Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

A SIMULATION OF DYNAMIC TASK ALLOCATION IN A DISTRIBUTED COMPUTER SYSTEM

Ed Andert
Aerojet ElectroSystems
P.O. Box 296, Bldg 160, Dept 4216
Azusa, CA 91702, U.S.A.

ABSTRACT

Distributed processor systems are currently used for advanced, high-speed computation in application areas such as image processing, artificial intelligence, signal processing, and general data processing. The use of distributed and parallel processor computer systems today requires systems designers to partition an application into at least as many functions as there are processors. Spare processors must be allocated and function migration paths must be designed to allow fault tolerant reconfiguration. The parallel process/ parallel architecture control simulation (PPCS) models parallel task allocation on a distributed processor architecture. Parallel task allocation is a first step in designing a dynamic parallel processor operating system that automatically assigns and reassigns application tasks to processors. Advantages of this approach are: dynamic reconfigurability removing the need for spare processing power reserved for failures; the reduced need for fallback and recovery software for fault detection; more optimized partitioning of functions; and better load balancing over available processors. PPCS models various distributed processing configurations, task dependencies, and the scheduling of the tasks onto the processor architecture. The PPCS system implements fifteen different heuristic scheduling algorithms to map a set of tasks onto the processing nodes of a distributed computer. The simulation shows the feasibilty of using fast algorithms to heuristically schedule a system of multiple processors allowing dynamic task allocation.

1. INTRODUCTION

PPCS is a parallel process/ parallel architecture control simulation that models various distributed processing configurations, task dependencies, and the mapping or scheduling of the tasks onto the processor architecture. A computer program can be written in such a manner that tasks related by their data and order of execution are defined to take advantage of inherent parallelism in the algorithm. Often these task dependencies are represented by directed graph data-flow diagrams. Certain tasks must precede other tasks or groups of tasks which requires sequential execution of the tasks. Some tasks may occur simultaneously with one another which allows parallel or concurrent execution. This paper will call the relationship of many such tasks in a single job a task dependency structure. It is well known that programs and algorithms can be written to take advantage of inherant parallelism using parallel operations in a multiprocessor environment. Data-flow languages such as LUCID (Wadge et al. 1985) and OCCAM (May 1980) and other programming languages such as ADA are available that allow the encoding of parallel operations. Task dependency mappings can be generated from programs written in these languages.

The PPCS system implements various scheduling algorithms to map a set of tasks onto the processing nodes (resources) of a distributed computer. The primary goal of scheduling algorithms is to minimize the total time required to execute a set of tasks on a finite system of processors. Most current research in this area concentrates on problems in which all tasks and their characteristics are known apriori, thus requiring static partitioning algorithms and techniques. The PPCS system on the other hand is directed at researching and developing task scheduling algorithms that are dynamic and adaptive. A major concern of this effort is the development of general models and algorithms that, coupled with simulation results, can be expanded upon to build a working parallel control operating system. This generality is contrary to much of the research in this area which concentrates on scheduling models for specific applications (Degroot et al. 1981) (Hennings et al. 1977) (Nutt 1977) (Stankovic et al. 1985) (Stone 1985) (Sullivan et al. 1977) (Thomas et al. 1974).

2. TASK ALLOCATION MODEL

The assumptions about the characteristics and types of algorithms needed for the task allocation scheduling problem are as follows. The scheduling algorithms for this stage of the research are confronted with a single set of tasks and task dependency relationship. Depending on the scope and nature of a particular application, expanding the model used here to encompass a dynamic task set and/or dependency relationship is not a difficult problem. The fundamental goal of each scheduling algorithm is to minimize execution time through the system for all tasks without violating the task dependency structure. An additional concern for a scheduling algorithm is the data-flow routing costs (hops). A hop occurs when the

data that results from one task executing on a processor must be transferred to another processor for input to a successor task. None of the algorithms contained in this discussion have explicit capability for the minimization of routing costs, but all are monitored for their performance in this respect in the simulation. Preemption of tasks is not considered since in this model it is not clear that preemption could offer any advantage. The processing nodes are assumed to be identical (homogeneous) and loosely coupled (each processor has it's own memory). Hetrogeneous processors often occur in real systems but the scheduling of hetrogeneous processors is complex and resource dependent (Miller 1982), which makes it difficult to show non-application-dependent simulation results.

