A Communication-Ordered Task Graph Allocation Algorithm

John D. Evans
Robert R. Kessler

UUCS-92-026

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

April 21,1992

Abstract

The inherently asynchronous nature of the data flow computation model allows the exploitation of maximum parallelism in program execution. While this computational model holds great promise, several problems must be solved in order to achieve a high degree of program performance. The allocation and scheduling of programs on MIMD distributed memory parallel hardware, is necessary for the implementation of efficient parallel systems. Finding optimal solutions requires that maximum parallelism be achieved consistent with resource limits and minimizing communication costs, and has been proven to be in the class of NP-complete problems. This paper addresses the problem of static allocation of tasks to distributed memory MIMD systems where simultaneous computation and communication is a factor. This paper discusses similarities and differences between several recent heuristic allocation approaches and identifies common problems inherent in these approaches. This paper presents a new algorithm scheme and heuristics that resolves the identified problems and shows significant performance benefits.

A Communication-Ordered Task Graph Allocation Algorithm

John D. Evans and Robert R. Kessler

April 21,1992

Abstract

The inherently asynchronous nature of the data flow computation model allows the exploitation of maximum parallelism in program execution. While this computational model holds great promise, several problems must be solved in order to achieve a high degree of program performance. The allocation and scheduling of programs on MIMD distributed memory parallel hardware, is necessary for the implementation of efficient parallel systems. Finding optimal solutions requires that maximum parallelism be achieved consistent with resource limits and minimizing communication costs, and has been proven to be in the class of NP-complete problems. This paper addresses the problem of static allocation of tasks to distributed memory MIMD systems where simultaneous computation and communication is a factor. This paper discusses similarities and differences between several recent heuristic allocation approaches and identifies common problems inherent in these approaches. This paper presents a new algorithm scheme and heuristics that resolves the identified problems and shows significant performance benefits.

1 Introduction

A primary goal of program allocation and scheduling is to minimize program run time. The goals of program allocation vary, however, with the circumstances of hardware and software requirements. Memory size limits, restrictions on external I/O access and processor failure are several considerations that may take precedence over execution time. In addition, the existence of shared memory, distributed memory and the possible duplication of computations to eliminate communication, may significantly alter the nature of the allocation problem.

The inherently asynchronous nature of the data flow computation model offers promising improvement in performance over the control flow programming model. Data flow research has focused on developing high speed data flow multiprocessors, on recognizing the potential parallelism in sequential programs[12], achieving parallel program performance, developing parallel languages [1]. Recent research has also addressed the extension of data flow concepts to the execution of programs on non data flow architectures.

The data flow computation model describes computations as a collection of work units called **tasks**[10]. Individual tasks are sequential segments of program execution which can run from start to completion without synchronization. Tasks require input data to begin execution and the computational results of tasks are then used as the input to subsequent tasks. A task is eligible for execution when all of its inputs are ready. Precedence relationships are defined between pairs of tasks if the results from one task are

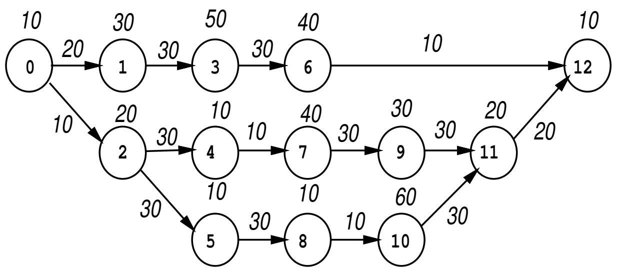

Figure 1: Data Flow Program Representation

required for the computation of the other. These precedence relationships define a partial ordering of the task set that allows the asynchronous parallel program execution.

Programs may be represented as directed weighted data flow graphs (see Figure 1). Graph nodes represent tasks which are weighted proportionally to the amount of computation done. Arcs define **data flow** between nodes and represent the requirement for the results of one node as input to another. The arcs define a precedence relationship between producer and consumer and are weighted proportionally to the amount of data transferred. Loops and nesting of subgraphs are possible within the data flow graph model as is recursive nesting. Data flow languages have been developed [1] to exploit the properties of data flow graphs. The data flow graph model can also be used to represent programs in traditional languages [12].

Several researchers have focused on the problem of generating directed acyclic graphs for parallel programs [28] and also on the problem of extending static graph models to mixed static/dynamic graphs[23]. These are both ongoing areas of research. For the purposes of studying program allocation, graphs are often translated into non-looping non-nested form as DAGs.

A graph representation may be used to characterize the parallel architecture. Graph nodes represent processors and are weighted according to processor performance characteristics. Connecting arcs represent communication pathways between processors and are weighted proportionally to the cost of communication between processors.

Several data flow architecture models have been developed. These include the Manchester Data Flow Machine [13], MIT's Tagged-Token Dataflow Machine [2] and the DDM-1 Computer[8]. Other architectures such as hypercube[16], HP Mayfly [9] and Banyan networks[15] are amenable to data flow program representations. Architectural characteristics common to many or all of these architectures include distributed memory systems and MIMD control. An additional characteristic of many of these architectures is the possibility of overlapping computation and communication. Separate hardware handles communication allowing a processing node to execute tasks and communicate simultaneously.

A number of research efforts have addressed the allocation of programs to distributed memory multiprocessor systems using the data flow program model for analysis. In the following sections this paper categorizes and discusses several related approaches to this problem, and identifies common problems inherent in these approaches. This paper presents a new algorithm scheme and heuristic cost procedure that resolves the identified problems and shows significant performance benefits. The remainder of the paper is organized into sections as follows: Section 2 defines the allocation problem. Section 3 categorizes related work. Section 4 gives working concept definitions. Section 5 describes specific related algorithms. Section 6 discusses performance of these algorithms. A new algorithm scheme is presented in Section 7. A new heuristic allocation procedure is presented in Section 8. Section 9 describes simulation tests and results. A complexity analysis is given in Section 10. Finally, Section 11 presents Conclusions.

2 Problem Definition

This paper focuses on the problem of minimizing program run time for programs allocated to distributed memory multiprocessors where significant inter-processor communication costs exist. The task graphs are assumed to be directed acyclic graphs and to be statically determined at allocation time. It is also assumed that parallelism between computation work and communication work is a factor in the multiprocessor system[9, 21, 16]. Architecture configuration and unit interprocessor communication costs are variables. Processor utilization time is broken down into three components:

Task computation time is the time spent on the execution of tasks.

Local communication time is the time spent by a processor locally to effect communication.

Non-local communication time is the time spent during communication in which a processor may carry out other activities.

Task computation time and local communication time are present irrespective of the allocation and together comprise task weight. Non-local communication time potentially causes delays but may overlap with local communication time or task computation time if there is work available for the processor to carry out. Non-local communication time is represented by arc weight. In the processor network configuration interprocessor communication delays vary depending on topology.

3 Categories of Related Work

Other program allocation research efforts utilize problem descriptions similar to that given above. Many of the related program allocation algorithms have distinguishing characteristics that can be generally classified as follows:

List schedule algorithms walk the task graph allocating tasks to processors to build lists of tasks to be executed by each processor [17, 24, 14]. This type of algorithm usually allocates the task graph in a single pass.

Task Clustering algorithms cluster tasks into groups or lists and then allocate the groups to processors [3, 25, 18]. This may be a multi-phase of single phase process.

