A Comparison of Heuristics for Scheduling DAGs on Multiprocessors

C.L. McCreary, A.A. Khan, J. Thompson, M.E. McArdle,
Department of Computer Science and Engineering
Auburn University, AL 36849
mccreary@eng.auburn.edu

Abstract

Many algorithms to schedule DAGs on multiprocessors have been proposed, but there has been little work done to determine their effectiveness. Since multi-processor scheduling is an NP-hard problem, no exact tractable algorithm exists, and no baseline is available from which to compare the resulting schedules. This paper is an attempt to quantify the differences in a few of the heuristics. The empiracle performance of five heuristics is compared when they are applied to ten specific DAGs which represent program dependence graphs of important applications. The comparison is made between a graph based method, a list scheduling technique and three critical path mathods.

1. Introduction

One of the primary problems in creating efficient programs for multiprocessor systems with distributed memory is to partition the program into tasks that can be assigned to different processors for parallel execution. If a high degree of parallelism is the objective, a greater amount of communication will be required between the tasks. On the other hand, if communication is restricted, potential parallelism will be lost. The goal of partitioning heuristics is to divide the program into the appropriate size and number of tasks to balance the two conflicting requirements of low communication overhead and a high degree of parallelism. The most effective partitioning scheme will minimize the parallel running time (elapsed time from the start of the first processor to the completion of the last processor.)

A sequential program is commonly represented as a Program Dependence Graph (PDG) which is a directed acyclic graph (DAG) with node and edge weights [21]. Each vertex in a PDG denotes a task and a weight, which represents its processing time. Each edge denotes the precedence relation between the two tasks, and the weight of the edge is the communication cost incurred if the two tasks are assigned to different processors. Given a PDG, the graph is partitioned into appropriately sized grains which are assigned to processors of a parallel machine. A good assignment will shorten the execution time of the program. The partitioning and assignment is called the *scheduling* problem. The problem is also known as *grain size determination* [12], *the clustering problem* [11, 24], and *internalization pre-pass* [21]. The problem is important because solution methods can be used to generate efficient parallel programs.

The partitioning/scheduling problem is intractable, and heuristics are required to find sub-optimal solutions. In addition, there are no known performance guarantees for scheduling heuristics for general graphs. Many researchers have proposed algorithms, but little work has been done to determine the performance of the resulting schedules. Since neither analytic nor absolute results are known, the goal of this paper is to determine the relative performance of some promising techniques. The techniques are applied to graphical representations of some real programs.

In classical scheduling, communication costs are not considered [3, 7]. Introducing the communication cost is necessary because communication between processors does take time in real parallel systems, especially in distributed memory systems where communication costs tend to be high relative to processor speed. The challenge in the extended scheduling problem is to consider the trade-off between communication time and degree of parallelism [17].

Many researchers have studied the problem and proposed solutions. This paper considers algorithms from three caregories: critical path heuristics, list scheduling heuristics and graph analysis.

•Critical path heuristics [4, 5, 11, 21, 23, 24]: For DAGs with edge weights and node weights, a path weight is defined to be the sum of the weights of both nodes and edges on the path. A *critical path* is a path of greatest weight from a source node to a sink node. Extending the critical path method due to Hu [9] in classical scheduling, these algorithms try to shorten the longest execution path in the DAG. Paths are shortened by removing communication requirements (zeroing edges) and combining the adjacent tasks into a grain. This approach has received the most attention. A taxonomy of these techniques as well as a comparison of four specific heuristics can be found in the work of Gerasoulis and Yang [4]. In this paper we include experimental results from three critical path algorithms, DSC, Linear Clustering and MCP.

•List scheduling heuristics [1, 10, 12, 13, 14, 18, 20]: These algorithms assign priorities to the processes and schedule them according to a list priority scheme. For example, a high priority might be given to a task with many heavily weighted incident edges or to a task whose neighbors have already been scheduled. Extending the list scheduling heuristic in classical scheduling [6], these algorithms use greedy heuristics and schedule tasks in a certain order. Task duplications have been used in [12,1, 20] to reduce the communication costs. One list scheduling method, the LAST algorithm, is included in this comparison study.

•Graph decomposition method [15, 16]: Based on graph decomposition theory, the method parses a graph into a hierarchy (tree) of subgraphs. Communication and execution costs are applied to the tree to determine the grain size that results in the most efficient schedule. The only published method in this category, CLANS, is included for comparison.

This paper describes five partitioning/scheduling schemes, executes those schemes on ten different PDGs, identifies the resulting schedules, and compares the parallel time and efficiency. The PDGs include very regular and very irregular graphs representing real program dependence graphs.

2. Problem Definition and Assumptions

The problem of parallelizing the PDG to achieve minimal parallel time is NP-hard [21]. Two distinct steps are usually described in this parallelization process: *partitioning* and *scheduling*. *Partitioning* combines tasks from the PDG onto groups of tasks called *grains* to be executed on the same processor. The optimal size of a grain is dependent on the characteristics of the target architecture. Efficient programs on processors whose inter-processor communication costs are high relative to processing costs require larger grains than they would on processors with low inter-processor communication costs. *Scheduling* assigns grains to processors, insuring that all grain inputs are available at the scheduled start time. Associated with each scheduled grain is a processor number, a start time and a completion time (start time plus execution time).

To achieve valid comparisons between scheduling heuristics, the execution models, underlying architectures and objective functions for the heuristics must be identical. The following assumptions hold for all heuristics described in this paper:

(1) All methods use a DAG representing the PDG as input and consider the costs of inter-task communication. Two tasks scheduled on the same processor incur no communication costs, and any two tasks scheduled on two different processors incur the communication cost specified by the edge weight no matter what the processor assignments.

(2) The architecture is a network of an arbitrary number of homogeneous processors. This simplified model allows schedulers to concentrate on the essential problem of task scheduling.

(3) Duplication of the execution of tasks in separate grains is not allowed. Several heuristics [12, 1, 20] have been developed that take advantage of this option, and these algorithms deserve further analysis. However, it is beyond the scope of this paper to consider them.

(4) Tasks communicate only before starting and after completing their execution. Communication occurs simultaneously with computation and tasks communicate via asynchronous mechanisms whether message passing or shared memory. Tasks may send and receive data in parallel and may multicast messages to more than one receiver.

(5) The objective function of all the heuristics is to minimize parallel time. Some scheduling heuristics attempt to minimize communication time and others, such as load balancing, attempt to keep all processors busy. Since the goal of high performance computing environments is to execute programs quickly, parallel time is the objective that is most meaningful for the scheduling problem.