A general model for a multiprocessor scheduling system of this type proposed by Graham (1972) has the following constraints:

1. A set of identical processors P_i , $i = 1,\ldots,m$.

2. A set of tasks T_j , $j = 1,\ldots,n$ to be processed by the P_i .

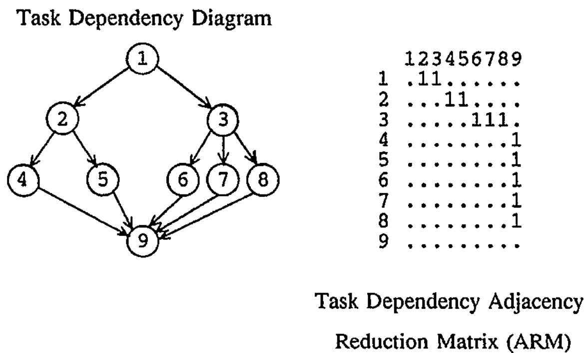
3. A task dependency relation, $<$, on T which is anti-symmetric and transitive.

4. A function, $M: T \rightarrow (0,\infty)$, which denotes the execution time of each task, T_j .

The performance goal of this model is to minimize the total time required to execute the set of tasks, T_j , on the set of processors, P_i , without violating the task dependency relation, $<$. This problem has been shown (Ullman 1973) to be polynomial complete even if all tasks T_j require only one unit of execution time, or all tasks T_j require one or two units of execution time and there are only two processors P_i . Thus, optimal task scheduling according to this model is computationally intractable, i.e. computations increase exponentially as the number of tasks and processors grow.

PPCS is a simulation of the operation of scheduling algorithms that circumvent the NP complete problem of the multiprocessor model by forcing limitations to the model and/or producing suboptimal schedules. Two major categories of algorithms are used in the simulation. The first type of algorithm produces a level schedule and requires that the task dependency structure be "N-free" to produce level schedules that are nearly optimal. The level scheduling algorithms must be implemented in a central operating system since they create a single schedule for the entire set of tasks. The second category of algorithms follows a general multiprocessor system (GMS) model in which processors are assigned tasks from a ready que whenever the processors become free. Tasks are added to the ready que as soon as all of their predecessors have been executed. The GMS algorithms are best implemented in a distributed manner with each processor executing the scheduler between tasks. A more detailed discussion of both the PPCS simulation and scheduling algorithms is contained in the following sections.

The task dependency directed graph can be represented in an adjacency reduction matrix (ARM). A task dependency ARM is a matrix of boolean values representing the dependency of one task to another. The matrix rows and columns represent the tasks of a particular job or computation. A location in the matrix that contains a one indicates that the corresponding column-task is dependent on the corresponding row-task. An example of a directed graph data-flow diagram and its associated adjacency matrix is shown in Figure 1. The ARM representation of task dependencies is used in the PPCS system as a convenient storage mechanism and processing interface between the scheduling algorithms and the task dependency mappings.



Task Dependency Adjacency Reduction Matrix (ARM)

Figure 1

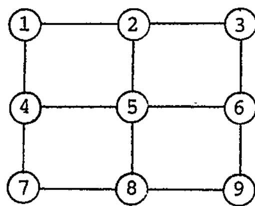
Distributed architectures are similarily represented in an ARM in the PPCS system. A distributed architecture topology usually consists of independent processing nodes and node-resident memory with fixed interconnections. The rows and columns of the interconnection ARM represent the processors of the architecture. A location of the matrix that contains a one indicates that there is a interconnection from the corresponding row-processor to the corresponding column-processor. An example can be seen in Figure 2.

3. LEVEL SCHEDULING

A level schedule is one that partitions tasks into computation levels according to the task dependency structure. Those tasks with no predecessor tasks are assigned to the first computation level, followed by the tasks whose predecessors were assigned to the first level being assigned to the next level, etc. Each computation level of tasks is executed on all of the processors at one time with the next computation level

executing after the complete execution of the previous level. The different types of level algorithms in PPCS assign tasks to processors for each level via differing methods. A simple example of a level schedule with in-order (of tasks found for a new level) task assignment to processors is shown in Figure 3.

Processor Topology Diagram




```
  123456789
1 .1.1.....
2 1.1.1....
3 .1...1...
4 1...1.1..
5 .1.1.1.1.
6 ..1.1...1
7 ...1...1.
8 ....1.1.1
9 .....1.1.
```

Processor Topology Adjacency Reduction Matrix (ARM)

Figure 2

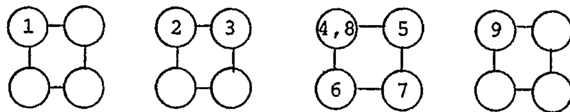
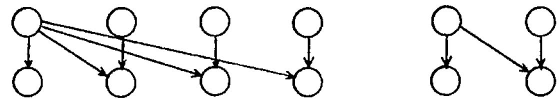


