Parallel Task Execution in a Decentralized System

MARIO J. GONZALEZ, JR., AND C. V. RAMAMOORTHY

Abstract—The overhead involved in the real-time multiprocessor execution of parallel-processable segments of a sequential program is investigated. The execution follows a preprocessing phase in which the source program is analyzed and the parallel-processable segments are recognized. A number of representations of a parallel-processable program are possible. A table representation is used, and a technique is developed to efficiently interpret this table in a system in which a number of identical processors share a common memory.

A number of program graphs based on real problems are used to investigate the overhead involved in parallel task execution. The investigation is conducted in the context of two organizations—the centralized, in which processor assignments are totally under the control of the operating system, and the decentralized, in which processors are allowed to determine their own next assignment from the program they are executing. In the decentralized system, processor assignment and other control information is embedded within the program segments themselves. Within each of these organizations, two configurations are examined. In the first, the operating system periodically polls a program to determine if a need for a processor exists; in the second, a program generates an interrupt to request a processor when a task becomes ready.

Results of simulation in which the number of processors and the task durations are permitted to vary over a wide range show that the decentralized organization and the polling configuration provide the most improvement over conventional uniprocessor execution. This improvement, which includes consideration of overhead due to table interpretation, exists even when the number of parallel paths is not large, provided the average task duration is not small.

Index Terms—Centralized, decentralized, operating systems, parallel processors, program graphs, task schedules, task tables.

Manuscript received January 26, 1972; revised August 1, 1972.

M. J. Gonzalez, Jr., was at the University of Texas at Austin, Austin, Texas; he is now with Texas Instruments, Inc., Austin, Texas 78757.

C. V. Ramamoorthy is with the Department of Electrical and Computer Sciences, University of California, Berkeley, Calif. 94720.
In the current discussion, a brief summary of background material, including the table representation of a program, will be given, and a statement of the problem will be provided. This will be followed by a discussion of assumptions made regarding the system environment under which the parallel-processable program is to be executed. The main part of the paper considers the execution of a diversity of programs under two operating system configurations—the centralized, in which resource allocation authority stems from a single source, and the decentralized, in which authority is distributed among the resources themselves, subject to certain constraints. Finally, the cost of the allocation schemes is investigated, and recommendations are made on the basis of conclusions obtained from extensive simulation of the schemes that are examined.

Background

Given the permissible transition graph of a computer program, the maximal strongly connected (MSC) subgraphs in the given graph can be uniquely determined [1]. A final program graph, called the reduced graph of the system, can be derived by replacing each MSC subgraph with a single vertex. In this graph, a vertex that initiates a branching operation will be called a decision or branch vertex. The next step is to derive the parallel task graph and its connectivity matrix C. The elements of C are determined by analyzing the inputs of each vertex in the reduced graph. An element \( C_{ij} \) is a \( \text{"1"} \) if and only if the jth task of the reduced graph has as one of its inputs the output of task i; otherwise \( C_{ij} = \text{"0"} \).

The parallel task graph also can be represented by means of a task table, as shown in the example of Fig. 1. This table is a modification of the tables generated as the output of the Fortran Parallel Task Recognizer [3]. The task table is actually two separate tables; the first is of dimension \( NX3 \) (where \( N \) represents the number of nodes in the graph), and the second table, the successor table, is of dimension \( (E+1) \) (where \( E \) is the number of edges in the graph). In the example graph, the exclusive-or symbol (\( + \)) is used to indicate two conditions: a branch node (that is, a node which, upon termination, causes the initiation of only one of a number of possible successors), and a join node, which requires for its initiation the completion of only one of its possible predecessors.

The first column in the task table contains a set of pointers to the successor table. In the successor table, all the possible successors of all the tasks are listed. For example, the second entry in the first column of Fig. 1(b) contains the number 2. This means that a listing of the successors of task number 2 begins in the second entry of the successor table. To determine the number of successors of a particular task, determine the difference between adjacent entries in the first column of the task table. Continuing the example, the difference between the second and third entries in the first column of the task table is three. This means that task 2 has three successors, beginning with the second entry of the successor table. These successors are tasks 3, 4, and 5.

The second column merely represents the number of predecessors of each task. With the exception of tasks that have mutually exclusive inputs, this number is equal to the number of edges incident to the task. The third column is nonempty only in those cases in which the task is a branch. In those cases, the entry in the third column represents the run-time successor of the branch task.

The task table, then, contains all the information necessary to process the corresponding program in a multiprocessor environment. Notice, however, that no optimal scheduling information is built in to the task table. In the above example, for instance, upon completion of task 2, all that is known is that tasks 3, 4, and 5 may be begun. In the event that the number of available processors is less than three, a decision must be made as to which of the three tasks should be initiated in order to achieve minimum execution time (i.e., an optimal schedule). This choice is a function of the program configuration and the task duration; given this information, techniques of the type given in [8] can be used to generate an optimal schedule. In this discussion, however, the top member of a set of candidates in the successor table always will be chosen first. This corresponds to execution of tasks according to their position in the set of earliest time precedence partitions [1], [4], [8].

Before the primary objectives of this paper are addressed, some comments will be given regarding the role of a parallel task recognizer in a multiprocessor system.

Recognizer–System Interface

In previous work [2], the recognizer had been used on a stand-alone basis to perform parallel task analysis on programs written in a particular language. In general, however, a parallel task recognizer of the type referred to here should fit into an overall system in a manner similar to that shown in Fig. 2.

