A COMPARISON OF TASK PARTITIONING TECHNIQUES

Technical Report CSE-91-14

C. L. McCreary, M. E. McArdle

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849-5347

December 1991
A COMPARISON OF TASK PARTITIONING TECHNIQUES

C.L. McCreary* and M.E. McArdle*
Department of Computer Science and Engineering
Auburn University, AL 36849

*This research is sponsored by NSF grant no. CCR-8909253
1. THE PARTITIONING PROBLEM

With the advances in multiprocessor technology comes the problem of creating programs that take full advantage of the capabilities of these systems. An important research area in parallel computing is the partitioning and scheduling of an algorithm for concurrent execution. Partitioning groups the steps of a program into sets, or grains that execute sequentially on a single processor. Scheduling assigns the grains to physical processors to minimize execution time. It is desirable to detect potential parallelism in the sequential algorithm and partition it for concurrent execution in the most efficient manner [SAR89]. The most important goal is that the transformed algorithm executes as quickly as possible on the target architecture. For a distributed memory system, this involves the conflicting goals of maximizing parallel execution of code and minimizing interprocessor communication.

The first phase in compiling a program for parallel execution is finding dependencies in the program that will prevent parallelism. There are two types of dependence. Data dependence occurs when different blocks of program code reference or change a variable in a way that forces an order of execution on those blocks. Control dependence occurs when the execution of the statements in one block determines whether another block executes. Both data and control dependencies in the program must be identified and the program
should possibly be reorganized to permit parallelism [ZIM90, ALL88]. Two examples of tools that perform this task are PTRAN [BUR88, ALL88] and START/PAT [APP89]. The output of this phase is a program dependence graph, or PDG [ZIM90, FER87].

The program dependence graph (PDG) is a directed acyclic graph (DAG) which represents the dependencies in a program. Vertices represent statements or blocks of statements in the program. Edges represent dependencies between blocks that must be satisfied in any execution of the program and impose an order between the vertices in the graph. A grain is a group of processes that execute sequentially on one processor. The problem is determining what the proper grains are for a particular algorithm on a particular multiprocessor system [McC89, KER71]. After establishing the grains, they must be scheduled for execution. It has been shown that this problem is NP-Complete [ULL75]. Therefore, approximate solutions are used. Often, the scheduling problem blends into the partitioning problem, and the two are solved together.

The most viable solutions accept a PDG as input and group its vertices into grains that are scheduled on processors. When the target architecture is a distributed memory multiprocessor system (DMS), the solution cannot consider only the cost of executing statements on the processors. It must account for the cost of communicating data between processors [McC89, KER71, KIM88, GYL76]. Currently, there exist several PDG based partitioning strategies. Three competing approaches were chosen for evaluation: 1) Localized Allocation of Static Tasks (LAST) [BAX89], 2) Linear Clustering
(LC) [KIMth], and 3) Clan Based Parsing (CLANS) [McC89, McC90].
LAST, LC, and CLANS are chosen because they have several important characteristics in common:

1) All methods use a PDG as input and consider the effect of intertask communication.

2) All methods can produce graph partitions assuming the target architecture is fully connected.

3) The ultimate goal of all the methods is to produce schedules that minimize the total throughput. The objectives of other partitioning schemes may differ. Some algorithms attempt to minimize communication time, and others try to keep processors busy.

4) All methods produce static schedules that can be compared.

5) The three methods competing come from different classes of partitioning schemes. Linear clustering is a critical path technique; the LAST algorithm partitions based on node neighborhoods; and CLANS analyzes the PDG structure.

The three approaches are evaluated against each other to determine which is the best solution to the partitioning and scheduling problem. If winners vary, the characteristics of the PDG that favor one scheme over the others are determined. Included in this evaluation is a comparison of schedules produced by the methods for a variety of input DAGs and comparisons of strengths and weaknesses of each method. Of course, the most important criterion for choosing the superior method is fastest completion time. Section 2 contains descriptions of the task partitioning
methods. Section 3 compares the schedules produced by these techniques and summarizes the results. Section 4 concludes the report.