Figure 3: Example of Level Schedule of Figure 1 Task Dependency onto 4-Node Nearest Neighbor Topology

A major attraction of level algorithms for scheduling is that limitations on the general multiprocessing model can reduce the worst case bound on the length (in time) of level schedules. The worst case bound on the length of the schedule produced by many known algorithms is (2 - 1/m) times as long as the optimal schedule where m is the number of processors (Chen et al. 1974) (Nett 1977). However, if the task dependency relationship is "N-free" then the level schedule is bounded by 3/2 times the length of the optimal schedule (Nett 1977), An example of an "N" in a graph is shown in figure 4. Task dependency structures that have been reviewed and generated in this research have not contained "N" structures and it is difficult to specify program data flow structures for real-world problems that contain such structures. Thus, using a scheduling algorithm that builds better schedules (in the worst case) with the omission of "N" structures is appealing for general multiprocessor scheduling problems. It should be noted that the proofs for the bound on level schedules in (Nett 1977) were done with the assumption of unit task execution times. The scope of this research did not include the presentation of a proof for a worst case bound for an "N-free" level algorithm over varying execution times, but it is assumed that level algorithms will produce good schedules where task dependencies are "N-free".



Task Dependency Containing "N"s　　Actual "N" Structure

Figure 4

Another attraction of level scheduling for a real-world multiprocessor system is their symplicity. The basic level scheduling algorithm as explained previously does not require excessive comparisons and calculations on the task dependency relationship which in essence make up the time cost of a scheduling algorithm. This simplicity does not come without a price. Since task execution time is not of unit length, it is guaranteed that at each computation level all processors. except for the one with the task load of longest execution time, will remain idle waiting for the computation level to end.

The level scheduling algorithm builds computation levels and assigns tasks to processors for all tasks and processors in the system at one time. For this reason the level scheduler would be best implemented in a central processor. Thus, in the PPCS system, level scheduling algorithms are considered centralized scheduling algorithms.

3.1 Level Scheduling Algorithms

All of the level scheduling routines share the same simulation environment program. The level scheduling simulation programs differ only in the building of levels and assigning of tasks.

The building of levels is done by two different algorithms that vary only slightly. They produce two different types of level assignments, one top loaded and one bottom loaded. The top loaded version of the algorithm assigns all tasks to the earliest possible computation level while the bottom loaded version assigns tasks to the latest possible computation level. The difference between the two level assignments is minimal or non-existent for most data-flow task dependencies. In other words, the more irregular the structure of the forks and joins in the task dependency relationship, the larger the difference between the top and bottom loaded level assignments.

The different level algorithms assign tasks to processors in different order. Thus, this function varies for every different level schedule. There are two different processor assignment

routines for each type of level scheduling algorithm; one to handle top loaded level assignments and one to handle bottom loaded level assignments. The processor assignment routines are essentially the same except that they operate in reverse order of one another.

The in-order top and bottom loaded task assignment to processor algorithms schedule tasks to processors in the same order that tasks are found and assigned to levels by the level making routines. These algorithms are called LEVELTOP and LEVELBOT.

The decreasing time order algorithms assign tasks to processors in decreasing task execution time order which has been used as an scheduling criteria in both bin-packing problems (Coffman et al. 1977) and multi-processor problems (Jensen 1977). An array is created during the environment portion of the level program that contained the execution time distribution from a corresponding specified file. This array of execution times is indexed by task number and contains the estimated execution times from the file. A quick-sort routine is used that sorts integer pointers to an array of double-precision real numbers into an order in which they point to the real numbers in increasing order. The quicksort algorithm used is a modification of the standard quicksort algorithm (Aho et al. 1974) to sort a pointer array. The decreasing time order task assignment to processor algorithms are called LDECRTOP and LDECRBOT.

The increasing time order algorithms assign tasks to processors in increasing task execution time order. The LINCRTOP and LINCRBOT algorithms are very similar to the decreasing time order algorithms.

The random order assignment algorithms (LRANDTOP and LRANDBOT) assign tasks to processors in a randomized order. This has been used for a scheduling criteria for multi-processor problems and can be used as a comparison for ordered assignment algorithm performance (Jensen 1977).

Assignment of tasks to processors by pairwise interchange (LRPAIRTOP and LTPAIRBOT) will produce the most nearly optimal schedule for all of the level algorithms. Tasks are assigned to processes in-order for each level and then compared two by two until no improvement can be made to each computational level execution time. This technique was shown by Graham (1972) to produce a schedule execution time worst case bound of (2 - 2/(m + 1)) times the optimum schedule execution time where m is the number of processors and there is no task dependency relationship. This bound applies at each computation level of the schedule guaranteeing that each computation level will be executed in close to the shortest possible time.

