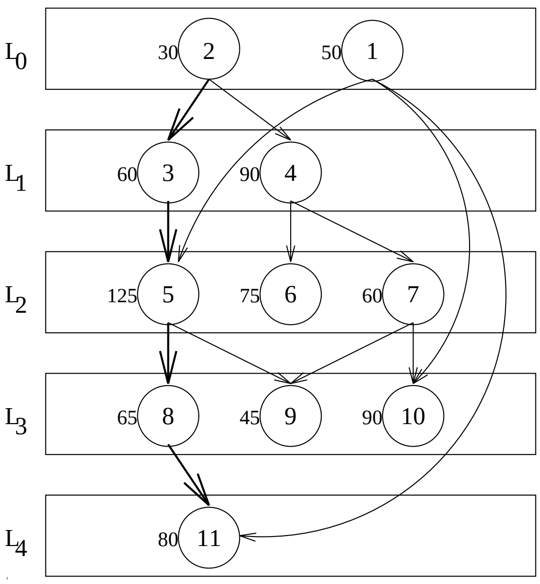

Figure 8: *LR*-Graph *L* with no merging of tasks.

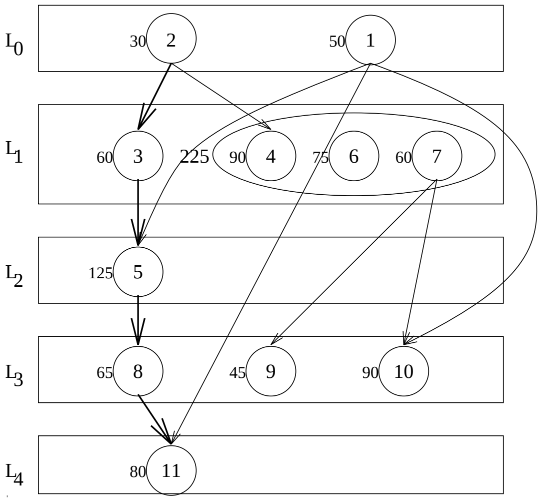

Figure 9: LR-Graph L with tasks 4, 6, and 7 merged.

into a LR-graph L representing the initial schedule for the input DAG D. In computing this initial schedule an arbitrary number of processors is assumed. Once L is constructed the *Coalesce-Layer-Merge* step checks whether or not all tasks in successive layers of L should be merged into a single task. Lastly, the *Layer-Merge* portion of the algorithm is responsible for minimizing task creation costs by merging tasks within each layer of L such that no more than k tasks exist. In the sections that follow we describe each component of the algorithm in detail.

Ω-HEURISTIC(**Dag** D, **Arch** *ArchObj*)
  // This routine determines an near optimal schedule for the input DAG D.
  Dag D;
  LR-**graph** L, L_1, L_2;

  $L_1 \leftarrow$ Ω-CUT-MERGE(D,*ArchObj*);
  $L_2 \leftarrow$ COALESCE-LAYER-MERGE(L_1,*ArchObj*);
  $L \leftarrow$ INTRA-LAYER-MERGE(L_2,*ArchObj*);

  return L;
end

Figure 10: Major components of the Ω-Heuristic

4.3.1 Ω-Cut Merge

The Ω-Cut Merge determines how to partition the critical path of D in an effort to achieve maximum parallelism without performing an exhaustive search. The critical path of D, P_ω, consists of all tasks along the highest cost path. Since all tasks in P_ω must be executed sequentially, our heuristic attempts to execute all other tasks in at least ω time units. For example, the graph previously presented in Figure 7 illustrates a dependency graph with $T_1 = 770$, $P_\omega = \{2, 3, 5, 8, 11\}$, $\omega = 360$, and $\alpha = 20$. For this example we also assume all communication edges have cost 10, i.e., $\forall_{e_{j,j'}} \beta(j, j') = 10$. Since we wish to execute all tasks within ω time units, an alternative approach might be to simply merge all tasks in P_ω into a single task p', and then try to schedule the remaining tasks of D with p'. However, this approach usually fails because of existing dependencies between tasks in P_ω and tasks not in P_ω. In Figure 11 we illustrate the ill effects of merging all tasks along P_ω. In particular, p_k represents some task along P_ω which is dependent upon some task in $Pre - P_\omega$. Also, there is some task in $Post - P_\omega$ which is dependent upon p_k. By merging all tasks in P_ω to form

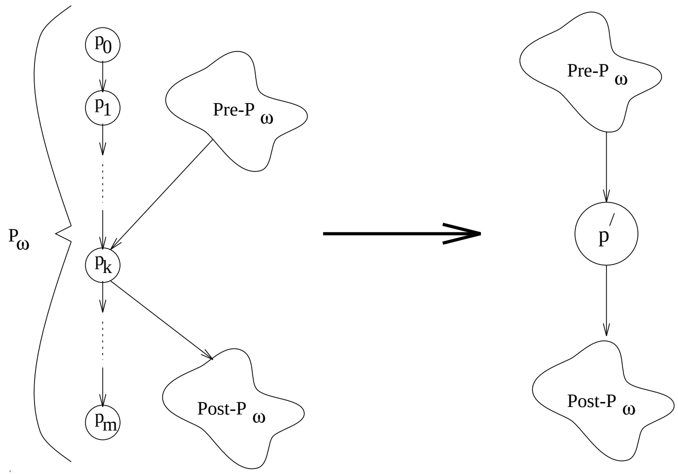

Figure 11: Merging all tasks along P_ω

p' the start time of p' is delayed by the minimum cost of executing all tasks in $Pre - P_\omega$. Similarly, the starting time for tasks in $Post - P_\omega$ is also delayed. To solve this problem our algorithm searches for a partitioning p' of tasks in P_ω which allow the most tasks not in P_ω to execute currently p'. For instance by letting p' consist of tasks $p_0, \ldots, p_{k-1}$, tasks in $Pre - P_\omega$ can be executed concurrently with p' as illustrated in Figure 12.

We show that by analyzing only tasks along P_ω a near optimal schedule can be computed. The cost of executing a particular partition is given by $\Omega(p_i)$ where p_i corresponds to the i-th task along P_ω. In computing $\Omega(p_i)$ the first i tasks along P_ω are merged. For example, $\Omega(p_3)$ for the graph given in Figure 7 is computed by first merging tasks $\{2, 3, 5\}$ together.

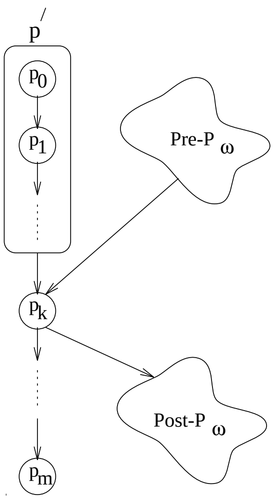

Figure 12: Partitioning of tasks along P_ω

Figure 13 contains the DAG produced after performing the merge. Note that the merged task saves the cost of creating two tasks, i.e., $2 * \alpha$, as well as the communication costs $\beta(2,3)$ and $\beta(3,5)$. However, the highest cost path, ω, has been extended drastically due to the added delay of task 4 which must now wait until the newly merged task, i.e., $\{2,3,5\}$, completes. Also, task 1 must now be executed prior to the newly merged task.

To determine the partitioning of tasks in P_ω the Ω-Cut algorithm successively merges tasks along the path comprising P_ω. If $p_0, p_1, \ldots, p_{|P'_\omega|-1}$ represent these tasks then the first iteration initially sets $p' = p_0$. Successive iterations merge the next task along the path until $p' = P_\omega$. For each instance of p' the amount of cost savings in terms of execution time is computed. The iteration which minimizes the execution time is called the Ω-Cut and determines the first partition of P_ω. Similarly, successive partitions are found by removing tasks comprising p' for the Ω-Cut and repeating the same procedure.

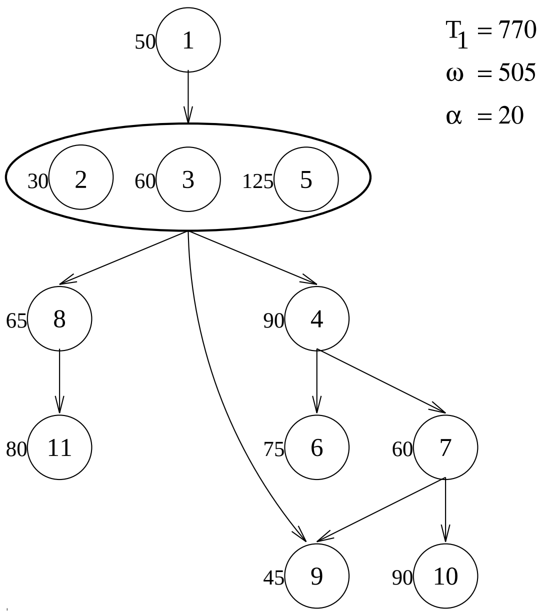

Figure 13: Sample DAG corresponding to $\Omega(p_3)$

The algorithm to find the Ω-Cut is as follows:

Algorithm:	**Ω-Cut Merge**	
Input:	D	DAG generated by the *Hierarchical Dependency Analyzer.*
	ArchObj	an object containing the specifics of the target architecture.
Output:	LR-graph L.	
Description:	Computes the LR-graph L by successively finding the Ω-Cut's of D.	

1. $D' = D$
2. $\delta_{\max} = -\infty$
3. compute P_ω, the critical path of D', and initially set P'_ω to P_ω.
4. **while** $(|P'_\omega| > 0)$

   do

   (a) $z = 0$

   (b) number the tasks in P'_ω as $p_0, p_1, \ldots, p_{|P'_\omega|-1}$ such that for $0 \leq i < |P'_\omega| - 1, p_i \prec p_{i+1}$.