3. The Heuristics

This sections describes the five competing heuristics. They are divided into three categories: critical path techniques; list processing techniques; and graph analysis. The three critical path techniques include linear clustering, the work of Kim and Browne; the dominant sequence clustering (DSC) algorithm of Gerasoulis and Yang; and modified critical path (MCP) algorithm of Wu and Gajski. The list processing technique is LAST, the work of Baxter and Patel; and the graph analysis technique is the clan-based graph decomposition work (CLANS) of McCreary and Gill. The computational complexity of each heuristic is given in terms of the number of graph vertices (v) and edges (e).

3.1 Critical path techniques

In each of the critical path techniques, partitions are formed by examining the current critical path, zeroing an edge (combining the incident nodes into a cluster) in that path and repeating the process on the new critical path. Gerasoulis and Yang use the term *dominant sequence* to mean the critical path after the zeroing of one or more edges. The heuristics differ in their method of selecting the edge(s) to be zeroed and identifying the new dominant sequence.

3.1.1 Linear clustering

The linear clustering (LC) algorithm recursively groups all nodes in the critical path, zeroing all edges on the path in one step. In a second phase, it schedules on the same processor all partitions that do not execute concurrently.

The example DAG for LC shown in Figure 1 is different from the DAG that illustrates the other algorithms so as to demonstrate the second phase of the algorithm. The merging phase is the section where the non-overlapping clusters are grouped together and the critical cluster is merged with nodes in minor clusters for further reduction in parallel time. On Figure 2, LC generates the same the clustering as DSC. The computational complexity of LC is $O(v(e+v))$.

The Algorithm (LC):

Phase I: clustering

1. Determine the critical path
2. Create a partition by zeroing the edges on the critical path.
3. Cut (remove) all edges incident to the critical path.
4. Recursively apply steps 1, 2 and 3 until all paths have been examined.

Phase II: merging - reduce the number of processors required by combining clusters

5. Identify clusters whose busy times (from the cluster start time to the cluster completion time) do not overlap. Merge any such clusters.
6. Given the cluster with the longest estimated computation time, choose edge (i,j) where i is in the cluster and j is adjacent to the cluster. If the estimated completion time is reduced when vertex j is merged with the cluster containing i, then accept the merge.
7. Repeat step 6 until no further reduction is found.

3.1.2 Dominant Sequence Clustering (DSC)

DSC by Yang and Gerasoulis [4] is also an edge zeroing algorithm. The two major ideas behind DSC are to directly attempt to reduce the Dominant Sequence of the graph, and to create an algorithm with low computational complexity, This algorithm keeps track of the dominant sequence

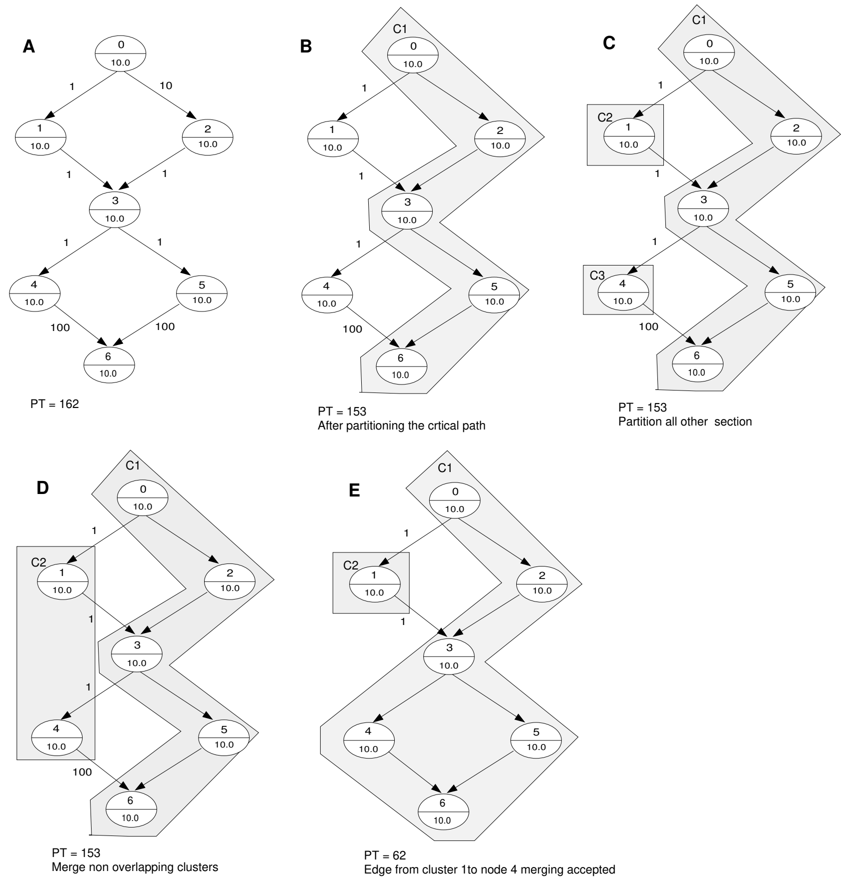

Figure 1
Example: LC

to reduce parallel time. The order of complexity of the algorithm is also reduced to *O((v+e)log v)* by decreasing the focus range of the steps in the execution.

To describe DSC we need to specify certain constraints and definitions.

Node types:

scheduled: A node is scheduled if it has been assigned to a processor.

free: A node is free if it is unscheduled and all its predecessors are scheduled.

partial free: A node is partial free if it is unscheduled and at least one of its predecessors is unscheduled.

The reduction in the focus range is achieved by maintaining lists of *free* and *partial free* nodes the lists are ordered according to priority as defined below. At any one iteration in the algorithm the focus is on the nodes on the first elements in these lists and the outgoing edges of the free node in question. This reduces the complexity by confining the range of edges to be zeroed.

The Algorithm (DSC):

```
Mark all nodes as unscheduled
WHILE there are unscheduled nodes DO
  In descending order of priority, sort list of free nodes
  In descending order of priority, sort list of partial free nodes
   Let n_x  be the free task with highest priority.
   Let n_y be the partial free task with highest priority.
   IF(priority(n_x)≥priority(n_y)) THEN
        IF ( CT1) THEN  accept zeroing for edge (c_i, n_x ) where c_i gives min(ST(c_i, n_x))
                  ELSE  schedule n_x on new cluster
        ENDIF
    ELSE
         IF ( CT2 & CT1) THEN  accept zeroing for edge (c_i, n_x ) where c_i gives min(ST(c_i, n_x))
                             ELSE  schedule n_x on new cluster
          ENDIF
     ENDIF
      Mark  n_x as scheduled
      Recalculate priorities & find new free & partial free nodes
ENDWHILE

DEFINE  CT1:  /* Constraint 1 guarantees that parallel time is not increased */
  For all clusters c_i that are parents of n_x (min_start_time = MAXINT)
  {
       IF ( min_start_time > ST( c_i , n_x)) THEN min_start_time = ST(c_i , n_x)
  }
   IF ( startbound(n_x) < min_start_time ) THEN
                        return TRUE
        ELSE
                        return FALSE
   ENDIF
 ENDDEF

DEFINE CT2: /* Constraint 2 guarantees that the starttime of partial free nodes is never increased */
 For all scheduled clusters c_i that are parents of n_y
 {
      IF (startbound(n_y) < ST(c_i , n_y )) THEN
                        return FALSE
     ENDIF
  }
  return TRUE
ENDDEF
```

