A Comparison of Multiprocessor Scheduling Heuristics *

A. A. Khan, C. L. McCreary, M. S. Jones
Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849
aakhan@eng.auburn.edu, mccreary@eng.auburn.edu, msjones@eng.auburn.edu

Contact person: Dr. Carolyn McCreary, Tel: (205)844-6308, Fax: (205)844-6329

Abstract

Many algorithms for scheduling 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 tractible algorithm exists, and no baseline is available from which to compare the resulting schedules. Furthermore, performance guarantees have been found for only a few simple DAGs. This paper is an attempt to quantify the differences in five of the heuristics. Classification criteria are defined for the DAGs, and the differences between the heuristics are noted for various criteria. The comparison is made between a graph based method, two critical path methods, and two list scheduling heuristics. The empirical performance of the five heuristics is compared when they are applied to the randomly generated DAGs.

*This work is supported by NSF grant number CCR-9203319

1 Introduction

One of the primary problems in executing programs efficiently on multiprocessor systems with distributed memory is to partition the program into tasks that can be assigned to different processors for parallel execution. If the partitioning results in a high degree of parallelism, a greater amount of communication will be required among 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 appropriate size and number of tasks to balance communication overhead and parallelism so that the completion time is minimized. A sequential program is commonly represented as a Program Dependence Graph (PDG) which is a directed acyclic graph (DAG) with node and edge weights [1]. Each vertex in a PDG denotes a task and a weight, which is 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. The partitioning and assignment known as the scheduling problem. The problem is also called *grain size determination* [2], the *clustering problem* [3, 4], and *internalization pre-pass* [1].

The partitioning/scheduling problem is intractable, and heuristics are required to find sub-optimal solutions. As a result, there are no 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 experimental results are known, the goal of this paper is to determine the relative performance of some promising techniques.

In classical scheduling, communication costs are not considered [5, 6]. 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 [7]. Many researchers have studied the problem and proposed solutions. Based on the techniques employed, the earlier methods can be classified into the following three categories:

- *Critical path heuristics* [1, 3, 4, 8, 9, 10]: 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 [11] 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 [8]. In this paper we include experimental results from two critical path algorithms, DSC and MCP.

- *List scheduling heuristics* [2, 7, 12, 13, 14, 15, 16]: These algorithms assign priorities to the tasks 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 [2, 12, 16] to reduce the communication costs. Two list scheduling methods, the Mapping Heuristic (MH), and multiprocessor Hu by Lewis and El-Rewini are included in this comparison study.

- *Graph decomposition method* [17, 18]: Based on graph decomposition theory, the method parses a graph into a hierarchy (tree) of subgraphs representing the independent/precedence relationship among groups of tasks. 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 a random set of 2100 PDGs, identifies the resulting schedules, and compares the parallel time and efficiency. The PDGs are classified according to grain size, out degree and node weight range. Section 2 presents the underlying model; Section 3 gives the graph classification; Section 4 gives the numerical results; Section 5 provides a summary and conclusion, and the appendix explains the scheduling heuristics.

2 Problem Definition and Assumptions

The problem of parallelizing the PDG to achieve minimal parallel time is NP-complete [1]. This process usually involves two distinct steps: partitioning and scheduling. Partitioning combines tasks from the PDG into 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. Processors whose interprocessor communication costs are high relative to processing costs require larger grains than processors with low inter-processor communication costs. Scheduling assigns it 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 in the PDG no matter which two processors are involved.

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 tasks in separate grains is not allowed. Several heuristics [2, 12, 16] have been developed that take advantage of this option, but duplication adds additional complexity to an already intractable problem that none of our competing methods use.

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.

3 Graph Classification

The objective of the comparison is to identify the best scheduling heuristic under various conditions that the scheduling algorithm may encounter. So far results on classifications and characterizations of application programs are few. Therefore, a subgoal of the research is to define classes of DAGs that may be useful in distinguishing the performance of the heuristics. The first metric for classification is called *granularity*. Granularity is a measure of the ratio of execution time to communication time. When this ratio is high, communication delay is short (relative to execution time) and a high degree of parallelization can be effecively realized. On the other hand, when it is low, the overhead of communication costs outweighs the reduction in parallel time obtained by parallelization. Another graph property that might distinguish the heuristics is the branching factor or degree of the nodes. With random generation of the graphs, we were unable to control the degree to the full extent of determining an exact distribution of the nodes of each degree, so we approximate this value with a metric called *anchor out-degree*. We conjectured that some heuristics could more effectively accomodate larger variations in node weights than others. To test this conjecture, we included tests on various *node weight ranges*. This sections more precisely defines these three graph classifications and gives the values used in our testing.

3.1 Granularity

For parallel programs, granularity as defined by Sarkar in [1] is: *the average size of a sequential unit of computation in the program*. This does not consider the weight of the communication between the tasks. To take into account of the edge weights, a new definition of granularity of a parallel program based on the PDG is proposed here.

We define *granularity* of a weighted DAG as the average ratio of node weight to maximum adjacent outgoing edge weight. The formula for granularity is:

$$Granularity = \frac{1}{N-S}\left(\sum_{i=0}^{N-S} \frac{w_i}{max[w_{e_{ij}}]}\right)$$

where:

- N is the total number of nodes in the graph.

- S is the total number of sink nodes in the graph.[1]
- w_i is the weight of the i^{th} node that is not a sink.
- $max[w_{e_{ij}}]$ is the maximally weighted outgoing edge from node i to some node j.

The granularity of a DAG determines the amount of useful parallelism available. In general, a granularity of 1 indicates an even trade-off between communication and execution costs. For granularities less than 1, more efficient schedules are produced when PDG nodes are combined more frequently and when the granularity is greater than 1, more parallelism can be extracted from the PDG. Gerasoulis et. al. [9] define a similar notion of granularity and call graphs with $granularity > 1$ *coarse* grained. It is proven that for any coarse grained graph any list scheduling heuristic gives a schedule within a factor of 2 of that of the optimal schedule [6].

For the comparison of CLANS, DSC, MCP, MH and HU, the following five granularity levels were chosen since they correspond to the most interesting results in many of our testings of the algorithms.

- Granularity < 0.08
- $0.08 <$ Granularity < 0.2
- $0.2 <$ Granularity < 0.8
- $0.8 <$ Granularity < 2.0
- $2.0 <$ Granularity

3.2 Anchor Out-Degree

The *out-degree* of a node is equal to the number of out-going edges from the node. The *anchor out-degree* of a graph is defined as the mode of the out-degrees of the nodes. We will use the word anchor and anchor out-degree interchangeably. Anchor out-degree is a good measure of program branching factor. The effect of branching on the schedules generated by the heuristics was of particular interest to this investigation. The comparison of the three algorithms in our work involved graphs with anchor out-degrees from 2 to 5.

3.3 Node Weight Range

The *node weight range* of a graph is the interval $[w_{min}, w_{max}]$, where w_{max} is the maximum node weight and w_{min} is the minmum node weight. The node weights of the graph are distributed in the range specified by the w_{min} and w_{max}. For the comparison of the three given algorithms, three node weight range sets were chosen: $[20-100]$, $[20-200]$ and $[20-400]$.

[1] Note that sink nodes do not contribute to communication delay. Therefore, they are left out of the average.

Table 1: Classification and distribution of graphs.

	ANCHOR 2		ANCHOR 3		ANCHOR 4		ANCHOR 5	
Granularity	Node Weight Range	# of Graphs	Node Weight Range	# of Graphs	Node Weight Range	# of Graphs	Node Weight Range	# of Graphs
G<0.08	10-100	35	10-100	35	10-100	35	10-100	35
	10-200	35	10-200	35	10-200	35	10-200	35
	10-300	35	10-300	35	10-300	35	10-300	35
0.08<G<0.2	10-100	35	10-100	35	10-100	35	10-100	35
	10-200	35	10-200	35	10-200	35	10-200	35
	10-300	35	10-300	35	10-300	35	10-300	35
0.2<G<0,8	10-100	35	10-100	35	10-100	35	10-100	35
	10-200	35	10-200	35	10-200	35	10-200	35
	10-300	35	10-300	35	10-300	35	10-300	35
0.8<G<2.0	10-100	35	10-100	35	10-100	35	10-100	35
	10-200	35	10-200	35	10-200	35	10-200	35
	10-300	35	10-300	35	10-300	35	10-300	35
2.0<G	10-100	35	10-100	35	10-100	35	10-100	35
	10-200	35	10-200	35	10-200	35	10-200	35
	10-300	35	10-300	35	10-300	35	10-300	35

4 Performance Evaluation

This section gives the numerical data from running CLANS, DSC, MCP, MH and HU on test graphs generated randomly. The 2100 graphs tested are divided into 60 separate sets. The division into the sets was based on the three classification criteria: *granularity*, *node weight range*, and *anchor out-degree*. Table 1 illustrates how the sets are divided over the three classes. The analysis is also divided over the three classification criteria. In each case we have a preliminary set of comparisons which gives:

- The number of graphs for each heuristic schedule whose running time exceeds the serial time (i.e. speedup < 1).
- Average *speedup* for each heuristic over a graph subset.
- The average *normalized relative parallel time* for each heuristic over a graph subset.
- Average *efficiency* for each heuristic over a graph subset.