2. TASK PARTITIONING TECHNIQUES

All the task partitioning techniques described in this section expect a PDG as input. Each vertex represents a sequential task in the computation. The vertices are labelled with computation time and the edges are labelled with communication time. When allocating the vertices of the graph to multiprocessors, it is assumed that as many processors are available as are needed for an efficient schedule. Another assumption made is that the target multiprocessor's nodes are fully interconnected. This is justifiable because there exist distributed memory systems such as the Intel hypercubes that behave as though they were fully interconnected. Communication between processor nodes that are not directly connected is not significantly more costly than nearest neighbor communication [BOM89].

2.1 Localized Allocation of Static Tasks (LAST)

Localized allocation of static tasks is similar to other techniques that examine neighborhoods of vertices and is described in [BAX89]. LAST assigns processes to processors based on their proximity to groups of processes that have already been assigned. LAST successively allocates sets of task vertices to processors until a complete mapping is found. In general, the next task is
chosen based on its connectivity with already assigned tasks; then
it is allocated to a processor based on the estimated time in which
this assignment can be computed. The algorithm is capable of
considering the connectivity of the target system when estimating
the impact of interprocessor communication but is evaluated only
for the case where the target architecture is fully connected.

Some definitions are needed to discuss this algorithm. A
group, $g_i$, is a subset of vertices in the PDG that are assigned to
processor $i$. A vertex can be assigned to at most one processor (or
group). A frontier, $f_i$, contains all vertices not assigned to any
group with at least one edge connecting them to some vertex in
group $i$. An edge $(j,k)$ is defined if either adjacent vertex $j$ or $k$
is a member of any group. The value $D\_EDGE_{j,k}$ is 1 if $(j,k)$ is
defined, 0 otherwise. $COMM_{j,k}$ is the communication cost on the edge
$(j,k)$.

LAST has two sub-algorithms, task selection and task alloca-
tion. The task selection phase maximizes what is known about
vertices in the PDG before choosing one for processor allocation.

Each vertex $j$ has a value $D\_NODE_{j}$ which is the percentage of
vertex $j$'s communication costs that are defined edges. The formula
to compute $D\_NODE_{j}$ is below. $k$ and $i$ represent adjacent vertices
to $j$.

$$D\_NODE_{j} = \frac{\sum (COMM_{k,j} \times D\_EDGE_{k,j}) + \sum (COMM_{j,i} \times D\_EDGE_{j,i})}{\sum COMM_{k,j} + \sum COMM_{j,i}}$$
To choose the next vertex for allocation, search through all frontiers to find the vertex with the largest D_NODE value. The frontiers eliminate vertices that are not connected to any group to reduce the search space.

After a vertex is chosen, it is allocated to a group. The STRENGTH$_{j,i}$ is a rough measure of how highly connected vertex $j$ is to group $i$. For vertex $j$, strength is computed for every group in which $j$ is on the frontier.

$$\text{STRENGTH}_{j,i} = \sum \frac{\text{COMM}_{k,j}}{\text{COMP}_{j}} + \sum \frac{\text{COMM}_{j,l}}{\text{COMP}_{j}} - \sum \frac{\text{COMP}_{j}}{\text{COMM}_{j,l}}$$

for all vertices $k, l \in g_i$ and edges $(k,j), (j,l)$

The first two sums are the amount of communication vertex $j$ has with group $i$. The negative term attempts to estimate the gain obtained by assigning vertex $j$ to a group other than $i$.

Before allocation, the D_NODE of vertex $j$ is appraised. If this value is greater than some threshold, then $j$ is forward allocated. Otherwise $j$ and a set of related vertices are backward allocated. The threshold value indicates the minimum percentage of communication that must be on defined edges for a vertex to be forward allocated. An allocation of a set of tasks is necessary when there really is not enough known about task $j$ for it to be assigned individually.