The execution simulation program uses the schedule produced by a level routine and simulates the schedule execution compiling execution time, communication data transfer (hop), and algorithm compute time statistics. The execution simulation routine loops around the specified processors using them over and over in the order given when there are more tasks assigned than processors specified. The algorithm CPU time usage is monitored on a VAX 11/785 which includes some VMS operating system overhead CPU usage. For comparison reasons the simulations were run under similar VAX loading situations.

4. GMS SCHEDULING

General multiprocessor system (GMS) scheduling is a way of scheduling a task dependency relation onto a set of processors suggested by Graham (1972). This type of scheduling assumes the general multiprocessing system constraints explained in the earlier conceptual overview. A GMS schedule proceeds as follows: Initially, at time 0, all tasks that have no dependency constraints (ie. have no predecessor tasks) are available for execution and inserted in a ready que. All processors are free and attempt to remove a task from the ready que and execute it. If no tasks are in the ready que and a processor is available then the processor will remain idle until a task is entered into the que. As a processor completes a task and proceeds to look into the ready que for more work, tasks that are successors to the completed task and have no other unexecuted predecessors are entered into the ready que. This continues until all tasks have executed.

A major attraction for using the GMS model in solving the multi-processor scheduling problem is the model's strong theoretical research background. Graham (1969) has done extensive research into timing anamolies encountered in the GMS model. Much related research has been done in finding worst-case bounds in one-dimensional bin packing problems. Optimal and sub-optimal algorithms have been developed to evenly fit different sized objects into multiple bins or buckets of equal size (all bins filled to the same height) (Graham 1972) (Johnson 1974) (Demers et al. 1974). The bin packing problem in one of its forms, namely sequencing to minimize make-span, is very similar to the general multiprocessor scheduling problem under discussion here. In this problem the set of objects (records, tasks) must placed into a fixed set of minimum sized bins (storage units, processors) which is equivalent to the general multiprocessing system defined earlier without a task dependency relation.

Another, perhaps more practical, attraction for the GMS algorithms is their capability for being implemented in a distributed manner. Each processor can execute a portion of the scheduler upon it's completion of a task and before it's search for a new task to execute. Upon completion of a task

the processor can search the task dependency relation for new tasks and enter them into the ready que. The processor can then remove a task from the ready que and execute it. This distributed scheduling capability is especially appealing with the GMS type algorithm since it is much more complicated and computationally intensive than the Level type algorithm.

4.1 GMS Scheduling Algorithms

The GMS scheduling environment consists of an implementation of the GMS scheduling simulation, statistical data accumulation and reporting. Data gathering is similar to the level schedule simulation with schedule execution time, communication data tranfers (hops), and algorithm compute time statistics reported. All of the GMS scheduling routines in PPCS use the same main GMS scheduling algorithm. All of the actions in the algorithm are shared by all GMS scheduling programs except for the ordering of the ready que tasks. The methods used to order the ready que is where GMS algorithms can be tuned for higher performance. The GMS routines in the PPCS system each use a different method for ordering the ready que.

A common bin packing algorithm that assigns each object to the bin with the lowest current level is exactly the GMS algorithm without task dependency considerations. The objects can be ordered into increasing size or decreasing size improving the worst-case performance of the algorithm in terms of the schedule produced compared to the optimal schedule (Coffman et al. 1977). Because of these results and the simplicity of the algorithm, increasing and decreasing estimated execution times are used a as sorting criteria in two GMS ready que ordering algorithms. The same quick-sort routine that was used in creating Level schedules is used here. The quick-sort sorts the ready que as indexes to the array of estimated execution times for the tasks in the system. A decreasing ready que ordering algorithm is used in GMSDECR and increasing order algorithm is used in GMSINCR.

Beyond the increasing and decreasing sorts of the ready que, there are no algorithmic performance improvements with a foundation in theoretical research that can be applied to the GMS scheduling model. However, it seems probable that some use of the task dependency structure in weighting the tasks in the ready que would yield further performance improvements. Thus, the concept that the level scheduling model was based on will be used here as a heuristic sorting criteria. The algorithm used in the GMSHLF program is a highest level first (number of vertices from a task to it's most distant successor) sort of the ready que. This criteria was used in a simulation of a somewhat similar multiprocessing model involving the parallelization of sequential FORTRAN compilations. The highest level first algorithm was found to produce better results than random or first-in first-out orderings of a ready que (Jensen 1977).