The definitions of these measures are given below.

$$Speedup = \frac{SerialTime}{ParallelTime}$$

and

$$Efficiency = \frac{Speedup}{NumberofProcessors}$$

ParallelTime(X) is the parallel time of the schedule produced by heuristic X (where X is CLANS, MCP, DSC, MH or HU), while *BestParallelTime* is the shortest of the five heuristic schedule execution times. The relative parallel time compares the performance of heuristic *X's* schedule with the best schedule among the five algorithms. The best algorithm will have a relative parallel time of 0.00. The terms *normalized relative parallel time* and *relative parallel time* will be used interchangeably in this paper

$$NormalizedRelativeParallelTime(X) = (\frac{ParallelTime(X)}{BestParallelTime}) - 1$$

4.1 Granularity Analysis

4.1.1 Speedup < 1

The first result shows the number of input graphs for which each one of the heuristics gives *speedup smaller than 1.* This information indicates the range of granularities where the heuristic is unsuitable for scheduling. Table 2 shows the performance of each of the five heuristics. At low granularities the critical path and

Table 2: Number of graphs for which each heuristic gives a speedup less than 1 for the given granularity (each range has 420 graphs).

	CLANS	DSC	MCP	MH	HU
G < .08	0.00	234.00	262.00	212.00	420.00
.08 < G < .2	0.00	114.00	148.00	109.00	411.00
.2 < G < .8	0.00	2.00	1.00	2.00	230.00
.8 < G < 2	0.00	0.00	0.00	0.00	59.00
2 < G	0.00	0.00	0.00	0.00	0.00

list scheduling heuristics are extremely ineffective and produce schedules that retard execution time. In particular, within the $G < 0.08$ range, the critical path schedulers and the list schedulers give a retardation instead of a speedup for over 50% of the graphs. On the other hand CLANS, can never produce a speedup less than 1. The algorithm has a speedup check at every linear node of the parse tree. If the parallelization decision at a linear node causes a negative speedup, the children of that node are scheduled on the same processor. This gives CLANS a macro level control over the schedule in the later stages of the algorithm, as the parse tree is traversed from the leaves to the root, which is not reflected in the other four algorithms. The other four heuristics make very localized decisions at all stages of the scheduling process.

4.1.2 Normalized Relative Parallel Time

The *normalized relative parallel time* for each set of graphs and heuristics is presented in Table 3. If one heuristic produced the best schedule for all input graphs, the normalized relative parallel time for that heuristic would be 0. The Table 3 shows that DSC, MCP, DSC, and HU perform poorly in comparison to CLANS when G is low, which corresponds to the case when communication delay is greater than task time.

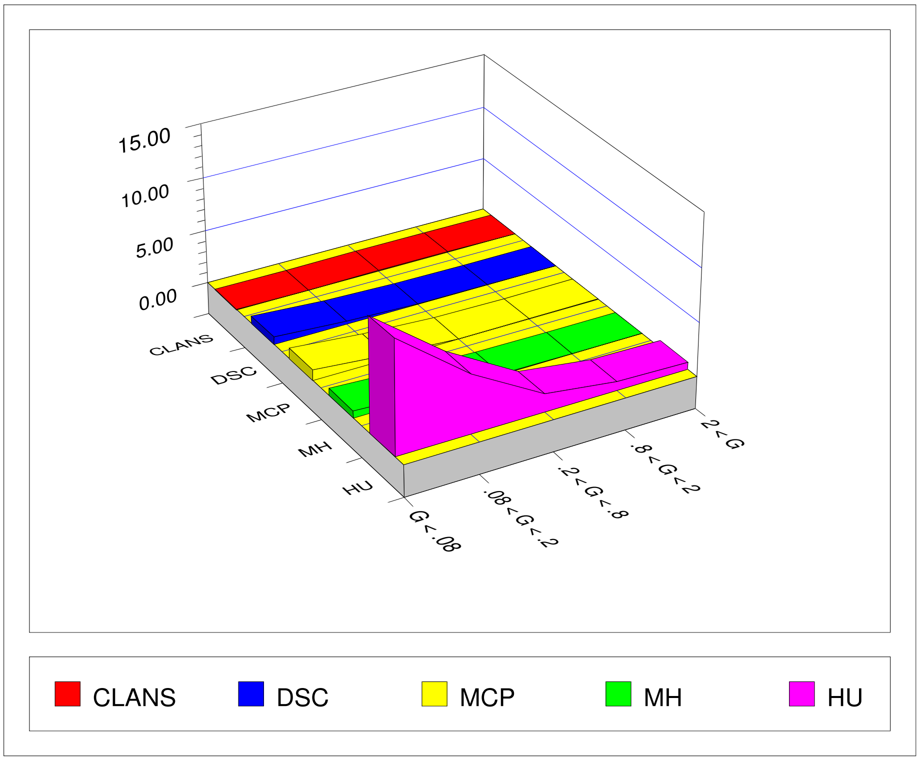


Figure 1: Average relative parallel time comparison with the increase in granularity.

Table 3: Average relative parallel time for the given granularity ranges.

	CLANS	DSC	MCP	MH	HU
G < .08	0.01	0.69	1.01	0.66	10.63
.08 < G < .2	0.02	0.30	0.41	0.33	5.50
.2 < G < .8	0.06	0.06	0.02	0.09	1.70
.8 < G < 2	0.06	0.02	0.00	0.04	0.84
2 < G	0.02	0.02	0.04	0.03	0.67

The last row in Table 3 suggests that for coarse grain graphs, while HU is behind the others, CLANS, MCP, DSC and MH all converge to give effective schedules. This is not surprising since performance bound for these graphs are known.

Figure 1 shows that CLANS gives the most consistent performance with respect to variation in granularity. It is consistently within 6.5% of the best performing heuristic. At high granularities the performance of critical path and list scheduleing heuristics improves considerably while CLANS lags a little behind. In the low granularity range MCP, DSC, MH and HU give very high relative parallel time, due to the tendency of all of them to give speedups of less than 1.

4.1.3 Average Speedup

Greater speedup was recorded by all heuristics at high granularity (see Table 4). It is expected that an increase in granularity corresponds to an increase in parallelism within the DAGs, due to the reduction of communication cost relative to parallelism. The combination of heavy nodes and light weight edges increased the possibilities available to the heuristics for parallelization. The *average speedup* comparison as illustrated in Figure 2 and Table 4 indicate that lower granularities result in lower usable parallelism, as expected.