In forward allocation, $j$ is tentatively assigned to group $i$, chosen by the evaluation of strength. The cost of adding this task to the group is calculated. The cost estimates how quickly
processor i can compute task j. Cost accounts for the computation time of tasks that must complete before j can begin. It also estimates communication costs for the task's defined edges. Using the cost estimate as a maximum value, the task is also tentatively assigned to processors that are connected to processor i. As long as quicker completion times are estimated, adjacent processors are tried until the minimum cost allocation is found.

Backward allocation is necessary when more of vertex j's edges need to be defined before assigning j to a processor. Therefore, the vertex is allocated with a set of its ancestors. The set includes all vertices, not yet allocated, whose outputs contain a path to vertex j and is referred to as the input tree. The tree is compressed into one task vertex. Strengths for the new vertex are computed and a group is chosen. In this case, not only is the new vertex tentatively assigned as in forward allocation, but the unassigned parents of vertex j are also backward allocated. The combinations of allocations are tried on different processors recursively until the cheapest overall mapping is found.

As an example of how LAST works, consider the PDG in Figure 1. The vertices are labelled with a node number and a computation value. Assign the root vertex 1 to group 1. Now the vertices with defined edges are 2, with a D_NODE value of 0.5 and 3, with a D_NODE value of 0.333. Vertex 2 is chosen for the next allocation. Assuming a threshold value of 0.65, then D_NODE2 is less than the threshold. Vertex 2 must be allocated with its input tree. However, in this case there are no unallocated ancestors of vertex
2, so essentially this task is forward allocated. Because the vertex is only in frontier 1, tentatively assign it to processor 1 ($P_1$). The estimated completion time for this task is 30. Task 1 executes for 10 units of time, then 2 executes another 20. The only defined communication for vertex 2 so far is the edge from vertex 1. Because task 1 is assigned to the same processor, the communication does not affect the estimated completion time.

Now vertex 2 must be assigned to neighboring processors to determine if a better allocation is possible. An assignment of the task to any other processor results in an estimated time of 35, due
to the communication delay from task 1. Therefore, vertex 2 is allocated to \( P_1 \) with vertex 1.

After allocating vertex 2 to a processor, those vertices with defined edges are 3 and 5. Both are in frontier 1 and have \( D_{\text{NODE}} \) values of 0.333 and 0.5 respectively. Choose vertex 5 to allocate. Because \( D_{\text{NODE}} \) is less than the threshold, it must be backward allocated along with the unallocated vertices of its input tree, nodes 3 and 4 (Figure 1). After compressing the input tree into a single vertex, the cost of assigning the compressed node onto \( P_1 \) is computed as 150. Next, the compressed node is allocated to an adjacent processor. The estimated cost is higher, 155, due to communication from vertex 2, so a ceiling of 150 is placed on the cost of allocating the nodes. Alternative mappings of the group of nodes must be explored. Basically, forward allocate vertex 5 to a group, then recursively backward allocate its parents. Whatever combination produces the best cost is the final schedule that is kept. The final allocation of \( P_1 = \{1, 2, 5\} \) and \( P_2 = \{3, 4\} \) has a completion time of 140.

2.2 Linear Clustering (LC)

Linear clustering is based on critical path analysis and is detailed in the work by S. J. Kim in [KIMth]. Like other critical path approaches, LC finds the path in the program dependence graph with the greatest length in terms of computation and communication. The next longest path is found in the PDG until all such paths are determined. These paths are referred to as linear clusters. This
subgraph represents a list of units of computation that are sequentially dependent and are assigned to the same processor.

LC is done in two phases, clustering and merging. Clustering cuts the PDG into a set of disjoint connected subgraphs. A linear cluster is a special subgraph where every vertex has at most one input edge and one output edge. Merging combines clusters that cannot execute in parallel and assigns them to processors.

LC assumes all graphs have a single source (root) and a single sink (terminal). Clustering locates linear clusters within the PDG as follows: find the longest path (critical path) by combining computation and communication costs from the root vertex to the terminal vertex in the graph. It traverses this path backwards from the terminal vertex to the root, cutting incoming and outgoing edges incident on the vertices on the path. The result is the first linear cluster based on the critical path. Continue this process on remaining subgraphs until the PDG is broken into a set of linear clusters. A simple example in Figure 2 illustrates this process. All vertices and edges have the same weight in the example.