(c) compute ω, the cost of executing tasks along P'_ω, as follows:

$$\omega = \sum_{i=0}^{|P'_\omega|-1} W(p_i)$$

(d) compute the set B representing the tasks not in P'_ω, i.e., $B = V(D') - P'_\omega$.

(e) **foreach** p_i along P'_ω compute $\Omega(p_i)$ as follows:
do

i. merge tasks $p_0, p_1, \ldots, p_i$ into a single task p'. Note $W(p') = \sum_{j=0}^{i} W(p_j)$.

ii. calculate δ, the cost savings achieved in performing the merge in terms of both task creation costs and communication costs:

$$\delta = i\alpha + \sum_{j=0}^{i-1} \beta(p_j, p_{j+1})$$

iii. compute the set of tasks X such that each task $x \in X$ has the property that there exists tasks p_m and p_n in the previously merged task p' s.t. $p_m \prec x \prec p_n$, i.e.,

$$X = \{x \in B | p_m \prec x \prec p_n, 0 \leq m < n \leq i\}$$

once computed, merge all task in X with p'.

iv. adjust δ by:

$$\delta = \delta + |X|\alpha + (\text{communication costs internal to } X)$$

v. let B' consist of the set of tasks which must be executed before p', i.e., the set of all ancestors of tasks in p' which are themselves not in p'.

vi. **if** $B' \neq \emptyset$ then adjust δ by:

$$\delta = \delta + (|B'| - 1)\alpha + (\text{communication costs internal to } B')$$

vii. let b' represent the task created by merging all tasks in B'.

viii. calculate the set of available tasks A that may be executed concurrently with p'. That is:

$$A = \{x \in V(D') | x \notin p', x \notin B', x \notin P_\omega, p' \not\prec x, \text{and} x \not\prec p'\}$$

ix. perform the *Independent Subgraph Bin-Packing* algorithm, described below, on all tasks in A. Assign the resulting LR-graph L_{rslt} to $L[z]$, an array of LR-graphs.

x. determine the ω-extension, i.e., recalculate the highest cost path, ω', of the DAG induced on D' by merging tasks according to the layered graph $L[z]$. If $\omega' > \omega$ then adjust δ by:

$$\delta = \delta - (\omega' - \omega)$$

xi. **if** $\delta > \delta_{\max}$ **then**
do

A. $\delta_{\max} = \delta$

B. $iter_{\max} = z$. This saves the index of the LR-graph corresponding to the highest cost δ value encountered thus far.

done

xii. $z = z + 1$

done

(f) merge the layers corresponding to $L[iter_{\max}]$ to the existing LR-graph L.

(g) remove all tasks from D' which exist in L.

(h) set P'_ω to the highest cost path of D'.

done

5. **return** L.

4.3.2 Independent Subgraph Bin-Packing

The *Independent Subgraph Bin-Packing* (ISBP) algorithm determines the tasks from A, the set of available tasks, that should be executed concurrently with p'. In doing so, the available tasks are merged to create a new task whose estimated execution cost is as close to $W(p')$ as possible without exceeding it.

Definition 2 *An* **Independent Subgraph** *of a DAG D, denoted D_I, has the following properties:*

1. *Each vertex v_i of D_I is reachable by at most one source node of D. Hence, no vertex of D can be in D_I if it has more than one source vertex as an ancestor.*

2. *The number of independent subgraphs for a DAG D is equal to the number of source vertices in D. This follows immediately from 1.*

Therefore, an *Independent Subgraph* of a DAG D consists of a source vertex s_i along with all vertices reachable from s_i which are not reachable from any other source vertex s_j, where $i \neq j$.

Hence, the *Independent Subgraph Bin-Packing* algorithm identifies all of the Independent Subgraphs of D'_A, the induced subgraph of D' consisting of only tasks in A which may be executed with p'. Consider the DAG in Figure 14 consisting of tasks which may all be executed with p'.

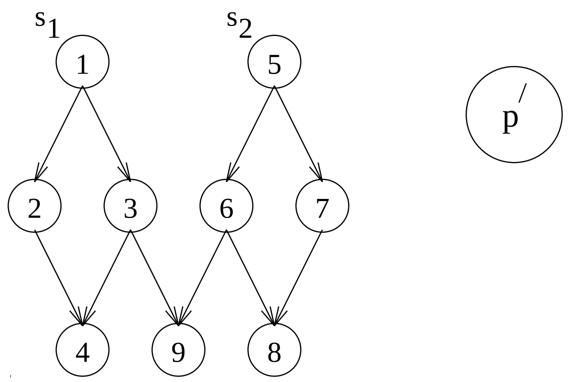

Figure 14: Induced subgraph D'_A with source tasks s_1 and s_2.

The corresponding *Independent Subgraphs* for D'_A are illustrated in Figure 15. Note that task 9 can not belong to any subgraph since it is reachable from more than one source node, i.e., both s_1 and s_2.

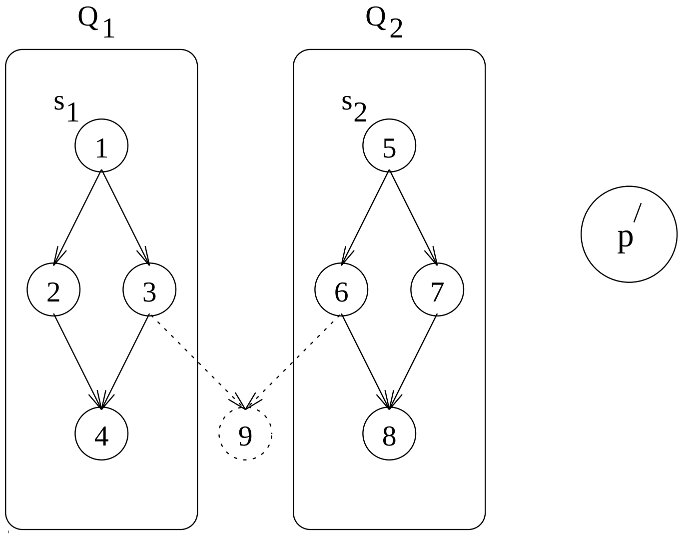

Figure 15: Independent subgraphs of D'_A represented by Q_1 and Q_2.

Once all subgraphs have been identified the binpacking portion of algorithm is invoked. To accomplish this portion we must first determine the number of bins to create. This is done by computing the total cost of each Independent Subgraph Q_i, denoted W_i. If $W_i \geq W(p')$ we associate Q_i with bin B_i. Otherwise, we begin merging Independent Subgraphs until the new cost of the merged subgraph $W_{j,\ldots,n} \geq W(p')$. Once found we associate the merged Independent Subgraphs, i.e., $Q_j, \ldots, Q_n$, with bin $B_{j,\ldots,n}$. Hence the number of bins created is at most equal to the number of Independent Subgraphs.

We then begin packing nodes from each subgraph into the appropriate bin, ensuring that no task is packed until all of its ancestors have been packed, since precedence constraints must be maintained.

The algorithm to perform the identification of Independent Subgraphs and packing of bins is as follows:

Algorithm:	**Independent Subgraph Bin-Packing**	
Input:	D'	working copy of the DAG D generated by the *Hierar-. chical Dependency Analyzer*
	p'	merged task to be scheduled.
	A	the set of available tasks in D' which may be executed concurrently with p'.
	B'	the set of tasks which must be executed prior to p'.
	b'	the single task created by merging all tasks in B'.
	$ArchObj$	an object containing the specifics of the target architecture.
Output:	LR-graph L.	
Description:	Finds all *Independent Subgraphs* for D' which are then used to create tasks to execute concurrently with p'.	

1. create the subgraph D'_A from D' by removing all vertices from D' which are not in the set A. All incident edges to the vertices removed are also deleted.
2. **foreach** source task $s_1, s_2, \ldots, s_k$ of D'_A identify the *Independent Subgraph* $W_1, W_2, \ldots, W_k$, as follows:
   (a) set all task labels of D'_A to 0.
   (b) initiate a DFS from each source node s_i of D'_A. As each node is visited do the following:
      i. **if** the node is labeled with 0, label it with i.
      ii. **else** label it with ∞.
   (c) the Independent Subgraphs, $Q_1, Q_2, \ldots, Q_k$, are identified by the nodes labeled with $1, 2, \ldots, k$, respectively.
3. let W_i represent the cost of executing all tasks in Q_i, i.e., $W_i = \sum_{v \in Q_i} W(v)$.
4. to determine the number of processes to execute with p' we first calculate the number of bins that may be efficiently packed and then associate the Independent Subgraphs

with each bin as follows:

(a) sort the W_i's in decreasing order of cost.
(b) **if** $W_i \geq W(p')$ **then** create a bin B_i and associate it with Independent Subgraph Q_i.
(c) **else** (i.e., $W_i < W(p')$), find a collection of W_j's, $i \neq j$, from the sorted list such that the total cost of W_i plus the collection is either equal to $W(p')$ or just exceeds it. Create bin $B_{m_1,m_2,\ldots}$ where $m_1, m_2, \ldots$ correspond to the collection of subgraphs used to create the bin. For example, if $W_1 < W(p')$, and the collection found consists of W_3 and W_8, then the bin created would be called $B_{1,3,8}$ and would be associated with Independent Subgraphs Q_1, Q_3, and Q_8; denoted $Q_{1,3,8}$.