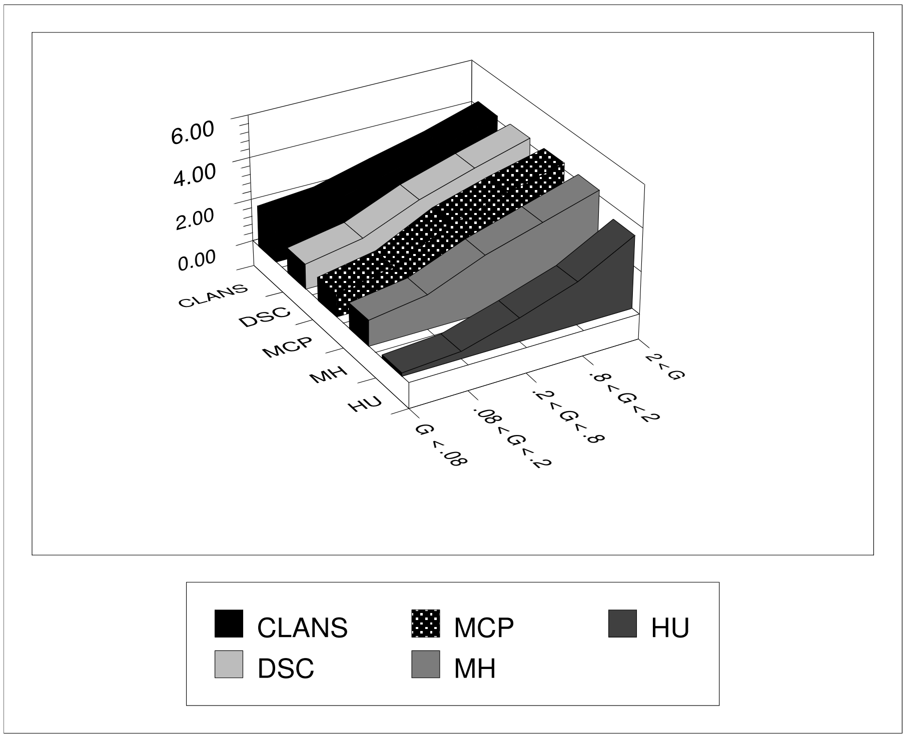


Figure 2: Trend illustrating the increase in speedup with the increase in granularity.

Table 4: Average speedup for the given granularity ranges.

	CLANS	DSC	MCP	MH	HU
G < .08	1.95	1.19	1.08	1.21	0.18
.08 < G < .2	2.16	1.67	1.60	1.62	0.34
.2 < G < .8	2.80	2.77	2.91	2.69	1.12
.8 < G < 2	3.45	3.57	3.65	3.49	2.01
2 < G	4.23	4.31	4.32	4.28	3.41

At low granularity CLANS had nearly 2 times the speedup of DSC, MCP, MH and HU although CLANS only performed better than HU at high granularity. Although these results are interesting, they still do not give sufficient detail about the relative advantages and disadvantages of using a particular heuristic.

Note that in Figure 2, the average speedup curves for all five algorithms varied almost linearly with granularity G. This indicates that granularity defined in this paper is a useful metric for predicting the speedup performance.

4.1.4 Efficiency

Efficiency of the algorithms reveals the average percent of time the processors are active. CLANS shows greater efficiency with lower granularity because it consistently uses fewer processors than the rest of the algorithms.

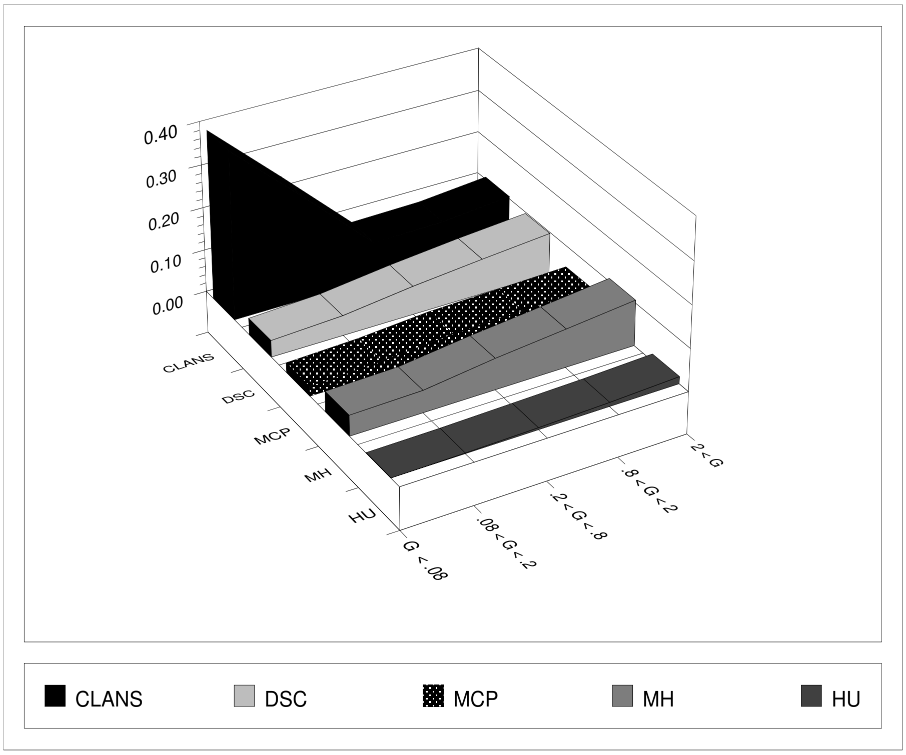


Figure 3: Average efficiency comparison with the increase in granularity.

Table 5: Average efficiency for the given granularity ranges.

	CLANS	DSC	MCP	MH	HU
G < .08	0.39	0.04	0.02	0.05	0.00
.08 < G < .2	0.24	0.05	0.03	0.05	0.00
.2 < G < .8	0.09	0.07	0.04	0.07	0.01
.8 < G < 2	0.09	0.08	0.05	0.09	0.01
2 < G	0.10	0.10	0.05	0.11	0.02

Figure 3 shows that the efficiency of critical path and list scheduling algorithms is slowly increasing as the granularity increases. At low granularity, CLANS becomes much more efficient. For many of graphs of these sets, CLANS chooses to place all the processes on a single processor, so that the efficiency of serial processing is 100%. As the granularity passes into the the third stage ($0.2 < G < 0.8$) the efficiency of

CLANS comes close to that of DSC and MH.

4.2 Node Weight Range Analysis

4.2.1 Speedup < 1

The results obtained from graph sets defined by node weight range were not as clear as those obtained from granularity. In the preliminary study, the cases where the five heuristics produce schedules with speedups less than 1 were studied most closely. CLANS does not produce any schedule that retards the speed to less than serial time. Table 6 illustrates a sudden jump in the number of speedups that are less than 1 when the

Table 6: Number of schedules with speedups less than 1 in the given Node Weight Range (each set has 700 graphs).

	CLANS	DSC	MCP	MH	HU
20 - 100	0.00	55.00	80.00	41.00	331.00
20 - 200	0.00	147.00	167.00	140.00	393.00
20 - 400	0.00	148.00	164.00	142.00	395.00

node weight range is increased to 20 - 200 for DSC, MCP, MH, and HU. These initial results encourage us to look at the parameter more closely.

4.2.2 Relative Parallel Time

A further experiment using the relative parallel time criteria gave some additional insight into the performance of critical path heuristics. Figure 4 and Table 7 indicate how the relative parallel times of MCP,

Table 7: Average relative parallel time for each heuristic in the given Node Weight Range.

	CLANS	DSC	MCP	MH	HU
20 - 100	0.04	0.10	0.13	0.09	2.75
20 - 200	0.03	0.27	0.38	0.29	4.50
20 - 400	0.03	0.29	0.38	0.31	4.36

DSC, MH, and HU increase with the increase in node weight range. The role of node weight range clearly needs further investigation.

4.2.3 Average Speedup

In the speedup analysis, another interesting result showed that the increase in node weight range resulted in decrease of speedup. The average speedup of CLANS was slightly greater than that of MCP, DSC, MH and HU.

