Denelcor

Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444
TWX 910-931-2201

TABLE OF CONTENTS

RESEARCH PAPERS

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

THE EVOLUTION OF A SUPER-COMPUTER

BY DR. B. J. SMITH

THE EVOLUTION OF A SUPERCOMPUTER

Burton J. Smith
Denelcor, Inc.
Denver, Colorado 80205

Abstract -- The HEP computer system, originally a digital replacement for analog computers, has gradually evolved into a high-performance scientific machine in the supercomputer class. The problems encountered during this metamorphosis are pointed out together with the solutions that were adopted, and conclusions are made based upon this experience.

Initial Designs

In 1973, several of us at Denelcor, Incorporated decided that a digital computer could be designed and built to replace the analog machines that had been our traditional products. Our intent was to retain the speed and parallelism of the analog computer while improving on its programmability, flexibility, and reliability. Our approach was straightforward: the functional units of the analog computer would be implemented in digital form and the patch panel would be replaced by a high-speed bus controlled by a scheduler processor which would transfer data among the functional units. Since some of the functions would have data dependent execution times, it was decided that the synchronization of the data flow in the computer would not depend on the timing of scheduler programs; instead, every function unit would have input and output locations accessible from the high-speed bus which could be either full or empty. The function would be performed when all input locations were full and all output locations were empty, and would empty the inputs, compute the function, and then fill the outputs with the answers. The scheduler processor would be able to perform the opposite operations on the input and output locations to move output values of one function unit to the input locations of other function units. The machine was to have 32 bit arithmetic and be capable of about ten million floating point instructions per second.

We wrote a few programs for some typical analog computer problems and discovered that the traffic on the high-speed bus was limiting the performance of the system. At this point, we decided to replace the add and multiply function units by an algebraic module - really a simple shared resource MIMD computer[1] to reduce the bus load. The algebraic module was pipelined to improve the utilization of the addition and multiplication logic, and had several program counters so that several expressions could be evaluated simultaneously. The registers of the algebraic module could be filled or emptied by the scheduler via the high-speed bus as well as by instructions within the algebraic module. We called this system HEP, standing for Heterogeneous Element Processor.

We built a prototype HEP based on these concepts and were happy with the effectiveness of the architecture. The prototype was built under contract to the U.S. Army Ballistic Research Laboratory (BRL) in Aberdeen, Maryland, and executed their benchmark at a quite respectable speed, expecially considering that the implementation suffered from noise problems and did not run at design clock rate. BRL was impressed enough with the concept to award Denelcor a contract in 1976 to design and build a HEP system incorporating four algebraic modules and 64 bit arithmetic. In order to guarantee that the noise problems of the prototype would be solved, the contract stipulated that initially a single algebraic module was to be built and demonstrated to BRL running benchmarks at design speed; only then would we be allowed to complete the design and construction of the full system. The contract also specified that a high-level language be furnished.

Enhancements

The high-level language problem had us concerned for a while because we could see no good way to decompose a program into a scheduler part and several algebraic module parts, even if the decomposition was specified by the programmer, because of the very special nature of the scheduler instruction set. We decided with BRL's concurrence to eliminate the scheduler and the high-speed bus, and to provide the communication functions of these components with a data memory which would be accessible by all algebraic modules via a switch. The full-empty property would be available at every location in the shared memory to facilitate synchronization of processes running in parallel in different algebraic modules. This idea allowed us to implement an extended version of FORTRAN in which a programmer can write explicit parallel code. In particular, a subroutine in HEP FORTRAN may be invoked by a CREATE statement rather than by a CALL statement. This causes a process to execute the subroutine in parallel with the creating process. In addition, variables having the full-empty property (called asynchronous variables in HEP FORTRAN) are identified by a "$" as the first letter of the variable name and are used to synchronize parallel processes in a producer-consumer fashion.

We realized that explicit parallel programming was not the only way in which HEP could be used effectively, and that multiple independent jobs could be run concurrently if protection were provided by the hardware. The cost of including the necessary protection mechanisms turned out to be just as expensive in terms of hardware complexity, system cost, and schedule as we all had predicted; we implemented the protection hardware primarily because a "single, highly opinionated, forceful individual" [2], Max Gilliland, insisted on it. The ability of HEP to use the parallelism provided by multiple jobs executing simultaneously is certainly an important feature, and has greatly simplified the design of the operating system. It was at about this time that we started calling the algebraic modules "processors" and realized that HEP might be usable as a general-purpose computer system.

While the processor required by our contract with BRL was being built, our attention turned to the switch that was to connect the four processors of the BRL system to data memory. Our original intent was to implement a crossbar switch, but two properties of HEP made a crossbar undesirable. First the interconnection of large numbers of processors and memories is very expensive if a crossbar (or any other $O(n^2)$ switch) is used, and second, HEP is a physically large system and the wire lengths needed to interconnect widely separated units to a centrally controlled switch would result in very wide data paths to maintain the necessary throughput rates. The switch network that we eventually came up with uses packet switching techniques to allow the control of the switch to be distributed among its nodes. The HEP switch also has advantages in configuration flexibility, versatility, and fault tolerance over our original scheme. The major problem in designing it was making it fast enough; how well we succeeded can perhaps be inferred from the fact that the switch propagates messages at one-fifth the speed of light with a bandwidth approaching 80 megabytes per second for each processor or memory connected to it. A succinct description of the HEP switch and HEP as a whole may be found in [3].

Experiences

The HEP processor that we had been building for demonstration to BRL executed its first HEP FORTRAN program successfully in June of 1979, and we have since demonstrated that processor to BRL and others. Most of the programs that have been written for HEP are benchmarks that were obtained from interested parties and rewritten in HEP FORTRAN by Robert Lord of Washington State University. Some of this benchmarking work has led to more generally applicable MIMD algorithm development [4,5].

In sharp contrast to our experience with the prototype, the first HEP processor was extremely easy to get running. We made a conscious decision not to use unproven technologies, and this undoubtedly explains part of our success. The key ingredient, however, was that we were extremely conservative in our approach to the packaging, maintenance, and electromagnetic field theoretic aspects of the implementation. The HEP computer system is speed independent in design, and will run at any clock rate less than or equal to its maximum rate of 40 MHz. This feature allows us to test and debug any part of the system (including the system itself) using IML, our interactive maintenance language. Our experience with the multiplier function unit is instructive. HEP printed circuit boards are about 45 cm wide and 35 cm long and are populated with an average of 208 SSI and MSI circuits apiece. After the nine boards of the multiplier had been debugged individually and at low speed using IML, we plugged them all into the processor and had the multiply instructions working at full speed about ten days later, this in spite of the fact that the multiplier was designed by five people.

Conclusions

Several conclusions can be drawn from this history in addition to the obvious ones about knowing when to stop designing and start manufacturing and the like. First, it is probably better to design a computer for a single application that you know very well than to design a computer for a number of them hoping for a bigger market. At least you will please a few customers in the first case, and the compromises you make in designing a more general purpose computer may wind up pleasing no one. In addition, you may be surprised to find that your single-application machine can do other things too; I think the experience of Floating Point Systems in this regard is especially interesting.

Second, it is not enough to merely use the best attainable technology when a large and innovative digital system is to be built. It is equally as important to make the system easy to maintain in a general sense: easy to manufacture, easy to test, easy to repair, and perhaps even easy to understand. Innovation in architecture can overcome problems stemming from slow parts, insufficient connector pins, or inadequate component densities, but cannot overcome deficiencies in implementation of that architecture.

Finally, it is not really necessary to know the exact architecture of the computer one is building before one starts working on it. Far more important is the maintenance of an open-minded approach to the problems one is trying to solve with the computer and a willingness to change the whole design if it seems appropriate. Most of the time that passes between the point of inception and the point of obsolescence of a computer system passes after the first prototype is running, and good architectures are surprisingly long-lived and fruitful assets in the marketplace. This is especially significant when compared to the rate at which advances in electronic technology change our implementations of those architectures.

References

[1] Flynn, M.J., "Some Computer Organizations and Their Effectiveness", IEEE-C21, (September, 1972).

[2] Lincoln, N.R., "It's Really Not As Much Fun Building a Supercomputer As It Is Simply Inventing One", Symposium on High Speed Computer and Algorithm Organization, (April, 1977).

[3] Smith, B.J., "A Pipelined, Shared Resource MIMD Computer", International Conference on Parallel Processing, (1978).

[4] Lord, R.E., Kowalik, J.S., and Kumar, S.P., "Solving Linear Algebraic Equations on a MIMD Computer", submitted to International Conference on Parallel Processing, (1980).

[5] Deo, N., Pang, C.Y., and Lord, R.E., "Two Parallel Algorithms For Shortest Path Problems", submitted to International Conference on Parallel Processing, (1980).

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

A PIPELINED, SHARED RESOURCE MIMD COMPUTER

BY DR. B.J. SMITH

Tomorrow's Computers. . . Today

A PIPELINED, SHARED RESOUCE MIMD COMPUTER

Burton J. Smith
Denelcor, Inc.
Denver, Colorado 80205

Abstract -- The HEP computer system currently being implemented by Denelcor, Inc., under contract to the U.S. Army Ballistics Research Laboratory is an MIMD machine of the shared resource type as defined by Flynn. In this type of organization, skeleton processors compete for execution resources in either space or time. In the HEP processor, spatial switching occurs between two queues of processes; one of these controls program memory, register memory, and the functional units while the other controls data memory. Multiple processors and data memories may be interconnected via a pipelined switch, and any register memory or data memory location may be used to synchronize two processes on a producer-consumer basis.

Overview

The HEP computer system currently being implemented by Denelcor, Inc., under contract to the U.S. Army Ballistics Research Laboratory is an MIMD machine of the shared resource type as defined by Flynn [1]. In this type of organization, skeleton processors compete for execution resources in either space or time. For example, the set of peripheral processors of the CDC 6600 [5] may be viewed as an MIMD machine implemented via the time-multiplexing of ten process states to one functional unit.

In a HEP processor, two queues are used to time-multiplex the process states. One of these provides input to a pipeline which fetches a three address instruction, decodes it, obtains the two operands, and sends the information to one of several pipelined function units where the operation is completed. In case the operation is a data memory access, the process state enters a second queue. This queue provides input to a pipelined switch which interconnects several data memory modules with several processors. When the memory access is complete, the process state is returned to the first queue. The processor organization is shown in Figure 1, and the over-all system layout appears in Figure 2.

Each processor of HEP can support up to 128 processes, and nominally begins execution of a new instruction (on behalf of some process) every 100 nanoseconds. The time required to completely execute an instruction is 800 ns, so that if at least eight totally independent processes, i.e. processes that do not share data, are executing in one processor the instruction execution rate is 10^7 instructions per second per processor. The first HEP system will have four processors and 128K words of data memory.