5. **foreach** bin previously identified, i.e., $B_{m_1,m_2,\ldots}$, pack the bin from the tasks in its corresponding Independent Subgraph $Q_{m_1,m_2,\ldots}$ as follows:

   do

   (a) $b = W(p')$; b denotes the maximum size of each bin.
   (b) sort the available source nodes in decreasing order of estimated execution cost. Let L_s represent the list of sorted source vertices.
   (c) **if** there exists a source vertex $s_i \in L_s$ such that $W(s_i) \leq b$
   do

      i. **while** there exists a source vertex $s_i \in L_s$, such that $W(s_i) \leq b$ **and** $b > 0$
      do

         A. let s' represent the largest cost vertex in L_s s.t. $W(s') \leq b$
         B. pack s' into the corresponding bin B and remove s' from Q.
         C. add any new source vertices created by removing s' from Q to the sorted list L_s.
         D. set b to the residual space left after packing s', i.e., $b = b - W(s')$

      done

   done

   (d) **else** (i.e. no source task s_i exists such that $W(s_i) \leq b$) place the smallest cost source task from L_s in the corresponding bin.
   (e) adjust the cost savings δ by:

   $$\delta = \delta + (\#\text{ tasks merged } - 1)\alpha + (\text{communication costs saved for } B_{m_1,m_2,\ldots})$$

   (f) let $t_{m_1,m_2,\ldots}$ represent the single task created by merging all tasks in bin $B_{m_1,m_2,\ldots}$.

   done

6. create an empty LR-graph L.
7. **if** there are tasks which must be executed prior to p', i.e., $B' \neq \emptyset$; **then** insert the task b' into layer L_0 of L, where b' is the single task created by merging all tasks in B'.

8. add the following tasks to the current layer, L_i (L_0 if $B' = \emptyset$, L_1 otherwise)
   (a) p'
   (b) each task $t_{m_1,m_2,\ldots}$ from each corresponding bin $B_{m_1,m_2,\ldots}$.
   (c) update δ, by
   $$\delta = \delta + (\sum_{t_j \in L_i \wedge t_j \neq t_k} W(t_j) \;) - W(t_k)$$
   where t_k is the highest cost task in the current layer, L_i.
9. **if** $B' \neq \emptyset$ add all necessary communication edges from task b' in layer L_0 to tasks in layer L_1. Note that tasks in $L_0 \prec L_1$ since L_0 consists of all tasks which must be executed before tasks in p'.
10. **return** L. The entire LR-graph L is stored in $L[z]$ of the caller, i.e., L_0 and L_1, if it exists, are stored in $L[z]$.

4.3.3 Coalesce Layer Merge

Once the layered DAG, L, has been computed the effects of merging all tasks within individual layers as well as merging all tasks within successive layers are examined.

We begin by constructing two m x m matrices M_1 and M_2 where m is equal to the number of layers of L. The matrix entries $c_{i,j}$ of M_1 provide the sequential cost of executing layers L_i through L_j and are calculated as follows:

$$c_{i,j} = Seq(i,j)$$

where

$$Seq(i,j) = \sum_{k=i}^{j} W(L_k) \; + \; \alpha$$

The entries $c_{i,j}$ of M_2 provide the optimal cost of executing layers L_i through L_j. The matrix entries $c_{i,j}$ of M_2 are calculated as follows:

$$c_{i,j} = Opt(i,j)$$

where

$$Opt(i,j) = \begin{cases} \min \; (C_i, Seq(i,i)) & \text{if } i = j \\ \min\limits_{i \leq k < j} \; (Seq(i,j), Opt(i,k) + Opt(k+1,j) + \sum\limits_{l=i}^{k} \sum\limits_{p=k+1}^{j} \beta'(L_l, L_p)) & \text{otherwise} \end{cases}$$

and C_i, the cost of executing all tasks in layer L_i of L, is defined as:

$$C_i = \max_{j \in L_i} W(j) + \ n_i \alpha$$

where n_i is the number of tasks in L_i. The task creation costs are assessed sequentially; hence the $n\alpha$ term. The solution of $Opt(i,j)$ is computed by using k to partition the layers of L into two distinct partitions consisting of layers i through k, and layers $k+1$ through j, and the communication costs between them. For each of these partitions the optimal cost would have been previously computed. Also, $\beta'(L_l, L_p)$ is the sum of all communication edges which have a source task in the first partition and and a sink task in the second partition.

Since we are only interested in entries where $i \leq j$, both M_1 and M_2 will be upper triangular. This allows us to store values of $\beta'(L_i, L_j)$ in $c_{j,i}$ of either M_1 or M_2. Entry $c_{0,m}$ of matrix M_2 will contain the best cost to execute the tasks in the layered graph. For example, the layered graph illustrated in Figure 16 was produced by the Ω-Cut Merge algorithm for the original DAG D given in Figure 7, with $\beta(j,j') = 10$. The matrices, M_1 and M_2, produced by the *Coalesce Layer Merge* are illustrated in Figure 17 produced.

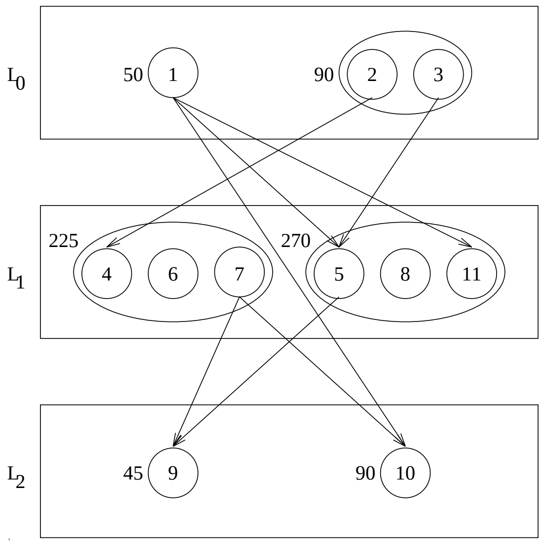

Figure 16: Layered graph produced by Ω-Cut Merge algorithm.

Communication costs are stored in the lower-triangular portion of M_1; hence communication overhead between any two layers are readily accessible. For example, entry $c_{1,0}$ contains the communication overhead between layers L_0 and L_1. The sequential execution cost is located in $c_{0,2}$ of M_1. This value is simply $T_1 + \alpha$, where $\alpha = 20$. Note that corresponding matrix entries in the upper-triangular portions of M_2 are strictly less than those in M_1, indicating that no merging of tasks or successive layers was necessary to increase performance. The optimal execution time located in $c_{0,2}$ of M_2 indicates that the optimal execution schedule produced by this algorithm has a schedule length of 650.

Seq(i,j)

M_1	0	1	2
0	160	655	790
1	40	515	650
2	10	30	155

(a)

Opt(i,j)

M_2	0	1	2
0	130	480	650
1	X	310	470
2	X	X	130

(b)

Figure 17: Result matrices: (a) M_1: sequential execution cost entries, $Seq(i, j)$, with $\beta'(L_i, L_j)$ stored in $c_{j,i}$; (b) M_2: optimal execution cost entries $Opt(i, j)$.

The algorithm to perform the *Coalesce Layer Merge* follows:

Algorithm:	**Coalesce Layer Merge**	
Input:	L	layered graph produced the Ω-*Cut Merge* algorithm .
	$ArchObj$	an object containing the specifics of the target architecture.
Output:	LR-graph L.	
Description:	checks whether all tasks within a layer; or all tasks within successive layers should be merged to increase performance.	

1. create two m x m matrices M_1 and M_2 where m is the number of layers in L.
2. create an m x m solution matrix, SOL, with each element initialized to 0.
3. $\forall_{i=0}^{m-1}$

   do

   (a) $M_1(i, i) = Seq(i, i)$
   (b) $M_2(i, i) = C_i$
   (c) $SOL(i, i)$ =PARALLEL

   done
4. $\forall_{i=0}^{m-2} \forall_{j=i+1}^{m-1}$
   $M_1(i, j) = Seq(i, j)$
5. $\forall_{i=0}^{m-2} \forall_{j=i+1}^{m-1}$
   $M_1(j, i) = \beta'(L_i, L_j)$
6. $\forall_{j=0}^{m-1} \forall_{i=j}^{m-1}$

do

(a) tmp =MAXFLOAT

(b) $\forall_{k=i}^{j-1}$

do

i. $comm\_cost = 0$

ii. $\forall_{z=i}^{k} \forall_{p=k+1}^{j}$

$comm\_cost = comm\_cost + M_1(p, z)$

iii. $fval = M_2(i, k) + M_2(k + 1, j) + comm\_cost$

iv. **if** $fval < tmp$

do

A. $tmp = fval$

B. $M_2(i, j) = fval$

C. $SOL(i, j) = k$

done

done

(c) **if** $M_1(i, j) < M_2(i, j)$

do

i. $M_2(i, j) = M_1(i, j)$

ii. $SOL(i, j) =$ SERIAL

done

done

7. Process the solution matrix, SOL, described below.
8. **return** L

Before returning, the layers of L are processed according to the entries in the generated solution matrix, SOL. The solution matrix describes how the entire layered graph should be manipulated. The layered graph L is processed by first checking entry $SOL(0, m - 1)$, which describes how layers $L_0 \rightarrow L_{m-1}$ should be processed.

The entries are processed as follows:

$$\text{if } SOL(i,j) = \begin{cases} \text{SERIAL} & \text{merge all tasks in layers } L_i \rightarrow L_j\text{, inclusive, into a single task} \\ \text{PARALLEL} & \text{leave layer } L_i \text{ as is} \\ k\text{, where } k < j & \text{process layers } L_i \rightarrow L_j \text{ in terms of entries } SOL(i,k) \text{ and } SOL(k+1,j) \end{cases}$$

For example, the solution matrix generated by the *Coalesce Layer Merge* algorithm for the layered graph illustrated in Figure 16 is provided in Figure 18.

SOL(i,j)

	0	1	2
0	PARALLEL	0	0
1	0	PARALLEL	1
2	0	0	PARALLEL

Figure 18: Solution matrix generated by the *Coalesce Layer Merge* algorithm.