Another heuristic that can be used to order the ready que is the number of successors to each task. This can be done by sorting the tasks based on their number of immediate successors. This was also simulated in the above mentioned study and found to produce schedules that were no better than those produced using a FIFO or random ordering of the ready que (Jensen 1977). The algorithm for ordering the que based on immediate successors is much simpler than any of the other heuristic orderings used here and therefore should be much faster than the others. It also seems intuitive that a most immediate successors first heuristic should perform much better than a random ordering if the task dependency relationship is of a parallel rather than sequential nature. The most immediate succesors first algorithm was used in the GMSMISF program.

A more accurate heuristic for ready que ordering based on the number of successors to each task is a most total successors first method. In previous simulation efforts this criteria was found to produce resulting schedules of lengths similar to the highest level first ordering (Jensen 1977). The most total successors first algorithm was used in the GMSMTSF program.

The most accurate heuristic for producing the shortest total execution time for a set of tasks can be found in the critical path method used in management science scheduling theory. The critical path method suggests that the best schedule can be found by searching for the critical path in the task network (dependency relationship) and giving it the highest priority (executing it first). The critical path is defined as the path through the network from beginning to end that requires the most total time to execute. This concept can be used in the GMS scheduling model by ordering the tasks in the ready que based on the path length of each task. The path length in the GMS model can be defined for each task as the longest sum of execution times required to get from the task to the last possible successor task(s) for every possible path. A more complicated (yet computationally easier) version of this heuristic using likelyhood probabilities for each path (Martin et al. 1969), used in the simulation discussed earlier, was found to produce schedules of length similar to the level and total successors heuristics (Jensen 1977). The longest path length first ordering of the ready que algorithm was used in the GMSLPLF program.

5. SIMULATION RESULTS AND ANALYSIS

The PPCS simulation programs were run for three different task dependency structures, three different processor topologies, and three different execution time distributions.

Running all of the PPCS algorithms on the varying parameters created 432 schedules and execution simulations.

Uniform, gaussian, and bi-modal execution time distributions were created by a BUILDTIME program. The uniform distribution contained double-precision real number values for task execution times uniformly (randomly) distributed between 0 and 100. This creates task execution values that represent a uniform mixture of long, medium, and short task computation times. The gaussian distribution contained real values between 0 and 100 with a mean of 50 and a standard deviation of 10. This represents a set of tasks all having about the same execution time. The bi-modal distribution was made up of real values between 1 and 100 with two modes that are equally likely. One mode was at 30 and the other at 70 both with a standard deviation of 5. The bi-modal distribution represents a set of tasks with either long or short execution times. Also, the estimated execution times used by the scheduling algorithms were gaussian approximations to each execution time value. Each estimated time was picked from a distribution with a mean of the actual execution time and a standard deviation of 5.

The task dependency relationships used in the simulation runs attempt to represent general multiprocessing problems. The task dependency structure used in GEN1 shown in figure 5 was suggested in (Jensen 1977) as a model of a computation graph considering compiler analysis of typical programs. The task dependency graph was modified within the constraints of the graph model in (Jensen 1977) to emphasize parallel operations rather than sequential. Even with these enhancements the structure for GEN1 is very sequential and looks like a parallelization of the operations specified by a sequential language program. GEN2 shown in figure 6 is a modification of GEN1 to allow as much parallelism as possible. Thus, GEN2 represents a sequential computation with many parallel operations at different stages. GEN3 as shown in figure 7 is the most parallel of all of the task dependecy relationships. It is a binary tree with branches and leaves pruned randomly to decrease the regularity of the graph. GEN3 represents a common search graph found in many artificial intelligence and image processing applications.

The processor topologies used in the simulation runs were made up of 12 or less processors. Any more processors would limit the utility of the scheduling algorithms due to the size of the task dependency relationships. The largest of the three task sets contained 198 tasks but by the nature of the task dependencies, the task sets often could not offer much more than 12 competing tasks at any one time. Simulating larger task sets and processor topologies would produce limited additional general multiprocessor system results and is thus beyond the scope of this effort. The three processor topologies, NEAREST, PLANAR-4, and CUBE as used in the simulation are shown figure 8. These topologies were chosen because of their size and varying interconnection schemes.