Basically, the components (statements) of the source program are examined in parallel by the translator (compiler) of the source program language and the recognizer; these two system components are, in turn,
in communication with each other. In practice, the recognizer would not be active at all times. It would be activated upon user request and only after the source program was determined to be free of errors. As has been shown in [2], the recognition process requires large amounts of time, and should be used only when the source program is to be run a large number of times without modification. After the object code and the corresponding task table are generated, the operating system controls a number of identical processors sharing a common memory to execute the source program.

Ideally, the mode of operation described above should be adaptable to any language that the system can recognize as long as the precedence integrity of the program tasks is maintained. An additional characteristic of this type of system should be versatility. This versatility should include the ability to execute the program under different operating system disciplines and various hardware configurations, such as a variable number of processors, including uniprocessor operation.

The discussion that follows assumes the existence of the task table representation of the parallel-processable task graph of a sequentially organized program written in a higher level language.

**Problem Definition**

The basic problem can be simply stated as follows. Given the table representation of a program and a number of identical processors sharing a common memory, how can the table be interpreted dynamically such that the various processors can be assigned to the program tasks in a manner that minimizes program execution time? An answer to this question will be provided in several steps. The first of these steps considers the dynamic interpretation of the task table.

**Task Table Interpretation**

On the surface, the interpretation of the task table is straightforward. When a processor completes a task, the contents of the predecessor column of each of the successors of the completed task are decremented by one. If, as a result of this action, the contents of the predecessor column of any of the successor tasks goes to zero, the successor task may be begun. Notice, however, that additional information is needed to relate tasks as represented in the task table to their actual counterparts in the original code stream. Thus, for example, when we say that upon completion of task $T_n$, tasks $T_{n+1}$, $T_{n+2}$, and $T_{n+3}$ may be begun, what is meant is that a processor should be assigned to the execution of the code represented by each of the task names.

In order to implement the table interpretation scheme outlined above, the following items are needed.

1) The address of the first instruction of each of the program tasks. Thus when we say that "task $T_j$ can be initiated," what is meant is that a processor should begin execution of code at the address that corresponds to the first word of task $T_j$.

2) A means of identifying the boundaries of a task and the task name.

3) A means of causing task table updates that preserves the integrity of the precedence ordering of the tasks.

**Address-Task Association**

In order to associate an address with a task name, a degree of coordination is needed between the parallel task recognizer and the compiler that generates the object code. Briefly, the compiler examines a line of source program code and passes it to the recognizer after it is found to be syntactically correct. Just before the compiler examines a new statement, it has available to it the contents of the program counter. The contents of this counter represent the relative address of the last object code instruction of the previously examined statement. This address, plus one, represents the relative address of the first instruction of the next statement to be analyzed. Thus when the compiler transmits a copy of a statement to the recognizer, it can, upon the request of the recognizer, also transmit the contents of the program counter. In this manner, the recognizer can associate an address with a task name entry in the first column of the task table.

**Task Name and Boundary Identification**

Even though the beginning address of each task may be identified, provisions must still be made to prevent a processor from executing code beyond the limit of its assigned task. An obvious reason for this is that tasks represented by adjacent sets of object code may be executed simultaneously. And unless a processor is caused to terminate execution at a particular point, it may attempt to initiate execution of a task that has already been assigned to another processor.

To eliminate this problem, several solutions were considered [4], and the one chosen was the following. Imbed two additional instructions at the end of each task. The effect of the first is to set a task-complete bit, and the second causes an unconditional transfer to a set of control code that informs the processor of its new assignment (a new task or release to a processor pool, for example).

Information for the first of the two instructions can be provided to the compiler by the recognizer. When the compiler transmits a copy of the source program statement to the recognizer, the recognizer can increment its task counter by one and return the number to the compiler. The compiler will then automatically increment its program counter by two after adding the required
instructions, and will continue to analyze source code. Utilizing this scheme, a set of object code, modified for parallel execution in a multiprocessor environment, can be illustrated, as shown in Fig. 3. BRANCH tasks and their successors are handled in a different manner to overcome the problem that would result from a failure to execute the two added instructions due to the nature of a BRANCH task [4].

**Task Table Updating**

In the previous paragraphs, a scheme was developed that provides an identification of a completed task by the termination of code execution after the name of the completed task is recognized. The problem now is to accomplish the following: decrement the predecessor count of each of the successors of the completed task, and indicate in some manner which of the successor tasks, if any, may be initiated. At the same time, task ordering should be preserved, and processor idle time and additional memory requirements should be minimized.

That the problem arises at all is due to the fact that the entire process of task execution, table update, and new task assignment is all occurring simultaneously in an asynchronous manner. Note that to cause a table update really means the execution of a set of predefined code. In the table interpretation scheme described earlier, the update scheme begins with the name of the completed task. Once this name is known, the predecessor count of its successor tasks is decremented by one. When the predecessor count of task reaches zero, the task may be initiated.

To accomplish these objectives, only one copy of the task table and one copy of the table-update code are maintained. Provisions are made for the update code to be locked and unlocked by the processor making a table entry. Because the code will often be unavailable when a processor seeks to record the fact that it has completed a task (and since it is undesirable to make a processor wait for the update code to become available), provisions are made for the name of a completed task to be entered in a Task Complete Queue (TCQ). In a similar manner, a Task Ready Queue (TRQ) is maintained to permit a processor to indicate that more than one task is ready for execution as a result of a table update (i.e., a FORK situation).