Graph Folding algorithms use graph theoretic operations to transform the program task graph to match the processor network graph topology. Algorithms of this type generally rely upon uniformity of the input graph. Task level allocation algorithms [7] may assume a standard unit arc weight (and thus a standard data size) for communication between adjacent tasks. Alternately, a process graph may be used where processes are considered to contribute a processing load that is uniform over time [19, 4].

Search algorithms iteratively make modifications to an initial allocation attempting to make successive improvements. Included in this type are simulated annealing [26], steepest descent search [5], and branch and bound search [20] algorithms. These algorithms often have relatively higher complexity than the others due to the number of iterative improvements required.

The algorithm types considered for comparison here are of the task clustering and list schedule types. Graph folding algorithms are not considered because of the limitations placed on the graph model. Search algorithms are not considered because of their relatively higher complexity.

4 Operational Definitions

4.1 Graph Ordering

Several notions of graph ordering have been useful in program allocation research:

Topological ordering (See Figure 2) assigns a number to each task such that the descendents of a task always have higher (lower) number. There are usually many possible topological orderings for a particular graph.

Level ordering (see Figure 2) is similar to topological ordering. Level ordering assigns to each task the number that is equal to the number of tasks in the longest length precedent chain of tasks from the root to the task. This defines a partial ordering of the tasks since multiple tasks may cohabit the same level.

Precedent ordering (see Figure 3[1]) assigns a number to a task relative to the greatest completion time (or start time) of its precedent tasks for a particular program allocation. Tasks are numbered therefore in completion time (or start time) order. Precedent ordering satisfies the topological ordering requirements but also considers the processing time requirements of allocated tasks.

Critical path ordering (see Figure 4) assigns a number to a task that is equal to the longest exit path weight for that task. The exit path weight is the largest sum of precedence related task weights from the task being numbered to the end of the program.

[1]Task Weights are shown above the tasks. Interprocessor arc weights are also shown.

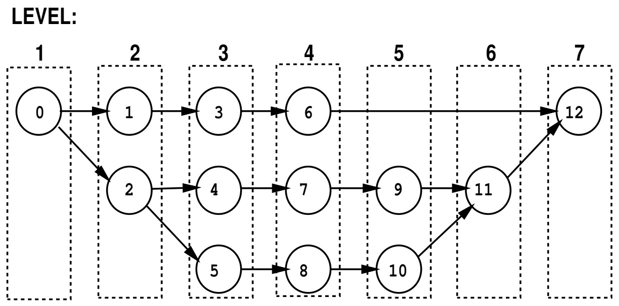

Figure 2: Topological Ordering (with Level Ordering)

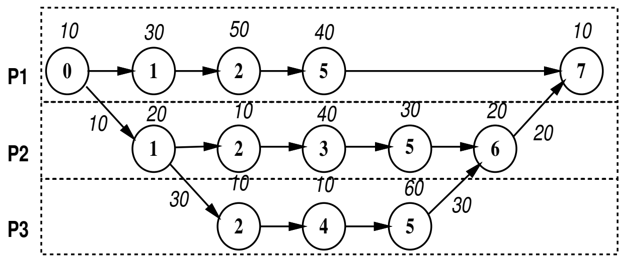

Figure 3: Precedent Ordering Showing Processor Grouping

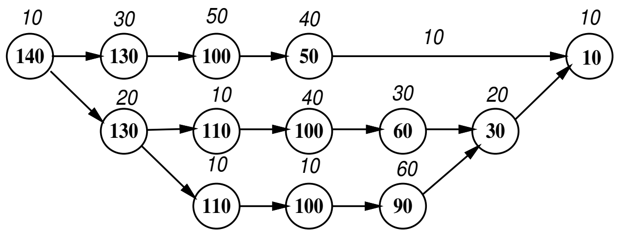

Figure 4: Critical Path Ordering

4.2 List Schedule Scheme

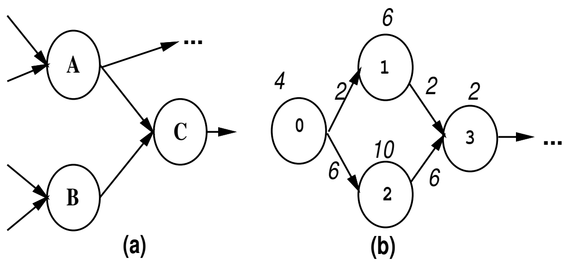

Figure 5: Data Flow Graph Fragments

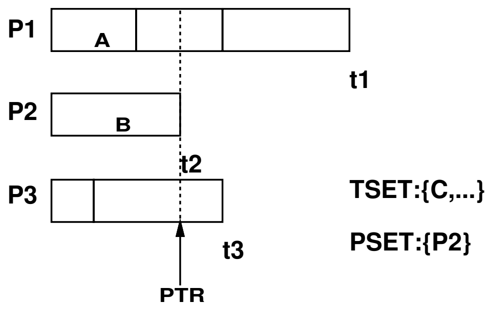

Figure 6: List Schedule Scheme

List schedule algorithm types build processor task schedules by iteratively adding tasks to processors. In operation, tasks are inserted into an eligible task set when their precedents are met and the processors become eligible for allocation when idle. Heuristic cost functions and/or graph ordering are then used to prioritize tasks and processors. The algorithms then match tasks to processors based on the determined priorities. In addition to the task graph and processor graph definition, the list schedule algorithms (and some clustering algorithms) maintain similar sets of dynamic information about the task graph and processor task sets during allocation. Figure 6 shows possible list schedule scheme information for the graph fragment in Figure 5(a). This information organizes the progression of the graph traversal. In some of the algorithms, one or more of these features may be degenerate. Conceptually this information includes:

CNT_i A counter of the number of allocated precedent tasks is maintained for each unallocated task. This is used to determine when a task becomes eligible for allocation.

$\mathrm{SCHEDULE}_p$ A sequence of tasks and their predicted completion times (T1,T2,T3 in Figure 6) for each processor. This is used to determine when precedents are satisfied for unallocated tasks and when allocated tasks will complete.

TSET The set of tasks eligible for allocation. Set representation varies for different algorithms.

PTR A pointer tracking completion (or starting) times of allocated tasks and processors. A Task is added to TSET when PTR has passed all its precedent tasks. PTR steps through the task set in completion time order. Alternately start time, topological position or level order may be used.

PSET The set of processors that eligible for allocation. Set representation varies.

The heuristic cost functions are localized in that they make use of local information about each task considered, such as exit path weight, task length, communication costs and precedent relationships. They also make use of information about the schedule of tasks already allocated to each processor, but do not consider the graph as a whole.

4.3 Task Clustering Scheme

Task clustering algorithms separate allocation of a task graph into two steps, clustering and assignment. Heuristic cost functions are used to cluster tasks into groups and to allocate the groups to processors. These steps may be combined into a single phase or left as separate phases. In either case the phase(s) iteratively process tasks, usually in a manner similar to the list schedule scheme. In addition, heuristic notions of graph ordering are used to evaluate assignment cost and to estimate execution time.