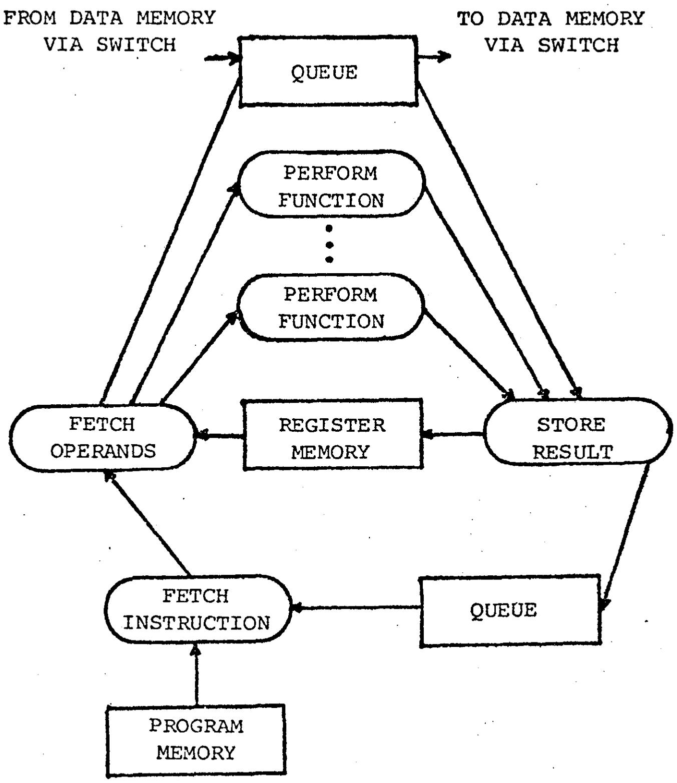


Figure 1. Processor Organization

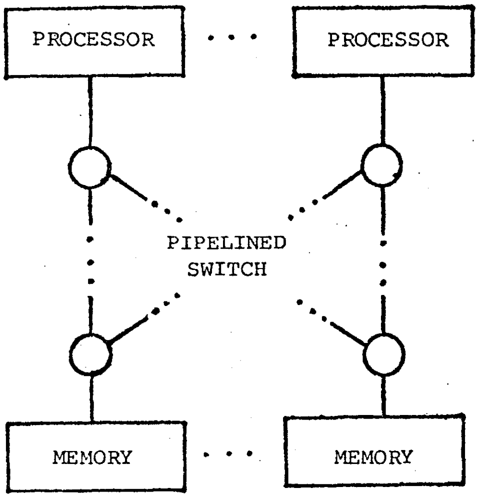


Figure 2. Overall System Layout

HEP instructions and data words are 64 bits wide. The floating point format is sign magnitude with a hexadecimal, seven-bit, excess-64 exponent. All functional units can support one instruction execution every 100 nanoseconds except the divider, which can support this rate momentarily but is slower on the average.

Tasks

Since HEP attains maximum speed when all of its processes are independent, a simple set of protection mechanisms is incorporated to allow potentially hostile users to execute simultaneously. A domain of protection in HEP is called a task, and consists of a set of processes with the same task identifier (TID) in their process state. The TID specifies a task status word which contains base and limit addresses defining the regions within the various memories accessible by the processes in that task. In this way, processes within a task may cooperate but are prevented from communicating with those in other tasks. Processes in different tasks or processors may communicate via data memory if they have an overlapping allocation there.

Processes are a scarce resource in HEP; in addition, the synchronization primitives used in HEP make processes difficult to virtualize. As a result, the maximum number of processes a task will use must be specified to the system when the task is loaded. It is the job of the operating system to insure that its total allocation of processes to tasks does not exceed the number available, so that a create fault (too many processes) can only occur when one or more tasks have created more processes than they were allocated. In this event, the offending task or tasks (not necessarily the task that actually caused the create fault) are removed from the processor.

Protection violations, create faults, and other error conditions arising within a process cause traps. A trap is the creation of a process executing in a supervisor task. There are a total of sixteen tasks available in each processor; eight of these are user tasks and the other eight are corresponding supervisor tasks. When any process in it, and a process is created in the corresponding supervisor task to handle the condition. This scheme is not used for create fault, however; a create fault suspends execution of all processes (regardless of task) except those actually handling the fault.

Create fault occurs before all processes have been used to allow any create instructions in progress within the pipeline to complete normally and to allow for the creation of the create fault handler process. All other traps in HEP are precise in the sense that no subsequent instructions will be executed from the offending task, a useful feature when one is trying to debug a concurrent algorithm.

Synchronization

The synchronization of processes in HEP is made simple by virtue of the fact that any register or data memory location can be used to synchronize two processes in a producer-consumer fashion. This requires three states in general: a reserved state to provide for mutual exclusion, a full state, and an empty state. The execution of an instruction tests the states of locations and modifies them in an indivisible manner; typically, an instruction tests its sources full and its destination empty. If any of these tests fails, the instruction is reattempted by the process on its next turn for servicing. If all tests succeed, the instruction is executed; the process sets both sources empty and the destination reserved. The operands from the sources are sent to the function unit, and the program counter in the process state is incremented. When the function unit eventually writes a result in the destination location that was specified in the instruction it sets the destination full. Provisions are made to test a destination full rather than empty, to preserve the state of a source, or to totally override the state of a source or destination with the proviso that a reserved state may not be overridden except by certain privileged instructions. Input-output synchronization is handled naturally by mapping I/O device registers into the data memory address space; an interrupt handler is just a process that is attempting to read an input location or write an output location. I/O device addresses are not relocated by the data memory base address and all I/O-addressed operations are privileged.

Switch

The switch that interconnects processors and data memories to allow memory sharing consists of a number of nodes connected via ports. Each node has three ports, and can simultaneously send and receive a message on each port. The messages contain the address of the recipient, the address of the originator, the operation to be performed by the recipient, and a priority. Each switch node receives a message on each of its three ports every 100 nanoseconds and attempts to retransmit each message on a port that will reduce the distance of that message from its recipient; a table mapping the recipient address into the number of a port that reduces distance is stored in each node for this purpose. If conflict for a port occurs, the node routes one of the contending messages correctly and the rest incorrectly. To help insure fairness, an incorrectly routed message has its priority incremented as it passes through the node, and preference is given in conflict situations to the message(s) with the highest priority.

The time required to complete a memory operation via the switch includes two message transmission times, one in each direction, since the success or failure of the operation (based on the state of the memory location, i.e. full or empty) must be reported back to the processor so that it can decide whether to reattempt the operation or not. The propagation delay through a node and its associated wiring is 50 nanoseconds. Since a message is distributed among two (or three) nodes at any instant, the switch must be two-colorable to avoid conflicts between the beginning of some message and the middle part of another. When the switch fills up due to a high conflict rate, misrouted massages begin to "leak" from the switch. Every originator is obliged to reinsert a leaking message into the switch in preference to inserting a new message. Special measures are taken when the priority value reaches its maximum in any message to avoid indefinite delays for such messages; a preferable scheme would have been to let priority be established by time of message creation except for the large number of bits required to specify it.

FORTRAN Extensions

Two extensions have been made to FORTRAN to allow the programmer to incorporate parallelism into his programs. First, subroutines whose names begin with "$" may execute in parallel with their callers, either by being CREATEd instead of CALLed or by executing a RESUME prior to a RETURN. Second, variables and arrays whose names begin with "$" may be used to transmit data between two processes via the full-empty discipline. A simple program to add the elements of an array $A is shown in Figure 3. The subroutines $INPUT and $OUTPUT perform obvious functions, and the subroutine $ADD does the work of adding up the elements. There are a total of 14 processes executing as a result of running the program.

```
C     ADD UP THE ELEMENTS OF
C     THE ARRAY $A
      REAL $A(1000),$S(10),$SUM
      INTEGER I
      CREATE $INPUT($A,1000)
      DO 10 I=1,10
      CREATE $ADD($A(100*I-99),$S(I),100)
   10 CONTINUE
      CREATE $ADD($S,$SUM,10)
      CREATE $OUTPUT($SUM,1)
      END

C     NOELTS ELEMENTS OF $V
C     ARE ADDED AND PLACED IN $ANS
      SUBROUTINE $ADD($V,$ANS,NOELTS)
      REAL $V(1),$ANS,TEMP
      INTEGER J, NOELTS
      TEMP=0.0
      DO 20 J=1,NOELTS
      TEMP=TEMP+$V(J)
   20 CONTINUE
      $ANS=TEMP
      RETURN
      END
```

Figure 3. HEP FORTRAN Example

Applications

As a parallel computer, HEP has an advantage over SIMD machines and more loosely coupled MIMD machines in two application areas. The first of these involves the solution of large systems of ordinary differential equations in simulating continuous systems. In this application, vector operations are difficult to apply because of the precedence constraints in the equations, and loosely coupled MIMD organizations are hard to use because a good partition of the problem to share workload and minimize communication is hard to find. Scheduling becomes relatively easier as the number of processes increases [3], and is quite simple when one has one process per instruction as in a data flow architecture [4].

A second type of application for which HEP seems to be well suited is the solution of partial differential equations for which the adjacencies of the discrete objects in the model change rapidly. Free surface and particle electrodynamics problems have this characteristic. The difficulty here is one of constantly having to rearrange the model within the computer to suit the connectivity implied by the architecture. Tightly coupled MIMD architectures have little implied connectivity. Associative SIMD architectures of the right kind may perform well on these problems, however.

Conclusion

The HEP system described above represents a compromise between the very tightly coupled data flow architectures and more loosely coupled multicomputer systems [2]. As a result, it has some of the advantages of each approach: It is relatively easy to implement parallel algorithms because any memory location can be used to synchronize two processes, and yet it is relatively inexpensive to implement large quantities of memory. In addition, the protection facilities make it possible to utilize the machine either as a multiprogrammed computer or as an MIMD computer.

References

[1] Flynn,M.J. "Some Computer Organizations and Their Effectiveness", IEEE-C21 (Sept. 1972).

[2] Jordan,H.F. "A Special Purpose Architecture for Finite Element Analysis", International Conference on Parallel Processing (1978).

[3] Lord,R.E. "Scheduling Recurrence Equations for Parallel Computation", Ph.D. Thesis, Dept. of Computer Science, Wash. State Univ. (1976).

[4] Rumbaugh,J. "A Data Flow Multiprocessor", IEEE-C26, p. 138 (Feb. 1977).

[5] Thornton,J.E. "Parallel Operation in the Control Data 6600", Proc. FJCC vol 26, part 2, p. 33 (1964).