The process of clustering helps facilitate the compromise between the conflicts of maximizing parallel execution and minimiz-
ing interprocess communication. Linear clustering favors parallel-
ism by partitioning a graph into disjoint lists. These lists have
the potential to execute in parallel and therefore are assigned to
different processors.

Linear clustering also helps reduce interprocessor communi-
cation by allocating the tasks in a linear cluster to one proces-
sor. Because tasks in a cluster are sequentially dependant on each
other, allocating them to the same processor eliminates communi-
cation between them. Each linear cluster becomes a vertex in a new
graph called the virtual architecture graph (VAG). The VAG
represents an optimal processor connection scheme for the computa-
tion. Adjacent clusters should be allocated to adjacent physical
processors to reduce communication distances between them. Since
the processor configuration is assumed to be fully connected,
linear clusters are simply allocated to any processor.

After clustering is complete and the VAG is available, some
clusters that are highly interdependent are merged. The merging
step combines clusters that cannot execute in parallel, thus
further reducing unnecessary communication. Merging utilizes the
levels in the PDG of the task vertices. The level number of a
vertex is the length of the longest path from the root to that
vertex. If two vertices have the same level number, they may
execute in parallel. Also, if vertices have different level
numbers, there may exist some dependence that prevents them from
executing in parallel.
There are two cases when clusters are merged; when they are *sequentially strong-dependent* or *mutually strong-dependent*. Two linear clusters are sequentially strong-dependent if their tasks have no common level numbers and if the trailer vertex of one cluster precedes the header vertex of the other. So, the first task of one cluster can only begin after the last task of the other has completed. Two linear clusters are mutually strong-dependent if their tasks have no levels in common and the following conditions hold for the tasks $T_1$ and $T_2$ in the first cluster. $T_1$ is a direct ancestor of $T_2$. $T_1$ is the direct ancestor with the highest level number of all parents of the header vertex in the second cluster. $T_2$ is the direct descendant with the lowest level of all children of the trailer vertex in the second cluster. When pairs of linear clusters meet either of the dependency conditions they are merged into one cluster.

Once dependent clusters are merged, an additional refinement is performed. Given the cluster with the longest estimated computation time, an edge $(i,j)$ is chosen, where $i$ is in the cluster and $j$ is adjacent to the cluster. Temporarily merge the two vertices into one. If the result of assigning the new vertex to either cluster reduces the estimated completion time, then the refined cluster is retained. This refinement is repeated until no more reduction in the estimated schedule time is found.

After clustering and merging, the resulting VAG must be mapped onto the physical processor configuration. Because it is assumed that the target machine is fully interconnected, each node in the
VAG represents the list of tasks assigned to one processor. I.e., the final linear clusters represent the schedule of tasks onto processors.

As an example, consider the graph in Figure 3. The linear clusters are \{0, 2, 3, 5, 6\}, \{1\}, and \{4\}. Note that the simple clusters containing vertices 1 and 4 are sequentially strong-dependent. They are merged into one cluster. \(C_1 = \{0,2,3,5,6\}\) and \(C_2 = \{1,4\}\).

Let \(e_i\) represent the execution time of vertex \(i\) and \(c_{i,j}\) the communication time of edge \((i, j)\). The estimated completion time of \(C_1\) includes communication delays of 2 from edges \(c_{0,1}\) and \(c_{1,3}\) and 101 from edges \(c_{3,4}\) and \(c_{4,6}\) for a total of \(e_0 + e_2 + e_3 + e_5 + e_6 + c_{0,1} + c_{1,3} + c_{3,4} + c_{4,6} = 153\). Two edges, \((0,1)\) and \((3,4)\) are edges from a parent in \(C_1\) to a child outside of \(C_1\). If vertex 1 is merged with vertex 0, the estimated completion time increases to 161. In this case the benefit gained by parallelizing tasks 1 and 2 would be lost if task 1 were placed on the same processor. If vertex 4 is merged with 3, and placed in the first cluster, the estimated completion time decreases to 62. This is a result of eliminating the costly communication between tasks 4 and 6. After cluster merging and testing the vertices adjacent to the longest path, the final cluster assignment is \(\{0,2,3,4,5,6\}\) and \(\{1\}\).
2.3 Clan Based Parse (CLANS)