Entry $SOL(0,2)$ indicates how layers $L_0 \rightarrow L_2$ are to be processed; since $SOL(0,2) = 0$, the layers are processed in terms of entries $SOL(0,0)$ and $SOL(1,2)$, i.e., $k = 0$. Entry $SOL(0,0) =$ **PARALLEL**; hence, layer L_0 is executed in parallel, i.e., no merging is performed. However, entry $SOL(1,2) = 1$ which indicates that layers $L_1 \rightarrow L_2$ are processed in terms of entries $SOL(1,1)$ and $SOL(2,2)$. Since both of these entries are set to **PARALLEL**, no merging is performed. Therefore, for this example, the layered graph remains unchanged.

4.3.4 Intra-Layer Merge

The *Intra-Layer Merge* algorithm is called as the last step of the Ω-Cut heuristic. It is used to enforce the processor requirement for the specified target architecture, i.e., we must guarantee that the number of tasks in any layer L_i of L is less than or equal to k, the number of processors available.

The algorithm to perform the *Intra-Layer Merge* follows:

Algorithm:	**Intra-Layer Merge**	
Input:	L	Layered graph produced by the *Coalesce Layer Merge.* algorithm
	$ArchObj$	an object containing the specifics of the target architecture.
Output:	LR-graph L where each layer of L has at most k tasks.	
Description:	Alters the input layered graph L by merging tasks within each layer until the number of tasks within each layer is no more than k, the number of processors available.	

1. $k = ArchObj.GetCPU()$
2. **foreach** layer L_i of L
   (a) **if** $|L_i| \leq k$ **then**
       continue
   (b) **else**
      i. create k bins each of size b where
      $$b = \frac{\sum_{m \in L_i} W(m)}{k}$$
      ii. label the bins $b_1, b_2, \ldots, b_k$
      iii. let b_j^r represent the residual space available in bin b_j; initially $b_j^r = b$ for $1 \leq j \leq k$.
      iv. **while**$(|L_i| > 0)$
         A. let t' represent the highest cost task in L_i.
         B. place t' in the bin with the most available space, i.e, bin b_l such that $b_l^r \geq b_j^r \quad 1 \leq j \leq k$.
         C. $b_j^r = b_j^r - W(t')$
      v. replace all tasks in L_i with the k tasks formed by merging all tasks in each of the bins $b_1, \ldots, b_k$.

The *Intra-Layer Merge* algorithm first determines which layers of L have more than k tasks. Then, for each of these layers found, tasks are successively merged until exactly k tasks exists. The algorithm uses a binpacking scheme where the number of bins is equal to the number of processors available. The size of each bin is computed by dividing the total task cost for the layer by k. The algorithm then iteratively packs each bin by placing the highest cost task from L_i into the bin with the most available space. Once completed, k tasks are formed by merging all tasks within each of the k bins.

For example, Figure 19 illustrates the intra-layer merging performed on a sample layered graph with $k = 2$. Since layer L_0 exceeds the processor requirement, two bins are created,

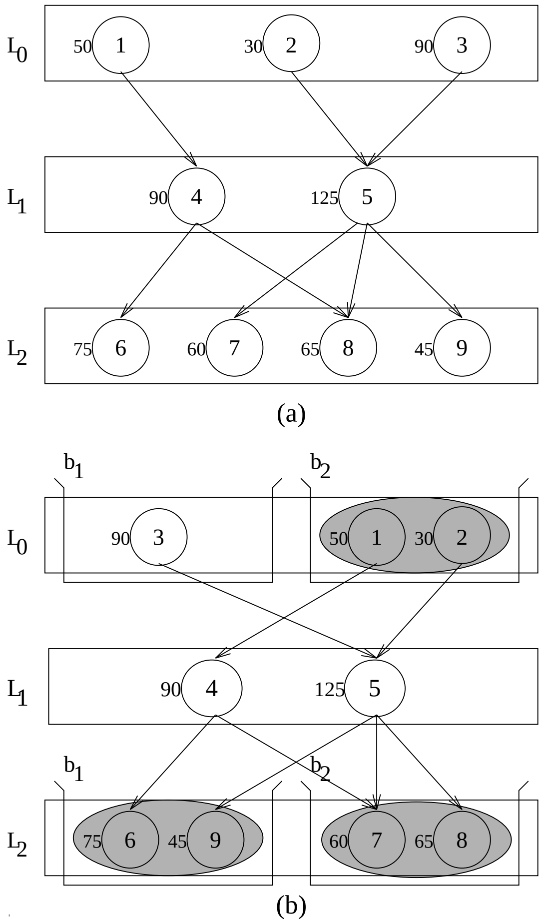

Figure 19: Layer Merging: (a) Sample layered graph; (b) Sample graph after performing *Intra-Layer Merge* algorithm with $k = 2$.

i.e., b_1 and b_2, with binsize $\sum_{m \in L_0} W(m)/k$, or 85. The highest cost task, i.e, t_3 is then placed in b_1 and b_1^r, the residual space left, is set to $b_1^r - W(t_1)$, or -5. The next highest cost task, t_1, is placed in b_2, since this bin has the highest residual space available. The residual space for b_2 is updated, i.e., $b_2^r = b_2^r - W(t_1)$, or $b_2^r = 35$; and finally, t_2 is placed in b_2, since $b_2^r > b_1^r$. No merging is needed in L_1 since it satisfies the processor requirement. In L_2, two merges are necessary. The processing of this layer is analogous to L_0, except for the size of the two bins created, i.e., $b = 122.5$.

The merging performed by this algorithm, though, not explicitly necessary, enforces the computation to proceed in what would normally be expected if no merging were performed. For example, if processor p_1 and p_2 start executing tasks t_1 and t_3 at time 0, respectively; then after 50 time units, processor p_1 would become available and proceed to task t_2, since it is the only task ready for execution; hence, by explicitly merging tasks as described, needless task creation overhead can be eliminated.

4.4 Performance of Ω-Heuristic

The outline of the Ω-Heuristic was previously provided in Figure 10. The *Independent Subgraph Bin-Packing* routine called at step 4(e)ix. of the Ω-*Cut Merge* algorithm runs in $O(n^2)$ where n is the number of tasks. This call occurs in a nested loop, i.e., step 4 and 4(e), which runs in $O(n^2)$; hence the Ω-*Cut Merge* portion of the algorithm runs in $O(n^4)$. The *Coalesce Layer Merge* portion runs in $O(n^3)$. This is due to the fact that the computation of the *comm_cost* in step 6(c)ii can be precomputed since the matrix M_1 has already been initialized in step 3. It is presented this way for purposes of clarily only. The *Intra-Layer Merge* portion runs in $O(n \log n)$, since all $O(n)$ elements must be sorted. The algorithm is dominated by the Ω-*Cut Merge* portion of the algorithm; hence, the worst-case performance of the Ω-Heuristic is $O(n^4)$. However, for most example programs tested, we observe an execution time of $O(n^3)$. This is due to the fact that the task, p', in the LR-graph selected at step 4(f) of the Ω-*Cut Merge*, typically consists of several merged tasks in order to compensate for the task creation cost.

4.5 Example

In this section we present a detailed example of the entire Ω-*Heuristic* algorithm using the DAG illustrated in Figure 7. This example was chosen over the MinMax example presented earlier to better illustrate the subtleties of the algorithm. For this example all communication overhead costs are assumed to be 10, i.e., $\forall_{e_{i,j}} \beta(t_i, t_j) = 10$; and the task creation cost is 20, i.e., $\alpha = 20$.

The Ω-Heuristic is invoked with the DAG D to be scheduled and an object containing all target architecture specifics (see Figure 10).

The **Ω-Cut Merge** algorithm is called to determine the initial layered graph for D. The critical path of D, P_ω is computed, i.e., $P_\omega = \{2, 3, 5, 8, 11\}$. The set B contains all tasks from D which are not in P_ω; hence $B = \{1, 4, 6, 7, 9, 10\}$. In step 4(e) of the algorithm, the tasks in P_ω are iteratively merged in an effort to find the best cut along P_ω. In Figure 20, the layered graph produced by the first iteration is illustrated.

Initially, p' is created by taking the first task along P_ω, i.e., $p' = \{2\}$. The cost savings in constructing p' is 0, since no merging was performed. Now, if there are any tasks $b_k \in B$

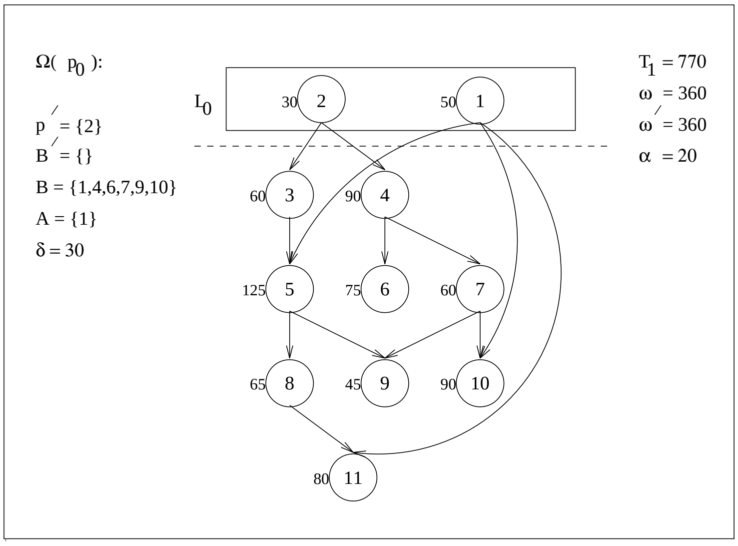

Figure 20: Layered graph generated for $\Omega(p_0)$

such that there exist two tasks $t_i \in p'$ and $t_j \in p'$ where $t_i \prec b_k \prec t_j$; then all such b_k's are merged with p'. Since no such tasks exist, p' remains unchanged. All tasks that must be executed prior to p' are placed in the set B'; since there are no such tasks, $B' = \{\}$. The set A contains all tasks which may be executed concurrently with p'; hence $A = \{1\}$, since all other tasks depend on p'. Once A is computed, the ISBP algorithm is called. A single bin is created for the lone source node in A. The size of the bin, b, is set to $W(p')$; hence $b = 30$. Step 5(d) is performed since there is no task $a_i \in A$ such that $W(a_i) <= b$. This step places task 1 from A in the single bin created. In step 8(c) of ISBP, δ is set to 30, and the the layer represented by L_0 in Figure 20 is returned. Upon returning from the ISBP algorithm, the critical path extension, ω', is computed for the DAG created by performing the merges represented by the returned LR-graph, $L[z]$. Since no merges were performed in $L[z]$, $\omega = \omega'$; hence δ remains unchanged. Therefore, the first iteration yields $\delta = 30$.