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

A PARALLEL OPERATING SYSTEM FOR AN MIMD COMPUTER

BY DR. R. A. SCHMIDT

A PARALLEL OPERATING SYSTEM FOR AN MIMD COMPUTER

Rodney A. Schmidt
Denelcor, Inc.
Denver, Colorado 80205

Abstract -- The HEP computer system developed by Denelcor, Inc. under contract to the U.S. Army Ballistics Research Laboratory is an MIMD machine of the shared resouce type as defined by Flynn. In this type of organization, it is of paramount importance that the parallelism inherent in a user program not be compromised by serialization or deadlock in the operating system. The HEP operating system solves this problem by limiting its resource management activities through resource preallocation and subdivision of resources into separately managed pools.

Overview

The HEP computer system developed by Denelcor, Inc. under contract to the U.S. Army Ballistics Research Laboratory is an MIMD machine of the shared resource type as defined by Flynn [1]. The architecture of this machine has been covered earlier in a paper by Smith [2]. Briefly, processes in HEP reside within tasks, which define both a protection domain and an activitation state (dormant/active). Tasks reside within processors, all of which access a shared data memory. Multiple tasks may cooperate by sharing a common region in data memory. Cells in data memory have the property of being "full" or "empty" and the execution of instructions in processes may be synchronized by busy waiting (in hardware) on the full/empty state of data memory cells. Other than the state of data memory, processes and tasks in different processors have no means of synchronization or communication.

High-level language (e.g. FORTRAN) programs in this machine are explicitly parallel. Subprograms are made to run in parallel with the main program by an explicit CREATE statement analogous to CALL in ordinary FORTRAN. Code within a subprogram is SISD. The objective of the HEP operating system is to preserve the parallelism of the user program by executing in parallel during the performance of I/O and related supervisory functions. The operating system must:

1.) Allow all user processes to execute during I/O related supervisory computation;

2.) Allow multiple concurrent supervisory I/O computations;

3.) Allow reentrant use of code in the supervisor and the user program;

4.) Provide maximum user performance by consuming minimum resource in both time and space.

In SISD computers, reentrancy is usually obtained with some form of dynamic memory allocation. Concurrency of the operating system and the user is not possible due to the SISD nature of the machine.

In HEP, most dynamic memory allocation would generate considerable serialization of code around the resource lock required to safeguard the memory allocation data structure. In addition, HEP cannot allow any memory used by the system to be writeable by the user since the user is running truly in parallel with the system and could destroy any location at any time.

User Memory Management

In the HEP operating system, the available general purpose registers (about 2,000 of them) are divided a priori into groups of uniform length. When a process is created, the creating process must obtain a register environment from a table of available groups. This operation is relatively infrequent and inexpensive. All register environments are identical, and no state is retained in them.

Main memory (data memory) environments are obtained at the subprogram level by each subprogram as it is invoked. Space is obtained from a pool of data memory environments peculiar to that subprogram. The user must specify at link time how many such environments should be allocated for each subprogram. Control of an environment is obtained via a table of free environments, but the table is local to the subprogram. Thus, serialization for access to an environment is only between multiple, nearly simultaneous, invocations of the same subprogram, and is much less damaging to performance.

Data memory environments are a resource not visible to the user, and as such can contribute to deadlock problems. Given the user's ability to increase the amount of data memory resource allocated to a subprogram, the deadlock problem can be circumvented without much difficulty.

Concurrent I/O presents its own set of problems. In FORTRAN, a single I/O is implemented with multiple calls to I/O formatting services. State must be retained by the formatter during this process. This state is bound to the I/O unit, not the subprogram. Further, the amount of space required is not known until run time. Thus, some type of run time memory management is required, and the resource thus allocated is invisible to the user. The space must be allocated in an area accessible to all processors in a multi-processor job, so that all tasks may share the same I/O units.

The strategy employed in HEP is to allocate I/O buffers for a logical unit upon the first I/O to the unit. The space is then consumed for the duration of the program, even if the I/O unit is closed. If the I/O unit is re-opened for another file, the record length of the new file must be less than or equal to that of the old file. In this implementation, space can be allocated from a top-of-memory pointer which moves in only one direction. Serialization of processes occurs only on simultaneous first I/O operations, and only for the few microseconds required to move the pointer. This contrasts with the substantial serialization introduced by the normal scheme of a linked list of available space with garbage collection.

Consideration is being given to allowing a user to supply his own logical record buffer, with only the fixed portion of the I/O state held at the top of memory. This would allow the user greater dynamism in the logical record size, at the expense of managing his own resources.

Supervisor Memory Management

HEP supervisors require two types of dynamic memory: registers to use while copying logical records to/from physical records, and data memory to hold file parameters for open files. Of these, the register allocation is the simplest. Since the users register requirement can be determined from the number of processes requested (a control card parameter), all remaining registers in the register memory partition can be used for supervisor I/O operations. These registers are allocated from a bit table to active I/O operations.

Data memory allocation is more difficult. It is not known until run time how many files will be used, or how much logical record buffer space will be required by the user. Fortunately, the amount of supervisor space required per open file is constant. The operating system merely allocates supervisor space for enough files to accomodate the larger system programs (compiler, etc.) and leaves the remaining space for the user. The default limit on open files may be overridden with a control card for users with special requirements.

Future Directions

The present HEP system provides a high-performance low overhead environment for parallel computational activities. Our next activity will be to extend this capability with high-performance parallel I/O operation with speed comparable to our processing speeds. The parallel file system will include such features as record interlock within files and concurrent read/write capability from multiple jobs to the same file.

References

[1] Flynn, M.J., "Some Computer Organizations and Their Effectiveness", IEEE-C21, (September, 1972).

[2] Smith, B.J., " A Pipelined, Shared Resource MIMD Computer", Proc. of the 1978 International Conference on Parallel Processing (1978), pp 6-8.

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

A COMPARISON OF HEP AND VECTOR MACHINES
BY APPLICATION AREA

BY DR. B. J. SMITH

A COMPARISON OF HEP AND VECTOR MACHINES BY APPLICATION AREA

1. ORDINARY DIFFERENTIAL EQUATIONS

In this area of application, the utility of vector processing depends primarily on the similarities among the expressions that define the derivative vector. In the linear case, the vector machine performs very well; in the nonlinear case, each of the derivative expressions is typically unique. HEP was originally designed to attack this problem, and solves it easily assuming a reasonable scheduling algorithm to assign operations to processors. Vector machines are relatively useless for this class of problem because of two difficulties, namely a) scheduling the processor so that vector operations (especially add and multiply) occur in a suitable sequence, and b) addressing randomly located vector operands.

2. LINEAR ALGEBRA

This application area is a traditional strong point for vector architectures. If the matrices being manipulated are dense, then a vector machine should perform well. HEP also performs well in this case, since multiple processes executing identical programs on different rows or columns of an array can yield the maximum speed of which HEP is capable, i.e. 10^7 operations/second per processor. In the sparse matrix case, HEP has an advantage over vector machines in that the search for an appropriate array element can be done simultaneously and independently by a set of HEP processes, irrespective of the lengths of the searches, whereas a vector search of a number of rows or columns of a sparse array may result in a decrease in vector utilization due to masking while the last few elements are being found. The analogous problem in HEP is easily circumvented because processes that are through searching can acquire and search a new row or column.

3. IMPLICIT TECHNIQUES

For relatively simple classes of problems, a vector architecture can deal with relaxation effectively. Any of the following attributes, however, make it much more difficult to use a vector approach for the reasons indicated. First, arbitrary functions or any sort of conditional expression evaluation at the grid points will mask out vector elements and reduce efficiency. Second, any variability of connectivity in the problem such as might be caused by boundary motion will result in an addressing problem for a vector machine. Third, a complicated connectivity in general, irrespective of variability, will also give rise to vector addressing difficulties. HEP has no such problems, since each process can independently branch to a different spot in its program and can evaluate any address required to deal with variable or complicated geometries.

4. HEURISTIC PROGRAMMING

Many important problems in computing have the property that they require a prohibitively long time to solve completely on <u>any</u> computer, but approximate solutions can be discovered much more quickly using the techniques of heuristic programming. The basic approach is to devote the most computational effort to the most promising potential solutions. The usual implementation of this scheme on a single-process computer, vector or otherwise, is to remember alternatives to guesses made by the program and to explore those alternatives only after the current guess has been exhausted or seems "unpromising". HEP can be used to speed up this procedure; one can create a process to explore each possibility and let each process decide whether its own alternative is indeed promising or not. This approach may not be efficient on a single-process computer because of the overhead associated with changing the process that the processor is executing.

5. MULTIPASS ALGORITHMS

Many computations can be decomposed into several different passes or phases, each of which performs part of the work. Compilation and assembly, image processing, and data reduction are examples. Whereas vector machines are incapable of exploiting this potential parallelism, HEP can be used to execute all phases simultaneously by using a process to implement each phase and transmitting data between the phases. (This technique is sometimes called "macropipelining".) A compiler, for example, could be subdivided into a lexical analysis process, a parsing process, a semantics process, several optimization processes, and a code generation process. Current compiler design methodology often gives rise to this kind of a decomposition except that the subroutines or coroutines used to implement the phases do not in fact run in parallel.

6. EXISTING PROGRAMS

Much effort has been expended in attempting to detect and exploit parallelism in existing programs. Vector processors can be used to speed up loops of certain kinds by executing vector instructions that have the same effect that multiple executions of the loop would have. Unfortunately, much of the code in existing software is not subject to this kind of speedup, often because many kinds of loops are not "vectorizable". HEP, on the other hand, can not only exploit the vectorizable loops but can also execute a sequence of statements in parallel even when those statements do not involve vectors at all, because the dependencies among statements are not really very numerous. This technique is used to some extent in computers with "instruction lookahead", but the potential inherent in the HEP architecture is far greater because "lookahead" is not limited by availability of functional units or instruction stack size.

7. DATA BASE MANAGEMENT

Most of the operations of data base management exhibit a high degree of potential parallelism. Sorting, searching, and set-theoretic operations all can be sped up, but multiple I/O streams are required since only a small part of a data base will fit in primary memory. Moreover, vector operations are inappropriate since the data are variable length character strings. A number of HEP processes can perform I/O concurrently and search or sort in parallel using any number of published algorithms known to be suitable for MIMD machines such as HEP. Unlike most vector machines, HEP can address its memory a character at a time to facilitate string operations. The unmatched speed of the HEP I/O system should make it exceptionally attractive for data base applications, especially where numeric computations are to be done on the retrieved data.

8. MULTIPROGRAMMING

It is often useful to be able to execute many user jobs simultaneously on a computer. Machines which execute only one instruction at a time, vector machines included, accomplish this by switching the processor between jobs periodically. There is some overhead associated with this switching operation, depending primarily on how many registers of the processor must be saved and reloaded. There is no overhead whatsoever incurred by this activity on HEP. In fact, a good way to achieve speed via parallelism is merely to run multiple jobs. HEP provides protection hardware to prevent interference among the jobs, and at the same time offers all of the flexibility and resources associated with a parallel processor to each executing program.