5 Related Algorithm Types

NAME	DATE	ORDERING
List Schedule Algorithms		
GREEDY	69	PRECEDENT
CRITICAL PATH	76	PRECEDENT
GREEDY CPM	83,(90)	PRECEDENT
CAMPBELL ALG.	85	TOPOLOGICAL
WP ALGORITHM	87	LEVEL ORDER
RAVI ALGORITHM	87	PRECEDENT
Clustering Algorithms		
SARKAR ALG.	86,89	TOPOLOGICAL
LAST ALGORITHM	89	UNSPECIFIED
VERTICAL ALG.	91	LEVEL ORDER

Figure 7: Related Algorithms

Figure 7 lists several specific algorithm types relevant to the work reported here. These algorithm types fall into the classes of **list schedule** and **task clustering** algorithms. These types are considered primarily because they have been shown to produce reasonable results with minimum computational complexity and with minimal restrictions on graph and architecture properties. These algorithm types are further related in that they use the list schedule scheme and/or that they each employ one or more of the graph ordering concepts described in Section 4.1. Graph ordering concepts are used in these algorithms to order the sequence of execution of tasks allocated to a processor ($SCHEDULE_p$) and/or to sequence the allocation

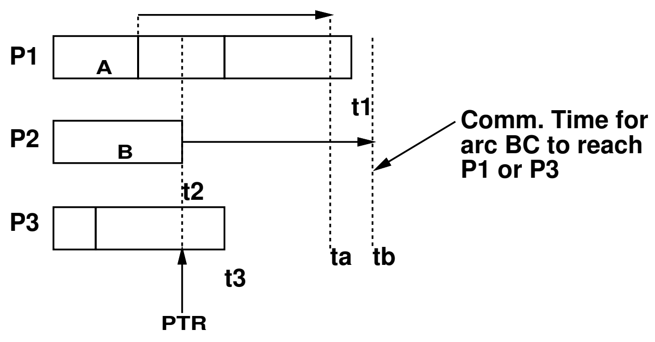

Figure 8: Scheme Showing Communication Times

of tasks. The complexity of these algorithms is discussed in Section 10. Specific algorithms types are discussed in more detail below.

1. Greedy Algorithm: A list scheduling algorithm in which tasks are added to the eligible task set when all parents are allocated which orders the allocation in **precedent order**. The eligible task set (TSET) is a simple FIFO queue, while the set of eligible processors (PSET) is the set of all processors. The heuristic allocation function here is to minimize communication costs. The algorithm assigns the next eligible task to the processor that can schedule it the earliest. Figure 8 augments Figure 6 with communication times for a uniform communication cost network. From the figure, P = Processor with $min(max(t_1, t_b), max(t_2, t_a), max(t_3, t_a, t_b)))$.

2. Critical Path Method [17]: The CPM list scheduling algorithm has been used as a basis of comparison for distributed memory allocation algorithms [18, 24]. In this algorithm the processor set (PSET) is a FIFO queue while the ready task set (TSET) is an exit path length ordered priority queue. Tasks are added to the eligible task set when all parents are allocated (**precedent order**). Communication cost is not considered.

3. Greedy Critical Path Method: This list schedule algorithm type was proposed by Ho and Irani [14] in 1983. It is a combination of the Greedy and CPM algorithms[2]. Tasks are added to the eligible task set (TSET) when all parents are allocated (**precedent order**). TSET is a critical path ordered priority queue. The heuristic assigns the highest priority available task to the processor that can run it the earliest (similar to Greedy above).

4. WP algorithm [22]: A list schedule algorithm that traverses the graph in **level order** (TSET is level ordered). PSET includes all processors. Several local heuristics (including CPM and task weight priority) are used to prioritize the tasks within levels. Barrier synchronization forces each task to execute in parallel with its level. Uniform communication time is assumed.

[2]A similar algorthim using the HLFET heuristic was proposed by Sih and Lee[27] in 1990.

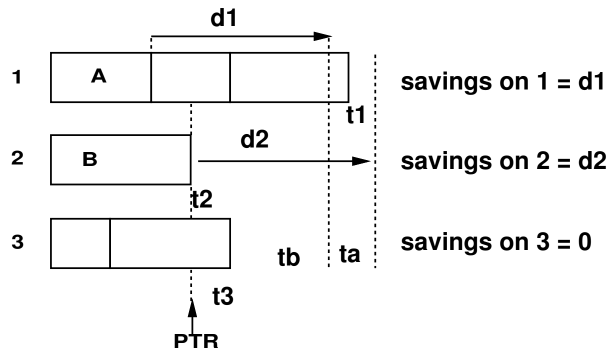

Figure 9: Communication Savings for Task C

5. Ravi Algorithm: This list scheduling algorithm type was investigated by Ravi, Ercogovac, Land and Muntz [24]. It also uses exit path length priorities making the task set a priority queue. Tasks are added to TSET in **precedent order** (when all parent tasks have been allocated). PSET is a stack queue causing the most recently idled processor to be the processor selected for allocation. The heuristics select the task from a range of high priority tasks. The task selected is the one which gives the maximum savings in communication time for the chosen processor. If a task is allocated to the same processor as one of its precedent tasks, then the communication cost between them is saved. Figure 9 shows the communication savings for task C from Figure 8 relative to its precedent tasks A and B.

6. Campbell Algorithm [6]: A list scheduling algorithm that traverses the graph allocating tasks in **topological order** (topological order TSET). Tasks are assigned to processors dependent on two heuristic cost functions. A communication cost function attempts to group tasks together that communicate, while a parallelism maximizing heuristic attempts to separate tasks that are not in each other's transitive closure. PSET includes all processors.

7. Sarkar Algorithm: Sarkar and Hennessy [25] proposed a two-phase clustering algorithm. The first phase groups tasks into clusters while the second phase allocates clusters to processors. The tasks within clusters (in phase one) and on processors (in phase two) are assumed to execute in **topological order**. In phase one the algorithm iterates simulating the graph run time repeatedly to find the best pair of clusters to merge. In phase two the algorithm iterates in a similar manner merging clusters onto processors.

8. Vertical Algorithm: A two-phase clustering algorithm[18]. Phase one repeatedly determines longest execution path among unallocated nodes and assigns that node sequence to a processor. Phase two determines the critical path (including inter-processor communication) for the allocation and tries to optimize it. Optimization is done by shifting tasks between processors to reduce communication costs along the critical path. The optimization phase is repeated until no further improvement can

be made. This algorithm assigns **level ordering** of execution for tasks assigned to each processor.

9. LAST Algorithm: A single phase clustering algorithm [3]. A Task is allocated singly if a sufficient percentage of the adjacent tasks are allocated. If not then the task is grouped with other adjacent unallocated tasks and the group is allocated as a unit. Tasks and clusters are assigned to a processor based on relative communication cost and processor completion time.

6 Discussion of Previous Work

Each of the algorithms studied makes use of one or more of the ordering concepts discussed except the LAST algorithm which leaves the ordering unspecified. These techniques attempt to characterize probable run time program behavior, in order to structure and sequence the allocation algorithms and/or the allocated program. Precedent ordering and topological ordering define the sequence of graph traversal in algorithms 1, 2, 3, 5 and 6. Precedent ordering and topological ordering also specify order of execution of allocated tasks in algorithms 1, 6 and 7. Level ordering is used to group tasks into horizontal layers that can execute in parallel in algorithms 4 and 8. Each of these techniques may introduce unnecessary run time delays into the task allocations generated. Of these, the delays caused by precedent ordering need special consideration because they also arise in the other two ordering strategies.