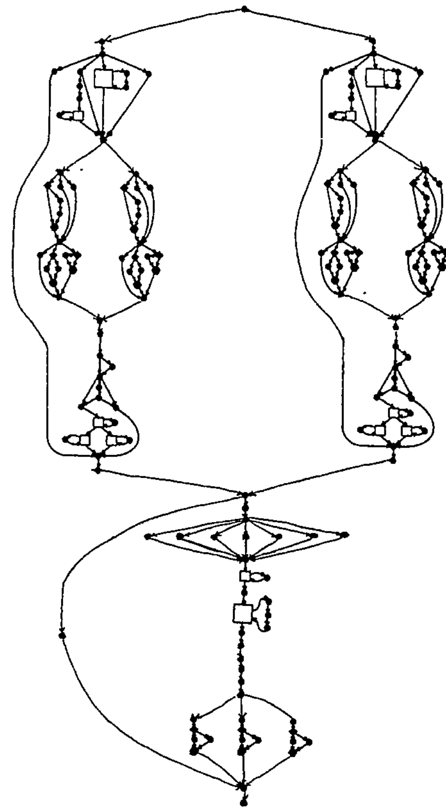


Figure 5: GEN1 Task Dependency Structure

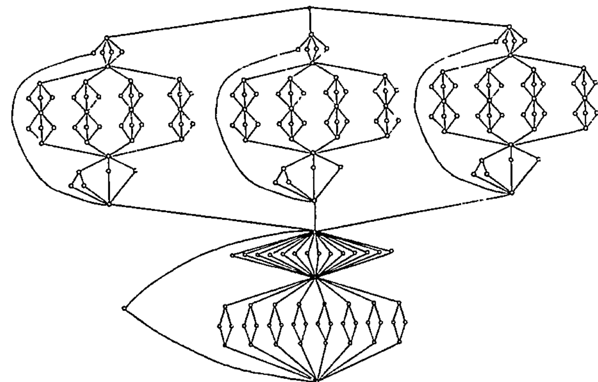


Figure 6: GEN2 Task Dependency Structure

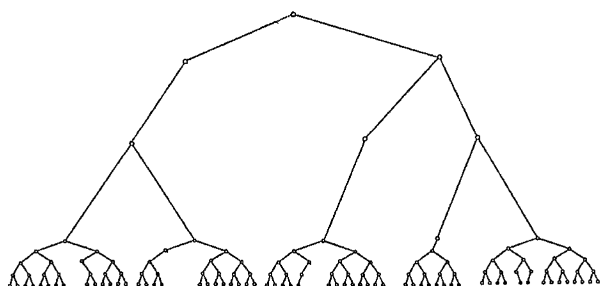
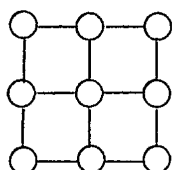
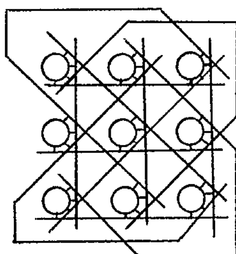


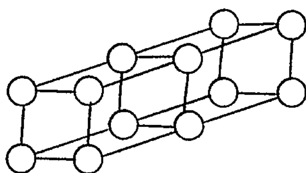
Figure 7: GEN3 Task Dependency Structure



Nearest Neighbor 9-Node Processor Architecture



Planar-4 9-Node Processor Architecture



Cube 12-Node Processor Architecture

Figure 8: Processor Architectures

terms of task execution time was fairly stable over the task sets but the level algorithm computation time in terms of top-loaded versus bottom-loaded level partitioning algorithms varied between task sets. The GEN3 task set caused the bottom-loaded algorithms to execute faster than top-loaded algorithms while the GEN1 and GEN2 task caused just the opposite. This not surprising considering that the top and bottom loaded partitions are done by searching the task dependency graph from opposite ends and would be easily affected by the width and depth of the search caused by the task dependency relationships.