9. MODULARITY

A HEP computer contains from one to fourteen processors, each of which executes 10^7 instructions per second, and has a data memory size ranging from 256K bytes up to two billion bytes. Expansion in the field is readily accomplished; moreover, failure of a single processor or memory merely results in decreased capacity of the system until repair is accomplished. This kind of modularity is not within the capabilities of vector machines; it is not possible to buy 1/2 of a vector processor nor is it possible to interconnect several such processors to obtain longer vectors. If a vector processor fails, the entire system is out of commission until the repair is accomplished. While error correction in the memory postpones the necessity for repair in both HEP and in vector machines, the requirement to repair a failing memory module brings the vector machine down but only reduces the performance of HEP temporarily.

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

STANDARD SYNCHRONIZATIONS IN HEP FORTRAN

BY DR. H.F. JORDAN

Tomorrow's Computers. . . Today

Standard Synchronizations in HEP Fortran

Harry F. Jordan

The basic synchronization mechanism supplied by HEP Fortran is that of elementary producer-consumer synchronization using busy waiting. This mechanism is accessed via the so-called asynchronous variables, the names of which begin with the $ symbol. With each such variable a state of FULL or EMPTY is associated so that reading (use in a right hand side expression or as a subscript) may only take place when the state is EMPTY and writing (assignment) may only take place when the state is FULL. Writing an asynchronous variable always sets the state to FULL and reading sets it to EMPTY with only a few exceptions. The PURGE statement may be used to set the state of one or more asynchronous variables to EMPTY regardless of previous state.

The elementary producer-consumer synchronization consisting of "wait until empty and then write" and "wait until full and then read" can be augmented by the passive logical functions FULL(a) and EMPTY(a) which test, but do not alter, the state of an asynchronous variable a. Furthermore, when an asynchronous variable appears inside the logical expression controlling an IF statement a wait until the state is FULL occurs but the state is not set to EMPTY when the expression is evaluated. The latter behavior can also be obtained within a right hand side expression or an index expression by use of the built in function SAVE(a) which delivers the value of an asynchronous variable a when it becomes full but does not set it empty.

Several types of synchronization other than single value produce and consume are useful in programming a multiple process machine such as HEP. Below we will treat several of the more important ones and exhibit their implementation using HEP Fortran. In the code given, a quoted string represents a manifest constant, usually an array dimension, which is to be replaced by a constant in any specific application of the code.

It is often necessary to apply producer-consumer synchroniation to a block of information so that no part of it is used until all of it has been written and no part of it can be written until all of it has been read. For the simple case of one producer and one consumer a straightforward implementation requires logical variables $EMPTY ,E, $FULL and F and appears as follows:

```
                          Initialization

                        PURGE $EMPTY, $FULL
                        $EMPTY = .TRUE.

     Producer                                      Consumer
   E = $EMPTY                                    F = $FULL
   Write block                                   Read block
   $FULL = .TRUE.                                $EMPTY = .TRUE.
```

Note that the values of the synchronizing variables $FULL and $EMPTY are unimportant. Only the state of the variable plays a role in the synchronization.

By using more code it is possible to do the above synchronization with only one asynchronous variable both the state and value of which are used in the synchronization.

Initialization

```
PURGE $FULL
$FULL = .FALSE.
```

```
        Producer                                    Consumer

10 FP = $FULL                             10 FC = $FULL
   IF (.NOT. FP) GO TO 20                    IF (FC) GO TO 20
   $FULL = FP                                $FULL = FC
   GO TO 10                                  GO TO 10
20 CONTINUE                               20 CONTINUE
   Write block                               Read block
   $FULL = .TRUE.                            $FULL = .FALSE.
```

The first solution is not only more straightforward but also easily expandable to the case of several producers and several consumers acting on a buffer with space for N blocks of data. In this case the synchroninzing variables become integers the values of which give the numbers of full and empty blocks in the buffer.

Initialization

```
PURGE $IFULL, $IEMPTY
$IEMPTY = "N"
```

```
      Each Producer                          Each Consumer
IE = $IEMPTY-1                         IF = $IFULL-1
IF (IE .NE. 0) $IEMPTY = IE            IF (IF .NE. 0) $IFULL = IF
Write block N-IE                       Read block IF+1
IF = $IFULL                            IE = $IEMPTY
$FULL = IF+1                           $IEMPTY = IE+1
```

In this code a producer fills the first empty buffer block and a consumer empties the last full block. We will consider the implementation of a first-in first-out strategy after treating the simpler concept of a critical section.

Critical sections of code executed by two or more parallel processes exclude each other in time. Processes may execute critical sections in any order but only one process at a time may be within a critical section. Either a single critical section of program may be shared by several processes or processes may execute distinct code sections. In HEP a section of code which begins by reading a given asynchronous variable and ends by writing it is a critical section with respect to any other code segment beginning with a read and ending with a write of the same asynchronous variable. Care should be taken in coding parallel processes for HEP that no more statements than necessary be placed between the read and subsequent write of an asynchronous variable since all processes sharing this code will run one at a time through this critical section whether or not that is the intent.

The producer-consumer synchronization on a multiple element buffer usually involves First In-First Out access to the individual items. A FIFO structure is usually implemented in software as a circular buffer. The simplest implementation uses critical sections to make the operation of inserting a new element into the FIFO (PUT) atomic with respect to the operation which extracts an element (GET). The critical section may be made a side effect of access to a variable needed to manipulate the FIFO in any case. This is the technique used below where the only asynchronous variable is $IN.

```
      BLOCK DATA

C  THE INITIAL STATE AND SIZE CONSTANTS FOR THE FIFO.
      INTEGER $IN ,OUT, LIM
      COMMON /FIFO/ $IN ,OUT, LIM, A("SIZE")
      DATA $IN , OUT, LIM /1,1, "SIZE"/
      END

      SUBROUTINE PUT(V, FULL)

C  PUT THE VALUE V INTO THE FIRST FREE SPACE IN THE FIFO
C     RETURNING FULL AS .FALSE. IF THE FIFO IS FULL PERFORM
C     NO OPERATION AND RETURN FULL AS .TRUE.
      LOGICAL FULL
      INTEGER $IN ,OUT, LIM
      COMMON /FIFO/ $IN ,OUT,LIM, A("SIZE")
      I = $IN
      IDIF = I-OUT
      IF (IDIF .LT. 0) IDIF = IDIF+LIM
      IF (IDIF .EQ. LIM-1) GO TO 10
      A(I) = V
      $IN = MOD(I, LIM)+1
      FULL = .FALSE.
      RETURN
10    $IN = I
      FULL = .TRUE.
      RETURN
      END

      FUNCTION GET(EMPTY)

C  THE FUNCTION RETURNS THE VALUE OF THE NEXT AVAILABLE FIFO
C  ELEMENT AND SETS EMPTY TO .FALSE. UNLESS THE FIFO IS EMPTY
C  IN WHICH CASE THE ONLY ACTION IS TO SET EMPTY TO .TRUE.

      LOGICAL EMPTY
      INTEGER $IN ,OUT, LIM
      COMMON /FIFO/ $IN ,OUT,LIM, A("SIZE")
      I = $IN
      IF (I .EQ. OUT) GO TO 10
      GET = A(OUT)
      OUT = MOD(OUT, LIM)+1
      $IN = I
      EMPTY = .FALSE.
      RETURN
10    $IN = I
      EMPTY = .TRUE.
      RETURN
      END
```

In the FIFO implementation above only one asynchronous variable is necessary to bound the critical sections. If it is desired to implement a FIFO the elements of which are larger than single values then a different approach can be used to increase the potential parallelism. In this case define integer valued functions IPUT(FULL) and IGET(EMPTY) which return an index to the next free FIFO space or the next available FIFO element, respectively. These indices can be used by the calling program to read or write elements of the FIFO outside of the critical sections associated with testing and updating the pointers. In this case, however, it is possible that parallel use of the FIFO by other processes may cause the value of $IN (OUT) to catch up to some previous value of OUT ($IN) which has not yet been used to access the FIFO element completely. Synchronization can be maintained in this case by making all variables of a FIFO element asynchronous. The time involved in accessing a FIFO element is thus removed from the critical section, which blocks all parallel access to the FIFO, and conflict is limited to the one other process which actually requires the same memory cells.

Another important synchronization is that of FORK and JOIN, in which a single instruction stream initiates the execution of ("forks into") several parallel instruction streams. After all of the parallel streams have reached a prescribed point at which parallel execution is to end, all but one stream are terminated and this single "joined" instruction stream is free to proceed. In HEP Fortran a CREATE operation is used to initiate a parallel instruction stream which will terminate when a RETURN statement is encountered. A CALL statement does a normal transfer of control to a code segment which will return control to the calling point when a RETURN statement is encountered. Thus several parallel instruction streams, all but one of which will eventually terminate (assuming no infinite loops), can be produced by a series of CREATE operations followed by a single CALL. The only difficulty is that the "live" code segment (the one invoked by a CALL) may finish before all the other instruction streams have terminated. A counter and reporting variable are used to determine that all parallel streams have reached the JOIN point in the code below which forks a single stream into N parallel executions of the subroutine PROC.

```
     Single Stream
              .
              .
    PURGE $IC, $FINISH
    $IC = 1
    DO 10 I = 1, N-1
    CREATE PROC(···)                  FORK operation
10  $IC = $IC+1
    CALL PROC(···)
    F = $FINISH                       JOIN operation
              .
              .
    Multiply Executed
       Process
    SUBROUTINE PROC(...)
              .
              .
    I = $IC-1
    IF (I .EQ. 0) GO TO 20
    $IC = I                           JOIN operation
    RETURN
20  $FINISH = .TRUE.
    RETURN
    END
```

Another well known synchronization is that of readers and writers on a shared data structure. Readers are defined to be processes which do not alter the overall structure of the data during their access to it. They may perform atomic write operations which do not alter the structure, and in HEP they may even perform read-modify-write operations on asynchronous variables provided the structure remains consistent. A writer alters the data structure during its access so that the structure can only be assumed consistent at the end of the writer access. A well known example is that of garbage collection or compaction of a dynamic data structure. An arbitrary number of processes may use the data structure at a time as readers, but the compaction process must have exclusive access.

In the first version of the synchronization the first reader locks the structure against access by the writer and the last reader unlocks it. The writer may have to wait indefinitely for a sequence of readers.