In summary, then, a task is initiated by causing a processor to execute an instruction that represents the first instruction of the related task. The end of a task is indicated by an unconditional transfer instruction explicitly inserted in the original object code stream by the compiler during the preprocessing phase. The completion of the task is communicated to the task table in an indirect manner. The processor completing the task enters the task name in a TCQ. At some point in time, when no task is waiting to be executed (i.e., the TRQ is empty) and the table-update code is found to be unlocked, a processor (not necessarily the same one that placed the task in the TCQ) will cause the task table to be updated, using the completed task as the key to the update code. If a ready task is found, it is placed in the ready queue, a processor is assigned to it, and the entire process is repeated.

**Task Schedules**

The schedules to be generated must include overhead of three different kinds: time spent in requesting and obtaining a processor, entering a completed task in the TCQ, and causing a task table update. As discussed in [4], approximately eight times as much code must be executed to cause a table update as is required to enter a completed task in the TCQ. Thus, based strictly on the amount of code executed, we say that the overhead associated with a table update is eight times that associated with entering a completed task in the TCQ. Equivalently, we say that if the latter requires 1 unit of time, the former requires two units of time.

This leaves only one overhead component to discuss: the overhead associated with requesting and obtaining a processor. We can safely assume that the system is set up to respond quickly to processor requests, since it is known that the processor-preemption, state-word-preservation technique is highly optimized. Thus it is safe to say that response to a processor request involves overhead of the same order as the table-update and task-complete overhead described earlier. For convenience, then, we will say that the average overhead associated with a processor request will be twice that associated with entering a completed task in the TCQ, that is, 3 unit.

To summarize, three types of overhead involved in the generation of task schedules have been identified. These are: \( t_r \), the average time required to obtain a processor once a request for one is recognized; \( t_u \), the time to search the task table and record the effect of a com-
completed task; and \( t_e \), the time required to place the name of a completed task in the TCQ and update its pointer.

The concepts outlined above are illustrated in Fig. 4 by means of a graph and its accompanying schedule. In this example, it is assumed that each task requires ten units of execution time, and that three processors are available for assignment to the parallel-processable program. The graph indicates that upon completion of task 1, tasks 2, 3, and 4 may be initiated; the completion of all three of these tasks indicates that task 5 may be initiated.

In this example, the advantages of parallel processing are immediately obvious. The program requires 39\( \frac{1}{2} \) units of time for its execution; of these, 9\( \frac{1}{2} \) units are attributable to overhead. If the same program would have been executed by a single processor, the time requirement would have been 50 units. What if, instead of three processors, only two were available? Under these circumstances, the schedule of Fig. 5 would result. Notice that now the schedule is 45\( \frac{1}{2} \) units in length, still under the 50 required by a uniprocessor schedule. The reduction in time is considerably less, however. Whereas before the reduction was more than 20 percent, it is now less than 10 percent. Thus we begin to get a hint of how critical are such things as the graph configuration, system response time, processor availability, and task duration. The critical nature of these and other items will be brought out more vividly in the discussion that follows.

**Assumptions**

This section will deal with assumptions regarding the environment in which the parallel-processable program is to be executed. In addition, assumptions regarding the nature of the program itself will also be discussed. These items are discussed in detail in [4].

**Memory Management**

It will be assumed that the parallel-processable program will be executed in a paged, multiprogrammed environment. The reason for this is that situations will often arise in which the computational power of a multiprocessor system is primarily in an idle state if this power is dedicated exclusively to a parallel-processable program. By allowing an idle processor to be assigned to a ready-but-waiting program, not only will processor idle time be reduced, but also, system throughput will be increased.

We will assume a "virtual memory" in which the memory resident programs acquire the pages of their "working set" on demand [9]. Activating a parallel-processable program is accomplished by bringing into memory the pages that contain the table-update code, the task table, and the TRQ and TCQ. The program is initiated by causing a processor to query the status of the TRQ that had task 1 entered into it at the end of the preprocessing phase. Upon determining that \( T_1 \) is ready to run, the processor will begin the issuance of virtual addresses that will cause the parallel-processable program to accumulate the pages that it requires.

It is suggested here that the pages that contain the table-update code and the task table be exempted from consideration when a need exists to replace a page in memory with a new page from secondary storage. Since this code is in almost constant use, it would be highly undesirable to replace one or more of these pages when it is virtually certain that they would be needed very soon thereafter.

As far as the object code is concerned, the nature of the program graph provides valuable information regarding page utilization. Since the graph is loop-free, once a task is completed, no subsequent reference will be made to the code that represents that task. Thus the pages that represent a completed task can be replaced without any concern for a page fault later on. Note that this is another strong argument for the use of a paged memory in an implementation of this type. Once a task is completed, the pages that represent that task will not be referenced again.

**Program Priority and Task Preemption**

As noted in [2], parallel task analysis requires large amounts of time. The primary reason for the investment of this time is that a need exists to process the analyzed program in as short a time as possible; in practice, this time should be more than trivially shorter than normal uniprocessor execution time. Thus it will be assumed that a parallel-processable program possesses a higher demand priority on system resources than the typical nonparallel-processable program.

Finally, since preemption in midstream of a processor assigned to a task in a parallel-processable program would only aggravate what is already acknowledged to be a significant amount of overhead, it will be assumed that in the proposed implementation, a processor cannot be preempted until it has completed its assigned task.