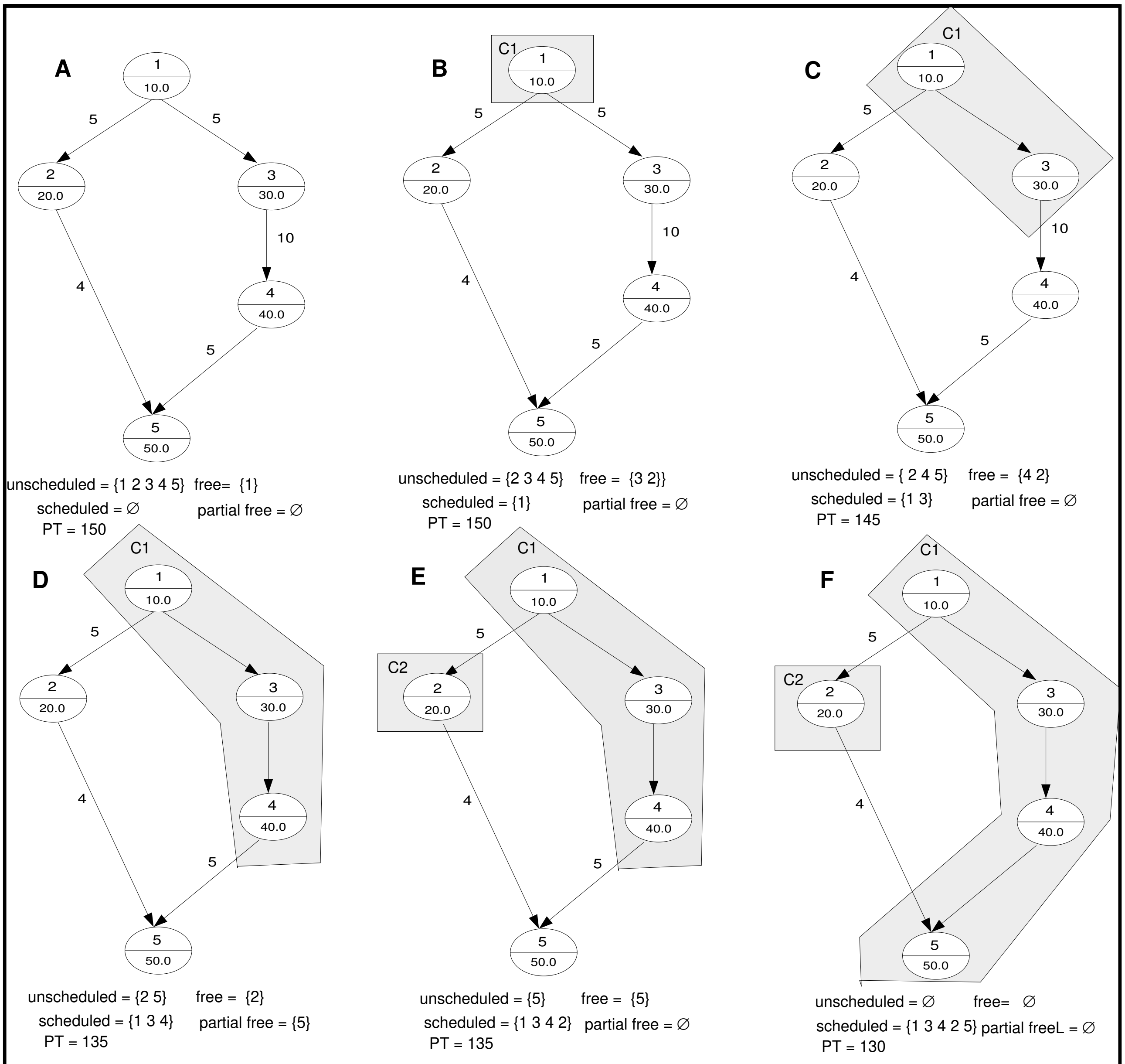

Figure 2
Example: DSC

Timing Values:

$ST(n_x)$: the starting time for n_x when scheduled on an independent cluster.

$ST(c_i, n_x)$: starting time for n_x when scheduled in cluster c_i.

$ET(n_x)$: the execution time of n_x.

$level(n_x)$: the length of the longest path from the start of n_x to an exit node.

$arrivetime(n_j, n_x) = ST(n_j) + ET(n_j) + e_{jx}$ where n_j is a scheduled predecessor of n_x, and e_{jx} is the cost of edge (n_j, n_x).

$startbound(n_x) = max(arrivetime(n_j, n_x))$, where $n_j \in SG \cap PRED(n_x)$. This is the lower bound for starting n_x .

$priority(n_x) = startbound(n_x) + level(n_x)$.

Figure 2 illustrates the steps in the execution of DSC.

3.1.3 Modified Critical Path (MCP)

MCP, proposed by Wu and Gajski, is similar to DSC in its basic heuristic to minimize the starting time of individual tasks. The term *as-late-as-possible* (ALAP) binding, represents the latest

The Algorithm (MCP):

List_of_PEs = ∅

1. Perform ALAP binding and assign the resulting ALAP time T_L to each node in the graph.

2. For each node n_i create a list $l(n_i)$ which consists of T_L's of n_i and all it's descendents, sort $l(n_i)$ in decreasing order of T_L.

3. Create L by Concatenating the T_L values for each $l(n_i)$ and sort in decreasing order.

4. Schedule *head(L)* to a Processing Element (PE_1) and remove *head(L)*
   List_of_PEs ← PE_1(*head(L)*)

5. While L is not empty
   Compute min(start_time) when *head(L)* is placed on PE_i in List_of_PEs.
   If min($start\_time_{exist\_cluster}$) > start time of head(L) when placed on a new PE
   then
   add new PE to List_of_PEs and place head(L) on new PE
   else
   place head(L) on PE that gives min(starttime)