Initialization

```
PURGE $ACCESS, $NREAD
$ACCESS = .TRUE.
$NREAD = 0
```

Reader

```
      IR = $NREAD
      IF (IR .EQ. 0) A = $ACCESS
      $NREAD = IR+1
C     DO THE READ ACCESS HERE.
      IR = $NREAD
      IF (IR .EQ. 1) $ACCESS = A
      $NREAD = IR-1
```

Writer

```
      A = $ACCESS
C     DO THE WRITE ACCESS HERE.
      $ACCESS = A
```

The second version of this synchronization ensures that no new readers may gain access to the buffer once a writer has requested access. The extra condition is handled by keeping a count of the number of writers which have requested use of the data structure.

```
          Initialization
      PURGE $ACCESS ,$NREAD, $NWRITE
      $ACCESS = .TRUE.
      $NREAD = 0
      $NWRITE = 0
```

```
          Reader
10    IF ($NWRITE .GT. 0) GO TO 10
      IR = $NREAD
      IF (IR .EQ. 0) A = $ACCESS
      $NREAD = IR+1
C     DO READ ACCESS HERE
      IR = $NREAD
      IF (IR .EQ. 1) $ACCESS = A
      $NREAD = IR-1
```

```
          Writer
      $NWRITE = $NWRITE+1
      A = $ACCESS
C     DO WRITE ACCESS HERE.
      $ACCESS = A
      $NWRITE = $NWRITE-1
```

This solution uses the passive (wait for full but do not set empty) access mechanism of the IF statement so that the testing of $NWRITE by a reader cannot lock the variable against access by a writer. A disadvantage of this solution is that if several readers are executing statement 10 they make no progress but occupy time slots in the process queue thus possibly reducing the overall machine throughput. This disadvantage can be reduced by preventing more than one reader from executing statement 10 at a time by placing it within a critical section. If $RMECH is initially full, then the statements R = $RMECH before line 10 and $RMECH = R after it will do the job.

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

SOLVING LINEAR ALGEBRAIC EQUATIONS ON AN MIMD COMPUTER

BY DR. R. E. LORD
DR. J. S. KOWALIK AND DR. S. P. KUMAR

Tomorrow's Computers. . . Today

SOLVING LINEAR ALGEBRAIC EQUATIONS
ON A MIMD COMPUTER

R. E. Lord
J. S. Kowalik
S. P. Kumar

CS-80-058

ABSTRACT: Two practical parallel algorithms for solving systems of dense linear equations are presented. They are based on Gaussian elimination and Givens transformations. The algorithms are numerically stable and have been tested on a MIMD computer.

KEY WORDS AND PHRASES: parallel algorithms, MIMD computer, linear equations, Gaussian elimination, Givens transformation.

CR CATEGORIES: 5.14, 5.25

1. Introduction

The problem of solving a set of linear algebraic equations is one of the central problems in computational mathematics and computer science. Excellent numerical methods solving this problem on uniprocessor systems have been developed, and many reliable and high quality codes are available for different cases of linear systems. On the other hand, the methods for solving linear equations on parallel computers are still in the conceptual stage, although some basic ideas have already emerged. The current state of the art in parallel numerical linear algebra is well described by Heller [3] and Sameh and Kuck [5].

Our investigation of methods for solving systems of dense linear equations on a MIMD computer includes Gaussian elimination with partial pivoting and Givens transformations. The first algorithm is commonly used to solve square systems of equations, the second produces orthogonal decomposition used in several problems of numerical analysis including linear least squares problems. We focus our attention on the cases where the number of available processors is between 2 and O(n), n being the number of linear equations. We take the view that is not presently realistic to assume that $O(n^2)$ processors will be soon available to solve sizable sets of equations. To verify our analytic results we have used a parallel computer manufactured by Denelcor Co. This computer, called HEP (Heterogeneous Element Processor), is a MIMD machine of the shared resource type as defined by Flynn.

2. Gaussian Elimination

If we consider a step to be either a multiplication and a subtraction, or a compare and multiplication then sequential programs for producing the LU decomposition of an n x n nonsingular matrix requires $T_1 = \frac{n^3}{3} + O(n^2)$ steps. The parallel method using $p \doteq (n-1)^2$ processors and partial pivoting requires $T_p = O(n \log n)$ steps. Thus the efficiency of such method for large n will be

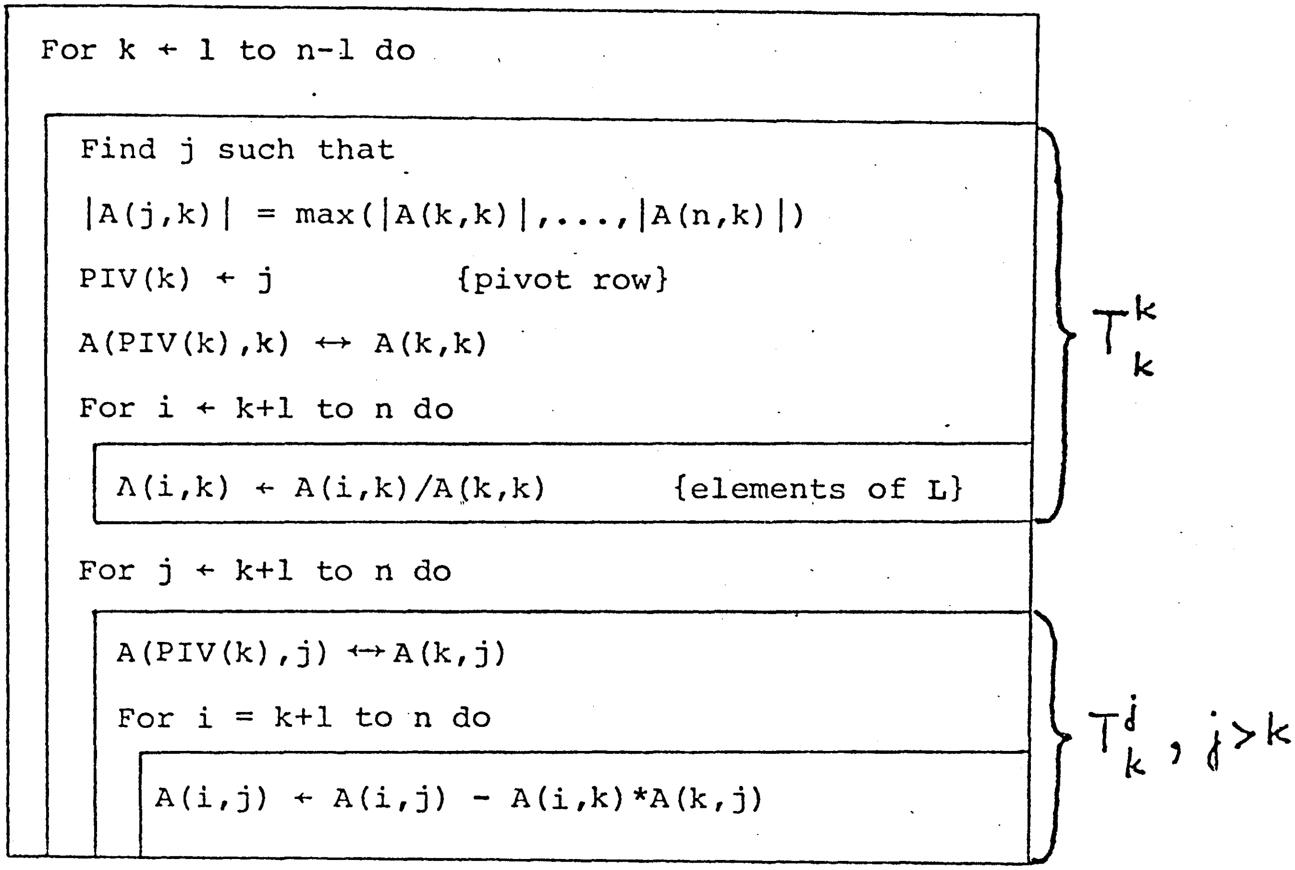
$$E_p = \frac{T_1}{T_p \cdot p} = \frac{1}{O(\log n)} .$$

Even if the cost of each processor in a parallel system is substantially less than current processor costs, this method will be economically unfeasible for n sufficiently large. We further observe that parallel computers which are or soon will be available will not provide n^2 processors for reasonable values of n. Thus, we restrict our attention to the case where the number of processors is in the range from 2 to O(n).

The algorithm which we present provides the LU decomposition of an n x n nonsingular matrix A using from 1 to $\lceil \frac{n}{2} \rceil$ processors and has an efficiency of 2/3 when $P = \lceil \frac{n}{2} \rceil$.

Consider the sequential program for LU decomposition with partial pivoting given in Fig. 1. In this program we shall consider a task to be that code segment which works on a particular column j for a particular value of k. We will denote these tasks by $J = \{T_k^j \mid 1 \le k \le j \le n, k \le n-1\}$.

Program LUDECOMP (A(n,n)).



```
For k ← 1 to n-1 do
    Find j such that
    |A(j,k)| = max(|A(k,k)|,...,|A(n,k)|)
    PIV(k) ← j            {pivot row}
    A(PIV(k),k) ↔ A(k,k)
    For i ← k+1 to n do
        A(i,k) ← A(i,k)/A(k,k)       {elements of L}
    For j ← k+1 to n do
        A(PIV(k),j) ↔ A(k,j)
        For i = k+1 to n do
            A(i,j) ← A(i,j) - A(i,k)*A(k,j)
```

Figure 1. Program for LU decomposition with illustration of tasks.

The precedence constraints imposed by the sequential program are

$$\lessdot = \{(T_k^j, T_m^\ell) \mid j<\ell \text{ or } k<m\}.$$

Thus, $C = (J, \lessdot)$ is the task system which represents the sequential program (Coffman, Denning [1]). The range and domain of these tasks are:

$$R(T_k^j) = \{A(i,j) \mid k \le i \le n\}$$

$$D(T_k^j) = \{A(i,j) \mid k \le i \le n\} \cup \{A(i,k) \mid k \le i \le n\}$$

and from this we can observe that, for example

$$\{T_k^{k+1}, T_k^{k+2}, \ldots., T_k^n\}$$

are all mutually noninterfering tasks and could be executed in parallel. More specifically we observe that $C' = (\mathcal{J}, \lessdot')$ where $\lessdot'$ is the transitive closure on the relation

$$X = \{(T_k^k, T_k^j) \mid k<j \le n\} \cup \{(T_k^j, T_{k+1}^j) \mid k<j \le n\}$$

is a maximally parallel system equivalent to C. This system is illustrated in Fig. 2.