In the second iteration the first two tasks along P_ω are merged to create p'; hence $p' = \{2, 3\}$. The layered graph produced by the second iteration is illustrated in Figure 21.

The sets B' and A are computed similarly to the first iteration. The noticeable difference in computing this iteration are with respect to the cost savings achieved. In particular, by merging tasks 2 and 3 to form p' a cost savings of $\alpha + \beta(t_2, t_3)$ achieved; hence $\delta = 30$ after the formation of p'. Also, in step 8(c) of the ISBP, δ is updated by 50 units, rather than 30, in the first iteration; hence the second iteration yields $\delta = 80$.

The layered graph produced by the third iteration is provided in Figure 22. The task p' is formed by merging the first three tasks along P_ω, i.e., $p' = \{2, 3, 5\}$. The cost savings

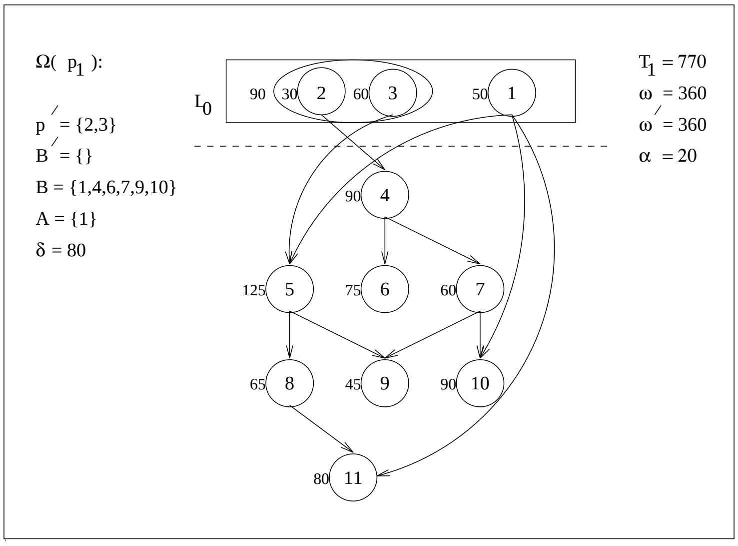

Figure 21: Layered graph generated for $\Omega(p_1)$

produced by this merge is $2\alpha + \beta(t_2, t_3) + \beta(t_3, t_5)$, i.e., $\delta = 60$.

However, in computing this iteration noticeable difference become apparent. In particular, t_1 must now executed prior to p', since $t_1 \prec t_5$ and $t_5 \in p'$. Therefore $B' = \{1\}$. Also, since there are no available tasks which may be executed concurrently with p', $A = \{\}$. The ISBP algorithm generates two layers, L_0 and L_1, where L_0 consists of the single task created by merging all tasks in B' and L_1 consists of p'. Note that no bins are created since there are no available tasks, i.e., $A = \{\}$. Therefore the ISBP algorithm produces no additional cost savings. In addition, the critical path is extended from 360 to 505 time units, which clearly eradicates the earlier cost savings achieved in creating p'. Therefore, the overall cost savings for the third iteration is $\delta = 60 + (360 - 505) = -85$.

In the forth and fifth iterations the extensions to the critical path are further amplified. This is due to the added delay of t_4 by $W(t_8)$ and $W(t_{11})$, respectively. Although the creation of p' saves both task creation costs and communications costs, the added increase to the critical path can not be overcome as illustrated by the graphs in Figure 23 and Figure 24 for the fourth and fifth iterations, respectively. In particular, the fourth iteration produces $\delta = -125$ and, in the fifth, a further increase is observed, as $\delta = -170$.

After all iterations are performed, the highest iteration value for δ is chosen, i.e. iteration 2 illustrated in Figure 21. Therefore, in step 4(f) of the Ω-Cut Merge algorithm, the LR-graph corresponding to iteration 2 is concatenated with the initially empty LR-graph L. Tasks t_1, t_2, and t_3 are then removed from D' and a new P'_ω is recomputed.

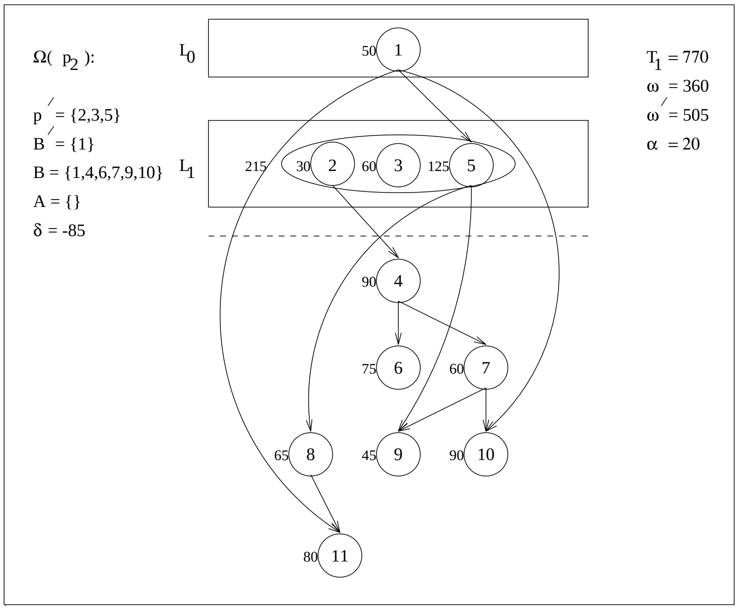

Figure 22: Layered graph generated for $\Omega(p_2)$

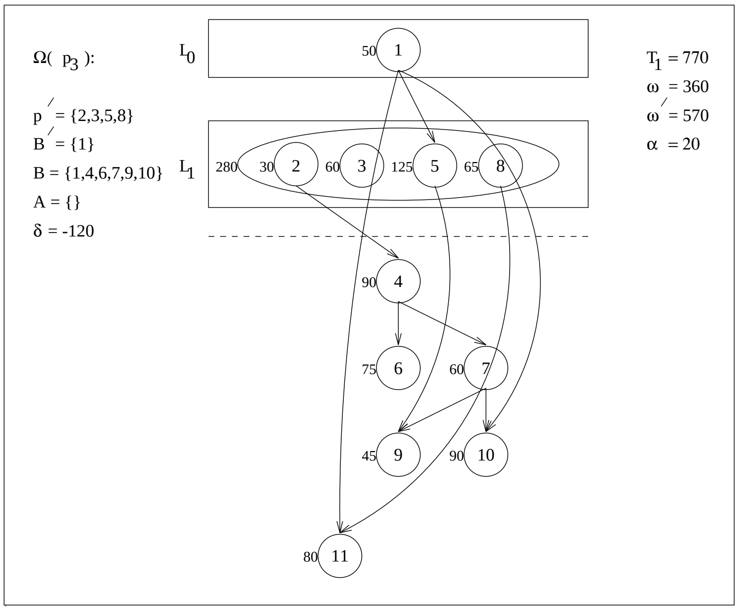

Figure 23: Layered graph generated for $\Omega(p_3)$

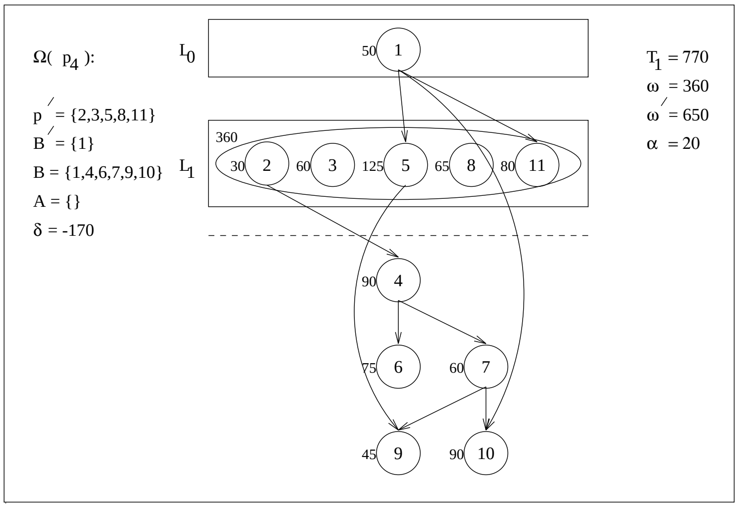

Figure 24: Layered graph generated for $\Omega(p_4)$

The "while-loop" at step 4 of the Ω-Cut Merge algorithm is iterated two more times, with each iteration generating a successive layer of L. The final LR-graph produced by the algorithm is provided in Figure 25. The dashed lines indicate the successive layers produced by step 4 of the algorithm.