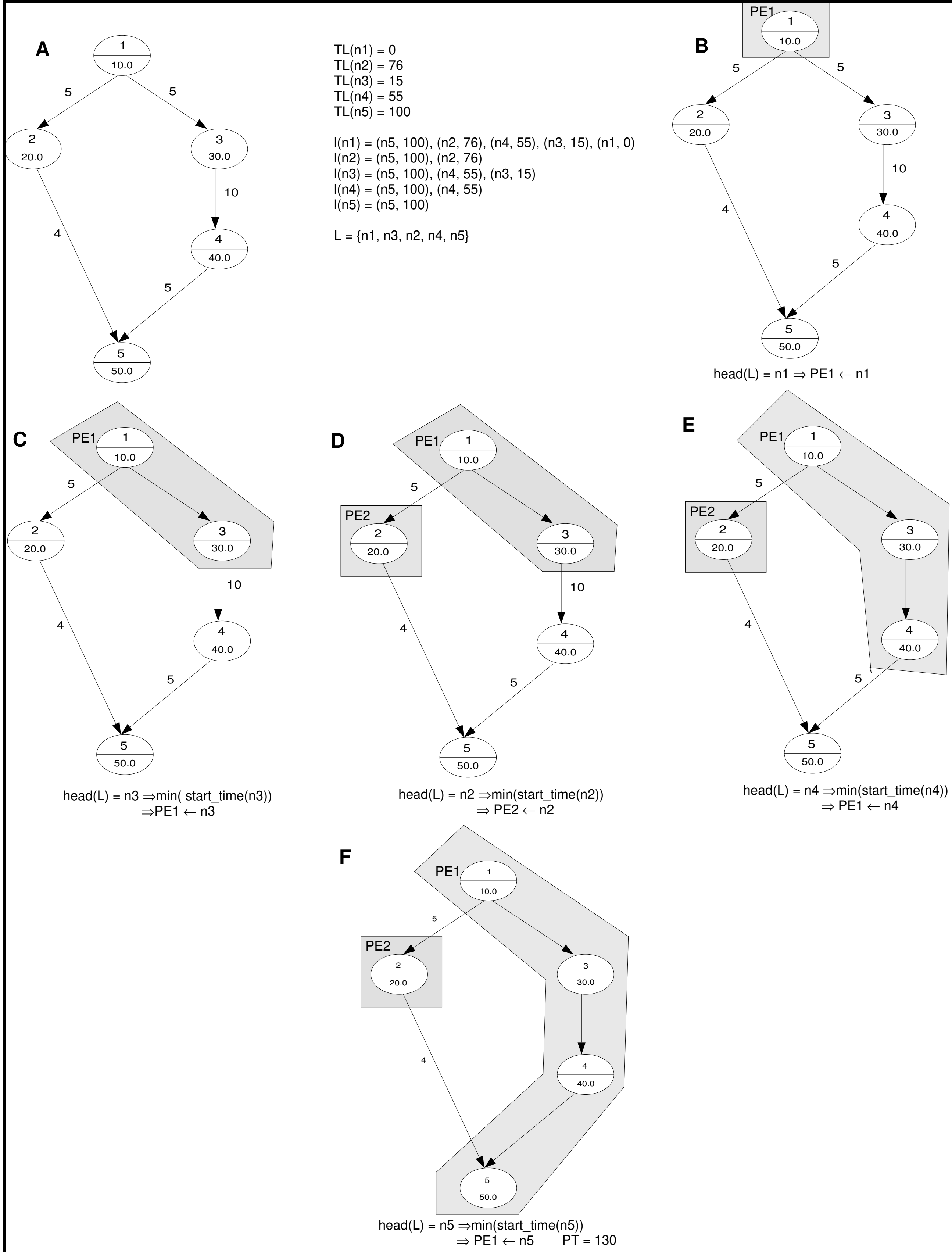

Figure 3
Example: MCP

possible start times for each task. The ALAP binding is found by traversing the DAG from the sink nodes to the source nodes and assigning the latest possible start time $T_L(n_i)$ to each node. The ALAP start times allow MCP to locate the critical path. For each node, the algorithm maintains a list $l(n_i)$ which consists of T_L's for n_i and all its descendents. The lists $l(n_i)$ are in turn ordered and the list L of nodes is created according to the order of lists $l(n_i)$. See Figure 3 for an example of its execution steps The computational complexity of MCP is $O(v^2 \log v)$.

3.2 List Scheduling Scheme

LAST (localized allocation of static tasks) allocates tasks to clusters of vertices based on the "strength" of their connectedness to vertices already allocated to clusters. Some definitions are required to understand the algorithm. A cluster, g_i, is a subset of vertices in the PDG that are assigned to processor *i*. A *frontier, f_i,* contains all vertices not assigned to any cluster and having at least one edge connecting them to cluster *i.* An edge (j,k) is *defined* if either adjacent vertex j or k is a member of any cluster. D_NODE_j gives the weighted average of adjacent nodes that are assigned to clusters. Figure 4 shows an example of the LAST clustering algorithm. The Baxter/Patel paper does not give

The Algorithm (LAST):

```
        D_NODE(source) = max int; set all other D-NODES to 0.
        While not all nodes have been examined, choose node v with highest
           D_NODE value.
          if D_NODE_v  > THRESHOLD then FORWARD ALLOCATE(v)
                                      else BACKWARD ALLOCATE(v).
            Compute parallel time
            For each node w in B-v BACKWARD ALLOCATE(w).
                                  FORWARD ALLOCATE(v)
                                  Choose allocation which gives least parallel time
FORWARD ALLOCATE(v)
        For each cluster adjacent to v, compute the time of v's completion if v is assigned to that cluster.
        Assign v to adjacent cluster which gives the smallest parallel time.
BACKWARD ALLOCATE(v)
        Create a backward cluster B containing v and  all nodes, not members of
         other clusters, whose outputs contain a directed path to v.
        FORWARD ALLOCATE(B).  Compute parallel time
```