**System Configurations**

In the preceding discussion, the table representation of a program was introduced, and a means was provided
for a processor to determine that a task was completed. Code was imbedded in the original object code stream to cause a processor to transfer to a set of control code at a specified location in memory. The purpose of this section is to investigate the options available to a processor once this transfer takes place. It is at this point that a processor first becomes a candidate for influence by the operating system. Since we assume nonpreemption of tasks, a processor cannot be given a new assignment until a task is completed.

In effect, we are asking the question: "How should a new assignment be given to a processor that has just completed its previous assignment?" There appear to be two main solutions to the problem. In the first, a processor will always revert to the control of the operating system upon completion of an assigned task. Upon determining that a processor request exists, the operating system can compare processor requests to processor availability, and award a resource of this type according to a predetermined algorithm. We will call this approach the centralized operating system (COS) approach since all authority for resource disposition stems from a single source.

In contrast to this, the second solution, referred to as the decentralized operating system (DOS), seeks to distribute decision-making policies among the user programs. What this means is that, upon completion of an assignment, a processor will be permitted to determine its next assignment without assistance from the operating system, where this assignment can be the execution of a ready task or a task table update.

The motivation for this solution is a belief that the operating system in present systems is already relied upon too heavily in an attempt to achieve proper system performance. To be sure, the performance of contemporary computer systems could not be achieved without well-tuned operating systems. Increasing computing demands, however, have caused the operating system to become as valuable and as scarce a resource as a processor, a memory module, or a data channel. The decentralized approach seeks to avoid the placement of an even heavier demand on an already burdened operating system. This reduction on demand comes at a cost of reduced control over the computing environment, for now the system must explicitly preempt a processor instead of having it become automatically available, as is the case in the centralized approach.

This paper, then seeks to compare program performance under the two philosophies—COS and DOS. Throughout these comparisons, the objective will be to execute a program in the shortest possible time. In the first comparison, Configuration II, a processor will always be assumed to be available, up to a certain maximum. In Configuration I, schedules were generated without regard to any kind of overhead. The function of Configuration I was to develop an understanding of the differences between COS and DOS. Configuration I will not be included in this comparison. In Configuration III, program priority will be considered. Processor assignment will be approached from two directions: in the first, the system will query the programs to determine whether or not they require an additional processor; in the second, the placement of a task name in either the TRQ or the TCQ will cause a program itself to request a processor. In these first two configurations, tasks will be assumed to be of equal duration. In the final comparison, Configuration IV, the equal task duration assumed in the first two configurations will be discarded. The comparisons of Configuration III will be repeated with tasks of unequal duration.

Comparison Environment

In order to compare the two operating philosophies, six programs of varying size and complexity were executed by means of a parallel processor simulator. With one exception, the graphs, adapted from a previous paper in this area, are representative of real programs [5]. The size of the graphs ranges from 29 to 195 nodes, and they are characterized by several degrees of inherent branching [4]. The graph of Fig. 6 contains 32 nodes and no branches, and is characterized by a large degree of natural parallelism. This graph will be used as an example throughout this discussion.

The simulator used to execute these program graphs uses the task table representation of the graph as input. For the case in which equal task durations are assumed, task durations of 4, 6, 8, 10, and 20 units were permitted. The number of processors in the system was allowed to vary from 1 to 20. Uniprocessor execution time \( t_u \) was not obtained by means of the simulator. Instead the following formula was used:

\[
t_u = \sum_{i=1}^{N} t_{d_i}
\]

where \( t_{d_i} \) is the duration of task \( T_{d_i} \) and \( N_i \) is the number of tasks executed in a graph of \( N \) tasks. Notice that the expression for \( t_u \) does not include any kind of overhead, since a program executed by a single processor in a conventional manner will not be represented by a table. Therefore, it will not incur any of the overhead associated with the dynamic interpretation of the table.

The following paragraphs will describe the various
conditions under which the centralized and the decentralized configurations were compared.

**DOS II**

Configuration II represents a condition that is somewhere between the ideal and the actual. Although it incorporates all the elements of overhead discussed earlier, it assumes that a processor will always be available one-half unit from the time it is needed. Thus as long as there are processors available in the system, a parallel-processable program request will be satisfied within one-half unit. Because it fails to consider system loading and program priority, this configuration serves the role of a benchmark against which the more realistic configurations that follow can be compared.

**COS III-Poll**

In this configuration, several concepts missing in Configuration II are introduced. First, when a task achieves ready status, it is not automatically initiated one-half unit later. Instead, it must wait for the operating system to examine the TRQ and the TCQ to see if anything is waiting to be done. In the tests conducted, the execution of the programs was simulated with the polling interval set at two, three, and four units. Second, after a ready task is detected, a processor is not automatically awarded after a fixed delay. Instead, to simulate system loading and program priority, a processor request formula is executed; if the result of the calculation exceeds a specified threshold, then the processor is awarded after a delay of one-half unit.

The processor request formula is given by the following expression:

\[ \text{REQ} = (N_p - N_a)P \]

where \( N_p \) represents the total number of processors available in the system, \( N_a \) represents the number of processors already awarded to the requesting program, and \( P \) represents the priority of the requesting program.

The matter of priority was handled in the following manner. Every time that the processor request expression was evaluated, a call was made to a random number generator (RNG). Results produced by the RNG are such that, on each call, a decimal fraction between 0 and 1 is returned with equal likelihood. The result returned by the RNG was substituted for \( P \) in the request formula. Furthermore, the request threshold was set at 0.5 throughout the experimentation. Thus as long as \( (N_p - N_a) \) is greater than zero, the worst situation that a parallel-processable program can face is that only half of its processor requests will be granted. The greater the difference between \( N_p \) and \( N_a \), the better the chances of an affirmative processor request.