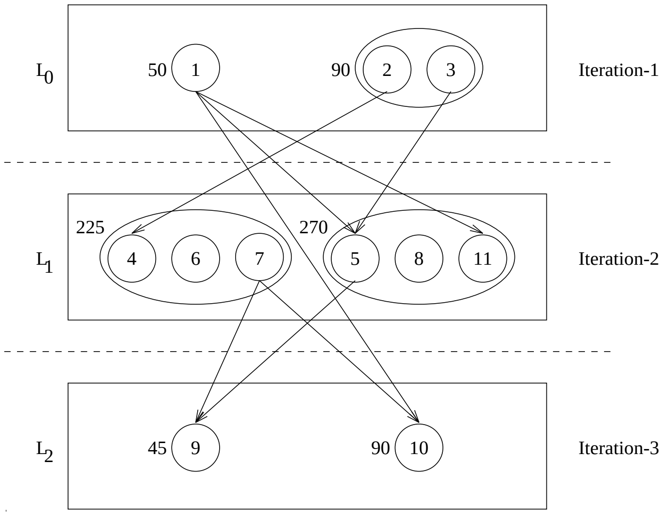

Figure 25: Layered graph produced by Ω-Cut Merge algorithm

Once L is computed, the *Coalesce Layer Merge* algorithm is called to check whether all tasks within a particular layer should be merged; or whether successive layers should be merged into a single task. This was previously done as an example in Section 4.3.3. The result produced no additional changes to the LR-graph. In the last portion of the Ω-*Heuristic*, the *Intra-Layer Merge* algorithm is called. This algorithm ensures that each layer contains no more than k-tasks, where k is the number of available processors for the current target architecture. Since each layer has no more than two tasks, any MIMD architecture satisfies this constraint; hence no changes to the LR-graph are necessary. Therefore, the LR-graph illustrated in Figure 25 is produced by the Ω-*Heuristic*.

In general, the Ω-Heuristic presented in Section 4.2 performs extremely well on all programs tested in our system. In fact, the heuristic produced schedules which are within 5% of the optimal solution, as the graph in Figure 26 demonstrates[1]. These results were attained by running the Ω-Heuristic on 1000 randomly generated DAG's for each value of n and comparing each result to the corresponding optimal solution for different values of α, i.e., $\alpha = 100$, 200, and 2500. The optimal solution is attained by enumerating all possible schedules for each DAG and selecting the schedule with the minimum cost. The task costs are randomly generated in each DAG and range from 100 to 10000 time units; similarly, communication costs range from 5 to 1000 time units.

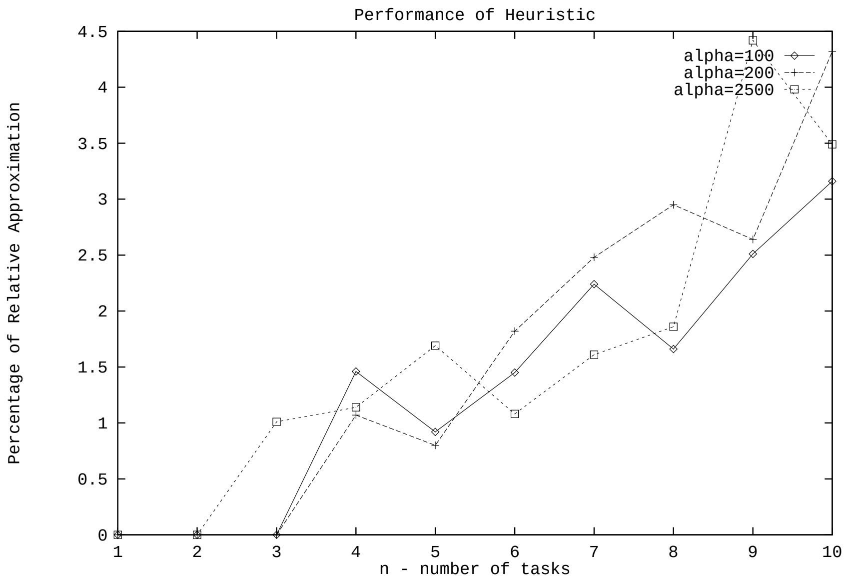

Figure 26: Results

Although we were unable to compute the optimal solution in a reasonable amount of time[2], we are able to compare the schedule generated by the Ω-Heuristic with the theoretical bound on the maximum speedup attainable (see Proposition 4.4). For values of n where $10 < n \leq 30$, the heuristic produced schedules which were within 40% of the maximum theoretical speedup attainable. In fact, these results are much closer, since the theoretical bound does not account for communication costs and task dependencies which are not along the critical

[1]The relative approximation is with respect to the optimal solution.

[2]For $n = 11$, the optimal solution takes approximately 300 minutes of CPU time on a Sparc 10.

path. We should also mention that problem sizes where $n > 30$ are quite rare, since these values correspond to the number of basic blocks at any particular hierarchical level in the source program. This is especially true of C-source code.

This problem has been addressed in earlier work. For a comprehensive description we refer to page 799 in [29]. Our work differs significantly in that our model uses a less restrictive cost function and a much more general program model (since the actual objects to be scheduled may be user defined). Also, we differ from Sarkar's work [29] by providing explicit support for multiple source languages and multiple target architectures. The objects manipulated by the scheduling algorithm are explicit instances of the source code represented with associated algorithms (methods) for data extraction and manipulation. Therefore, our scheduling algorithm works generically in the sense that the scheduler need not explicitly know what the object examined represents. Using our scheduling approach and hierarchical model we initially examine much fewer nodes in producing the schedule. Also, if the speed-up attained is less than desired, our model allows the scheduling algorithm to be applied recursively to nodes of the initial schedule. The results provided in the latter part of Section 5 demonstrate the effectiveness of our approach.

5 Target Code Generation and Evaluation

In this section we demonstrate the technique employed to generate target code for the near optimal schedule produced by the Ω-Heuristic. Our system was developed with the goal of targeting the class of general purpose parallel hardware architectures. This class of hardware architecture is generally referred to as the class *Scalable Symmetric Processors*. Some architectures in this class include the Sun 630 MP/4, Sequent Symmetry S81, Stardent Titan, and the UNISYS 6065. These hardware architectures provide both multiprogramming and multiprocessing environments; and allow for various processor configurations. Executing processes are allowed to migrate over the set of physical processors available, typically based on processor load, that is, there is a shared job queue containing tasks ready for execution. All of these architectures generally run some flavor of Unix as their respective operating systems. Our system may be used to target any architecture with the following features:

1. Unix operating system based on either AT&T Sys 5 or BSD.
2. Support for *semaphore* management.
3. Support for *shared memory* management.
4. ANSI C compiler.

In particular, all target code generation is handled directly by the *LayerRep* object produced by the *Compile-Time Scheduler*. ANSI C is generated as the target-code for each supported target architecture. This choice of target-code is very appealing since the native compiler of each architecture is used to generate the final executable. In particular, vendors of each architecture typically invest many years in developing compilers to exploit the underlying hardware. By generating a high-level language as target-code, and then using the language specific compiler supplied by the vendor, all architecture features and idiosyncrasies are easily accounted for. In fact, since some vendors compilers perform automatic vectorization, i.e., Stardent Titan, these types of optimizations can only further enhance performance.

Since each task represented by the *LayerRep* object is responsible for generating its own code, the *LayerRep* object need only assemble the code for each task and insert the necessary primitives for controlling communication and access of data. This greatly simplifies the complexity of the code generator since the low-level details of how to generate code for individual HICOR objects is handled by the respective objects themselves; the code generator merely delegates commands to and collects the respective results.

For example, the LR-graph provided in Figure 27 illustrates a simple task graph representing the computation of $c = a + b$. Both tasks in the first layer may be executed in parallel, once completed, the task in the third layer may executed.

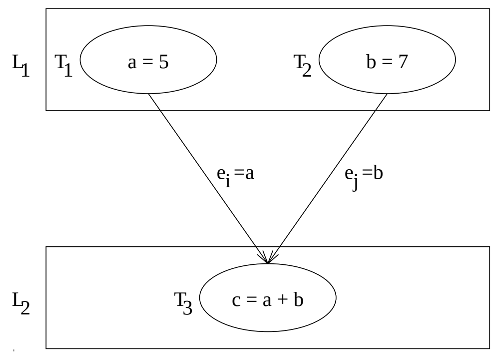

Figure 27: LR-graph containing three tasks, T_1, T_2, and T_3, with communication edges e_i and e_j

The corresponding code for the LR-graph is provided in Figure 28. The generated code closely resembles the structure of the LR-graph. Each task, T_i, is directly mapped into a corresponding function P_i. The use of $\textbf{shared}(x) = y$ declares variable x in shared memory with initial value y. If x was already declared by a prior **shared** statement, the new value is simply replaced. When $\textbf{shared}(x)$ occurs as a *rval*, i.e, $y = \textbf{shared}(x)$, the current value of x in shared memory is retrieved, and in this case, assigned to y.

```
P_1()
{
        int a;
        T_1.GenCode();
        shared(a_1) = a;
        signal(sem_{1,3});
}

P_2()
{
        int b;
        T_2.GenCode();
        shared(b_2) = b;
        signal(sem_{2,3});
}

P_3()
{
        int a, b, c;
        wait(sem_{1,3});
        wait(sem_{2,3});
        a = shared(a_1);
        b = shared(b_2);
        T_3.GenCode();
}
```

Figure 28: Output code for LR-graph. Each task is converted to a corresponding procedure with edges translated to shared memory and semaphore control statements

Our design incorporates the use of a driver routine which creates a separate Unix process for each function, P_i, via the **fork**() system call [30]. As each process executes, the control primitives, **signal** and **wait**, are used to control access to shared memory. Hence, the process executing P_3, can not execute the code for task T_3 until both executing processes for P_1 and P_2 have sent their respective signals. The actual driver routine generated by the system is provided in Figure 29. The driver routine is general enough to handle any arbitrary *LR*-graph, since the number of tasks to be created can be retrieved a priori, via the available methods.

```
driver()
{
        int i, pid;
        for(i = 1; i ≤ 3; i++)
        {
                pid =fork();
                switch(pid)
                {
                        case0: /* child executes here */
                                switch(i)
                                {
                                        case1: P1(); exit();
                                        case2: P2(); exit();
                                        case3: P3(); exit();
                                }
                        default: /* parent executes here */
                                continue;
                }
        }
}
```

Figure 29: Driver routine used to create processes to execute each task in the *LR*-graph.