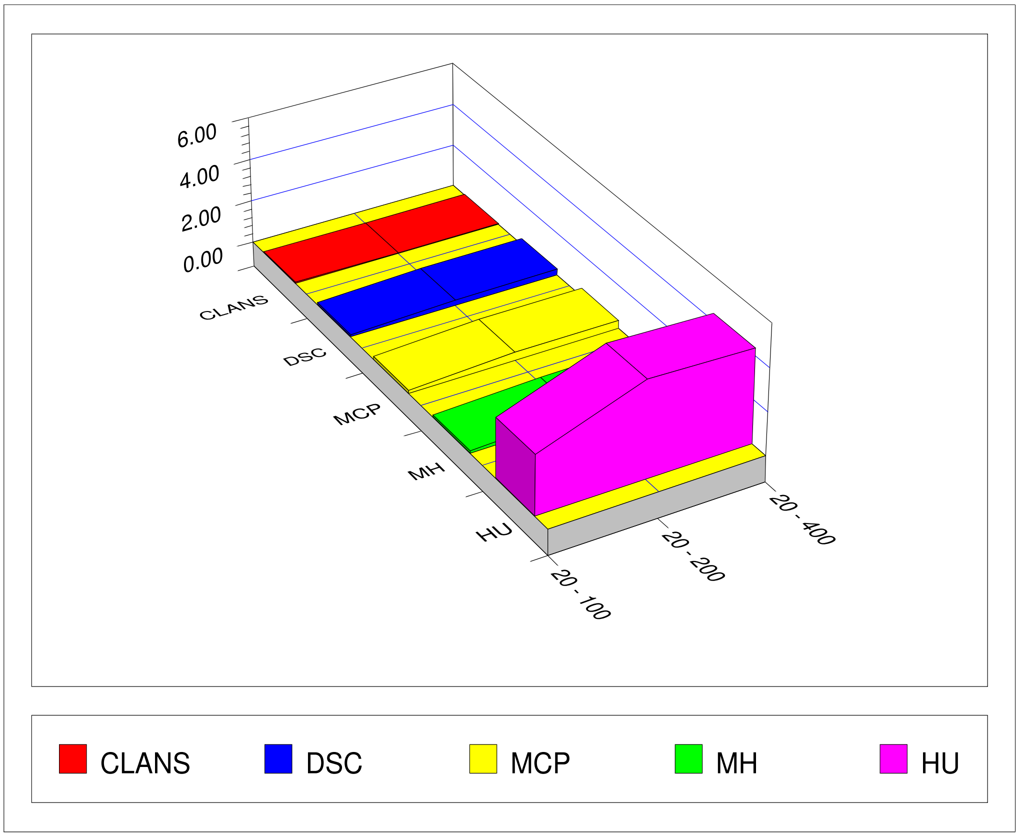


Figure 4: Average relative parallel time for the given Node Weight Range.

Table 8: Average speedup for each heuristic in the given Node Weight Range.

	CLANS	DSC	MCP	MH	HU
20 - 100	3.07	2.98	3.00	2.95	1.65
20 - 200	2.85	2.58	2.58	2.52	1.30
20 - 400	2.83	2.54	2.56	2.50	1.29

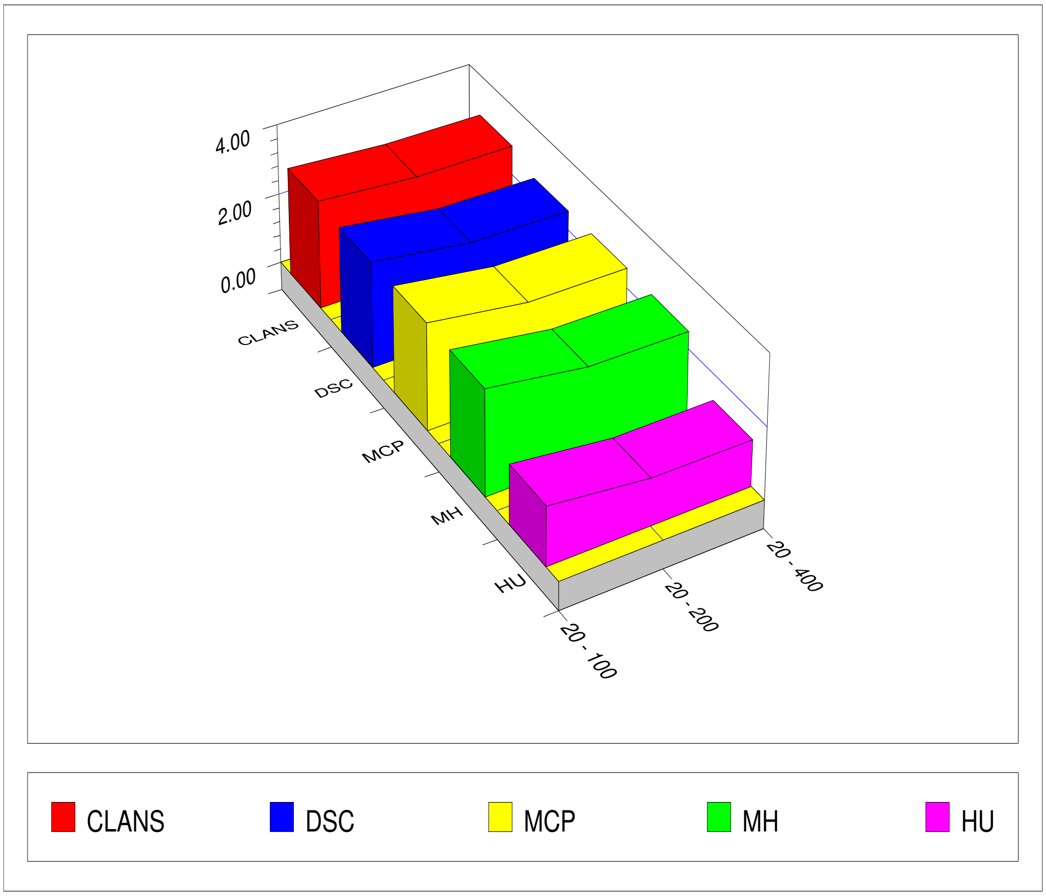


Figure 5: Average speedup for the given Node Weight Range.

4.2.4 Efficiency

A striking difference was revealed in the analysis between CLANS and the other techniques. Clans is dramatically more efficient in all classes.

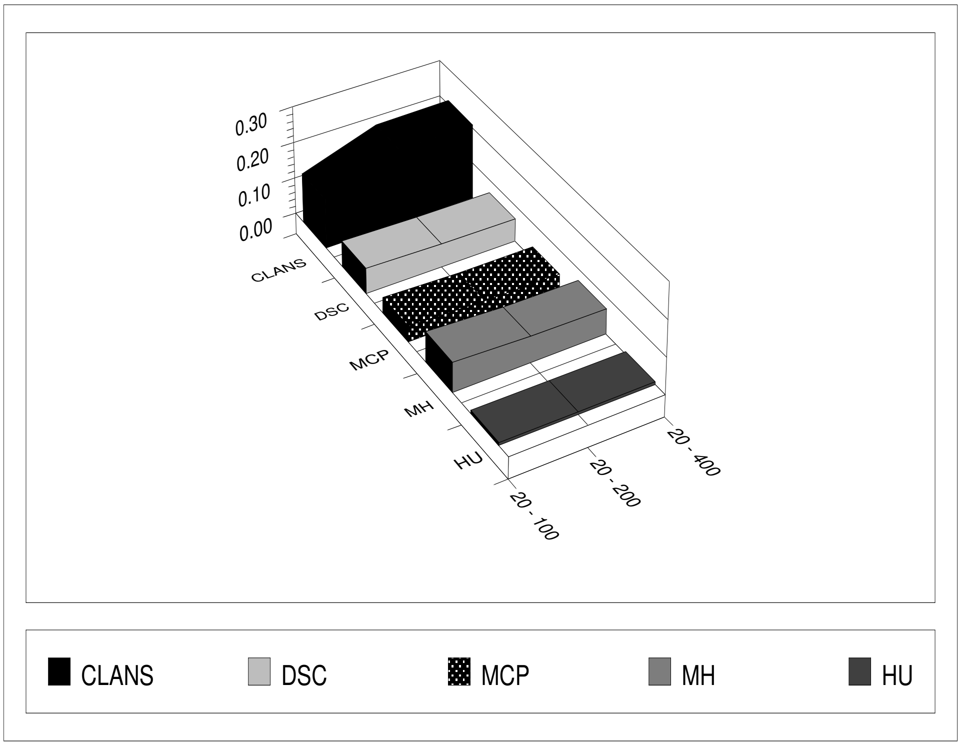


Figure 6: Average efficiency for the given Node Weight Range.

Table 9: Average efficiency for each heuristic in the given Node Weight Range.

	CLANS	DSC	MCP	MH	HU
20 - 100	0.14	0.07	0.04	0.08	0.01
20 - 200	0.21	0.07	0.04	0.07	0.01
20 - 400	0.21	0.06	0.03	0.07	0.01

4.3 Anchor Out-degree Analysis

The third graph parameter used to compare the heuristics was the anchor out-degree. This parameter did not provide any useful information as is seen in Table 10 and Table 11.

Table 10: Number of times each heuristic gives speedup less than 1 for the given anchor out-degree ranges.