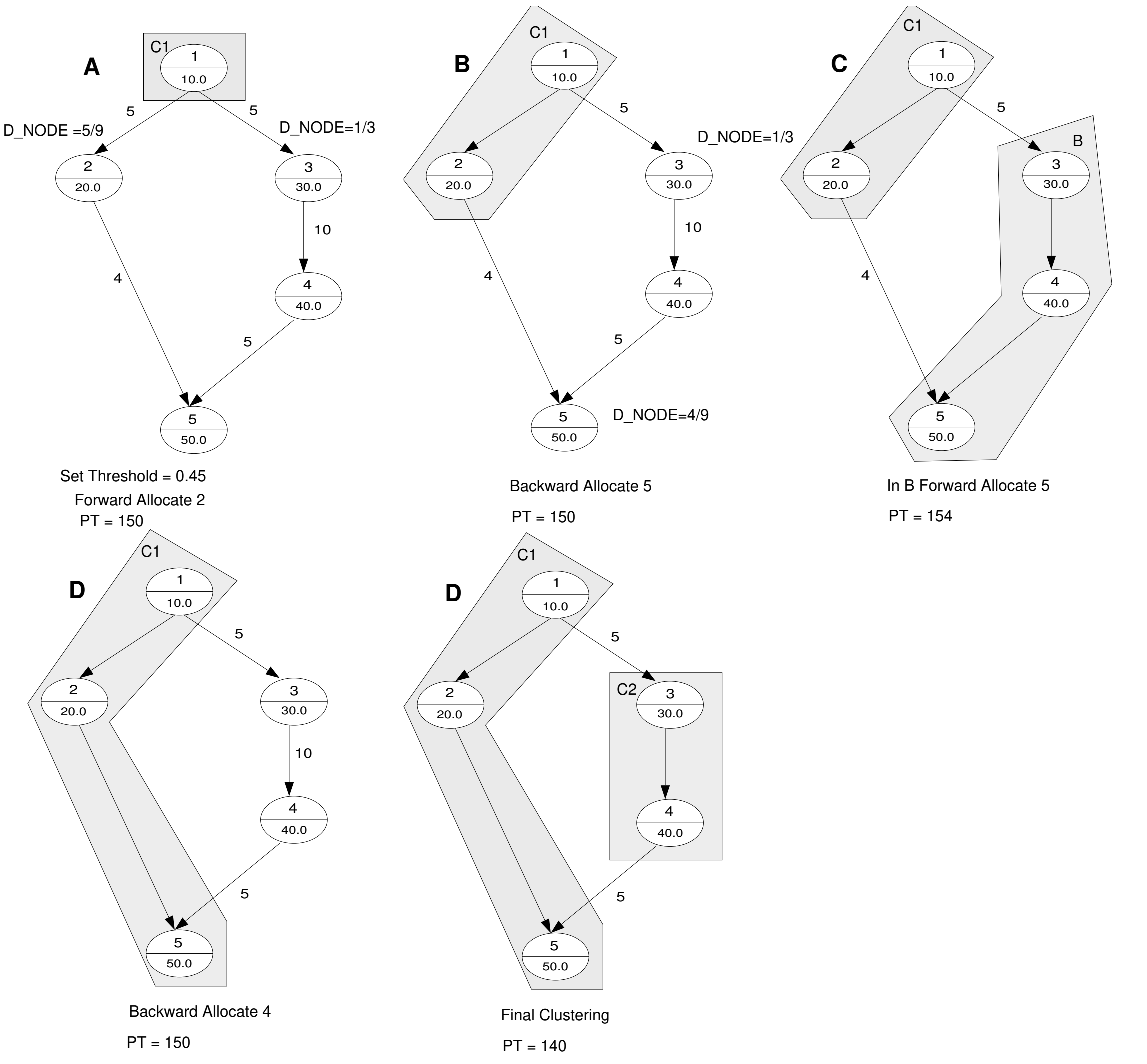

Figure 4
Example: LAST

the computational complexity of LAST.

3.3 Graph Decomposition

The partitioning scheme used in clan-based graph decomposition method (CLANS) analyzes the entire PDG before any clustering decisions are made. The graph is decomposed into a hierarchical *parse tree* of subgraphs known as clans. The top node of the parse tree represents the entire graph and its children represent clan subgraphs. For each clan in the tree, its children are in turn subclans of itself. Formally, a set of vertices C in graph G is a *clan* iff for all nodes x,y in C and z in G-C, (1) z is and ancestor of x iff z is an ancestor of y and (2) z is a descendant of x iff z is a descendant of y.

Clans have several properties that make them useful in the partitioning of a computation.

- A clan is a collection of nodes which have identical communication requirements with other nodes. Grouping by clans therefore yields a substantial reduction in communication overhead.
- Clans are classified as *linear, independent* or *primitive.* The nodes of the linear clans must execute sequentially, while the nodes of the independent clans may execute concurrently. Primitive clans are decomposed further into independent and linear clans.
- The clan structure derived from a PDG is unique and forms a hierarchy that can be regarded as a parse tree of the PDG. The parse tree contains linear and independent clans at alternating levels.
- The levels of the parse tree correspond to different partitioning sizes. When a cost model for a particular architecture is imposed on the parse tree, the optimal grain can be determined for that architecture.

The intention of this paper is not to explain the details of the algorithms, but to give some sense of the method and show the results of comparison testing on the five algorithms presented. For a detailed explanation of the algorithm see [15,16]. Only a high level description is given here.

> **The Algorithm**
> 1. Parse the PDG into a hierarchy of clans. The root is the entire graph and the leaves are the graph nodes. The original node execution costs are applied to the leaves.
> 2. Traverse the tree from the bottom up making local decisions at linear clans. The decision made at a linear clan is the best sequence of clustering and concurrency for its independent children.

Additional work is being done to improve the computational complexity of the parse. Currently. because of its dependence on matrix multiply, the algorithm is at least $O(v^3)$. Recursive

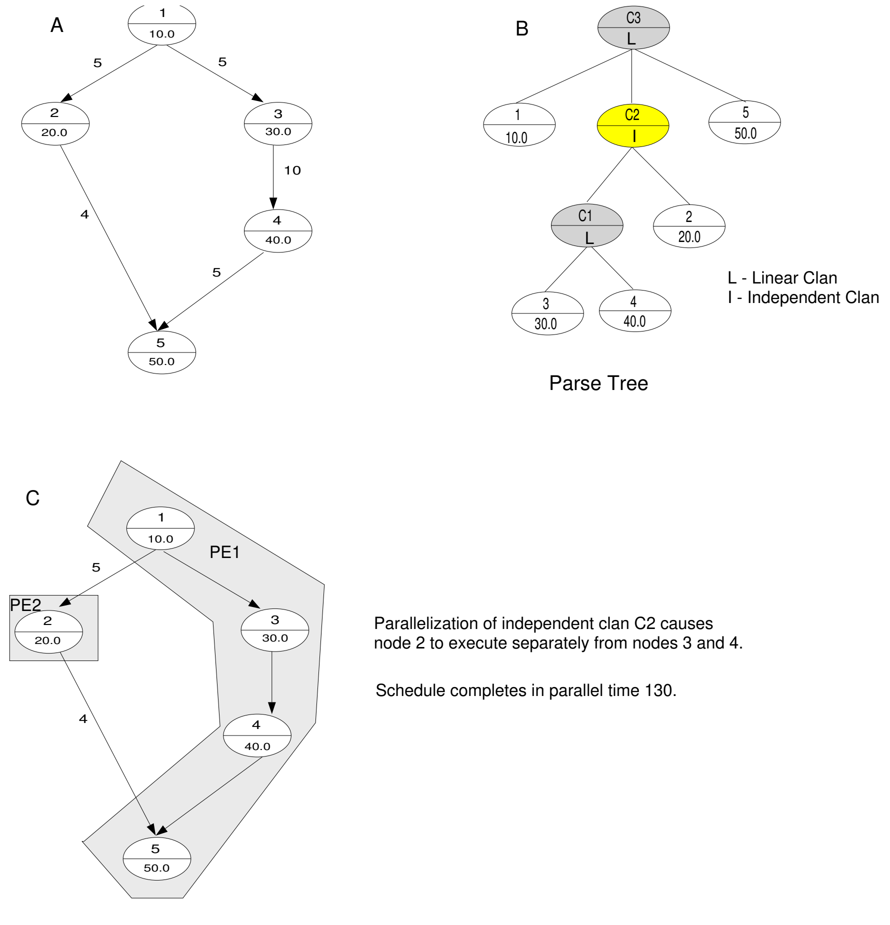

Figure 5
Example: CLANS

calls to decompose primitives cause additional overhead for some very irregular graphs. A new $O(v^2)$ parse is being developed and extensive testing will be done to calculate the expected computation cost.