Given the task system C' we now determine the execution time of the tasks and from that determine a schedule. We assume that one multiply and one subtract, or one multiply and one compare constitute a time step. Thus, neglecting any

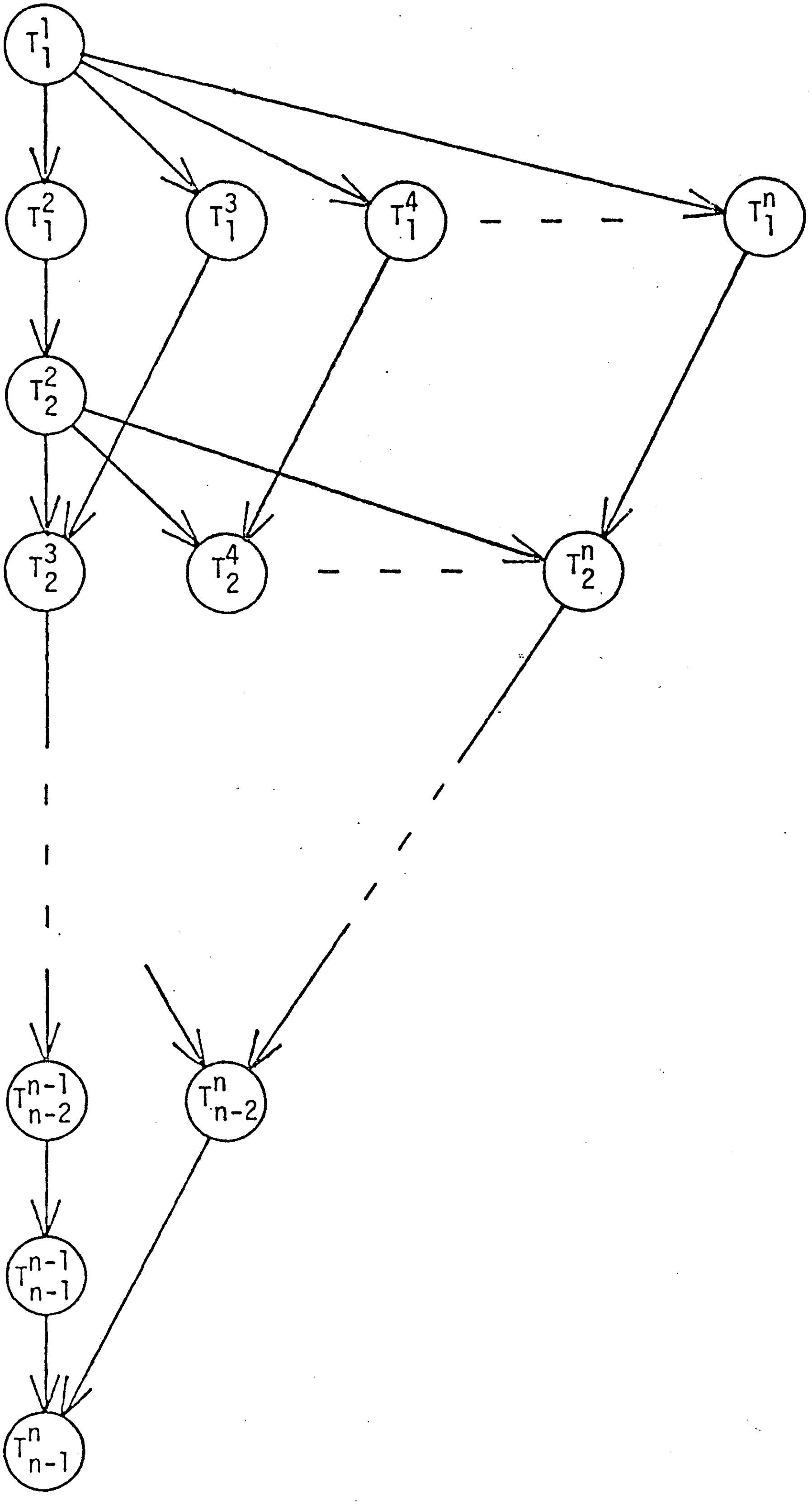


Figure 2: Maximally Parallel Task System Equivalent to C

overhead for loop control, the execution time $W(T_k^j)$ for each of the tasks is given by:

$$W(T_k^j) = \begin{cases} n+1-k & \text{if } k=j \\ n-k & k<j\,. \end{cases}$$

Treating the task system C' together with $W(T_k^j)$ as a weighted graph we observe that the longest path traverses the nodes: T_1^1, T_1^2, T_2^2, T_2^3, T_3^3,---, T_{n-1}^{n-1}, T_{n-1}^{n},. We will denote this path as S_1 and the length of the path by $L(S_1)$.

$$L(S_1) = n+1 + 2\sum_{j=2}^{n-1} j = n^2-1$$

Since the problem cannot be solved in time shorter than this path length we developed a schedule where the tasks constituting S_1 are assigned to processor 1 and the remaining tasks are assigned to $\left\lceil \frac{n}{2} \right\rceil - 1$ additional processors. Processor 2 will execute the tasks

$$T_1^3,\ T_1^4,\ T_2^4,\ T_2^5,\ T_3^5,\ \ldots,\ T_{n-2}^n$$

and, more generally, processor j will execute the tasks

$$T_1^{2j-1},\ T_1^{2j},\ T_2^{2j},\ T_2^{2j+1},\ \ldots,\ T_{n-2(j-1)}^n$$

and we will denote this as S_j. Note that this is not a path through the graph. For the case where n is even this schedule is illustrated in Fig. 3. Since this schedule has length n^2-1, the length of the longest path, then this schedule is optimal for n/2 processors. Using this schedule we note that:

$$\lim_{n\to\infty} S_p/p = \lim_{n\to\infty} \frac{n^3/3 + 0(n^2)}{(n^2-1)\ n/2} = \frac{2}{3}$$

and this efficiency is achieved to within 2% for relatively small n $(n \geq 50)$.

We now examine the question as to whether a schedule of length n^2-1 is achievable with fewer than n/2 processors. From the task system C as illustrated in Fig. 2 we note that task T_1^1 is a predecessor to all tasks and has an execution time of n steps. Consequently, any schedule for this system will have only one processor doing work during the first n steps. Similarly, T_{n-1}^n is the successor of all tasks and thus during the last time step only one processor can be doing work. Task T_{n-2}^{n-1} has all tasks except $\{T_{n-1}^{n-1}\} \cup \{T_j^n \mid 1 \leq j \leq n-1\}$ as predecessors, task T_{n-1}^{n-1} is a successor task and for the tasks $\{T_j^n \mid 1 \leq j \leq n-1\}$ each is a successor or predecessor of all other tasks in the set. Thus, for any schedule from the time that T_{n-2}^{n-1} commences execution, no more than 2·processors can be doing work. By similar argument, once T_{n-j}^{n-j+1} commences execution no more than j processors can be doing work. From this, we define the "computational area" of any schedule of C to be the product of the number of processors and the schedule length less the area where not all the processors can be doing work.

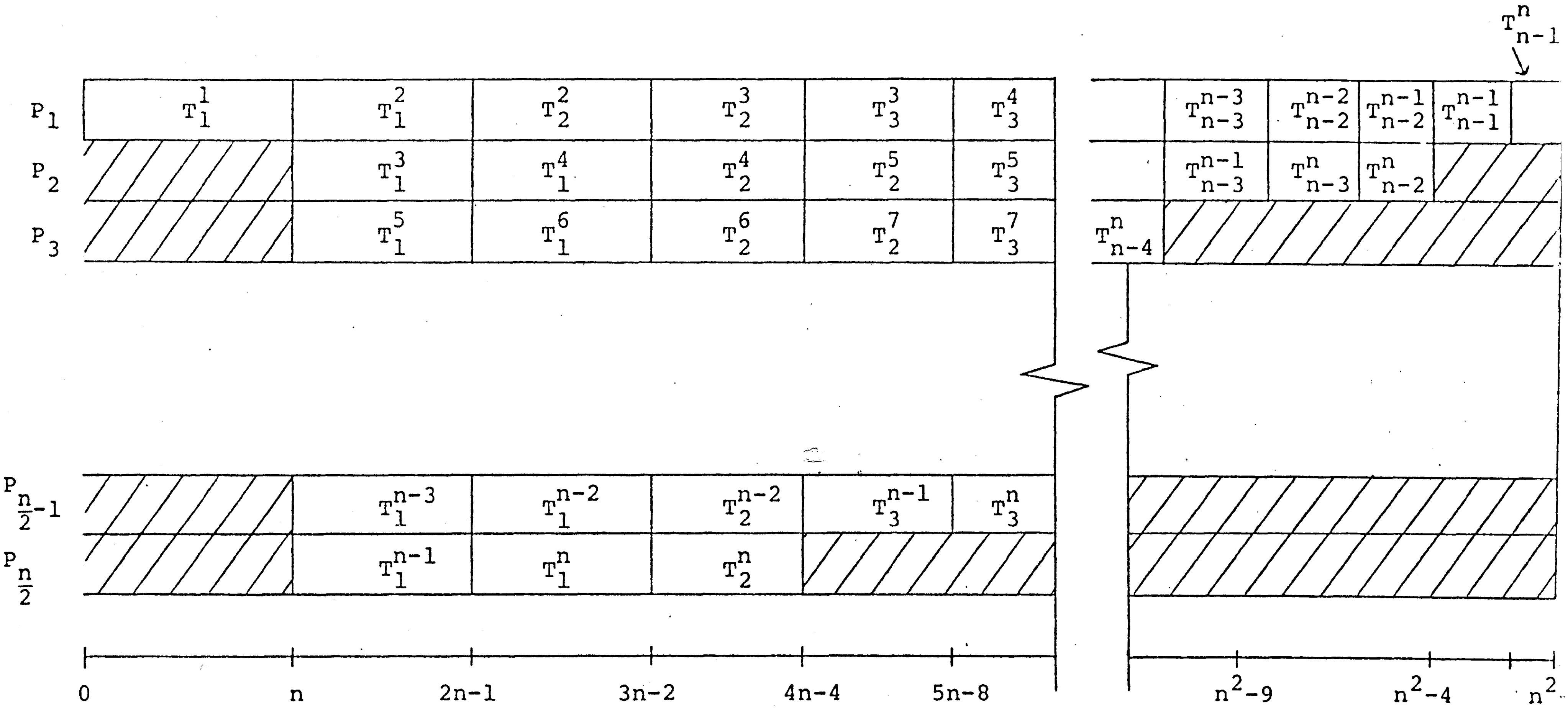


Figure 3. Schedule using $\frac{n}{2}$ processors (n even).