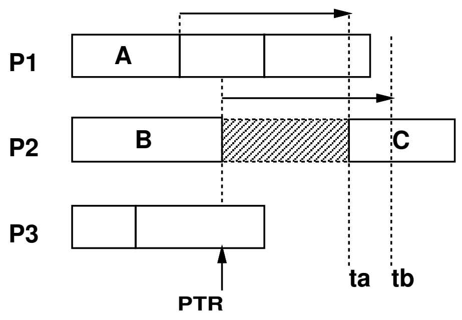

Figure 10: Precedent Order Allocation State

6.1 Precedent Ordering

The position of tasks in precedent ordering reflects an accurate estimation of program execution, and accounts for both communication costs and execution times for each allocated task. Certain execution delays may be attributed to precedent ordering.

As an example, Figures 5(a) and 8 depict a graph fragment and precedent order allocation state. At this point in the allocation, task C is added to the eligible task set because its parents have both terminated. If task C is allocated next, the state shown in Figure 10 results. A significant dead space has been introduced into the schedule for processor P2 because of communication delay due to arc AC. Several objections to this allocation arise:

1. Tasks with lower priorities might have been allocated to processor 2 to fill in the dead space without delaying task C.
2. Even if task C is delayed by a lower priority task the delay may not be as significant as the efficiency gain (particularly in systems with high communication cost).
3. Advancing the allocation pointer to further task completions on processors 1 and 3, may add tasks of even higher priority to the eligible task set. If these tasks could be assigned to processor 2 to execute during the dead space or the time interval of task C, then task C is executed in preference to a higher priority task. In this case a violation of the allocation heuristics arises as a result of precedent ordering.

6.2 Topological Ordering

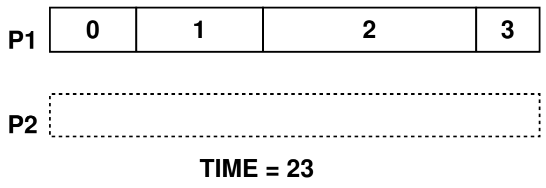

Figure 11: Topological Allocation Order

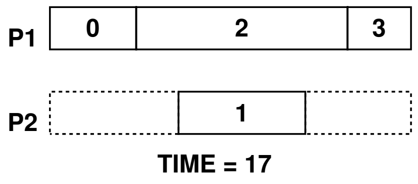

Figure 12: Non-Topological Allocation Order

Topological ordering sorts tasks based on precedence, but does not consider communication delays. Because of this it may introduce the same allocation delays caused by precedent order allocation. Because topological ordering does not consider the execution length of tasks, it may also introduce sequencing delays when the lengths of all tasks are not equal, or when work cannot be evenly divided among the available processors.

The graph fragment shown in Figure 5(b) is used as an example of execution delay caused by a topological allocation order. When the graph is allocated to two processors, task 1 is allocated before task 2 and is placed with task 0 (see Figure 11) to minimize communication cost. If the topological allocation ordering is not used and both task 1 and 2 are considered eligible, then critical path priorities or minimization of communication cost would result in the optimal allocation shown in Figure 12.

The graph shown in Figure 1 is used as an example of delays introduced by requiring an particular topological ordering of execution. Figure 13 shows an optimal allocation of the graph on two processors. This allocation does not preserve this topological ordering of execution of tasks because task 6 is executed

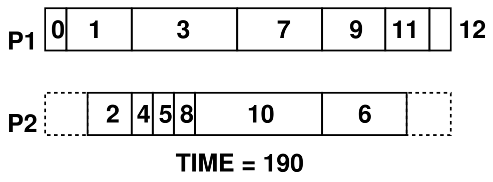

Figure 13: Optimal Allocation (true optimal)

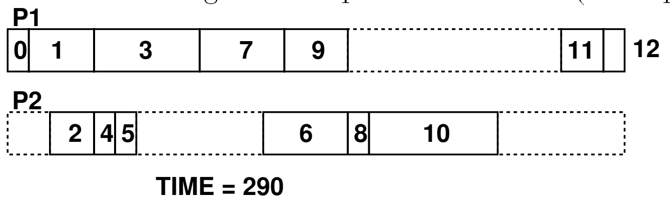

Figure 14: Optimal Allocation (forcing topological execution order)

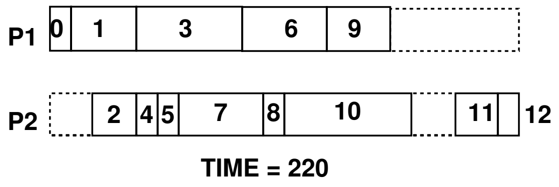

Figure 15: Optimal Allocation Given Topological Execution Order

after task 10. Forcing this topological ordering of execution on this task assignment extends the execution by more than 50 percent (see Figure 14). Furthermore, exhaustive search reveals that no allocation exists that preserves this topological ordering of execution and has run time within 15 percent of optimal run time (see Figure 15).

6.3 Level Ordering

Level ordering is intended to relate topological position of tasks to their potential for parallel execution. The entrance and exit path lengths, however, have a strong impact on the actual time that tasks are executed and may not correspond well to topological level. In addition, since inter-processor communication time is not considered, timing delays (as discussed in section 6.1) may occur. Interval graphs [11] connect pairs of nodes that coexist over some interval and serve to illustrate the potential inaccuracy of level ordering. The interval graph in Figure 16 corresponds to the level ordering shown in Figure 2. The optimal allocation is shown in Figure 13 and the corresponding interval graph in Figure 17. Path lengths in the graph allow tasks to shift levels easily, and optimal allocation requires this. The result is a low correlation between the level ordering and optimal allocation. Only 3 of the 8 arcs in the level order interval graph match the optimal interval graph, and 5 arcs from the optimal interval graph are omitted.

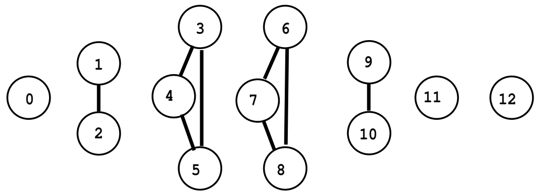

Figure 16: Level Order Interval Graph

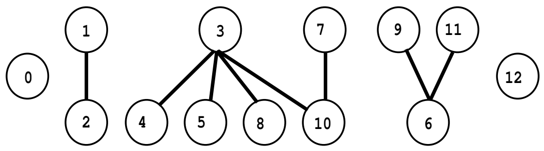

Figure 17: Optimal Allocation Interval Graph

7 Communication-Ordered Scheme

The previous algorithms generate allocation delays because they poorly estimate execution time state relationships. The **communication-ordered** scheme extends the list scheduling scheme to overcome the liabilities of the ordering techniques used in previous algorithms. This scheme adopts a discrete event simulation model where task execution times and communication delays are modeled as events. The sequence of allocation is determined by simulated time, and a task becomes eligible for allocation on a processor when simulated time reaches the point when all precedent tasks have completed, all required data could arrive at that processor and task execution could actually begin. In this scheme, tasks become eligible for allocation at different simulated times on different processors due to variations in communication delays across the processor network. We call this allocation ordering concept **communication time ordering**.