Example: For the graph of Figure 5(A), the non-trivial clans to be placed in the tree include: linear clan C1{3, 4}, independent clan C2{2, {3, 4}}, and linear clan C3{1, {2, {3, 4}}, 5}. The parse tree created in step 1 of the algorithm is shown in Figure 5(B). In step 2 of the algorithm, costs are assigned in a bottom up fashion. The leaves have the costs of the corresponding nodes in the original graph.

C1 is the first linear clan to be considered. It has no independent children for which parallelization would be an option, so both of its children are placed in the same cluster for a cost of 30 + 40 = 70. At C3, a choice is presented for one of its children: cluster the nodes in C2 for a cost of 20 +

70 = 90, or parallelize it by executing C1 and node 2 concurrently. The cost of parallelizing C2 is the maximum of the costs of its children including communication requirements for those executing on separate processors. If node 2 is executed separately, its cost with communication is 5 + 20 + 4 = 29. C1, executing on the same processor as the nodes with which it communicates, costs 70. The cost of parallelization of clan C2 is then the greater of 29 and 70, or 70. Since the entire graph is linear, the execution sequence is node 1, clan C2 and node 2 concurrently, followed by node 5, for a cost of 10 + 70 + 50 = 130. The schedule is shown in Figure 5(C).

4. Input Graphs

To perform an adequate comparison test of the partitioning algorithms, it is desirable to select a wide variety of input graphs. The expectation is that some schemes will work better on one type of graph than other schemes. If no one scheme emerges as robust for all cases, a suite of partitioning techniques should be developed along with an identification as to which works best for a particular type of graph. With this goal in mind, six dependence graphs corresponding to real problems have been chosen for analysis. The graphs range from simple trees to regular, symmetric graphs, to irregular asymmetric graphs. Arbitrary weights have been assigned to the nodes and edges.

Diagrams of the PDGs are provided in Figures 6 - 12.

Sum1 represents the sum of products computation. Nodes 1 through 8 perform multiplication and the remaining nodes do additions. The costs applied are 10 for the multiplication, 1 for the addition and 18 for each communication.

Sum2 is an upside-down tree that results from a divide and conquer algorithm. Again all communication costs are 18. Sink nodes, (nodes 8-15) have a weight of 10 and nodes 1-7 each cost 1.

K1 is an adaptation of a graph described by the author of the LC algorithm. It is similar to SUM1, but has a single source node and has some heavy edges. The edges within the heart of the graph alternate between values of 1 and 10. Neighboring pairs of paths differ in length. Because all vertices have computation values equal to 1 unit, communication is the dominating factor when deciding how to schedule this graph.

FFT, representing the computation of the fast Fourier transform, is another symmetrical graph. An important characteristic of this graph is that there is a path from every input vertex to every output vertex. Five versions of this graph are used, each with different computation and communication values. FFT1 and FFT2 have equivalent communication values of 25 units per edge. However, the bulk of the computation is distributed through the middle section of the FFT1 program graph and is located in the first half of FFT2. Table I contains the vertex labellings for the FFT graphs.

Vertex #	FFT-1	FFT-2	FFT-3	FFT-4
1- 8	1	60	10	20
9 - 12	20	50	272	20
13 - 16	30	5	272	30
17 - 20	20	5	272	20
21 - 28	1	5	12	5

Table I - FFT labeling

FFT3 is labelled with costs based on computation and communication time estimated for the Intel iPSC/2 hypercube. The communication cost is estimated to be 500 microseconds, an approximation to the amount of time it takes to transfer a message of 100 bytes or less. FFT4 shows widely varying communication costs. The edges from the second stage {9-12} to the third stage {13-16} each have a weight of 600 while the rest of the edges are of weight 5. FFT5 again has computation and communication values based on hypercube estimates, but contains 16 input and output vertices and assumes twice as much computation per node.

NEQ [11] is based on Gaussian elimination algorithm for solving four equations in four variables. The graph was chosen for its slightly irregular layout. The edges range in value from 55 to 116 as shown in Figure 11 and the vertex values range from 1 to 361.

IRR is an adaptation of a PDG of a physics algorithm [11], and was chosen because of the highly irregular layout. Each of the edges has weight 50.

5. Results & Analysis

The comparative analysis will be divided over three graph sets: trees, FFTs and irregular graphs. For each set the comparison will be based on three criteria: relative parallel time, speedup and efficiency. For each criteria we will discuss the strengths and weaknesses of the three classes of algorithms.

Relative parallel time for algorithm k is the ratio of the greatest parallel time produced by any of the algorithms to the parallel time given by algorithm k:

$$\text{Relative Time}_{\text{algorithm K}} = PT_k/\max(PT_i) \text{ where i ranges over all algorithms.}$$

Speedup is defined as the ratio of serial time to parallel time and efficiency is the ratio of speedup to the number of processors used. Table II lists the parallel times with the number of processors used by each algorithm for each DAG.

Parallel Time & Number of Processors (TIME / # of Processors used)						
DAGs	Serial Time	CLANS	DSC	MCP	LC	LAST
FFT1	296 / 1	124 / 4	124 / 4	148 / 8	172 / 8	146 / 4
FFT2	760 / 1	200 / 8	205 / 8	205 / 8	225 / 8	240 / 8
FFT3	3440 / 1	1860 / 4	1860 / 4	2350 / 8	2838 / 8	2220 / 2
FFT4	480 /1	405 / 2	710 / 12	710 / 8	710 / 8	170 / 8
FFT5	4704 / 1	1884 / 8	1904 / 4	2578 / 16	3066 / 16	3589 / 7
SUM1	87 / 1	42 / 4	34 / 7	67 / 8	50 / 7	59 / 4
SUM2	87 / 1	42 / 4	34 / 7	35 / 7	50 / 5	55 / 3
K1	32 / 1	11 / 8	10 / 16	11 / 15	11 / 15	17 / 17
NEQ	3164 / 1	1652 / 5	1597 / 7	1597 / 5	2132 / 4	2082 / 4
IRR	1330 / 1	725 / 7	605 / 12	605 / 7	710 / 8	840 / 3

Table II

5.1 Trees (Sum1, Sum2 & K1)

The tree group is a simple class which is uniform, and algorithms with sub-goals that optimize arbitrary forks and joins should work well on this class. K1 is placed in the class since it is very similar to Sum1 except it has a single source node.