In this manner, the parallel-processable program is implicitly given a high priority. In the worst case, only half of its requests are turned down. In practice, probably 75 percent of program requests were granted. This figure approached 100 percent when the number of processors was large (20) and the number of processors already in use by the program was small. The latter condition is always satisfied when a program first begins to generate a demand for system resources.

**DOS III-Poll**

The major distinction between this category and the previous one is the same as the major distinction between COS and DOS. In the former, a processor will always revert to system control upon completion of a task. In the latter, a processor will always check for the presence of something waiting to be done in the same program. If something is found, the processor will initiate the appropriate action on its own and continue to execute other tasks within the program. A free processor will permit the system to dictate its next assignment only if it finds nothing to do. Thus we would expect program performance to be markedly improved over the performance provided by the corresponding COS configuration. Experimental results to be related shortly show that this is indeed the case.

**COS III-Interrupt**

In this configuration, whenever a task becomes ready, a signal is sent to the system requesting a processor. We would therefore expect improvement over the poll scheme, since the program does not now need to wait for the system to interrogate the program to determine if a processor is needed.

One undesirable provision must be made in the configuration to cover a possible contingency. Suppose that task \( T_i \) represents a task that must be completed before any further processing can take place. Further, suppose that \( T_i \) has been completed and its name entered in the TCQ. At this point, a signal would be sent to the operating system requesting a processor to perform a task table update with \( T_i \) as the key. Consider the consequences of a negative reply to the request. If the request is denied, no other action pertaining to the program will ever cause a processor to be requested. Under these conditions, the program would remain in an idle state indefinitely if its memory residence were not questioned.

Therefore it is necessary for the system to periodically check on the status of the program to insure that it is not in this very damaging (at least as far as its execution time is concerned) state. Obviously, the interval cannot be very frequent since it would place a heavy burden on the system, on top of that created by the fact that the program is already demanding the attention of the system when tasks achieve a ready state. In the tests conducted, this status check was permitted to occur at intervals of 5, 10, and 15 units. Intuitively, we would expect program execution time to be improved over the pure polling scheme. Results of simulation, however, show that this is very much a function of the polling interval, with a positive gain achieved only when the interval is small.
DOS III-Interrupt

This configuration follows the general requirement given above for the COS scheme. That is, whenever a task achieves ready status, a signal is sent to the system requesting a processor to take care of the task. Unlike the COS scheme, however, no need exists for the system to check if the program is in an idle state, for this condition will never occur. In the worst case, the processor that caused a task to become ready will eventually take care of that task. This is a result of the characteristic of the decentralized approach in which a processor always looks for something to do within the program it has been executing. Thus, if under some situation no system support in the form of an additional processor is forthcoming, one of the processors already allocated to the program will eventually take care of a ready task. The only effect of this lack of support will be in the form of slightly degraded performances. However, provisions must be made to insure that the parallel-processable program always has at least one processor assigned to it. There will always be something for a single processor to do until the program terminates.

The next section will discuss the results of the simulation experiments performed under the various configurations. Fig. 7 shows a representative schedule that results under the DOS II configuration for the graph of Fig. 6. In this schedule, it is assumed that the number of processors in the system is five, and the task durations are assumed to be ten units.

Simulation Results

Each of the six test graphs mentioned earlier was executed by means of the parallel-processor simulator under each of the configurations listed above. Within each combination of test graph and system configuration, the task duration was set at 4, 6, 8, 10, and 20 units. That is, the ratio of task execution time to table-update time varied from a minimum of two to a maximum of ten. (Recall that the table-update time was set at two units.) For each task duration, the number of available processors was permitted to vary from 1 to 20. Therefore, a single data point is identified by four characteristics: 1) test graph name, 2) system configuration, 3) task duration, and 4) number of processors. Fig. 7 therefore represents a single data point; the schedule is generated by the simulator. No figures are actually drawn, of course; the result of one pass through the simulator is a numerical value that represents the simulated execution time of a program under a particular set of conditions.

Equal Task Duration

Figs. 8 and 9 show the simulation results for the example graph for the two extremes of task durations, 4 and 20 units. As shown in [4], these plots are representative of the conditions tested for all the test graphs. Notice that when the task duration is small, parallel processing yields no advantage over conventional uniprocessing. When the task duration is increased to 20 units, however, the advantages of parallel processing are immediately evident. Only COS III-Interrupt (for a poll interval of 10 and 15) and COS III-Poll (for a poll interval of 4) fail to yield an advantage when the number of processors drops below three.

Observations

For all the graphs tested, the first conclusion is the easiest to make, and is almost trivial to the point of not requiring any testing at all. Regardless of the graph con-
configuration, the longer the tasks, the better the performance of a set of processors executing in parallel. What is not trivial is determining what the minimum task length should be before parallel processing ceases to pay off. Notice that the answer to this question is not independent of the manner in which the processors are utilized (that is, the system configuration). In general, however, analysis of the simulator data shows that the task length should not be permitted to drop below ten units. This is equivalent to saying that the task execution time to table search time ratio should be of the order of five. (Recall that table search time was assumed to be fixed at two units.) Thus if we say that any one of a number of identical processors requires \( k \) units to perform a table update, then the task duration should be such that it requires \( 5k \) units to execute it. Tasks that
fall short of the desired duration can be made longer by the process of node coalescing [4]. This can be accomplished by combining two or more tasks into a single task.