	CLANS	DSC	MCP	MH	HU
A = 2	0.00	69.00	94.00	72.00	261.00
A = 3	0.00	71.00	81.00	64.00	260.00
A = 4	0.00	101.00	114.00	85.00	299.00
A = 5	0.00	109.00	122.00	102.00	300.00

Table 11: Normalized average relative parallel time for the given anchor out-degree ranges.

	CLANS	DSC	MCP	MH	HU
A = 2	0.03	0.24	0.33	0.25	3.98
A = 3	0.03	0.24	0.29	0.24	3.86
A = 4	0.04	0.21	0.30	0.22	3.84
A = 5	0.03	0.19	0.27	0.20	3.80

5 Conclusion

5.1 Comparison of Heuristics

The five heuristics discussed in this study fall into three categories:

- Graph Decomposition
- Critical Path
- List Scheduling

CLANS represents the graph decomposition technique while MCP and DSC represent the critical path methods and MH and HU represent the list scheduling technique. One of the subgoals of this work was to find graph domains in which the techniques differed and to point out the relative strength of the techniques for the separate graph domains. The comparison was highly conclusive with respect to the granularity domain, as illustrated in the results discussed in the previous section. In the node weight range domain the comparison was not as conclusive, though there is enough evidence to encourage further investigation of this domain, using a larger set of ranges. The results were inconclusive in the anchor out-degree domain, where no definite relationship was observed for any of the techniques.

The comparison of the heuristics indicates that CLANS is a more robust and generally applicable technique over a larger and more diverse set of graph sets. In particular, the study of granularity established the overall effectiveness of CLANS as the scheduler of choice at low granularities. MCP gave good results at high granularities. MCP, DSC, MH and HU gave extremely bad performances at low granularities, and as many as 50% of the DAGs with low granularity had speedups of less than 1.

The node weight range comparison gave very little conclusive evidence but study of both more selective and wider ranges is called for. The hypothesis yet to be tested is that at narrow node weight ranges, the critical path routines seem to perform better and at wider ranges CLANS performs better. To further explore this domain, a new set of graphs will need to be generated.

As expected, our results confirm that as granularity increases, the performance of all heuristics improves. This trend corresponds to an increase in useful parallelism in the DAGs.

The graph generation system generates graphs using a random parse tree generator. The graphs are then modified by removing and inserting randomly connected edges to match the given anchor out-degree. After the out-degree of the graph has been set, its parse tree does not resemble the randomly generated parse tree. This assertion was verified by a comparison of the randomly generated parse tree and the parse of the graph constituted by the out-degree setting routine. It is unclear whether the graph generation method provided a bias toward any of the heuristics. Further study is required.

5.2 Numerical Comparison Technique

Little work has been done to compare scheduling heuristics for DAGs. Comparison studies found in the literature survey have more to do with the comparison of complexity of the heuristics [19] rather than the performance of the heuristic at scheduling. Although the complexity comparison is very important, secondary to the ultimate objective of getting optimal speedup for all DAGs. As the scheduling problem is NP-hard and most heuristics have not been shown to give optimal solutions even for simple DAGs, a numerical comparison is clearly in order. The literature survey uncovered an abundance of scheduling heuristics, but contained very little data on how the heuristics perform.

All heuristics considered had the same objective function, minmization of total parallel time. General consensus on the execution model and the architectural model was also observed. The above two observations led to the idea of a numerical comparison testbed for scheduling heuristics. The preliminary results from this testbed are promising. The granularity parameter gives a very good overall measure of the useful parallelism in DAGs.

A parallelizing compiler will require the best scheduler to be selected and implemented. The best scheduler may be different for different classes of graphs. Variations in granularity, node weight range and other graph parameters may occur for different reasons. For example, variations may be caused by properties of the serial program dependencies, by the way the graph decomposed by the parallelizing tool, or by the properties of the multiprocessor architecture. The same serial code may therefore give different granularity when it is parallelized for a different multiprocessor, thus causing the compiler to choose a different scheduler for the new granularity. The availability of data indicating the strengths and weaknesses of various schedulers may help compiler designers choose between different algorithms.

Granularity is an important parameter that corresponds to the potential parallelism in a given PDG. The degree of parallelization that should be extracted from a serial program is highly dependent on the communication requirements of the program and the communication speed of the multiprocessor. A good area of further research would be to explore the granularity of DAGs generated from actual code and with edge weight derived from the charachteristics of different multiprocessors.

In this study the granularity range $0.08 < G < 0.2$ seems to be a threshold after which all heuristics perform relatively well. For coarse grains, all three heuristics perform similarly.

The numerical analysis led to the discovery that changes in granularity can point out weaknesses and strengths of heuristics. Granularity is therefore a parameter that may be used to choose schedulers. Similarly, other parameters can be established that may help the compiler designer in choosing the best scheduling heuristic.

Finally, the numerical analysis was also used to establish the best version of CLANS for the comparison performed here. The weaknesses of the first version were remove.

Other scheduling algorithms need to be added to give the study more meaning and to give more choices for the McClan group's parallelizing compiler project. The focus of this work was to validate the numerical comparison technique on a limited number of heuristics. We invite heuristics developed by all other research teams that use execution and architectural models similar to heuristics described here.

Numerical analysis as done here is not a completely satisfactory selection tool for determining the best scheduler. In particular, a next step for a testbed would be to use DAGs generated from real serial programs. These DAGs need to be classified into application classes, which would lead to a more realistic numerical comparison.

The numerical comparison technique a working part of the McClan group's search for the best scheduler. The results represented here indicate initial success in determining graph properties, heuristic properties and the performance of heuristics on sets of graphs with different charcteristics. The method should prove highly useful in determining good scheduling heuristics and their properties.

Acknowledgments

The DSC algorithm was provided by Apostolos Gerasoulis and Tao Yang of Rutgers University.

References

[1] Sarkar V. *Partitioning and scheduling parallel programs for execution on multiprocessors.* PhD thesis, Stanford Unversity, 1989.

[2] Kruatrachue B. and Lewis T. Grain Size Determination for Parallel Processing. *IEEE Software*, pages 23–32, Jan 1988.

[3] Kim S. and Browne J. A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures. In *Proceedings of 1988 IEEE International Conference on Parallel Processing*, pages 1–8. Institute of Electrical and Electronic Engineers, 1988.

[4] Yu W. *LU Decomposition on a Multiprocessing System with Communication Delay.* PhD thesis, University of California, Berkley, 1984.

[5] E. Coffman. *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, 1976.

[6] Graham R., Lawer E., Lenstra J., and Rinnooy Kan A. Optimization and Approximation in Deterministic Sequencing and Scheduling. *SIAM Journal of Applied Math.*, 17(2):416–429, March 1969.

[7] Papadimitrou C. and Ullman J. A Communication Time Tradeoff. *SIAM Journal of Computing*, 16(4):639–646, August 1976.

[8] Gerasoulis A. and Yang T. A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors. In *The Proceedings of the ACM 1990 International Supercomputing Conference*. Association of Computing Machinery, 1990.

[9] Gerasoulis A., Vengopal S., and Yang T. Clustering Task Graphs for Message Passing Architectures. In *Proceedings of ACM Interational Conference on Supercomputing*, pages 447–456, 1990.

[10] Wu M. and Gadjski D. Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Trans. Parallel and Distributed Systems*, 1(3):101–119, July 1990.

[11] Hu T. Parallel Sequencing and Assembly Line Problems. *Operations Research*, 9:841–848, 1961.

[12] H. Anger, Hwang J., and Y. C. Chow. Scheduling with Sufficient Loosely Coupled Processors. *Journal of Parallel and Distributed Computing*, 9:87–92, 1990.

[13] Hwang J., Chow Y., Anger F., and Lee B. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM Journal of Computing*, 18(2):1–8, April 1989.

[14] Lee C., Hwang J. Chow Y., and Anger F. Multiprocessor Scheduling with Interprocessor Communication Delays. *Operations Research Letters*, 7(3):141–147, 1988.

[15] Liu Z. A Note on Graham's Bound. *Information Processing Letters*, 36:1–5, October 1990.

[16] El-Rewini H. and Lewis T. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. *IEEE Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

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

[18] McCreary C., Thompson J., Gill H., Smith T., and Zhu Y. Partitioning and Scheduling Using Graph Decomposition. Department of Computer Science and Engineering CSE-93-06, Auburn University, 1993.