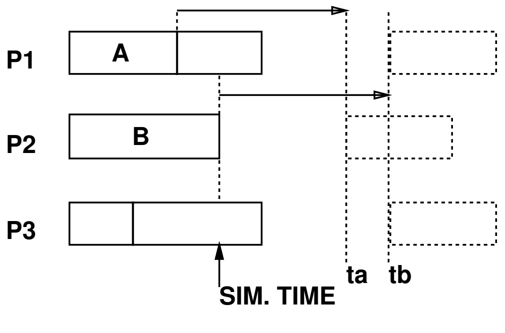

Figure 18: Communication-Ordered State

Figure 18 illustrates communication time ordering for the graph fragment shown in Figure 5(a). In simulated time, tasks A and B have completed. Earliest potential execution times for task C are shown as dotted rectangles, and task C will not be considered eligible until simulated time reaches those points. At this point only processor P2 is considered for scheduling. The other processors are active, and only tasks that can begin immediately are eligible for scheduling. This allocation policy does not introduce the dead space delays present in the other ordering strategies. Communication time ordering also maintains a more accurate view of program execution that minimizes imbalance between processor schedules. In comparison, the precedent order allocation in Figure 6 shows processor P1 to be two task completions ahead of processor P2. This state is not possible in communication-ordering because a task is not allocated until simulated time reaches the point where it may actually begin.

```
INPUT: Processor graph, task graph (annotated by exit path lengths)
FUNCTION: Determine the sequence of tasks to be allocated to each processor.

Initially:
    Clk = 0
    CQ = empty
    TQ = empty
    Pset = all processors
    $TS_i$ = empty for all processors.
    The root task is inserted into $TS_0$
Loop till Clk == $\infty$:
  1. Process task completion events for tasks $T_j$ in TQ with time == Clk:
       Add processor to Pset.
       For all immediate descendent tasks $T_i$: decrement counter and if counter == 0 then:
           Compute the start time list $ST_i$ for $T_i$
           Remove the earliest start time and schedule a communication event in CQ for this task.
  2. Process communication events for each task $T_i$ in CQ with time == Clk (three cases are possible):
       (a) $T_i$ has already been allocated: discard event.
       (b) This is the last entry in time list $ST_i$:
           remove $T_i$ from all processor task sets
           insert $T_i$ in global task set.
       (c) This is not the last time list entry in $ST_i$:
           insert $T_i$ in task set for the appropriate processor(s)
           remove the next start time entry from $ST_i$ and schedule a new event in CQ.
  3. Schedule tasks on processors:
       Compare global task set and local task set for each idle processor to determine processor/task pairs based on heuristics
           (see Figure 20). For each task to be allocated:
           Remove processor from Pset.
           Store start time and processor number.
           Insert task completion event in TQ.
           Delete task from all processor task sets if present.
           Delete task from global task set if present.
  4. Clk = min( $\infty$, next TQ time if any, next CQ time if any).
```

Figure 19: Communication-Ordered Scheme Algorithm

An outline of the communication-ordered allocation scheme is shown in Figure 19. The algorithm requires the task graph and processor graph definitions and in addition, the following information set is maintained:

Clk : Simulation time clock.

CQ : Communication event queue.

TQ : Task completion queue.

Pset : The set of eligible processors.

Ctr$_\mathbf{i}$: For each unallocated task: A counter of the number of its precedent tasks that have been allocated.

For each allocated task the task processor number and start time is stored.

ST$_\mathbf{i}$: For each unallocated task whose parents have completed there is a sorted list of start times. These are the times when the task could be started on the processors in the system.

GS : Global task set contains tasks eligible for execution on all processors (global tasks).

TS$_\mathbf{i}$: The set of eligible tasks for each processor i that are not globally eligible (local tasks).

8 Heuristic Allocation Procedure

The heuristic allocation procedure used in this algorithm is tailored to the communication-ordered scheme. The cost function uses greedy heuristics that attempt to minimize exit path length and reduce communication delays and overall communication cost. Exit path lengths are computed for all tasks prior to allocation. The communication-ordered scheme partially eliminates the need for heuristic cost functions to minimize processor delays since tasks are only considered for allocation when they may be run immediately. There is a distinction that can be made, however, between tasks in the local task sets (local tasks) and those in the global task set (global tasks). Global tasks are eligible for scheduling on all processors, while local tasks are only eligible on a subset of the processors. A distinction may also be made between processors that have tasks in their local task sets (locally runnable) and those that do not (idle). Scheduling global tasks on locally runnable processors may starve idle processors. This is true even when there are no immediate idle processors, since there may be at the next scheduling iteration. Scheduling local tasks on runnable processors reduces communication delays and the overall load on the communication network. The desire to keep as many processors busy as possible and to reduce communication costs must be traded against path length priorities. In priority order, the following heuristics are used:

1. Schedule global tasks on idle processors.

2. Prioritize global tasks by exit path length.

3. Schedule local tasks on runnable processors unless the exit path length of global tasks exceeds the exit path length plus the saved communication delay of any local task.

4. Prioritize local tasks by exit path length plus saved communication delay.

5. Give preference to processors of parent tasks for scheduling global tasks.

```
INPUT: IQ,RQ,GS,$TS_i$ for each processor.
FUNCTION: Allocate eligible tasks to processors.
OUTPUT: Return list of allocated tasks/processor pairs
  1. While IQ is not empty and GS is not empty:
       T = top(GS) (the highest priority global task)
       Remove T from GS.
       If a processor that ran a parent of T is in IQ then:
             P = parent processor
             else P = longest idle processor in IQ
       Remove P from IQ
       Allocate T to P
  2. While RQ is not empty:
       P = top(RQ)
       Remove P from RQ
       T1 = top($TS_P$)
       T2 = top(GS)
       If (priority(T1) + Saved Time) > priority(T2) then
            T = T1
       else T = T2
       Remove T from all appropriate queues.
       Allocate T to P
  3. Return list of allocated task/processor pairs (T,P)
```

Figure 20: Heuristic Allocation Procedure

An outline of the heuristic allocation procedure is shown in Figure 20. The data used by the heuristic allocation procedure includes the following:

GS : Global task priority queue (exit path length priorities).

$\mathbf{TS_i}$: The priority queue of eligible tasks for each processor i (exit path length priorities).

IQ : The idle queue of processors with no local tasks. IQ is ordered with longest idle processor first.

RQ : The runnable queue of processors with local tasks. RQ is ordered with processor with lowest top priority local task first. This ordering is useful when comparing local tasks to global tasks to ensure that the highest priority set of eligible tasks is selected for runnable processors.

T,P,T1,T2 : local variables.

The task sets GS and TS_i are also used in the scheme description. The processors queues IQ and RQ together comprise the processor set Pset from the scheme algorithm.

9 Simulation Tests

Simulation testing was used to evaluate the performance of the allocation algorithm presented. As a basis of comparison, several allocation algorithms were implemented based on algorithm types discussed in Section 5. The algorithms were implemented to operate on identical processor and task graph models as described in Section 2. The algorithm types implemented derived from the Greedy, Critical Path, Greedy Critical Path, Ravi, Vertical and LAST [3] algorithms. The graphs were allocated for mesh processor systems containing from 2 to 12 processors, and with average **cost ratio** (communication cost/computation cost) varying from 0 to 20. Three sets of allocation tests were conducted and are described in the following subsections. Figures 21 to 26 display results of the tests. The legend in Figure 22 is used in all comparison figures.