It also appears that for the graphs studied here, the number of processors available in the system should be at least three. If this number drops below three, the number of times that a processor request is denied increases quite rapidly. In addition, at any given time, one of the processors assigned to a program may be involved with performing a table update. If no other processor is assigned to the program, little real work gets accomplished for large periods of time. If the number of processors increases beyond three, then an adjustment can be made in the task length. For instance, if the number of processors is four, then a task duration of $4k$ units may be tolerated.

As far as system configurations are concerned, DOS III-Poll provides the best performance by far. In these tests, little loss in performance was noted as the poll interval was increased from two to four units. This is a very significant point, since it suggests that some tolerance can be built into the manner in which the operating system can react to the state of a timer that defines the poll interval. By increasing the poll interval from two to four units, the burden on the operating system is significantly reduced. In fact, the tests suggest that the poll interval can be increased even more while still providing a large improvement over a uniprocessor configuration. A tradeoff point is reached when the polling interval is increased to the extent that performance under this configuration is matched by one of the other less adequate configurations.

The other system configurations cannot be dismissed, however. Second place behind DOS III-Poll is shared by DOS III-Interrupt and COS III-Poll and Interrupt, provided that in the latter two cases the polling interval is high. For instance, in terms of the time scale used in these tests, COS III-Poll should not be attempted unless the polling interval is at least two units; the corresponding interval for COS III-Interrupt is five units.

If consideration is limited to performance alone, then DOS III-Interrupt appears to be the best of the three, although this configuration is marked by periods of erratic behavior. This occurs when a potential for parallel processing exists and a processor request is denied. The processor request will not be repeated until a subsequent task becomes ready. These periods of relative inactivity can be reduced by causing the operating system to periodically poll the program in the same manner that was done for COS III-Interrupt. In terms of case of implementation, these configurations do not rank high, however, since they require the attention of the operating system, both at the time that an interrupt is generated and also at the time that the system is required to poll the program.

COS III-Poll, on the other hand, requires no program-generated interrupts, although it does require polling at high frequencies in order to be profitable. An important feature of this configuration is that it provides a suitable alternative to the decentralized approach. Under this configuration, the system maintains close control over its resources at a cost of increased program execution time.

Unequal Task Duration

The tests discussed in the preceding section assumed that all tasks in a program graph required equal amounts of processing time. Although it is theoretically possible to transform a given program graph into one in which all tasks are of equal duration, it is still desirable to determine the effect of unequal task duration on the proposed parallel-processor configurations.

To accomplish this, reference was again made to a random number generator. In order to relate subsequent test results to preceding ones, each of the fractions supplied by the RNG was multiplied by 20, with only the integer part of the result retained. The result of these multiplications is that task durations are now in the range of tasks covered in the previous tests.

A close examination of the results of the simulation tests with unequal task times [4] shows that those results bear a very high qualitative resemblance to the results of the equal task duration tests. For example, we can say that for the example graph, the COS III-Poll scheme should not be used unless the poll interval is two and the number of processors is greater than or equal to three. Similarly, COS III-Interrupt should not be used unless the poll interval is at least five and the number of processors is at least three.

This type of remark can be made for all the test graphs and all the configurations with only minor modification in a few cases. Thus, for example, any conclusions reached for the case in which all task times were equal and the task duration was ten also can be applied to the case in which the tasks were unequal but whose average duration was ten units. This is a very significant result, for it implies that it is not necessary that all tasks be significantly longer than the table search time in order for parallel processing to pay off; it is only necessary that the average of the task durations be significantly longer than the table search time.

This is very important in terms of what it means to the compiler–recognizer relationship. The flexibility suggested by this result means a reduced burden on the compiler in the number of addresses it must supply the recognizer and the number of recognizer signals it must interpret.

Cost of Implementation

An implementation of the type proposed here involves several different types of overhead. The cost of the pre-processing phase has been discussed in [2]. The cost of the additional memory required to store the task table, the table-update code, and the control code imbedded in the object code stream will now be discussed. Let
these additional memory requirements be represented by \( M_t, M_u, \) and \( M_e \), respectively. Therefore, the additional memory requirement \( M \) can be expressed as

\[
M = M_e + M_u + M_t.
\]

To indicate task completion, two words are added at the end of each task, one to enter the task name in the TCQ and one to cause a transfer to the control code. These two words, however, are appended only to those tasks that are not branches. Each potential successor of a branch has three instructions added to it in front of the successor object code—one to indicate completion of the predecessor, one to add the execution time successor to the task table, and one to transfer to the control code [4]. Let the following constants be defined: \( N \), the number of tasks (nodes) in the program graph; \( N_B \), the number of branch tasks; \( E \), the number of edges in the graph; \( E_i \), the number of edges out of node \( i \); \( \delta_i \), equals 1 if task \( i \) is a branch, equals 0 otherwise. With these definitions, \( M_e \) can be expressed as follows:

\[
M_e = 2(N - N_B) + 3 \sum_{i=1}^{N} \delta_i E_i.
\]

As shown earlier, \( M_t \) can be expressed as

\[
M_t = 3N + (E + 1).
\]

The table update can be represented by approximately 25 lines of Fortran-like code. Assume that this would convert to approximately 50 words in their object code representation. Adding this number to (1) and (2), \( M \) is given by

\[
M = \left[ 2(N - N_B) + 3 \sum_{i=1}^{N} \delta_i E_i \right] + \left[ 3N + (E + 1) \right] + 50
\]

\[
M = 5N - 2N_B + E + 50 + 3 \sum_{i=1}^{N} \delta_i E_i.
\]

For the example graph of Fig. 1, \( N = 11, N_B = 1, E = 13, \sum_{i=1}^{N} \delta_i E_i = 2 \). Therefore, for this graph

\[
M = 5(11) - 2(1) + 13 + 50 + 3(2) = 122.
\]

Thus the total memory overhead would be 122 words. Of these, 47 words would be required to contain the task table. Thus the suggestion made earlier, that the pages containing the task table and the update code be kept in memory throughout the lifetime of the program, appears to be a reasonable one. The overhead due to table interpretation will now be discussed.