The code-generation scheme employed for explicit HICOR objects works equally well for exploiting loop-level parallelism. The *LoopBlock* object contains several methods for determining loop-level dependencies. In particular, a method called IsIterIndependent may be invoked to determine if loop iterations may be performed independently, i.e., there are no loop-carried dependencies. If so, the code-generator constructs a seperate procedure body to execute a range of loop iterations. The driver routine presented in Figure 29 remains unchanged. Other methods may be used to extract ranges of independent loop iterations adn to check for dependencies between linear array index equations. Other methods have been implemented to perform much of the loop-level parallelization techniques as described in [9, 13, 23].

In order to demonstrate the parallelization capabilities of the system several sample programs have been written and tested. The programs evaluated were written in both ANSI C and Fortran and contain no compiler directives. In particular, the following programs are used to demonstrate the overall effectiveness of the system.

- **Conv**: performs a convolution operation on an 50,000 element array.
- **MatMul**: initializes and multiplies two 1000x1000 element matrices.
- **MinMax**: locates both the minimum and maximum values within an 100,000 element array.
- **Vector**: multiplies the respective positions of two 10,000 element arrays together, and then adds a constant value to each result.

The target code generated by our system for each of the above example programs was compiled and executed on each architecture for all CPU values from 1 to n, where n is the maximum number of CPU's available. In all cases the C and Fortran versions produced the same results. The execution times (in seconds) along with the observed speedup for each program on the Sun 630 MP/4 is provided in Table 1.

As illustrated in Figure 30, the measured speedup attained for each program on the Sun 630 MP/4 is quite good. In fact, a near linear speedup may be observed with larger deviations at higher CPU values due to the increased overhead in accessing shared memory.

The execution times for the Stardent Titan are provided in Table 2. However, since the Titan provides a parallelizing compiler the results of our system may be compared directly to the parallel code generated by the native compiler. Hence, the execution times provided as unoptimized were obtained by executing the native compiler on the target code generated by our system with parallel optimizations disabled. Similarly, the optimized times were obtained by enabling parallel optimizations.

As illustrated in Figure 31, a near linear speedup may be observed for the Stardent Titan. As with the Sun 630 MP/4, the deviations at higher CPU values can be attributed to increased overhead in accessing shared memory.

In Figure 32 a comparison of the parallel code generated by our system with the parallel code generated by the native compiler for the Titan is provided. As illustrated, an average

Sun 630 MP/4			
Program Name	*CPU's*	*Execution Time*	*Observed Speedup*
Conv	1	460.90	1.00
	2	285.35	1.62
	3	223.50	2.06
	4	170.51	2.70
MatMul	1	30.00	1.00
	2	16.90	1.78
	3	15.07	2.00
	4	13.66	2.19
MinMax	1	25.24	1.00
	2	12.93	1.95
	3	11.00	2.30
	4	8.40	3.00
Vector	1	37.40	1.00
	2	19.00	1.97
	3	15.00	2.50
	4	13.98	2.68

Table 1: Execution time for each of the tested programs on the Sun 630 MP/4

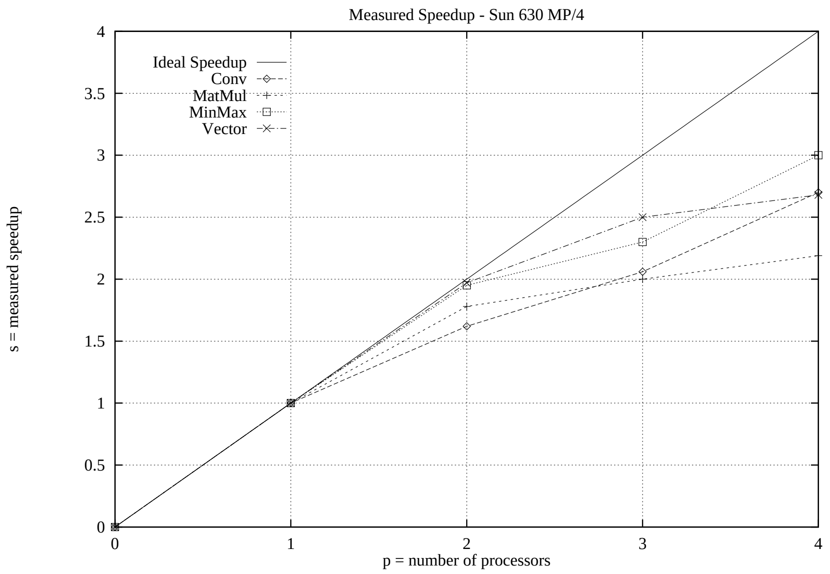

Figure 30: Measured speedup for the Sun 630 MP/4

Stardent Titan				
Program Name	*CPU's*	*Execution Time (unoptimized)*	*Execution Time (optimized)*	*Observed Speedup*
Conv	1	486.00	87.00	1.00
	2	369.87	71.08	1.31
	3	272.86	53.50	1.78
	4	218.08	43.10	2.23
MatMul	1	37.62	5.74	1.00
	2	20.45	4.81	1.84
	3	14.86	4.43	2.53
	4	11.73	4.13	3.21
MinMax	1	35.20	3.45	1.00
	2	18.00	1.87	1.96
	3	12.08	1.62	2.91
	4	9.28	1.23	3.79
Vector	1	53.29	4.13	1.00
	2	27.12	2.86	1.96
	3	18.18	2.07	2.93
	4	13.77	1.94	3.87

Table 2: Execution time for each of the tested programs on the Stardent Titan

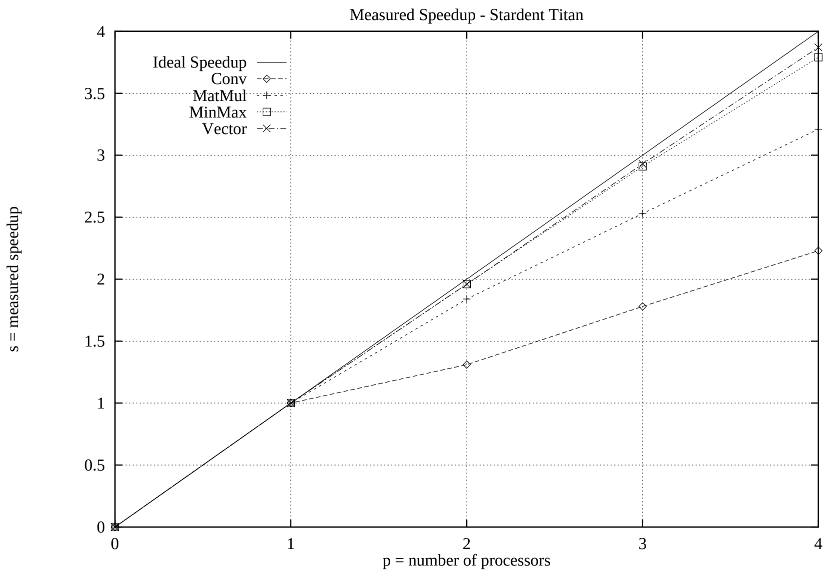

Figure 31: Measured speedup for the Stardent Titan

speedup of nearly two times is observed for each program tested. This additional speedup may be attributed to the implicit interprocedural analysis performed by our system, as the Titan compiler does not perform this type of analysis.

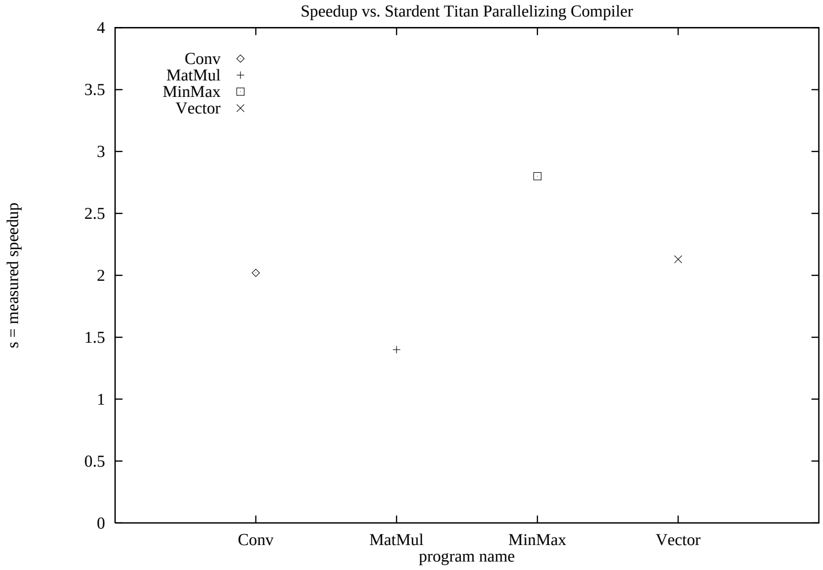

Figure 32: Measured speedup versus the Stardent Titan parallelizing compiler

The execution times for each program on the Sequent Symmetry S81 are provided in Table 3. Although, the Sequent has sixteen processors, a maximum of fifteen may be claimed by any single program; hence, no data can be obtained for sixteen CPU's.

As illustrated in Figure 33, a near linear speedup is observed for the Sequent Symmetry S81. As with the Sun and Titan, these deviations can be attributed to the added overhead of accessing shared memory. Although, the overhead on the Sequent is minimal since simultaneous reads from shared memory is supported.

The previous examples illustrate that our system can successfully parallelize sequential code for each of the tested architectures. In particular, a near linear speedup was observed for each architecture as the number of available CPU's was increased. Also, with respect to the Stardent Titan, the representation techniques developed, clearly provided better results than the existing parallelizing compiler.