Critical Path: In the tree set, critical path heuristics perform best. DSC implements an optimum clustering algorithm for a fork and thus it gets the best schedule in both cases. DSC tries to schedule both the normal graph and the backward graph (i.e. all edges reversed), handling Sum1 and Sum2 exactly the same way. For K1, DSC again comes up with the best time, and the reason again is the optimal fork & join sub-goal. MCP performs well for Sum2 where the initial decision is not as critical as in Sum1. In Sum1 when the ALAP bindings are equal for edges {1-8}, they are all put on separate processors so as to get the minimum start time for each task. The incorrect initial decision to parallelize all source nodes causes the heuristic to have an excessive number of communications in later stages. For K1, MCP performs better than is does on SUM1g+r IPPS-94 because the ALAP binding is not equal for nodes {1-16}. Alternating edge weights cause the variation in ALAP binding which results in the correct initial decisions. LC does well in K1 because it immediately reduces all paths with edge weights of 10 and then goes into the merge phase. For Sum1 and Sum2, LC is not very effective as all edges are of equal weight and there is no real advantage to zeroing the critical path when there are multiple critical paths.

List Scheduling: LAST is not very effective in this set of graphs since the D_NODE parameter does not properly prioritize the nodes. All nodes in these DAGs except the source nodes are of equal D_NODE value. This causes the heuristic to lose its measuring ability and in turn the ability to make good decisions.

Graph Decomposition: The efficiency of CLANS on these graphs is best of the algorithms, while the speedup is second best. For Sum1 and Sum2, CLANS parses the graphs causing all children at a join to be a clan or all parents at a fork to be a clan. For K1, all parents at each fork are a clan similar to Sum1. The parse trees formed from PDGs which are themselves trees are somewhat unbalanced. Nodes in the parse tree usually have children which in turn have widely varying numbers of children. A uniform and cost effective grain size which should be developed as the algorithm moves upward in the parse tree is more difficult to obtain in this type of parse tree.

5.2 Regular Symmetric Graphs (FFT1, FFT2, FFT3, FFT4 & FFT5)

All FFTs are symmetrical and regular, but have multiple critical paths. In this group, CLANS outperforms all the other algorithms with the exception of LAST on FFT4.

Critical Path: DSC is the best critical path algorithm for this set. DSC uses the CT2 (Constraint 2) as defined in the description of the algorithm to avoid making the same mistakes as MCP and LC. MCP has the same problem for this set of graphs as it has for Sum2, i.e. it assigns all the source nodes to independent processors to give each initial node the minimal

parallel time. For FFT2 where the source nodes are of higher weights, MCP gives equivalent performance to DSC. In all other cases, where the initial nodes do not have weights sufficiently large to parallelize, it performs much worse than DSC. LC again gives greater parallel times because it zeros all edges along one critical path while in all FFTs each path from source to sink is a critical path. All critical path algorithms have trouble with FFT4 where the parallel time for each algorithm's schedule is greater than the serial time. As a result the speedup is less than one! This result indicates situations where designers of critical path algorithms must be especially careful. The failure in all cases is due to the short sighted heuristic approach of the critical path algorithms. MCP and DSC zero edges along the critical path. MCP focuses on the head of the list L ordered according to ALAP binding while DSC focuses on the free and partial free nodes. At the second stage of FFT4, the decision to separately schedule nodes {9-12} has already been made. This causes at least four of the edges of weight 600 between the second and the third stage not to be zeroed. The only way to fix this problem in MCP and DSC is to backtrack, a technique not included by Gerasoulis and Yang because of the added computational cost.

List Scheduling: It is interesting to note that for FFTs, LAST's performance ranged from best of the algorithms (FFT4) to the worst (FFT5). LAST again performs according to the sensitivity of the D_NODE parameter. The THRESHOLD value of the D_NODE was varied to adjust the algorithm to give the best possible parallel time in each case. The best performance was registered for FFT4 where the edges with weight 600 were easily isolated by the D_NODE values of nodes {13-16}. In FFT5 the uniformity of the bulk of the interior nodes of the DAG caused the priority list to be of little use in giving any significant information on good clustering options thus resulting in poor parallel times. The variation in edge weights for FFT3 and the variation in node weights in FFT1 caused relatively good priority listings and accordingly good schedules.

Graph Decomposition: The performance of the CLANS algorithm on these graphs is superior to the other classes of algorithms on all measurements. The FFT graphs have the largest amount of available parallelism of those included. This high degree of possible concurrency translates to a parse tree with a well-balanced and deep structure. Scheduling decisions, made in going from leaves to higher tree levels, result in a uniform grain size that is appropriate to the cost of communication. The algorithm in effect discards potential for concurrency until the grain size is large enough for parallelization to be cost effective.

5.3 Irregular Asymmetric Graphs (NEQ & IRR)

This set of DAGs is hard to analyze and the results from this set may not indicate the actual performance capability of any of the algorithms over a large working set of truly random irregular asymmetric graphs. For all algorithms IRR and NEQ produced the lowest speedup of those graphs where communication costs did not dominate execution costs.

Critical Path: MCP and DSC are the two algorithms which tie for the best parallel time for NEQ, and DSC gives the best parallel time for IRR. LC fails to give a good schedule. It is significantly worse in the case of NEQ, due to the heavy weights of the nodes. When LC clusters the critical path it does not take into account that the majority of the weight was due to the node weights and not the edge weights. The relatively high node weights then cause a poor merging phase because fewer gaps exist between nodes. LC performs much better in IRR where the node to edge weight ratio is much lower.

List Scheduling: LAST does not do well for either one of these graphs. The priority list is diverse in each case and conveys more information than with regular graphs. However, the optimum THRESHOLD values for these two graphs are very hard to determine. We tried to vary the THRESHOLD between 0 and 1 by increments of .05 and obtained the listed parallel times. The excessive amount of BACKWARD ALLOCATE iteration due to an inappropriate THRESHOLD value may be the cause of this poor result. THRESHOLD is a very difficult parameter to establish if the graph is complicated, and LAST does not give any suggestion on possible values.

Graph Decomposition: The NEQ graph produces an extremely unbalanced parse tree. Every independent clan contains at least one singleton, making the formation of uniform larger grains impossible. A similar situation occurred with the tree graphs. The parallel time of CLANS on NEQ is about 3% greater than that of DSC and MCP and considerable less than (21%-23%) than that of LC and LAST. IRR contains many dependences that require code segments to be executed sequentially, and the corresponding parse tree illustrates this in that it affords little opportunity to create uniform parallel grains.

6. Conclusion

It is important for designers of parallelizing tools to have an idea of the capabilities of various scheduling algorithms with respect to variations in DAG properties. In this paper we have tried to illustrate three types of algorithms and their relative capabilities.