Specifically, for a schedule of length n^2-1 using p processors we have

$$CA = (n^2-1)p - (p-1) \cdot n-2\sum_{j=2}^{p-1} (p-j)j - (p-1)$$

$$= (n^2-1)p - (p-1)(n-1) - (p^3-p)/3.$$

The total amount of work (sum of the task weights) for the task system C is

$$TW = \frac{n^3}{3} + \frac{2n}{3} - 1.$$

Thus, a lower bound on the number of processors required to achieve a schedule of length n^2-1 is the smallest p for which $CA \geq TW$. For small even values of n the minimum P values are:

$2 \leq n \leq 8$	$p=n/2$
$10 \leq n \leq 14$	$p=n/2-1$
$16 \leq n \leq 22$	$p=n/2-2$
$24 \leq n \leq 28$	$p=n/2-3$
$30 \leq n \leq 34$	$p=n/2-4$
$36 \leq n$	$p \leq n/2-5$

For large values of n let $P=\alpha n$ and determine α such that

$$\lim_{n \to \infty} (CA/TW) = 1$$

Thus, an α to satisfy the above limit is a solution to:

$$3\alpha - \alpha^3 = 1$$

and an approximate solution to this is $\alpha = .34729$.

We note that this is only a lower bound and we do not know if it is achievable in general, however for n=10 we have found a schedule of length n^2-1 using n/2-1 processors and for n=16 a schedule using n/2-2 processors. The schedule for n=10 is shown in Fig. 4.

Should this lower bound be achievable then the efficiency for large n and using αn processors would be

$$\lim_{n\to\infty} (s_p/p) = \frac{n^3/3}{(n^2-1)\alpha n} = \frac{1}{3\alpha} \cong .9598.$$

Achievable Schedules

We now consider schedules similar to the one shown in Fig. 3 where the number of processors p is fewer than $\lceil n/2 \rceil$. The method we use is to assign to processor j the tasks comprising S_j, S_{j+p}, ..., S_{j+lp}. Where l is the largest integer such that $j+lp \le \left\lceil \frac{n}{2} \right\rceil$. The sequence of task assignments is such that the precedence constraints of the task system C' are meet.

Consider first the case when $n/4 \le p \le n/2$ so that processor 1 processes only the tasks of path S_1 and the tasks of S_{1+p}. This schedule will thus have length L equal to:

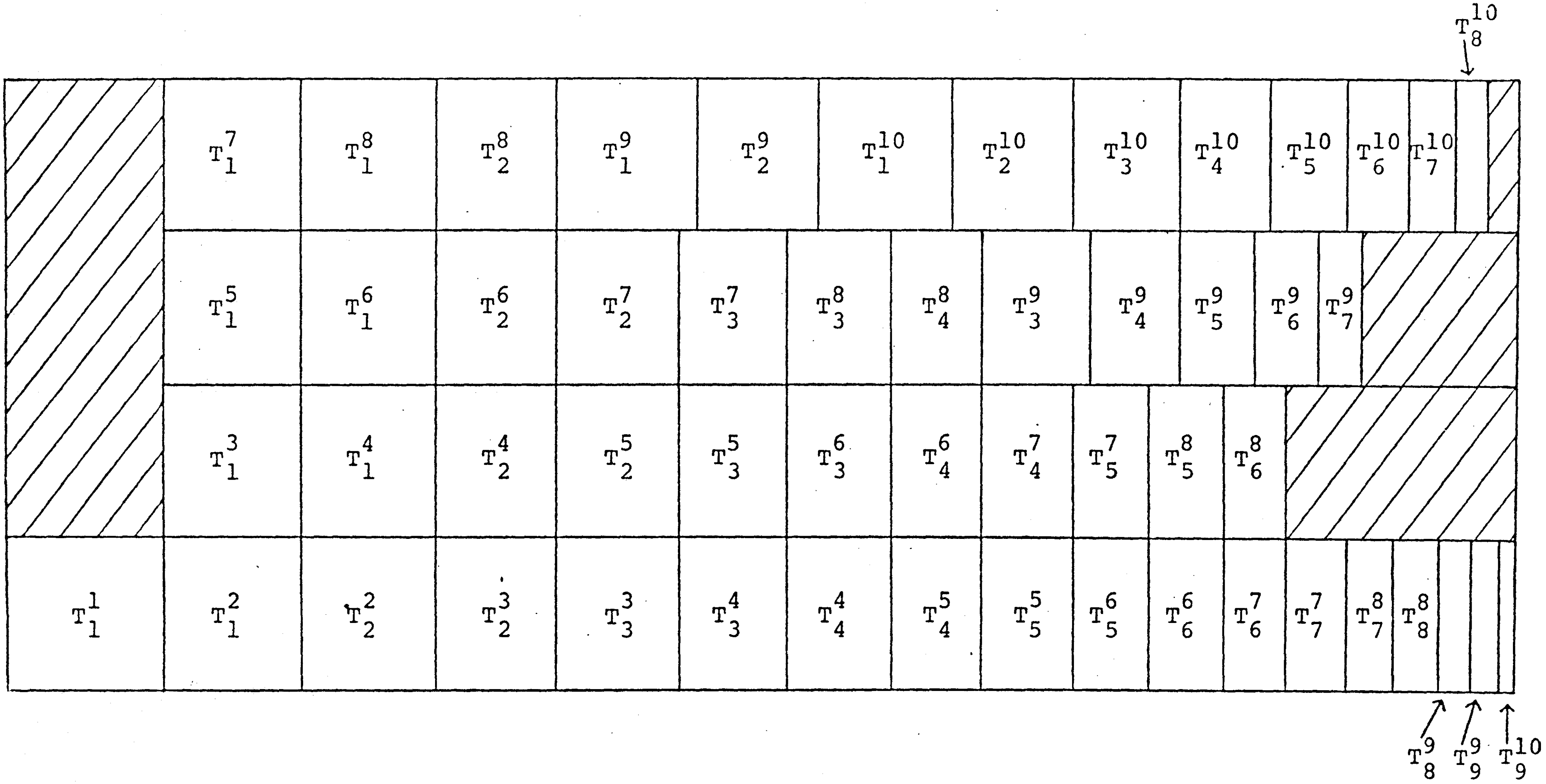

T^{10}_{8}
T^{7}_{1} T^{8}_{1} T^{8}_{2} T^{9}_{1} T^{9}_{2} T^{10}_{1} T^{10}_{2} T^{10}_{3} T^{10}_{4} T^{10}_{5} T^{10}_{6} T^{10}_{7}
T^{5}_{1} T^{6}_{1} T^{6}_{2} T^{7}_{2} T^{7}_{3} T^{8}_{3} T^{8}_{4} T^{9}_{3} T^{9}_{4} T^{9}_{5} T^{9}_{6} T^{9}_{7}
T^{3}_{1} T^{4}_{1} T^{4}_{2} T^{5}_{2} T^{5}_{3} T^{6}_{3} T^{6}_{4} T^{7}_{4} T^{7}_{5} T^{8}_{5} T^{8}_{6}
T^{1}_{1} T^{2}_{1} T^{2}_{2} T^{3}_{2} T^{3}_{3} T^{4}_{3} T^{4}_{4} T^{5}_{4} T^{5}_{5} T^{6}_{5} T^{6}_{6} T^{7}_{6} T^{7}_{7} T^{8}_{7} T^{8}_{8}
T^{9}_{8} T^{9}_{9} T^{10}_{9}

Figure 4. A schedule for n=10 using $\frac{n}{2}-1$ processors.

$$L = L(S_1) + \sum_{T_k^j \varepsilon S_{1+p}} W(T_k^j)$$

$$= n^2 - 1 + 2\sum_{j=1}^{n-2p} (n-j) - 2p$$

$$= 2n^2 - 4p^2 + 0(n).$$

Thus, for large n, p = αn and $1/4 \leq \alpha \leq 1/2$

$$\frac{S_p}{p} = \frac{1}{3(2\alpha-4\alpha^3)}$$

By similar analysis,

$$\frac{S_p}{p} = \frac{1}{3(3\alpha-20\alpha^3)}, \ \frac{1}{6} \leq \alpha \leq \frac{1}{4}$$

$$\frac{S_p}{p} = \frac{1}{3(4\alpha-56\alpha^3)}, \ \frac{1}{8} \leq \alpha \leq \frac{1}{6}$$

$$\frac{S_p}{p} = \frac{1}{3(5\alpha-120\alpha^3)}, \ \frac{1}{10} \leq \alpha \leq \frac{1}{8}$$

These efficiencies are plotted as a function of α = p/n in Fig. 5.

Actual Performance

The achievable schedules previously discussed were programmed using HEP FORTRAN and were executed on the HEP parallel computer. Although the program provided solutions to a set of linear equations, we present timing for only the LU decomposition part of the solution so that it may be compared with our

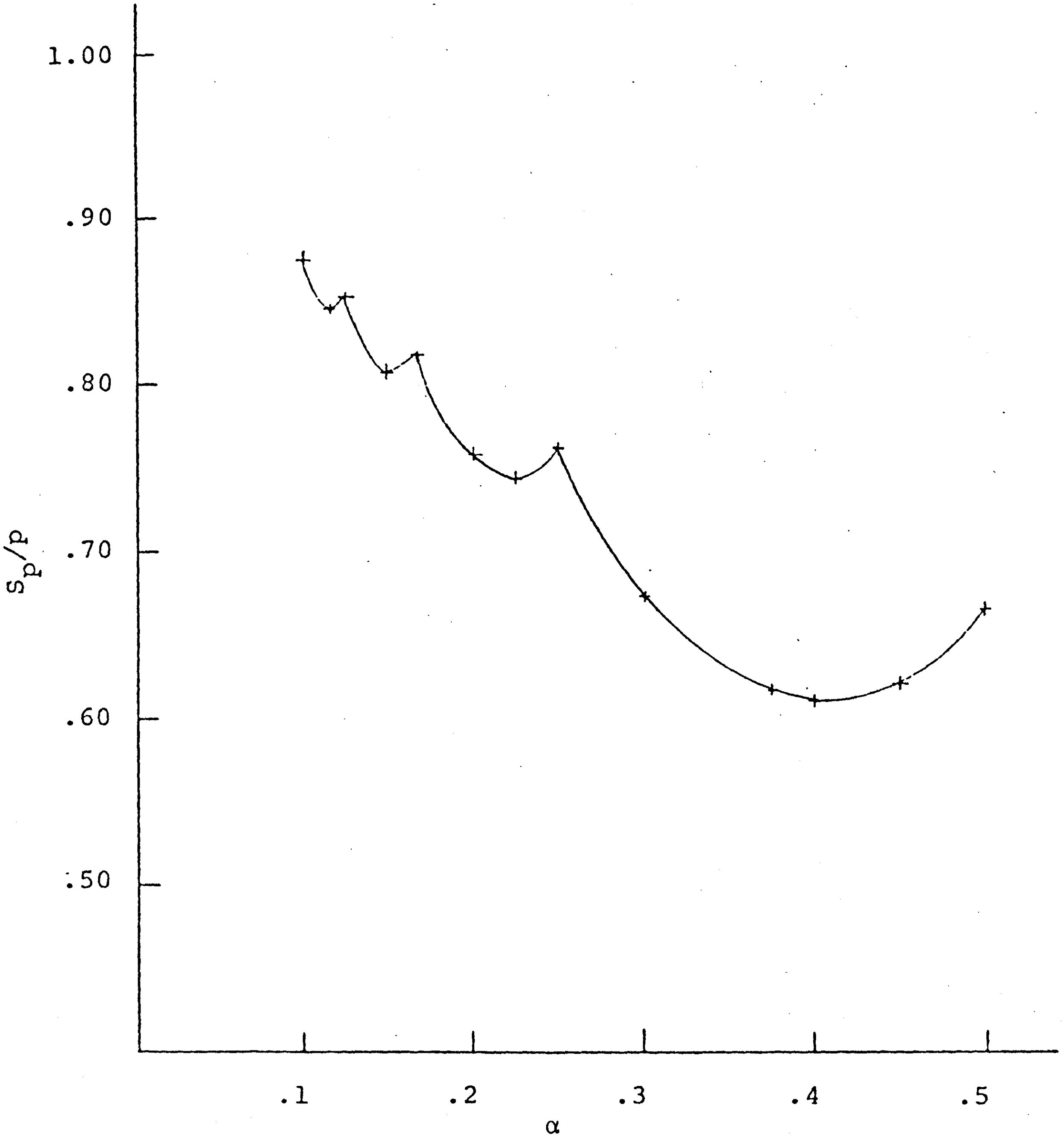


Figure 5. Efficiency versus $\alpha = p/n$.