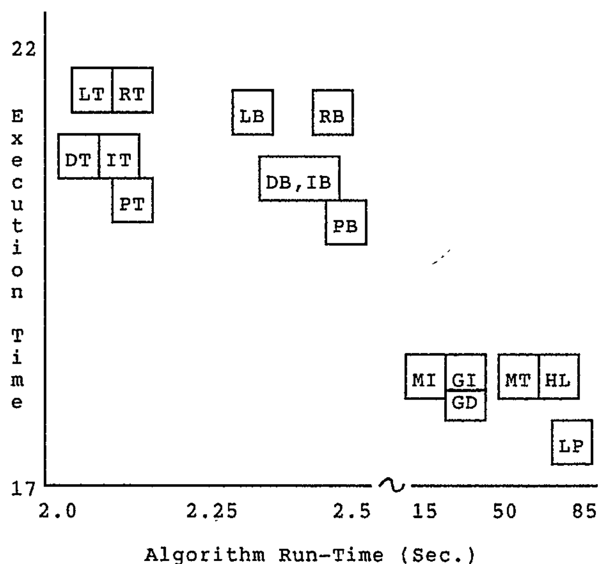


Figure 9: Algorithm Run-Time Performance

Graphs of the performance of the scheduling algorithms in terms of execution time for the schedules produced and the algorithm computation time are shown in figure 9 averaged over three representative parameter sets. The performance in terms of execution versus algorithm time of all of the simulation runs closely resemble those shown in this figure. No significant performance variation was found in the results from either the time distribution or the processor topology. There was, however, minor performance variation caused by the task sets. The performance of the level algorithms in terms of task execution time was fairly stable over the task sets but the level algorithm computation time in terms of top-loaded versus bottom-loaded level partitioning algorithms varied between task sets.

The most conclusive comparison that can be made from these simulations is between the performance of the different algorithms on any of the task sets. As can be seen from the graphs, the most striking observation is that the GMS algorithms require far more computation than the level algorithms, often in excess of ten times as much computation time. Although the performance of the GMS algorithms in terms of task execution time is better than that of the level algorithms, it is generally not much more than a 50% improvement. It is also important to note that in every case

that the GMSMISF algorithm performed equally as well as the GMSHLF and GMSMTSF algorithms at a substantially reduced computation cost. The GMSLPLF algorithm produced the best of all of the execution times in most cases but at a high computation cost and, as can be seen in figure 10, at a high communication data transfer cost. It is difficult to make a generalization about the performance difference between the GMSDECR and GMSMISF algorithms in terms of execution time or algorithm computation time, but as can be seen in figure 10, the GMSMISF algorithm often produces schedules with a lower data transfer cost. It is also difficult to make a performance comparison between the increasing and decreasing level algorithms beyond the fact that they perform better in terms of execution time than the random and in-order level algorithms. The pair-wise interchange level algorithms consistently offer the best execution times out of all of the level algorithms with little algorithm computation time penalty, but with a higher data transfer cost.

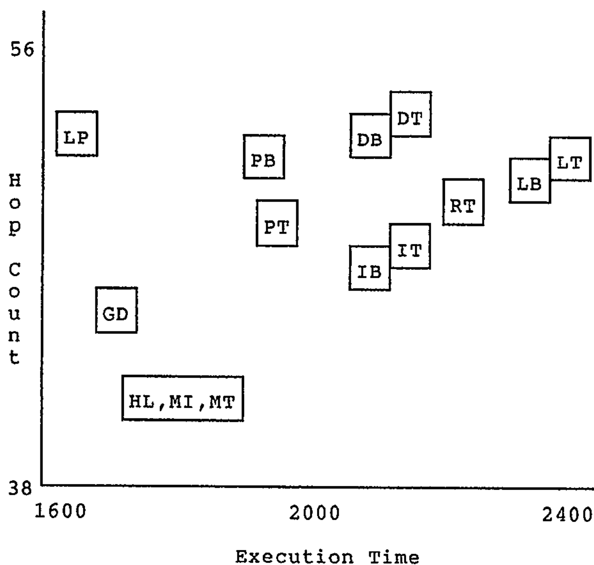


Figure 10: Hopcount Vs. Execution Time Algorithm Performance

The conclusions that can be drawn from these observations and applied to the general multiprocessor scheduling problem are as follows. Because of the regularity of the task execution time performance data over all of the parameter sets in terms of the different scheduling algorithms, it is safe to say that those algorithms that performed well in this simulation are likely to perform well for applications that fit into the multiprocessor model. For applications where the computation time spent in scheduling tasks is a large concern, then the global, level scheduling algorithms are the best choice. Where a small amount of computation time and data transfer time can be spared, the pair-wise interchange algorithm is the best choice. Otherwise the increasing and decreasing algorithms are the best alternative. The choice between a top-loaded and bottom-loaded level algorithm is application dependent requiring analysis of the expected task dependency relationships. Those applications that are most concerned with task execution time and/or data transfer costs are best implemented using the GMSMISF algorithm. If task execution time is the sole concern then the GMSLPLF algorithm is the best choice.

It is probable that the performance of the GMS algorithms could be significantly improved in terms of algorithm computation time by keeping more information about the tasks in the system than the ready que and task dependency structure. Making computed data reusable would significantly reduce the recomputation required by the GMS algorithms. This would, however, immediatly incur a large data communication penalty to offset the advantage of implementing the GMS algorithms in a distributed manner. An important improvement that would have to be made if the algorithms from PPCS were to be implemented in a system without an optimized hardware ARM processing capability, would be a sparse matrix implementation of the ARMs.