In the preceding discussion, three types of overhead were identified: \( t_r \), the average time required to obtain a processor once a request for one is identified; \( t_s \), the time to search the task table and record the effect of a completed task; and \( t_e \), the time required to place the name of a completed task in the TCQ and update its pointer. Of these, the first is highly dependent on the system configuration, that is, centralized or decentralized. Furthermore, this overhead is not truly due to the interpretation of table; rather it results from the assignment of multiple processors to multiple processes. Therefore, this type of overhead will not be discussed further.

The other two types of overhead, however, result directly from table interpretation and lend themselves to more direct analysis. The overhead due to task completion \( t_c \) occurs once for every task that is executed. For a program whose execution time path is highly data dependent, it is not possible to predict the exact number of tasks to be executed, although in many cases a fairly close estimate can be made. If we let \( N_t \) represent the number of tasks executed, then the following expression results for \( t_{tr} \), the total overhead due to task completion.

\[
t_{tr} = N_t \cdot t_r.
\]

Consider now \( t_s \), the overhead due to task table updating. As shown in [4], it is more nearly correct to speak of the average table search time \( t_s \). Of the \( N_t \) tasks executed, let \( N_{Bx} \) represent the number of executed branch tasks. Then \( \{N_t - N_{Bx}\} \) represents the number of executed tasks that are not branches. The total overhead due to table search \( t_{tr} \) therefore has two components. The first of these is due to table updates by non-branch tasks.

\[
t_{tr} = \sum_{i=1}^{N_{Bx}} \alpha_i E_i \delta_i
\]

where \( \alpha_i = 1 \) if \( T_i \) is executed and is not a branch, and is 0 otherwise.

The second component is due to table updates by branch tasks. The effect of completion of a non-branch task must be noted for all of its successors. The effect of the completion of a branch task on the other hand, must be noted only for its single execution time successor. However, it is necessary for the completed task to examine its potential successors until the actual successor is found. Therefore, one must speak of bounds when considering the table-update overhead due to the completion of a branch task. On one hand, the actual successor may be found on the first examination; on the other, all potential successors may be examined and the actual one not found until the final examination. The second component of \( t_{tr} \) therefore is expressed as follows:

\[
N_{Bx} \cdot t_s \leq t_{tr} \leq \sum_{i=1}^{N} \beta_i E_i \delta_i
\]

where \( \beta_i = 1 \) if \( T_i \) is an executed branch task, and is 0 otherwise. Equivalently, the lower bound may be written as

\[
\sum_{i=1}^{N} \beta_i \cdot t_r
\]
Combining the two components, the total overhead due to table searching can be expressed as
\[ t_{sr} = t_{sr1} + t_{sr2} \]
\[ t_{sr} \text{ (min)} = \tilde{t}_{s} \cdot \sum_{i=1}^{N} \alpha_i E_i + \bar{t}_{s} \cdot \sum_{i=1}^{N} \beta_i \]
\[ = \tilde{t}_{s} \cdot \sum_{i=1}^{N} (\alpha_i E_i + \beta_i) \]
\[ t_{sr} \text{ (max)} = \tilde{t}_{s} \cdot \sum_{i=1}^{N} \alpha_i E_i + \bar{t}_{s} \cdot \sum_{i=1}^{N} \beta_i E_i \]
\[ = \tilde{t}_{s} \cdot \sum_{i=1}^{N} E_i (\alpha_i + \beta_i). \]

The overhead due to both table search and task completion \( t_T \) now can be expressed as
\[ t_T = t_{sr} + t_{st} \]

In terms of the latest notation, \( t_{sr} \) can now be rewritten as
\[ t_{sr} = N \cdot t_{e} \]
\[ = \bar{t}_{s} \cdot \sum_{i=1}^{N} (\alpha_i + \beta_i). \]

If, as in the schedules of the previous sections, we assume that \( \bar{t}_{s} = 8t_{e} \), the following expressions result:
\[ t_{sr} = \frac{\bar{t}_{s}}{8} \cdot \sum_{i=1}^{N} (\alpha_i + \beta_i). \]
\[ t_T \text{ (min)} = t_{sr} + t_{st} \text{ (min)} \]
\[ = \tilde{t}_{s} \cdot \sum_{i=1}^{N} (\alpha_i + \beta_i) \]
\[ = \tilde{t}_{s} \cdot \sum_{i=1}^{N} \left[\frac{\alpha_i + \beta_i}{8} + \sum_{i=1}^{N} (\alpha_i E_i + \beta_i)\right] \]
\[ = \tilde{t}_{s} \cdot \left[\frac{\sum_{i=1}^{N} (8E_i + 1)}{8} \alpha_i + 9/8 \beta_i\right]. \]
\[ t_T \text{ (max)} = t_{sr} + t_{st} \text{ (max)} \]
\[ = \tilde{t}_{s} \cdot \sum_{i=1}^{N} (\alpha_i + \beta_i) + \bar{t}_{s} \cdot \sum_{i=1}^{N} E_i (\alpha_i + \beta_i) \]
\[ = \tilde{t}_{s} \cdot \left[\frac{\sum_{i=1}^{N} (\alpha_i + \beta_i)}{8} + E_i (\alpha_i + \beta_i)\right] \]
\[ = \tilde{t}_{s} \cdot \left[\frac{\sum_{i=1}^{N} (\alpha_i + \beta_i)(1/8 + E_i)}{8}\right] \]
\[ = \tilde{t}_{s} \cdot \left[\frac{\sum_{i=1}^{N} (\alpha_i + \beta_i)(8E_i + 1)}{8}\right]. \]