9.1 Small Graph Tests

The first test set used a group of small graphs (under 100 nodes) with very limited parallelism ((total work / critical path length) < 3.5). In allocation tests using the small graph set, the clustering algorithms underperformed the list schedule algorithms at low communication costs (cost ratio below 0.8). Among the list schedule algorithms reasonable speedup was obtained at low communication cost. At moderate to high communication costs (cost ratio above 2.0) the performance of all of the algorithms were similarly low. In this range average speedup was consistently below 2.0 (below 1.5 for cost ratio above 5.0) and was frequently nonexistent or negative. Speedups in this range tended to fluctuate up and down with increases in the number of processors. Low parallelism graphs seem to be equally suited then to any of the list schedule algorithms at low communication cost ratios, but no consistent and significant parallel improvement is apparent for high communication costs.

9.2 Large Graph Tests

A set of 25 randomly generated graphs with higher potential parallelism was used. These graphs ranged in size from 76 nodes to 2973 nodes. The task weights were assigned randomly from a uniform distribution. Communication cost weights were assigned by two methods: (a) Cost ratio x Task Weight and (b) Cost ratio x Uniform Random Variable. While different performance results were achieved using these two methods, very little difference in the relative merits of the allocation algorithms was observed.

[3]Values for the LAST type algorithm are approximate and were extrapolated from comparison data [3], where it was compared against the Greedy algorithm.

In tests using the large graph set, the relative performance of the comparison algorithms was consistent with earlier reported results [17, 14, 22, 24, 18, 3]. In comparison the communication-ordered allocation algorithm showed significantly superior response to increases in number of processors and increases in communication cost ratio. It generated consistently superior allocations when the cost ratio exceeded 0.6.

9.2.1 Algorithm Response

Figure 21 show performance response with increasing communication / computation cost ratio and increasing number of processors. Since speedup limits are graph specific, the figures shown are for a representative task graph. A baseline curve in bold indicates the speedup achieved with a CPM scheduler for a zero communication cost multiprocessor. Other curves are for the comparison algorithms when the indicated cost ratio is present. Consistent increases in speedup (above 9.0) are obtainable for the task graph on a shared memory multiprocessor. When communication cost ratio is below 0.5 (not shown) and when the available parallelism exceeds the available processors (i.e. less than 5 processors in Figure 21(a)), the list schedule algorithms performance is similar to the communication-ordered algorithm. When significant communication cost is a factor, run times degrade for all of the algorithms.

Characteristically, the communication-ordered allocator outperformed the range of comparison algorithms showing less performance degradation with increasing cost ratios. Also, characteristically, the communication-ordered allocator showed superior response to increased number of processors, maintaining speedups close to shared memory performance until reaching a maximum level and flattening sharply beyond the breaking point. In comparison, the other list schedule algorithms showed more gradual and uneven increases in speedup (and occasionally decline).

9.2.2 Overall Comparison

Figures 23 and 24 show the performance of the comparison algorithms relative to the communication-ordered allocator. In Figure 24 communication cost / computation time cost ratios in column 1 vary from 0 to 20. The figures indicate average relative performance at the breaking point of the communication-ordered allocator for all graphs at each communication cost ratio. The entries in Figure 24 are the average ratio of the run time for the comparison algorithm divided by the run time for the communication-ordered allocator. For example, a value of 2.0 indicates that the comparison algorithm produced an allocation that required 2.0 times the communication-ordered allocation run time. The "Best" column was computed by taking the best allocation achieved by any of the comparison algorithms for each test case and comparing that against the communication-ordered allocator. Figure 23 is a logarithmic plot of the data in Figure 24 omitting the BEST column.

At low cost ratios (below 0.6), the list schedule algorithms performed nearly as well as the communication-ordered allocator. At cost ratio 1.0 and above, however, the communication-ordered allocator consistently outperformed the comparison allocators individually and as a group. The relative performance of the list schedule algorithms continued to degrade with increasing communication cost. None of the algorithms