ACKNOWLEDGMENTS

Part of this study was performed by the author as a graduate research project at California State University, Fullerton.

REFERENCES

Aho, Hopcroft, and Ullman (1974). The Design and Analysis of Computer Algorithms. Addison Wesley, Mass. p 92.

Chen, N. and Liu, C. (1974). On a Class of Scheduling Algorithms for Multiprocessor Computing Systems. In: Parallel Processing Proceedings of the Sagamore Computer Conference. 1-16.

Coffman, Leung, and Slutz (1977). On the Optimality of First-Fit and Level Algorithms for Parallel Machine Assignment and Sequencing. In: Proceedings of the 1977 International Conference on Parallel Processing. Institute of Electrical and Electronics Engineers, San Francisco, California, 95-99.

Degroot, Jenevein, and Lipovski (1981). A Hardware Support Mechanism for Scheduling Resources in a Parallel Machine Environment. In: Proceedings of the 8th Annual Symposium on Computer Architecture. 57-65.

Demers, Johnson, and Ullman (1974). Worst Case Performance Bounds for Simple One-Dimensional Packing Algorithms. SIAM Journal on Computing 3, 299-326.

Graham, R. (1969). Bounds on Multiprocessing Timing Anomalies. SIAM Journal on Applied Math 17, 416-429.

Graham, R. (1972). Bounds on Multiprocessing Timing Anomalies and Related Packing Algorithms. In: Proceedings of AFIPS 1972 Spring Joint Computer Conference. 205-217.

Hennings, Schindler, and Steinacker (1977). Schedules for General Monitor Systems with a Minimal Number of Processors. In: Proceedings of the 1977 International Conference on Parallel Processing. Institute of Electrical and Electronics Engineers, San Francisco, California, 26-30.

Jensen, J. (1977). A Fixed Variable Scheduling Model for Multiprocessors. In: Proceedings of the 1977 International Conference on Parallel Processing. Institute of Electrical and Electronics Engineers, San Francisco, California, 108-117.

Johnson, D. (1974). Fast Algorithms for Bin Packing. Journal of Computers and System Science 8, 272-314.

Martin, D. and Estrin, G. (1969). Path Length Computations On Graph Models of Computations. IEEE Transactions on Computers C-18, 530-536.

May, D. (1980). OCCAM. IEEE SIGPLAN Notice 18.

Miller, L. (1982). A Heterogeneous Multiprocessor Design and the Distributed Scheduling of its Task Group Workload. In: Proceedings of the 9th Annual Symposium on Computer Architecture. 283-290.

Nett, E. (1977). On Scheduling Algorithms for N-Free Task Dependency Structures. Proceedings of the 197 International Conference on Parallel Processing, Institute of Electrical and Electronics Engineers, San Francisco, California, 100-107.

Nutt, G. (1977). A Parallel Processor Operating System Comparison. IEEE Transactions on Software Engineering SE-3, 467-475.

Shen, C. and Tsai, W. (1985). A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion. IEEE Transactions on Computers C-34, 197-203.

Stankovic, Ramamritham, and Shengchang (1985). Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems. IEEE Transactions on Computers C-34, 1130-1143.

Stone, H. (1985). Parallel Processing with the Perfect Shuffle. IEEE Transactions on Computers C-20, 1130-1143.

Sullivan, Bashkow, and Klappholz (1977). A Large Scale, Homogeneous, Fully Distibuted Parallel Machine I and II. In: Proceedings of the 4th Annual Symposium on Computer Architecture. 105-124.

Thomas, A. and Davidson, E. (1974). Scheduling of Multiconfigurable Pipelines. In: Proceedings of the 12th Annual Allerton Conference on Circuit and System Theory. 658-669.

Ullman, J. (1973). Polynomial Complete Scheduling Problems. Technical Report 3, Department of Computer Science, University of California at Berkeley.

Wadge, W. and Ashcroft, E. (1985). LUCID, the Dataflow Programming Language, Academic Press, London.

AUTHOR'S BIOGRAPHY

ED ANDERT is a member of the technical staff at Aerojet ElectroSystems company. He is also a part-time associate professor in the School of Computer Science at California State University, Fullerton. He received a B.A. in business management and computer science at CSU Fullerton in 1984, and a M.S. degree in computer science from CSU Fullerton in 1986. His current interests include advanced sensor processing, distributed and parallel systems, distributed operating systems, and artificial intelligence. He is a member of AAAI, ACM, and IEEE.

Ed P. Andert
Aerojet ElectroSystems
Bldg. 160, Dept. 4216
P.O. Box 296
Azusa, CA 91702-0296, U.S.A.
(818) 812-2571