Clan decomposition provides the basis of a method for aggregating program tasks into a grains which are then assigned to processors [McC89, McC90]. A grain is a set of program statements executed sequentially on a single processor. Basically, program tasks are aggregated into bigger grains when intertask communication outweighs benefits gained by executing the tasks in parallel. The clan partitioning strategy involves two phases. A unique parse of the PDG is built from its clans and organized into a parse tree. Scheduling decisions for the program tasks are then made using the parse.

Before describing the partitioning strategy, several key ideas must be defined. A Hasse Graph, $G_H$ is the transitive reduction of a directed graph, $G$, or more simply, the graph $G$ with all shortcut paths between pairs of vertices removed. $G_H$ retains the task ordering present in the original graph as illustrated in Figure 4.

A set of vertices, $X$, in $G$ is a clan iff for all $x, y \in X$ and all $z$ in $G-X$:

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 $y$. 

![Graph, G and Hasse Graph](image)

**Figure 4** Hasse Graph
Simply stated, a clan is a subset of nodes where every element outside of the set is related in the same way to every member in the set. Figure 5 illustrates the three types of clans that exist.

Clans have properties that make them useful when determining grains for parallel execution. Because all sources of a clan have the same set of parents, interprocessor communication is reduced by placing the members of a clan on the same processor. Similarly, the terminals of a clan have the same children, so one set of data can be broadcast from the clan to all processors containing the children.

The first phase of CLANS creates the parse tree of a PDG. Because the Hasse graph captures all data dependencies of the computation, it is substituted for the graph in the parsing phase. A series of quotient graphs are constructed in order to determine the parse of a graph. Given a graph G, find maximal clans of each type. Replace each clan with a vertex, resulting in a quotient graph, G'. Continue grouping vertices into clans until a quotient graph with only one clan is found. Primitive clans are broken into independent and linear clans by adding edges from the sources of the clan to the next level of vertices in the clan. This series of quotient graphs represent the hierarchy of the parse tree. An example parse is given in Figures 6-8.
One important characteristic of the parse tree is that it is bipartite. Linear and independent nodes are on alternating levels.
Children of a linear node must be executed in sequential order, whereas children of an independent node can be executed simultaneously.

The scheduling phase uses the parse tree to determine when to aggregate tasks into grains. Cost estimates are assigned to the nodes of the parse tree, bottom to top. Leaf nodes are assigned execution values from the corresponding vertices in the PDG. If a node is linear, no parallelization can occur. Therefore all its children are executed on the same processor. In this case, the node cost is the sum of the node costs of the children. To determine which independent nodes are executed in parallel, examine adjacent independent children of a linear node pairwise. Consider the case where these clans contain two nodes each. Figure 9 illustrates four possible scheduling choices: 1) the left pair of nodes and the right pair of nodes are parallelized, 2) the right pair is parallelized and the left pair is aggregated, 3) the left pair is parallelized and the right pair is aggregated, or 4) both pairs of nodes are aggregated. Choose the case with the shortest execution time. Label the nodes in the parse tree with their estimated computation times.
After costs are assigned to the parse tree, they are used when producing the final task schedule. If a node cost is equal to the sum of the cost of its children, assign all children to the same processor. If the node cost is less than the sum of its children, then group independent children into grains as determined in the previous illustration.

Figure 8 shows the final node labelling computed for the parse tree and also the resulting schedule.

3.0 COMPARISON OF METHODS

A variety of PDGs is used to compare the three task partitioning strategies. For each input, a schedule of tasks was produced by all three methods. Code implementing the LAST algorithm was provided by Jeff Baxter of the University of Illinois at Urbana-Champaign. The LAST algorithm produces a mapping of tasks to processors. The mapping produced for each PDG was then transformed into a processor time schedule. Code implementing the steps of LC was provided by Mandar Joshi of the University of Texas. The program produces a list of clusters and the tasks contained in each. Again, the output was transformed into a time schedule. Schedules based on CLANS were derived from the parse trees.