Sequent Symmetry S81								
CPU's	*Program Name*							
	Conv		MatMul		MinMax		Vector	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1	480.35	1.00	61.10	1.00	61.93	1.00	61.89	1.00
2	240.41	1.99	30.57	1.99	31.00	1.99	30.96	2.00
3	160.74	2.98	20.44	2.99	20.69	2.99	20.65	3.00
4	120.59	3.98	15.33	3.99	15.55	3.98	15.50	4.00
5	96.88	4.96	12.29	4.97	12.47	4.96	12.41	4.98
6	81.22	5.91	10.47	5.84	10.42	5.94	10.37	5.97
7	70.06	6.85	9.14	6.68	8.97	6.90	8.91	6.95
8	61.54	7.80	7.75	7.88	7.86	7.88	7.79	7.95
9	55.34	8.68	6.98	8.76	7.02	8.82	6.94	8.92
10	49.91	9.62	6.25	9.77	6.34	9.77	6.25	9.90
11	46.04	10.43	5.78	10.57	5.79	10.70	5.70	10.85
12	42.78	11.22	5.50	11.12	5.34	11.60	5.26	11.78
13	40.15	11.96	5.26	12.00	4.95	12.51	4.86	12.73
14	37.05	12.96	4.91	12.43	4.64	13.35	4.53	13.66
15	35.19	13.65	4.87	12.54	4.39	14.10	4.28	14.46

Table 3: Execution time for each of the tested programs on the Sequent Symmetry S81

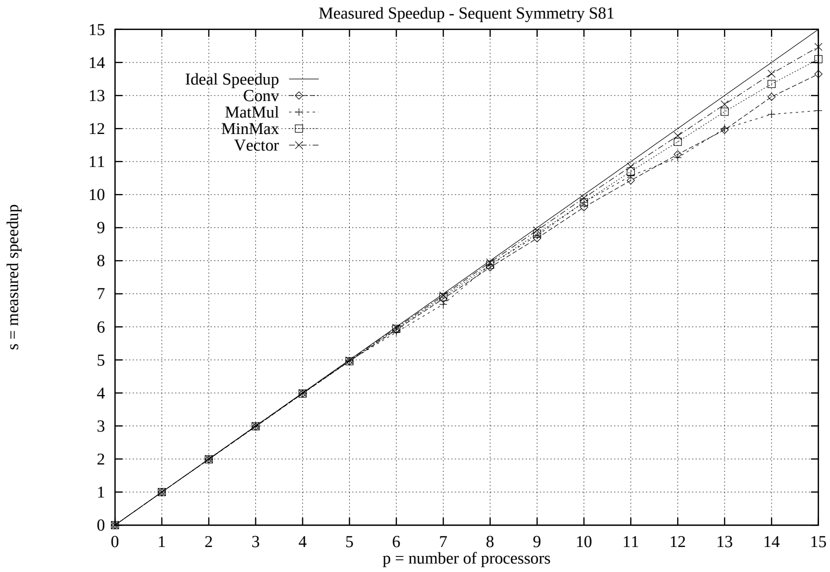

Figure 33: Measured speedup for the Sequent Symmetry S81

6 Conclusion

We have shown that the implementation of parallelization tools can be greatly simplified by using an object-oriented design to construct the intermediate representation. In particular, a tool to perform *Compile-Time Scheduling* was presented in Section 4. This tool differs from traditional scheduling in that merging of tasks is performed automatically to find the proper grain size for successful parallelization. The Ω-Heuristic was shown to produce schedules within 5% of the optimal solution with a near linear speedup as the number of processors increase. More importantly, by using the interface layer to HICOR, we have shown that the tool needs no knowledge of the underlying intermediate representation, and therefore cannot be affected by any internal changes. Tools constructed in this manner are inherently simpler since they can manipulate the representation at a much higher level of abstraction. Other tools constructed using this approach are described in [10, 11].

The object oriented approach employed clearly has its advantages over traditional approaches and has helped us greatly in the design and implementation of our system. This, in turn, has facilitated our ability to interface with other graph and algorithmic packages such as GraphPack [19] for visualization purposes and Maple [5] which generates C code. The success of our system is attributed to the construction of objects and associated methods to encapsulate the underlying intermediate representation. The design allows all high-level constructs available at the source code level to remain consistent throughout all phases of the compilation and therefore simplifies the extraction of data necessary to generate parallel code.

Acknowledgements: The authors wish to thank the three anonymous referees whose comments helped improve the presentation of the paper.

References

[1] A. V. Aho and M. Ganapathi. Efficient Tree Pattern Matching: An Aid to Code Generation. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 334–340, New York, NY, January 1985.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, March 1986.

[3] W. Baxter and H. R. Bauer, III. The Program Dependence Graph and Vectorization. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, Austin, TX, January 1989.

[4] M. Burke, R. Cytron, J. Ferrante, and Wilson Hsieh. Automatic Generation of Nested, Fork-Join Parallelism. *Journal of Supercomputing*, 2(3):71–88, July 1989.

[5] B. W. Char, K. O. Geddes, G. H. Gonnet, M. B. Monagagn, and S. M. Watt. *Maple Reference Manual*, 1988.

[6] K. D. Cooper and K. Kennedy. Fast Interprocedural Alias Analysis. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, Austin, TX, January 1989.

[7] R. Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, pages 5–16, February 1990.

[8] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[9] D. Gelernter, A. Nicolau, and D. Padua. *Languages and Compilers for Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[10] M. R. Gilder. *An Object Oriented Intermediate Code Representation For An Architecturally Independent Parallelizing Compiler.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1993.

[11] M. R. Gilder and M. S. Krishnamoorthy. A Viable Object Oriented Intermediate Code Representation for Parallel Architectures. Technical Report TR 92-35, Rensselaer Polytechnic Institute, Troy, NY, December 1992.

[12] M. R. Gilder and M. S. Krishnamoorthy. An Object-Oriented Intermediate Code Representation for the Development of Parallelization Tools. *Journal of Object-Oriented Programming (To Appear)*, 1994.

[13] M. Girkar and C. Polychronopoulos. Compiling Issues for Supercomputers. *IEEE Computer*, pages 164–173, September 1988.

[14] M. Girkar and C. Polychronopoulos. Automatic Extraction of Functional Level Parallelism from Ordinary Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, March 1993.

[15] G. Goff, K. Kennedy, and C. Tseng. Practical Dependence Testing. In *Proc. of the SIGPLAN '91 Conf. on Programming Language Design and Implementation*, volume 26, pages 15–29, June 1991.

[16] M. Harrold and M. Soffa. Computation of Interprocedural Definition and Use Dependencies. In *Proc. 1990 IEEE Int. Conf. on Computing Languages*, pages 297–306, New Orleans, March 1990.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, June 1988.

[18] K. Kennedy, K. S. Mckinley, and C. W. Tseng. Interactive Parallel Programming Using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[19] M. S. Krishnamoorthy, A. Suess, M. Onghena, and F. Oxall. Improvements to GraphPack: A System to Manipulate Graphs and Digraphs. In *DIMACS Workshop on Computational Support for Discrete Mathematics*, 1993 (To appear).

[20] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[21] C. McCreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32(9):1073–1078, September 1989.

[22] R. E. Miller and J. W. Thatcher. *Complexity of Computer Computations*. Plenum Press, New York, NY, 1972.

[23] D. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[24] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: An Algebraic Approach to Program Dependencies. Technical Report TR 90-1152, Cornell University, September 1990.

[25] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic, Boston, MA, 1988.

[26] J. R. Punin, M. R. Gilder, and M. S. Krishnamoorthy. A GUI for Parallel Code Generation. In progress.

[27] S. Richardson and M. Ganapathi. Code Optimization Across Procedures. *IEEE Computer*, pages 42–50, February 1989.

[28] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.

[29] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(5):779–804, September 1991.

[30] R. W. Stevens. *UNIX Network Programming.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[31] B. K. Szymanski. *Parallel Functional Languages and Compilers.* Addison-Wesley, Reading, MA, 1991.

[32] Y. Tanaka, K. Iwasawa, S. Gotoo, and Yukio Umetani. Compiling Techniques for First-Order Linear Recurrences on a Vector Computer. *IEEE Computer*, pages 174–181, September 1988.

[33] A. E. Terrano, S. M. Dunn, and J. E. Peters. Using an Architectural Knowledge Base to Generate Code for Parallel Computers. *Communications of the ACM*, 32(9):1065–1072, September 1989.

[34] J. D. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

A Notation

S : set of tasks to be scheduled.

n : number of tasks in S.

t_i : a particular task of S, where $1 \leq i \leq n$.

D : initial DAG computed by HICOR tool which consists of all tasks in S.

L : layered graph derived from D.

L_i : set of all tasks in the i-th layer of L.

D_{opt} : near optimal execution DAG derived from D.

P_ω : defines the critical path of D where the *critical path* is defined to be the highest cost path of D in terms of execution time.

p_i : the i-th task along P_ω.

ω_{opt} : optimal schedule time, i.e., the minimum completion time over all possible schedules.

W : function defining the cost, or execution time, for a given task in S.

α : represents the task creation cost for a given architecture.

β : function defining the communication cost, in terms of processing time, between any two distinct tasks in S.

$\beta'(L_i, L_j)$: function defining the overall communication costs of the layered graph L with respect to layers L_i through L_j, inclusive.

ω : execution time of all tasks along the critical path, P_ω.

k : number of available processors for the given architecture.

T_1 : serial execution time of D.

T_k : parallel execution time of D with k-processors.

C_i : function defining the cost of executing all tasks in layer L_i.

$Seq(i, j)$: function defining the sequential execution cost of all tasks in layers L_i through L_j of L, where $i \leq j$.

$Opt(i, j)$: function defining the optimal execution cost for all tasks in layers L_i through L_j of L, where $i \leq j$.

$S_{\max}$: optimal speedup attainable for D with unlimited processors.

$s_{\min}$: minimum speedup required to generate parallel code. This value is specified by the user.

S_k : optimal speedup with k-processors.