[19] Yang T. and Gerasoulis A. A Fast Scheduling Algorithm for DAGs on an Unbounded Number of Processors. In *The Proceedings of the 5th ACM International Conference on Supercomputing*, pages 633–642. Association of Computing Machinery, 1991.

[20] McCreary C. and Gill H. Efficient Exploitation of Concurrency Using Graph Decomposition. In *Proceedings 1990 IEEE International Conference on Parallel Processing*, pages II–199. Institute of Electrical and Electronic Engineers, 1990.

[21] El-Rewini H., Lewis T. and Ali H. Task Scheduling in Parallel and Distributed Systems. To Be Published. Prentice-Hall, Inc.

[22] Lewis T. and El-Rewini H. Parallax: A Tool for Parallel Program Scheduling. In *IEEE Parallel and Distributed Technology*, pages 62–71. May 1993.

A The Heuristics

This sections describes five heuristics. They are divided into three categories: critical path techniques, list scheduling heuristics and graph analysis. The two critical path techniques include the dominant sequence clustering (DSC) [19] algorithm Gerasoulis and Yang; and modified critical path (MCP) [10] algorithm by Wu and Gajski. HU and MH, written by Lewis and El-Rewini, are list scheduling techniques. The graph analysis technique is the clan-based graph decomposition work (CLANS) [20] by McCreary and Gill.

In both 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.

The list scheduling algorithms, MH and HU, each find a priority for each node in the graph. The nodes are put into a queue and scheduled according to their priority.

CLANS is 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.

A.1 Dominant Sequence Clustering

DSC by Yang and Gerasoulis [19] is an edge zeroing algorithm. The two major ideas behind DSC are to directly attempt to reduce the dominant sequence of the graph, and to reduce the complexity of the algorithm, to $O((v + e) \log v)$. This algorithm keeps track of the dominant sequence to reduce parallel time. The order of complexity of the algorithm is also reduced due to the decreasing focus range 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 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.

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/processor 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.
- $arrivetime(n_j, n_x) \equiv ST(n_j) + ET(n_j) + e_{jx}$ where n_j is a scheduled predecessor of n_x, and e_{jx} is the cost of edge (n_j, n_x).
- $startbound(n_x) \equiv max(arrivetime(n_j, n_x))$, where $n_j \in PRED(n_x)$. This is the lower bound for starting n_x.

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_y ) where c_i gives min(ST(c_i, n_y))
                        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: /*Guarantees that parallel time is not increased if node n_x is assigned to a Predecessor */
  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: /*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 7: DSC algorithm.

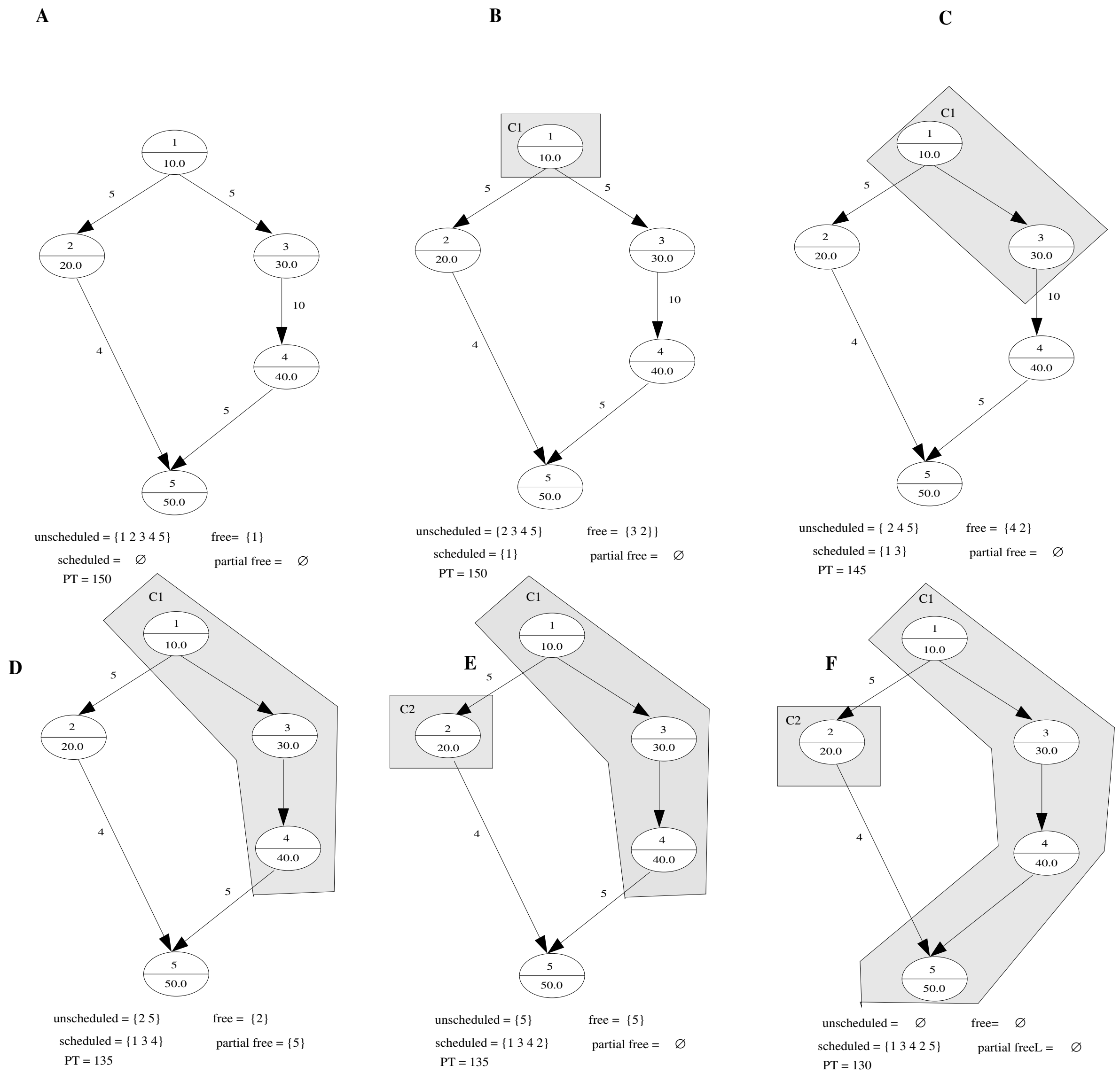


Figure 8: DSC example.

Using these definitions, we define priority for nodes in the free and partial free lists as follows.

$$priority(n_x) \equiv startbound(n_x) + level(n_x)$$

Figure 7 gives the DSC algorithm and Figure 8 illustrates the steps in the execution of DSC with an example.

A.2 Modified Critical Path

MCP, proposed by Wu and Gajski, is similar to DSC in that its basic heuristic is to minimize the starting time of individual tasks using an ALAP strategy. The term ALAP (as-late-as-possible) binding, refers to the latest 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 a T_L for each n_i and all its descendants. 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 9 for the algorithm and Figure 10 for an example of its execution steps. The complexity of MCP is $O(v^2 logv)$

A.3 Mapping heuristic

MH is a modified list-scheduling technique developed by Lewis and El-Rewini that considers processor speed, interconnection topology, and contention. Unlike the other heuristics, MH fits the PDG to various network topologies in an attempt to minimize communication delays. This is done by placing communicating tasks close together. Since the topology we use in our examples is fully-connected our experiment does not take advantage of this feature. In MH, each node is given a priority which is the level as defined by Gersaulis and Yang/citeYAN91. This heuristic picks a processor for a task such that the task could not finish any earlier on any other processor. The task is then allocated to the processor. [22]

A.4 Hu's heuristic

Lewis and El-Rewini modified the classic Hu algorithm to include communication costs. Hu obtains a priority by finding the level. All nodes that have no predecessors are put in a free list in decreasing priority order. After a task has been scheduled, its successors that have all of its predecessors scheduled are put into the free list. [21]

A.5 CLANS

The partitioning scheme used in the 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

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 descendants, 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 9: MCP algorithm.