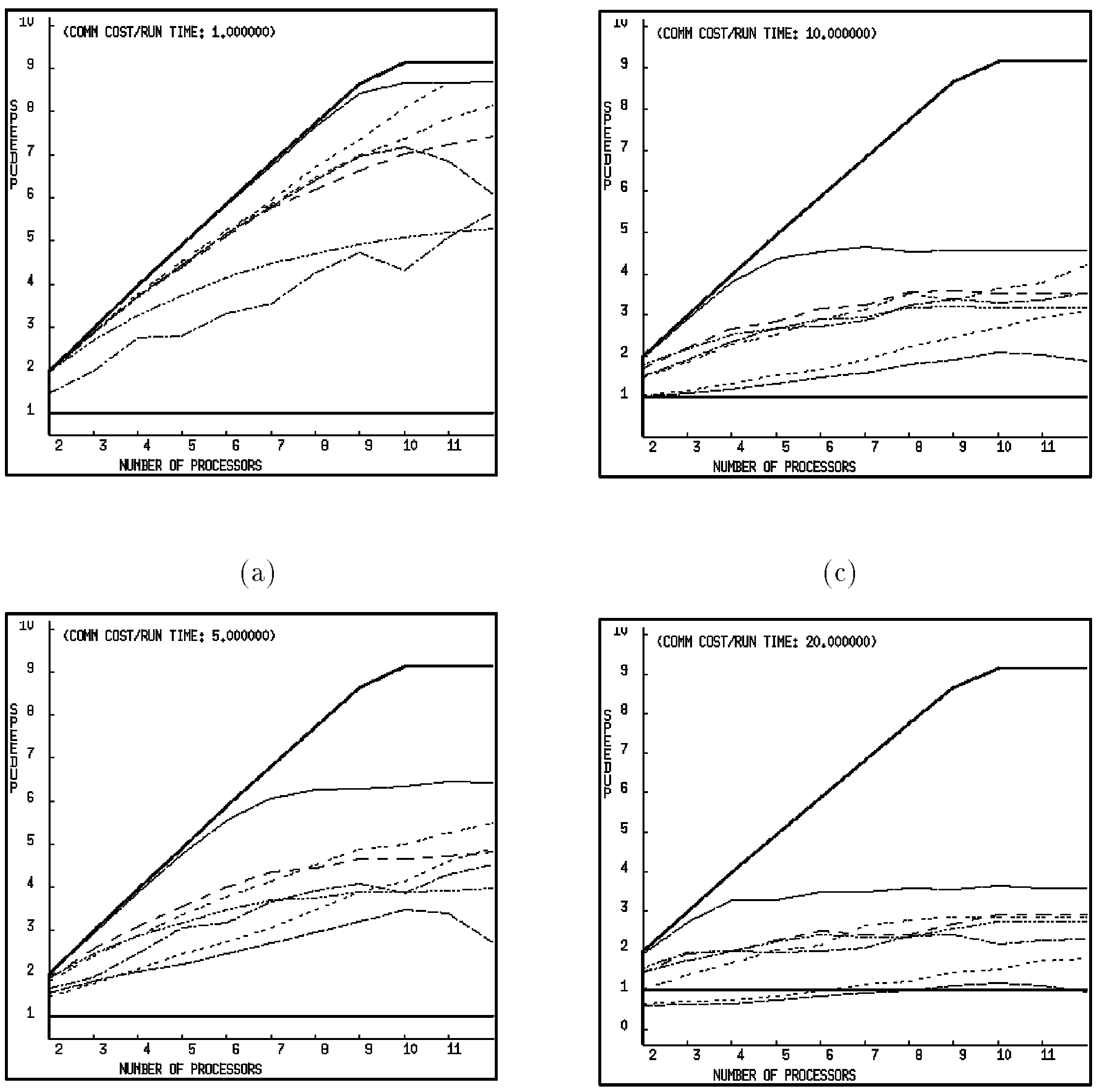

Figure 21: Speedup Graphs

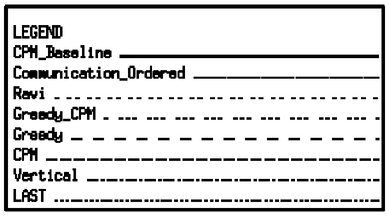

Figure 22: LEGEND

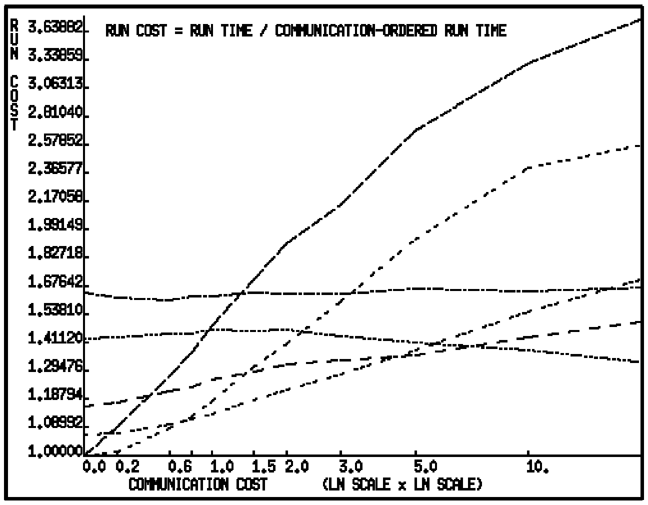

Figure 23: Overall Comparison

showed significant speedup with communication cost ratios above 20.0 for the task graphs used. The relative performance of the clustering algorithms is nearly level and the LAST algorithm shows gradual relative improvement, however, the potential speedup is exhausted long before it converges with the communication-ordered allocator.

C/T	RAVI	GCPM	GDY	CPM	VERT	LAST	BEST
0.0	1.002	1.068	1.163	0.999	1.644	1.431	0.999
0.2	1.013	1.072	1.175	1.091	1.619	1.436	1.012
0.6	1.087	1.102	1.216	1.279	1.609	1.452	1.068
0.8	1.127	1.119	1.233	1.372	1.629	1.455	1.091
1.0	1.182	1.136	1.261	1.485	1.624	1.469	1.118
1.5	1.312	1.186	1.294	1.710	1.644	1.465	1.180
2.0	1.408	1.224	1.320	1.908	1.639	1.470	1.221
3.0	1.605	1.284	1.339	2.148	1.640	1.444	1.275
5.0	1.932	1.378	1.361	2.701	1.667	1.412	1.335
10.0	2.410	1.552	1.435	3.301	1.653	1.379	1.360
20.0	2.597	1.727	1.511	3.813	1.674	1.330	1.329

Figure 24: Overall Comparison Results

9.3 Scheme Replacement Tests

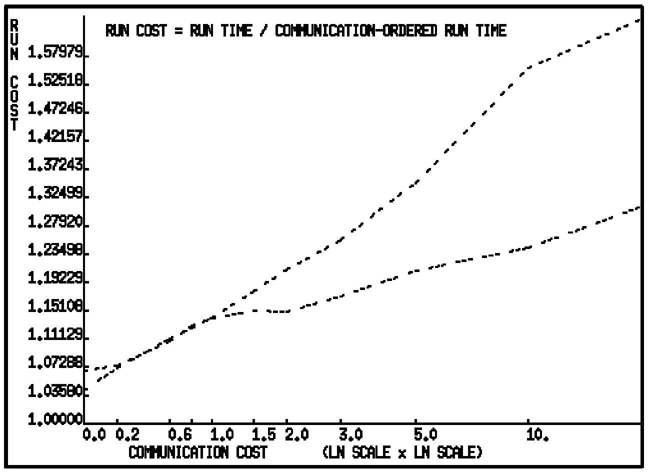

Figure 25: Scheme Replacement

In the scheme replacement tests the performance of the precedent ordered scheme of the Greedy and Greedy CPM algorithms is compared to the performance of the communication-ordered scheme using similar heuristic cost functions. Two modified algorithms are produced: Communication-ordered Greedy and Communication-ordered Greedy CPM. These algorithms use the heuristics of the base algorithms with the communication-ordered scheme. In the Communication-ordered Greedy algorithm, tasks are placed in a local FIFO queue when eligible on a processor, while in the Communication-ordered Greedy CPM algorithm, the tasks are placed in local priority queues. A global task queue is not used. Tasks are selected by taking the top task off of the local task queue for a processor. Figures 26 and 25 show the comparison of the Greedy CPM and Communication-Ordered Greedy CPM algorithms and the comparison of the Greedy and Communication-Ordered Greedy algorithms. Data was obtained as in Figures 24 and 23. The

C/T	GREEDY CPM	GREEDY
0.0	1.066	1.0431
0.2	1.072	1.0690
0.6	1.104	1.1109
0.8	1.126	1.1241
1.0	1.136	1.1360
1.5	1.174	1.1483
2.0	1.207	1.1466
3.0	1.251	1.1673
5.0	1.333	1.2079
10.0	1.542	1.2306
20.0	1.647	1.3020

Figure 26: Scheme Replacement Results

figures indicate average improvement above 10 percent for cost ratios above 0.2, and as high as 64 percent. These findings indicate that a significant improvement may be seen when using the communication-ordered scheme rather than the more typical precedent ordered scheme.

10 Complexity Analysis

The size of a precedence graph is bounded by the number of edges (e) in the graph rather than the number of nodes (n). Any scheduling algorithm that preserves precedence relations must have at least complexity $O(e)$. The size of a processor graph is p^2 since communication between processor pairs is considered even though it may be indirect.

The type of set structures used for processor and task sets varies with the heuristic allocation functions. Priority queue set operations require $O(\log(n))$ time while FIFO queues operations require O(1). The event and communication queues, however, are priority queues.

Step 1 of the outer loop in Figure 19 processes task completions. Computing start times during the duration of the algorithm requires examining the cost of all arcs for all processors in the worst case and requires $O(ep)$ time.

The worst case number of communication events is pn and each may require $\log(n)$ set access time. Step 2 processes communication events and requires $O(\log(n)pn)$ time.

Each loop iteration in the allocation functions allocate a task and require queue access. Step 3 is bounded though by the potential cost of deleting all tasks from all processor queues $O(\log(n)pn)$.

The outer loop is executed at most once for each task completion $O(n)$, and message $O(pn)$ and has $O(\log(n))$ complexity in Step 4. The result is $O(\log(n)pn)$. Worst case cost then is $O(ep + \log(n)pn)$.