A schedule indicates task execution intervals and communication delays. The most important detail of these schedules is the number of units of time taken for a partitioned program to complete.
Section 3.1 explains the PDGs that were used to evaluate the partitioning methods. Section 3.2 includes some examples of schedules derived by each method. The results of partitioning each PDG are summarized in this section. Section 3.3 discusses the relative merits of the methods.

3.1 The Program Dependence Graphs (PDG)

Diagrams of the PDGs are provided in Figures 10-16. To provide a variety of types of PDGs as input to the three partitioning methods, diverse graphs were chosen. The test attempts to be fair and not favor one method over the others. Those chosen range from simple trees to irregular graphs. All computation and communication values are given as units of time.

SUM1 represents the sum of products computation. Vertices 1 through 8 perform multiplication and have computation values of 10 units. The remaining vertices represent addition and have computation values of 1 unit. All edges represent communication costing 18 units.

Figure 10  SUM1
SUM2 is the reverse of SUM1. Where the potential parallelism occurs in the beginning, or top level, of SUM1, it now appears toward the bottom of this graph. All vertices and edges have the same weights as in SUM1.

K1 is an adaptation of a graph described by the author of the LC algorithm [KIMth]. It is another upside down tree similar to SUM1. However, edges within the heart of the graph alternate between values of 1 and 10. Neighboring pairs of paths differ in length. All vertices have computation of 1 unit.

FFT is another symmetrical graph, but it is not a tree [HOC8-8]. It represents the computation of fast fourier transform. An important characteristic of this graph is that every output vertex is connected to every input vertex. Three versions of this graph
were used, each with different computation and communication values.

**FFT-1:**
- vertices 1-8: 1
- vertices 9-12 and 17-20: 20
- vertices 13-16: 30
- vertices 21-28: 1

**FFT-2:**
- vertices 1-8: 60
- vertices 9-12: 50
- vertices 13-28: 5

**FFT-3** is labelled with costs based on computation and communication time estimates for the Intel iPSC/2 hypercube. The computation costs are derived from data on floating point operations for the 387 math coprocessor [Intel Microprocessors, 1990]. The edge cost represents an estimated amount of time (in microseconds) to send 100 bytes or less of data from one processor to another [BOM89, McC91]. This example indicates how costly communication is relative to fine grain computation on the iPSC system.

**FFT-3:**
- vertices 1-8: 10
- vertices 9-20: 272
- vertices 21-28: 12

**FFT-4** is similar to the previous fft graphs, except this version has 16 inputs and 16 outputs rather than 8. Again, the computation and communication values are based on hypercube estimates, but in this case the inner vertices are labelled with

![Figure 14 FFT-4](image-url)
136 units rather than 272. The change represents finer grain of computation at each vertex. All edges are still 500 units.

**FFT-4:**

- Vertices 1-8: 10
- Vertices 9-20: 136
- Vertices 21-28: 12

**NEQ,** found in [KWA90], is based on the 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, and the vertex values range from 108 to 256.

**IRR:** This is an adaptation of a PDG of a physics algorithm described by Kim, the author of the LC algorithm [KIM88]. It was chosen because of the highly irregular layout. An edge value of 50 and computation values ranging from 5 to 100 have been added to the graph.
3.2 Resulting Schedules

Sample schedules for graphs SUM1 and FFT-3 produced by each of the three methods are provided at the end of this paper. Shaded portions of the schedules represent communication delays. The number within the shaded area indicates the task from which the message is sent. For SUM1, CLANS and LAST produce a schedule completing in time 59. LC produces one that completes in time 50. For FFT-3, CLANS produces a schedule completing in time 1860, while the schedules produced by LAST and LC complete at times 2220 and 2838, respectively.

Table I summarizes the scheduled execution times computed by the methods for each of the input graphs. Usually, CLANS resulted in a better partition of the PDG. In most cases, CLANS did at least as well as LC or LAST.