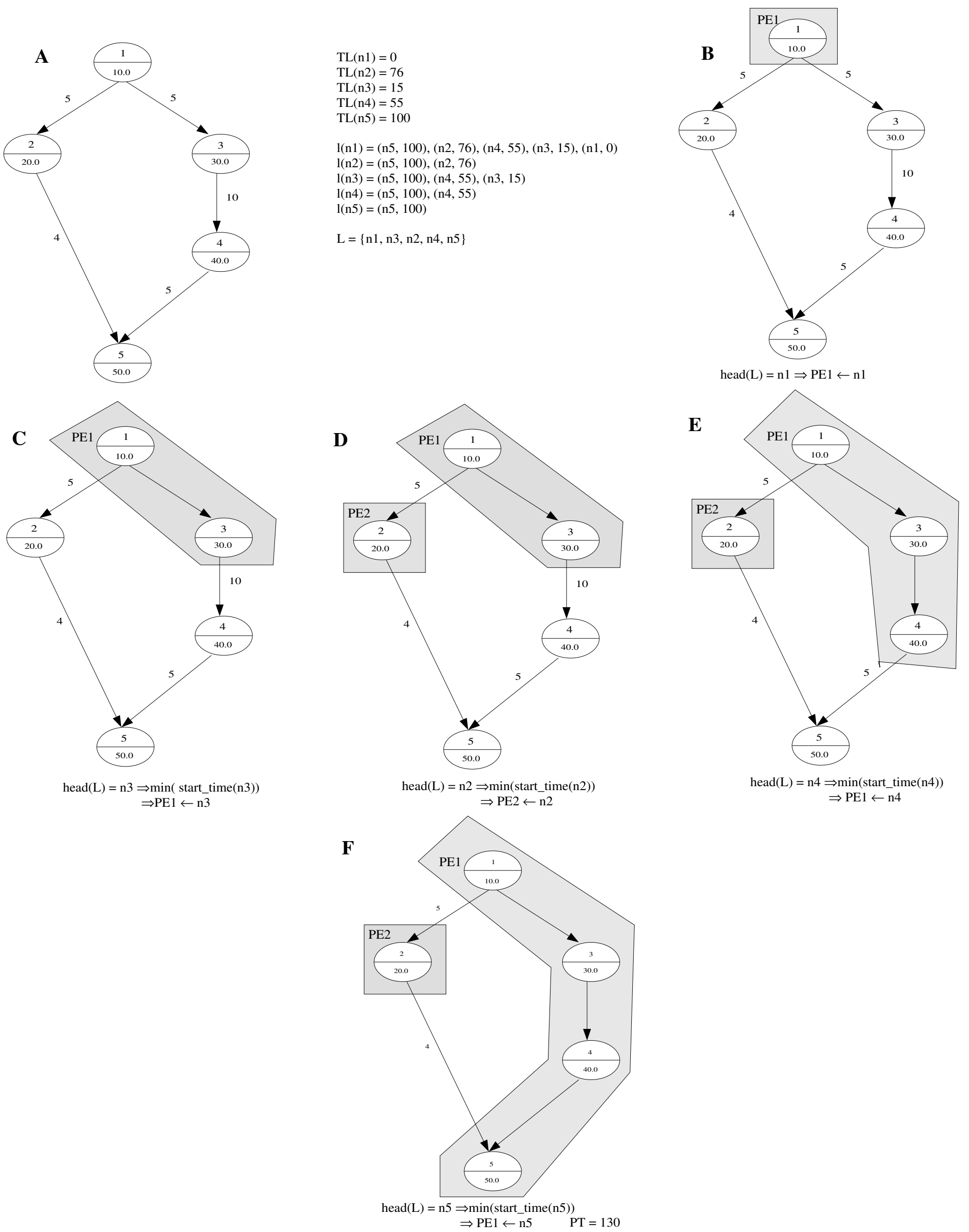
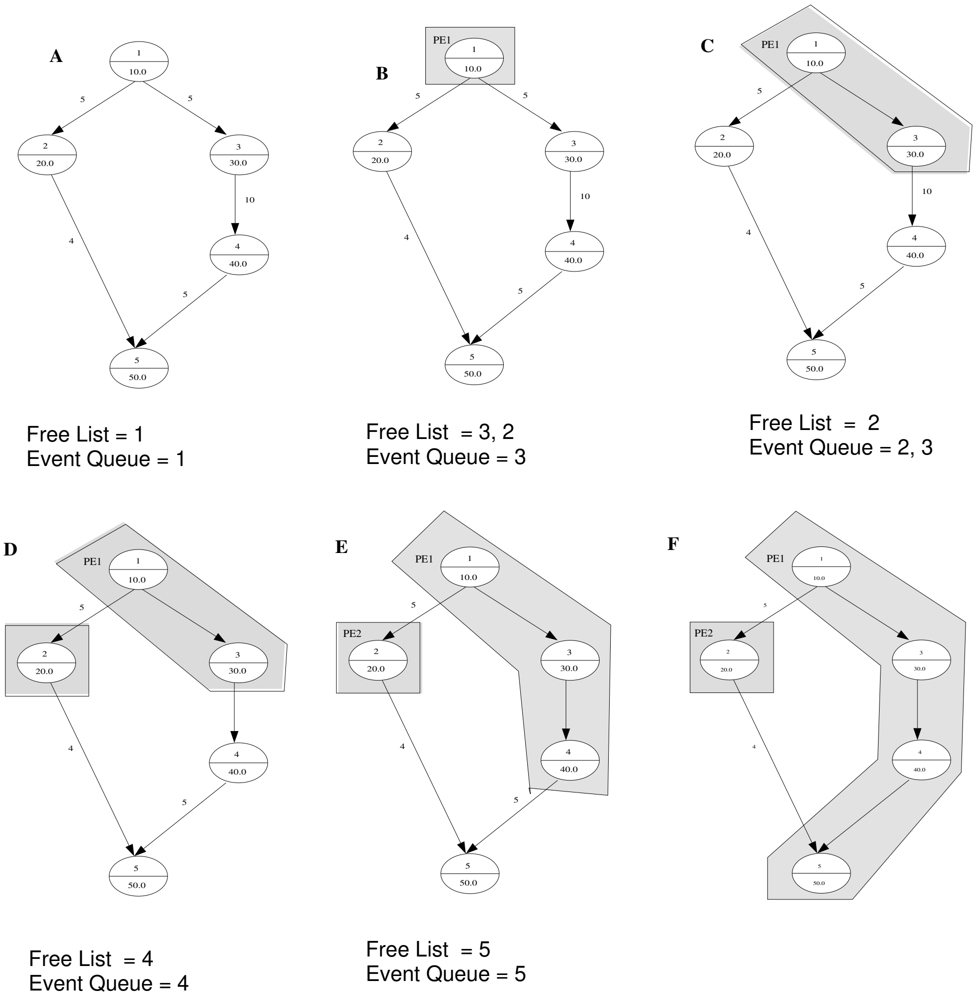


Figure 10: MCP example.

The Algorithm (MH)

```
Insert  a single exit node.  Edges to this node are given a weight of 0.
For each node n, compute level(n)
Initialize free list
Event list <- 0
Repeat
   Get task T with highest level from free list
   While T != nil
        begin
            Choose the processor on which T could start the earliest
             Allocate T to the processor
             Put T on Event List
             Get new T of highest priority from free list
          end
     Get the next event E off the event list
     While event list not empty
          if E has successors
               add to free list all successors which are satisfied and put in order of level
          get next event E from Event List
 Until iterated once for every node
```

Figure 11: MH algorithm.