In the case of the example graph, \( \beta_i \rightarrow 0 \) and \( \alpha_i \rightarrow 1 \) for all \( i \) (TG has no branches and all tasks are executed), and \( t_T \text{ (max)} = t_T \text{ (min)} = t_T \). For \( E_i = 47, N = 32 \), and \( \tilde{t}_{s} = 2 \), \( t_T \) can be calculated as follows:
\[ t_T = \frac{\bar{t}_{s}}{8} \cdot \sum_{i=1}^{N} (8E_i + 1) \]
\[ = \frac{\bar{t}_{s}}{8} \cdot \left[8(47) + N\right] \]
\[ = 1/4 \cdot \left[376 + 32\right] \]
\[ = 102. \]

This calculation indicates that, in the execution of the example graph, 102 time units can be attributed to table interpretation overhead, independent of the system configuration. Notice that some of the task completion updates occur at the same time that a table search is underway. Thus not all of \( t_T \) is expended in a strictly serial manner. Because \( t_T \) is significantly larger than \( t_{e} \), however, its effect can be ignored without introducing any serious error. If this factor is omitted from the calculation of \( t_T \), then \( t_T = 94 \).

In addition to being independent of the system configuration, \( t_T \) is also independent of the task durations. Regardless of how many processors are allocated to a program, the amount of real work done is still equal to the sum of the task execution times. For the case of unequal task times, the sum of the task execution times for the example graph equals 342 units. In order to accomplish this amount of real work, 94 time units of overhead must be invested. This is analogous to saying that for every four units of real computation, approximately one unit of overhead must be expended. Equivalently, if five processors are assigned to a program, one of them is devoted strictly to overhead matters.

**Summary**

In this paper, a technique has been introduced for executing independent segments of a sequentially organized program in a parallel-processor environment. A simulator was written to execute a set of program graphs under two different philosophies. In the centralized approach, a processor reverts to operating system control upon completion of an assigned task. In the decentralized approach, a processor always attempts to initiate a ready task before releasing to system control. Within each of these configurations, processors are acquired in one of two ways: either the operating system polls the program to see if anything is waiting to be done, or the program signals the system when a task becomes ready.

Extensive tests performed by simulation show that the decentralized approach in which the system periodically polls the program for the presence of a ready task offers the best performance by far. This is due primarily to the configuration characteristic that allows a processor to remain assigned to a program as long as something is waiting to be done. The performance observation is valid for tasks of equal and unequal duration as long as the average task duration is approximately five times the average table-update time duration. These times, of course, would be dependent on the
A General Class of Maximal Codes for Computer Applications

SE JUNE HONG AND ARVIND M. PATEL

Abstract—The error-correcting codes for symbols from GF $(2^b)$ are often used for correction of byte-errors in binary data. In these byte-error-correcting codes each check symbol in GF $(2^b)$ is expressed as $b$ binary check digits and each information symbol in GF $(2^b)$, likewise, is expressed by $b$ binary information digits. A new class of codes for single-byte-error correction is presented. The code is general in that the code structure does not depend on symbols from GF $(2^b)$. In particular, the number of check bits are not restricted to the multiples of $b$ as in the case of the codes derived from GF $(2^b)$ codes. The new codes are either perfect or maximal and are easily implementable using shift registers.

Index Terms—Adjacent-error correction, byte-error correction, error-correction code, maximal codes, shift-register implementation.

I. Introduction

The use of error-correcting codes to improve reliability is becoming an important technique in modern computers. Especially in the memory, be it a core, disk file, tape, or monolithic, and in the straight data transfer paths, benefits of error detecting and correcting codes are clearly recognized.

Random-error-correcting codes such as Hamming and BCH codes are suitable for bit per card or some homogeneous bit arrangements. Increasing speed and system efficiency demands have pushed the idea of bit per card to a cluster of bits per card type memory organization and, likewise, the data paths usually transfer the cluster of bits in parallel. This cluster of bits is often called a byte and hence the name, byte-oriented machine, describes most of the modern computers. A single fault in these systems, either in the memory or in the data path, is likely to affect many bits within a byte. Consequently, a byte-error-correcting capability is demanded of the codes to be used in these systems. The known multiple random-error-correcting codes, which do not make use of the error dependency of this application, require unduly high redundancy and complicated decoding procedure.

Another application of the byte-correcting code is in multichannel digital systems where the channel noise often affects more than one adjacent bit in each channel independently. A fixed size cluster of bits in each channel, when viewed as a byte, lends itself to the application of byte-error-correcting codes.

Throughout this paper, we shall use the term byte in a general sense. A byte means a cluster of $b$ bits of data that are likely to be affected together by channel noise or some hardware fault, due to the circuit packaging method or data format in recording. The byte length $b$, in general, is any positive integer. For $b = 1$, these codes are equivalent to the binary Hamming codes.

REFERENCES