<table>
<thead>
<tr>
<th>Method Graph</th>
<th>CLANS</th>
<th>LAST</th>
<th>LC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM1</td>
<td>59</td>
<td>59</td>
<td>50</td>
</tr>
<tr>
<td>SUM2</td>
<td>45</td>
<td>55</td>
<td>50</td>
</tr>
<tr>
<td>K1</td>
<td>11</td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td>FFT1</td>
<td>124</td>
<td>146</td>
<td>172</td>
</tr>
<tr>
<td>FFT2</td>
<td>200</td>
<td>240</td>
<td>225</td>
</tr>
<tr>
<td>FFT3</td>
<td>1860</td>
<td>2220</td>
<td>2838</td>
</tr>
<tr>
<td>FFT4</td>
<td>1760</td>
<td>2964</td>
<td>3066</td>
</tr>
<tr>
<td>NEQ</td>
<td>1597</td>
<td>2082</td>
<td>2132</td>
</tr>
<tr>
<td>IRR</td>
<td>660</td>
<td>770</td>
<td>710</td>
</tr>
</tbody>
</table>

The chart in Figure 17 illustrates the relative scheduled times for each example. The method that produced the least efficient schedule is shown with a value of 100% and the other
methods times are divided by the largest value. The chart shows the relative speed as a percentage of the worst case.

3.3 Summary of results

The chart in Figure 17 indicates that partitioning based on clan decomposition yields better schedules than other methods.

The main heuristic of the LAST algorithm is that tasks are grouped based on their connectivity with previously allocated tasks. If a graph has pockets of highly interdependent tasks, this localized allocation strategy is very efficient. However, in some cases, such as a graph that is highly symmetrical, this strategy may not be the best. It does not always recognize the symmetrical sets of vertices that should execute in parallel.
The heuristic of the linear clustering approach is to group the vertices on the critical path. Initially, the critical path provides an upper limit on the cost of executing any cluster. If two linear clusters have costly communication between them but run on different processors, the resulting schedule suffers. Also, in many graphs, clusters that intercommunicate are not merged because they do not meet the strict dependency criteria explained in section 2.3. Even though the grouping is refined to compensate for high communication across clusters, it cannot consider all interaction between clusters at the same time. Therefore, a beneficial merging may not be recognized. This approach works better for irregular graphs with diverse path lengths and for graphs where the communication time is small relative to computation times.

Clan based partitioning produces good schedules for all types of graphs. The parse tree orders the tasks hierarchically, with fine grains at the bottom of the tree. As a result, when deciding whether to aggregate or execute in parallel, all tasks are considered in order from bottom to top in the parse tree. Potential parallelism is explored from the fine grain case to the large grain. Also, the organization of tasks into the clan hierarchy exposes any linear ordering of sets of tasks.

CLANS is the only one of the techniques that analyzes the entire graph before making a partitioning decision. Both LAST and LC have a more greedy approach, making decisions based on local information only. For LC the locality is a path through the graph
and for LAST it is a neighborhood of a vertex. As is typical of NP-complete problems, greedy choices fail for some input cases. CLANS' emphasis on describing the structure of the entire graph before making parallel/aggregation decisions makes it a more robust partitioning technique as well as one that scales well to a fixed number of processors. The test data strongly supports Clan Decomposition as the method of choice for the scheduling/partitioning problem.

4.0 CONCLUSION

It has been shown that task partitioning based on clan decomposition is better than the competing methods of linear clustering and the LAST algorithm. This partitioning technique will be combined with other tools to compile algorithms for parallel systems. Included in the set of tools are a system that analyzes program dependence to produce the program graph, models of computation and communication for existing distributed memory systems, a task partitioning method, and a system to generate code based on the task partitioning. All these tools together perform the steps necessary to transform a sequential algorithm into a working parallel version.
## SUM1 Schedules

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>10</td>
<td>11</td>
<td>9</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Legend

- **8**: Processing
  - Vertex 8

- **6**: Communication
  - Delay Waiting for 6

- **Idle**
REFERENCES