ALGORITHM	COMPLEXITY
COMMUNICATION-ORDERED	$O(ep + pn\log(n))$
GREEDY	$O(ep)$
CRITICAL PATH	$O(e + n\log(n))$
GREEDY CPM	$O(ep + n\log(n))$
CAMPBELL	$O(n^3)$
WP	$O(e + n\log(n))$
RAVI	$O(ne\log(p))$
SARKAR	$O(n^2 e)$
LAST	$> O(n^3)$
VERTICAL	$O(n^3)$

Figure 27: Complexity Comparison

Figure 27 shows complexities of the algorithm types discussed[4]. The figure shows that the communication-ordered scheme is among the lower complexity algorithms. It significantly outperforms the test algorithms of lower complexity. Studies have shown that e is generally $O(n)$ with high probability so the expected performance bound is $O(pn\log(n))$.

[4]Complexity figures for GREEDY, CRITICAL PATH, GREEDY CPM, WP, RAVI and LAST were derived by the authors. The remainder were previously reported [6, 25, 18].

11 Conclusions

The problem of optimizing program run time for programs allocated to distributed memory multiprocessors where significant inter-processor communication delays exist has been considered. Classes of related suboptimal heuristic algorithms have been discussed. Particular attention has been given to two low cost and effective allocation algorithm classes, list schedule and task clustering algorithms. Specific algorithm types within these classes have been considered and common aspects of their schemes and ordering concepts identified and discussed. A new **communication-ordered** scheme based on discrete event simulation has been presented that avoids certain problems that occur in the previous strategies. A new heuristic cost procedure has been presented that is tailored to the **communication-ordered** scheme. Several sets of simulation tests have been conducted. A comparison of the performance of the new algorithm to other algorithm types has shown significant improvement in allocated graph execution time. The most dramatic improvement is shown in graphs that have significant parallel potential and when moderate to high communication costs are present. Comparison tests also indicate that some existing algorithms can be modified to use the new scheme and show significant improvement in so doing.

12 Acknowledgments

We would like to express appreciation for Hewlett-Packard Company for financially supporting this research. We are also grateful to Alan Davis for the insight, constructive criticism and moral support he contributed to this work.

References

[1] W. B. Ackerman, "Data Flow Languages," *Computer*, pp. 15–25, February 1982.

[2] Arvind, and D. E. Culler, "Dataflow Architectures," *Ann. Rev. Computer Science*, vol. 1, pp. 225–253, 1986.

[3] J. Baxter, and J. H. Patel, "The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm," In *Proceedings of the 1989 International Conference on Parallel Processing*, The Pennsylvania State University Press, pp. II–217 – II–222, 1989.

[4] F. Berman, and L. Snyder, "On Mapping Parallel Algorithms Into Parallel Architectures," In *Proceedings of the 1984 International Conference on Parallel Processing*, The Pennsylvania State University Press, pp. 307–309, 1984.

[5] S. H. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp. 207–214, March 1981.

[6] M. L. Campbell, "Static Allocation for a Data Flow Multiprocessor," In *Proceedings of the 1985 International Conference on Parallel Processing*, The Pennsylvania State University Press, pp. 511–517, 1985.

[7] M. J. Chamberlin, and A. L. Davis, "A Static Resource Allocation Methodology for a Dataflow Multiprocessor," Schlumberger Palo Alto Research Center.

[8] A. L. Davis, "The Architecture and System Methodology of DDM1: A Recursivly Structured Data Driven Machine," In *Proceedings of the 5th Annual Symposium on Computer Architecture*, pp. 210–215, 1978.

[9] A. L. Davis, "Mayfly: A General-Purpose, Scalable, Parallel Processing Architecture," *Lisp and Symbolic Computation Journal*, 1992 (to appear).

[10] A. L. Davis, and R. M. Keller, "Data Flow Program Graphs," *Computer*, pp. 26–41, February 1982.

[11] P. C. Fishburn, *Interval Orders and Interval Graphs*, Wiley-Interscience Publication, 1985.

[12] D. D. Gajski, D. A. Padua, and D. J. Kuck, "A Second Opinion on Data Flow Machines and Languages," *Computer*, pp. 58–69, February 1982.

[13] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester Prototype Data-flow Computer," *Communications of the ACM*, vol. 28, pp. 34–52, January 1985.

[14] L. Y. Ho, and K. B. Irani, "An Algorithm For Processor Allocation in a Dataflow Multiprocessing Environment," In *Proceedings of the 1983 International Conference on Parallel Processing*, The Pennsylvania State University Press, pp. 338–340, 1983.

[15] C. E. Houstis, and M. Aboelaze, "A Comparative Performance Analysis of Mapping Applications to Parallel Multiprocessor Systems: A Case Study," *Journal of Parallel and Distributed Computing*, vol. 13, pp. 17–29, 1991.

[16] e. a. J. P. Hayes, "A microprocessor-based hypercube supercomputer," *IEEE Micro*, vol. 6, no. 10, pp. 6–17, October 1986, Ncube/ten description.

[17] M. T. Kaufman, "An Almost-Optimal Algorithm for the Assembly Line Scheduling Problem," *IEEE Transactions on Computers*, vol. C-23, no. 11, pp. 1169–1174, November 1974.

[18] B. Lee, A. R. Hurson, and T. Y. Feng, "A Vertically Layered Allocation Scheme for Data Flow Systems," *Journal of Parallel and Distributed Computing*, vol. 11, no. 4, pp. 175–187, November 1991.

[19] V. M. Lo, "Algorithms for Static Task Assignment and Symmetric Contraction in distributed Computing Systems," In *Proceedings of the 1988 International Conference on Parallel Processing*, The Pennsylvania State University Press, pp. 239–244, 1988.

[20] P. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," *IEEE Transactions on Computers*, vol. C-31, no. 1, pp. 41–47, January 1982.

[21] "Paragon Parallel Processor," *I.E.E.E. Computer*, vol. 25, no. 1, January 1992, New Products Section.

[22] C. D. Polychronopoulos, and U. Banerjee, "Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds," In *Proceedings of the 1987 International Conference on Parallel Processing*, The Pennsylvania State University Press, pp. 410–420, 1987.

[23] B. R. Preiss, and V. C. Hamacher, "Semi-Static Dataflow," In *Proceedings of the 1988 International Conference on Parallel Processing*, The Pennsylvania State University Press, pp. 127–139, 1988.

[24] T. M. Ravi, M. E. Ercegovac, T. Lang, and R. R. Muntz, "Static Allocation for a Data Flow Multiprocessor System," Tech. Rep. CSD-860028, Computer Science Department, University of California at Los Angeles, Los Angeles, California, 1987.

[25] V. Sarkar, and J. Hennessy, "Compile-time Partitioning and Scheduling of Parallel Programs," In *Proceedings of the 1986 International Conference on Parallel Processing*, The Association of Computing Machinery, pp. 17–26, 1986.

[26] J. Sheild, "Partitioning Concurrent VLSI Simulation Programs Onto a Multiprocessor by Simulated Annealing," *IEE Proceedings*, vol. 134 Pt. E, no. 1, pp. 24–30, January 1987.

[27] G. C. Sih, and E. A. Lee, "Scheduling to Account for Interprocessor Communication Within Interconnection-Constrained Processor Networks," In *Proceedings of the 1990 International Conference on Parallel Processing*, The Pennsylvania State University Press, pp. I–9 – I–16, 1990.

[28] S. Skedzielewski, and J. Glauert, "IF1 - An Intermediate Form for Applicative Languages, Version 1.0," Tech. Rep. M-170, LLNL, 1985.