The critical path heuristics perform relatively well in cases where the edge weights are nearly uniform between intermediate stages of the graphs. In the case where the edge weights may be highly diverse as seen in FFT4 these heuristics tend to be extremely ineffective. Critical path heuristics make local decisions based on only one path in the graph because they are non-backtracking. They may give a schedule with a parallel time greater than serial time. Because they assume a completely parallelized initial graph with each node on a separate processor, a decision made early in the run may lead to an unwanted result later.

List schedulers are highly dependent on the priority parameter they use. In the case of LAST, D_NODE, a ratio of defined edge weights to undefined edge weights, was this parameter. LAST was generally less effective than the other algorithms except in the case of FFT4 where its priority parameter D_NODE identified the best stage of the FFT to aggregate so as to reduce the heavy edges. Priority based schemes like LAST are difficult to generalize and need to be considered on the merit of the priority parameter. LAST, in particular, was ineffective for most of the DAGs used for the comparison.

Among the three approaches considered, the graph decomposition method seems to produce the most uniformly good results. For every graph, its efficiency was best or second best. Its parallel time was best or second best in all cases but two, where it was third best. In no case did its parallel time exceed serial time. Rather than modifying the structure of a DAG or scheduling tasks from a priority list as used in the other two approaches, it derives the parallelism by analyzing the structure of the entire graph. The parse tree representation of the graph illuminates all opportunities of parallelism and all requirements for sequential ordering. Parse tree nodes represent clans, groups of DAG nodes with identical communication requirements to DAG nodes outside the clan. The clan is a logical cluster for reducing communication.

The goal of comparing scheduling algorithms is the identification of algorithms that work well on all graphs. If good algorithms for general graphs are not available, good algorithms could be developed for classes of graphs.

Through this work, particular strengths and weaknesses of the algorithms have appeared, and their discovery may be used to initiate improvements. There are many heuristics available, and this is the beginning of a study to analyze and classify them. We encourage readers who have developed other algorithms to add their results to ours. Send us your code and we willbe happy to run it against our data sets. Our work is continuing by comparing graphs generated randomly. We're extending the comparison to new input graph domains. We're looking at classifying graphs according to their relative execution/communication cost ratios, varience in node and edge costs, and density. Future directions for this work are a continued comparison based on randomly generated graphs, and the analysis of the running time of the scheduling heuristics.

Acknowledgments

The implementation of the LAST algorithm was provided by Jeff Baxter of the University of Illinois at Urbana-Champaign. Mandar Joshi of the University of Texas provided the implementation for some of the LC algorithm. The DSC algorithm implementation was provided by Apostolos Gerasoulis and Tao Yang of Rutgers University.

Bibliography

1. Anger, F.D.. Hwang, J.J., and Chow,Y.C. Scheduling with Sufficient Loosely Coupled Processors. *Journal of Parallel and Distributed Comput*., 9, 1990, pp. 87-92.
2. Baxter, J. and Patel, J.H., The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm. *Proceedings of the 1989 Int. Conference on Parallel Processing*, Vol. 2, 1989, pp. 217-222.
3. Coffman E.G., Jr., (Ed.) *Computer and Job-Shop Scheduling Thoery*. John Wiley & Sons, New York, 1976.
4. Gerasoulis, A. and Yang, T. A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors, *Journal or Parallel and Distributed Computing,* Dec. 1992, 16, pp. 276-2
5. Gerasoulis,A., Venugopal, S., and Yang,T. Clustering Task Graphs for Message Passing Architectures. *Proc. of ACM Int'l conf. on Supercomputing,* 1990, pp. 447-456.
6. Graham, R.L. Bounds on Multiprocessing timing Anomalies. *SIAM J. Appl. Math.*,

Vol. 17, No. 2, Mar. 1969, pp. 416-429.

7. Graham, R.L., Lawer, E.L., Lenstra, J.K. , and Rinnooy Kan, A.H.G. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Ann. Discrete Math.*, 5, 1979, pp. 287-326.
8. Horowitz E.,and Sahni, S. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, MD, 1984.
9. Hu, T.C. Parallel Sequencing and Assembly Line Problems. *Oper. Res.*, Vol. 9, No. 6, 1961, pp. 841-848.
10. Hwang, J.J., Chow, Y.C. , Anger, F.D., and Lee, B.Y. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM J. Comput.*, Vol. 18, No. 2, Apr. 1989, pp. 244-257.
11. Kim, S.J. and Browne, J.C. A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures. *Proc. of Int'l Conf. on Parallel Processing*, Vol. III, 1988, pp. 1-8.
12. Kruatrachue, B. and Lewis, T. Grain Size Determination for Parallel Processing. *IEEE Softw.*, Jan. 1988, pp. 23-32.
13. Lee,C.Y. , Hwang, J.J., Chow, Y.C. , and Anger, F.D. Multiprocessor Scheduling with Interporcessor Communication Delays. *Oper. Res. Lett.*, 7, 3, 1988, pp. 141-147.
14. Liu, Z. A Note on Graham's Bound. *Inform. Processing Lett.,* 36, Oct. 1990, pp. 1-5.
15. McCreary, C. and Gill, H. Automatic Determination of Grain Size for Efficient Parallel Processing. *Comm. of ACM*, Vol. 32, No. 9, Sept. 1989, pp. 1073-1078.
16. McCreary, C., Thompson, J., Gill, H., Smith, T., Zhu, Y., Partitioning and Scheduling Using Graph Decomposition. Auburn University Tech. Rpt., Dept. of Computer Science and Engineering, CSE-93-06.
17. Papadimitrou,C.H., and Ullman, J.D. A Communication-Time Tradeoff. *SIAM J. Comput.*, Vol. 16, No. 4, Aug. 19876, pp. 639-646.
18. Papadimitrou, C.H. and Yannakakis,M. Towards an Architecture-Independent Analysis of Parallel Algorithms. *SIAM J. Comput.*, Vol. 19, No. 2, April 1990, pp. 322-328.
19. Reingold, E.M., Nievergelt, J. , and Deo, N. *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.
20. Rewini, H.E. and Lewis,T.G. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. *Journal of Parallel and Distributed Computing*, 9, 1990, pp. 138-153.
21. Sarkar, V., *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. The MIT Press, Cambridge, Massachusetts, 1989.
22. Terrano,A.E., Dunn, S.M., and Peters, H.E. Using an Architectural Knowledge Base to Generate Code for Parallel Computer. *Comm. of ACM,* Vol. 32, No. 9, Sept. 1989, pp. 1065-1072.
23. Wu, M.Y., and Gajski, D.D. Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Trans. on Parallel and Distributed Systems,* Vol. 1, No. 3, July 1990, pp. 330-343.
24. Yu, W.H. LU Decomposition on a Multiprocessing System with Communication Delay. Ph.D. Theses, Dept. of EE and CS, University of California, Berkeley, 1984.