Node	Level
1	150
2	74
3	135
4	95
5	50

Figure 12: Mapping Heuristic example.

```
The Algorithm(HU)

 Find the level for each task and use it as the task's priority
 Put into free list all nodes which have no predecessors
       in order by priority
 Get task t off the free list
 Assign t to the first proecessor
 repeat
      update free list
      if (free list not empty)
             get next t off the free list
             Find processor with earliest start time
             Assign t to this processor
 until all tasks have been assigned to processors
```

Figure 13: Hu algorithm.

represent clan subgraphs. For each clan in the tree, its children are similarly subclans of their parent. Formally, a set of vertices C in graph G is a *clan* iff for all nodes x, y in C and z in $G - C$:

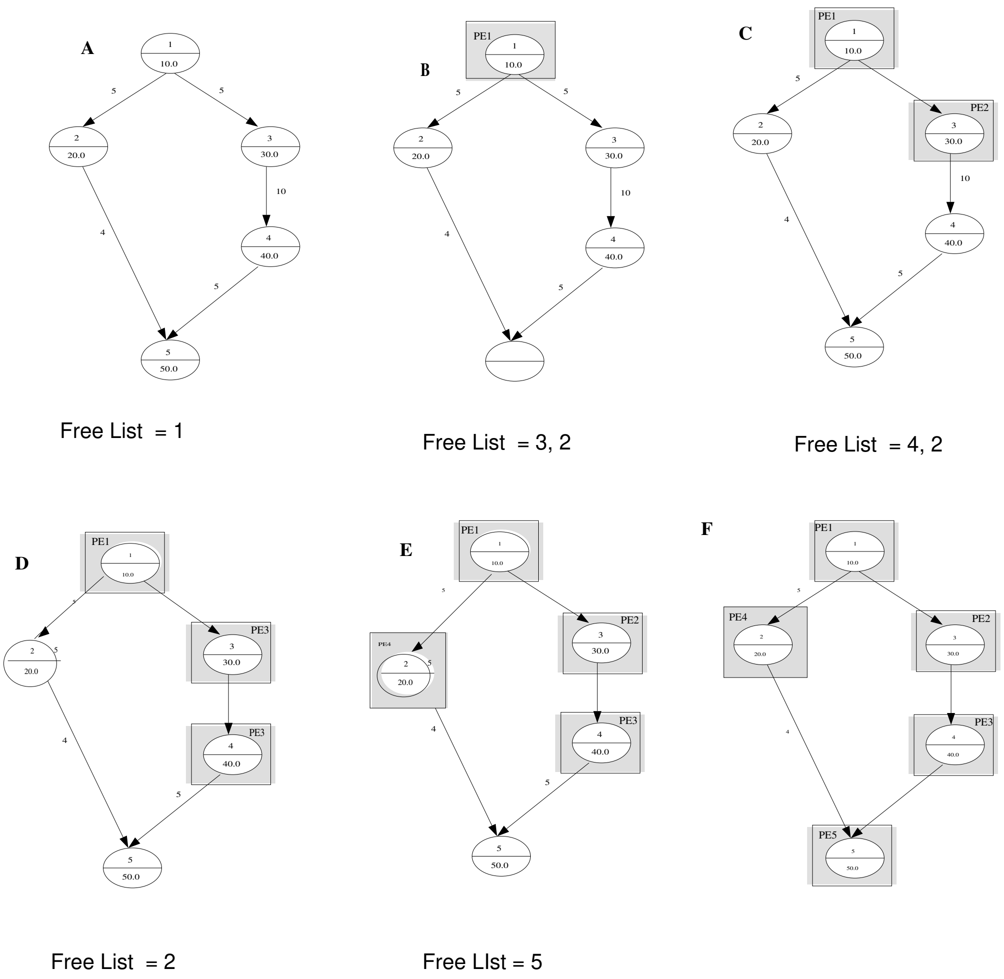
1. z is an ancestor of x iff z is an ancestor of y and;
2. z is a descendant of x iff z is a descendant of y.

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.
- 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 those that cannot be decomposed into independent and linear clans.
- The clan structure derived from a PDG is unique and forms a hierarchy that is called a *parse tree* of the PDG.
- 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.

For a detailed explanation of the algorithm see [20]. Only a high level description is given here. The current version of the parse is $O(n^3)$ but an $O(n^2)$ parser is under development.

For the graph of Figure 16 (A), the non-trivial clans to be placed in the tree include the linear clan $C_1\{3,4\}$, the independent clan $C_2\{2,\{3,4\}\}$, and the linear clan $C_3\{1,\{2,\{3,4\}\},5\}$. The parse tree created in step 1 of the algorithm is shown in Figure 16 (B). In step 2 of the algorithm, costs are assigned in a



Node	Level
1	150
2	74
3	135
4	95
5	50

Figure 14: Hu example.

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.

Figure 15: CLANS algorithm.

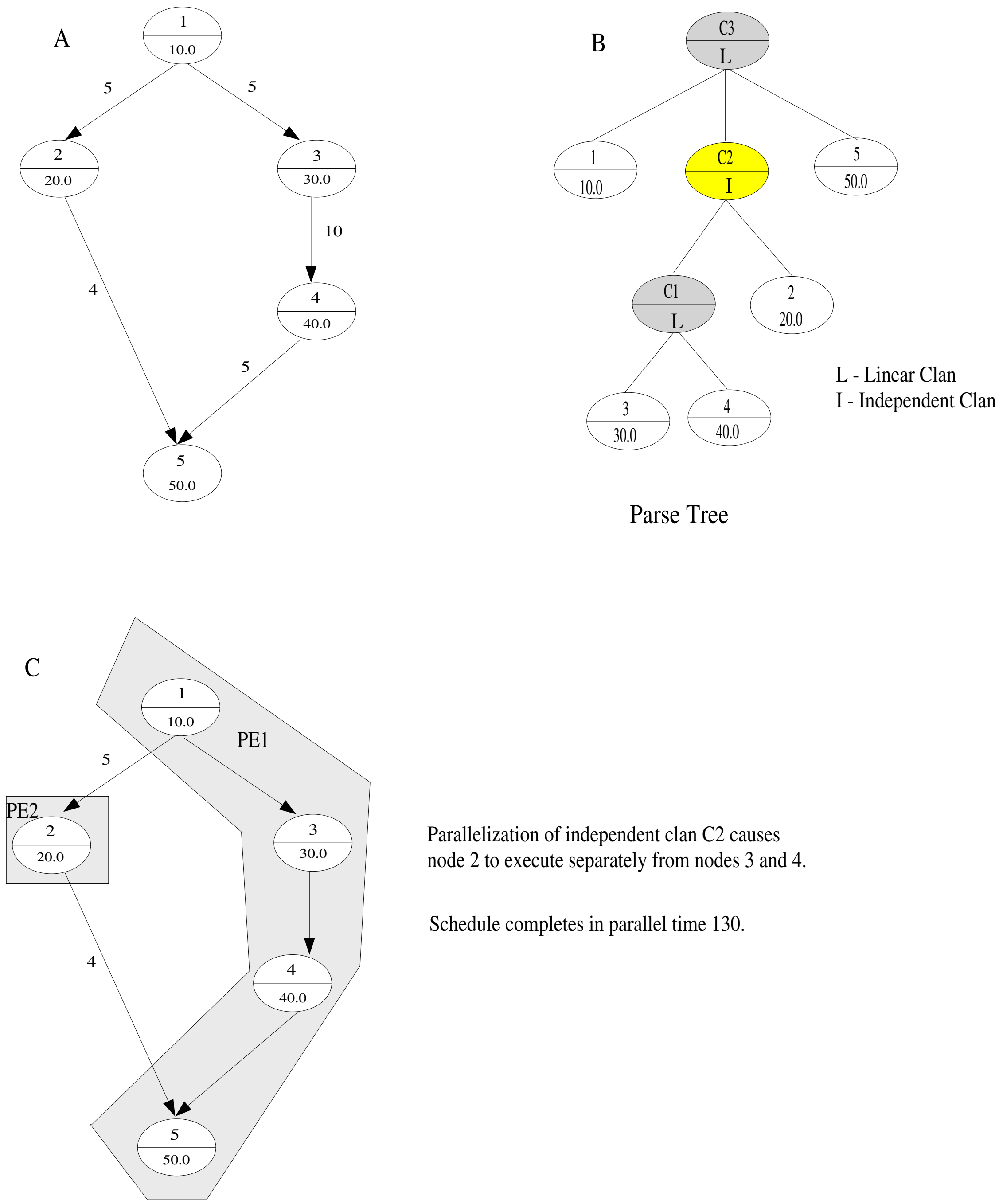


Figure 16: CLANS example.

bottom-up fashion. The leaves have the costs of the corresponding nodes in the original graph. C_1 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 C_3, a choice is presented for one of its children: cluster the nodes in C_2 for a cost of 20 + 70 = 90, or parallelize it by executing C_1 and node 2 concurrently. The cost of parallelizing C_2 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. C_1, executing on the same processor as the nodes with which it communicates, costs 70. The cost of parallelization of clan C_2 is then $70 = max(29, 70)$. Since the entire graph is linear, the execution sequence is node 1, clan C_2 and node 2 concurrently, followed by node 5, for a cost of 10 + 70 + 50 = 130. The schedule is shown in Figure 16 (C).