predicted results. Due to memory limitations of the machine to which we had access, we could only run programs with $n \leq 35$ and $1 \leq p \leq 8$. Table 1 gives the achieved results together with a comparison of the predicted results.

Although the actual results are limited by the restriction on the maximum value for n, we feel that the agreement between actual and predicted performance is sufficiently good to give credibility to our model of the algorithms performance and that the efficiencies are high enough to support the conclusion that parallel methods for solving linear equations are a viable alternative to sequential methods.

Fast Givens Transformations

To Solve the square system of equations $A\underset{\sim}{x}=\underset{\sim}{b}$ using the fast Givens transformations, due to Gentleman [2], we proceed as follows:

(i) the matrix A is kept in the factored form

$$A = D^{1/2}B$$

where D is a diagonal matrix.

Initially $D=I_{nxn}$, B=A where n is the number of equations.

(ii) Triangularize the matrix A by applying Givens rotations to the augmented matrix $[A, \underset{\sim}{b}]$ and obtain the factors Q, $\hat{D}$, R and $\hat{\underset{\sim}{b}}$ such that

$$Q[A, \underset{\sim}{b}] = Q[D^{1/2}B, \underset{\sim}{b}] = \hat{D}^{1/2}[R, \hat{\underset{\sim}{b}}],$$

where R is upper triangular, Q is the product of the orthogonal transformations used in the triangularization and $\hat{D}$ is diagonal.

	number of processors p							
	2	3	4	5	6	7	8	
n=10	.833	.719	.642	.633				A
	.852	.739	.678	.685				P
n=15	.888	.794	.740	.651	.618	.625	.581	A
	.900	.815	.766	.679	.652	.681	.633	P
n=20	.921	.843	.774	.758	.670	.623	.605	A
	.931	.863	.798	.789	.703	.656	.640	P
n=25	.934	.878	.830	.763	.755	.692	.642	A
	.944	.896	.855	.739	.788	.726	.675	P
n=30	.942	.892	.844	.818	.757	.744	.710	A
	.949	.911	.863	.843	.783	.777	.745	P
n=35	.948	.901	.862	.819	.790	.747	.741	A
	.956	.918	.880	.843	.827	.779	.769	P

Table 1. Actual and predicted efficiency.

The algorithm proposed in Kowalik et al. [4] produces the orthogonal matrix $Q = Q_{2n-3}\, Q_{2n-4} \cdots\cdot\, Q_2 Q_1$ where

$$Q_k = \{P_{i,j} \mid i<j = 1,\ 2,\ \ldots,\ n,\ i+\ \ = k+2\},$$

$k = 1, 2, \ldots, 2n-3$ and $P_{i,j}$ are applied in parallel.

For the purpose of this analysis and implementation we assume that the number of available processors is $p = \frac{n-1}{2}$ where n is odd. We also assume that every Givens rotation is performed sequentially, however, more than one rotation could be performed in parallel.

We derive now a parallel scheme to triangularize A from the sequential method given in algorithm 1.

Let a task T^i_j in algorithm 1 be defined by

$$T^i_j = \text{GIVENS}(i,j)$$

where GIVENS(i,j) performs the following calculations:

1. $\alpha = -B(j,i)/B(i,i)$
2. $\beta = -(D(j)/D(i))*\alpha$
3. $\gamma = 1-\alpha\beta$
4. $D(i) = (1/\gamma)D(i)$
5. $D(j) = (1/\gamma)D(j)$

$$\left.\begin{array}{l} 6.\ B(i,\ell) = B(i,\ell) + \beta\, B(j,\ell) \\ 7.\ B(j,\ell) = (B(j,\ell) + \alpha\, B(i,\ell) \end{array}\right\} \qquad i \le \ell \le n$$

Periodic rescaling of D and B to prevent underflows and overflows, and row interchanges for numerical stability are included in our implementation of the Givens routine.

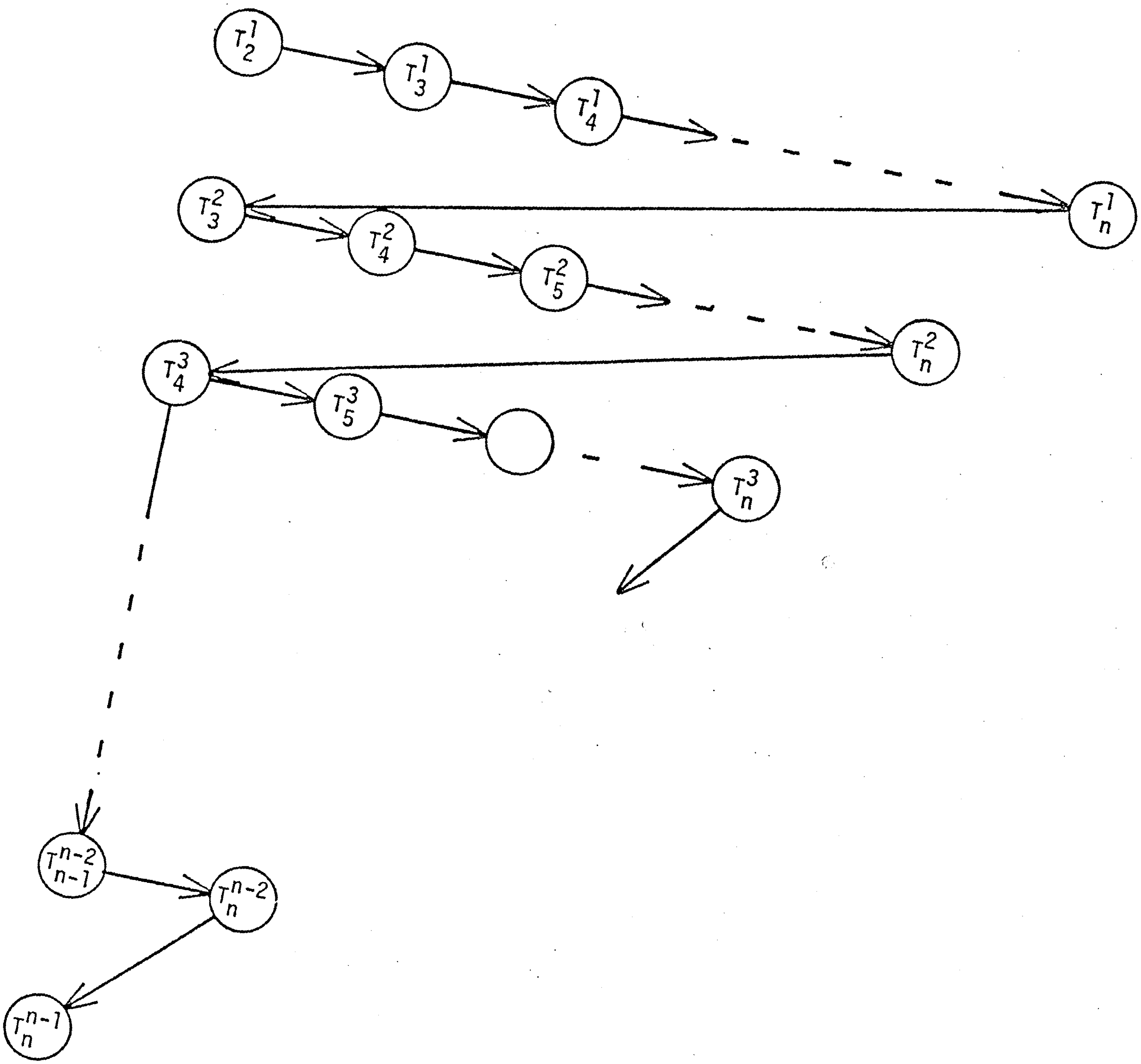


Figure 6: Task System C

The precedence constraints on the set of these tasks

$$\mathcal{J} = \{T^i_j \mid 1\le i\le n-1,\ \ i<j\le n\}$$

imposed by algorithm 1 are given by

$$\lessdot = [\{(T^i_j,\ T^i_{j+1}) \mid 1\le i\le n-2,\ i<j\le n-1\} \cup \{(T^i_n,\ T^{i+1}_{i+2}), 1\le i\le n-2\}]^*$$

where * represents the transitive closure of the set. Thus the system C = (J,$\lessdot$) is a task system with a graph shown in Fig. 6. The Range and Domain of these tasks are:

$$R(T^i_j) = (D(i),\ D(j),\ B(i,l),\ B(j,l) \mid i\le l\le n)$$

$$D(T^i_j) = (D(i),\ D(j),\ B(i,l),\ B(j,l) \mid i\le l\le n)$$

from this we can see that the tasks

$$\{T^i_j \mid i<j\le n,\ 1\le i\le n-1,\ i+j = k+2,\ k = 1,\ 2,\ \ldots,\ 2n-3\}$$

are mutually noninterferring tasks and can be executed in parallel. Hence we obtain a maximally parallel task system C' = ($\mathcal{J}$,$\lessdot$'), where

$$\lessdot' = [(T^i_j,\ T^i_{j+1}) \cup (T^i_j,\ T^{i+1}_j) \mid 1\le i\le n-2,\ i<j\le n-1]^*$$

equivalent to C.

This maximally parallel task system C' is shown in Fig. 7. We now assume that one arithmetic operation constitues a time step. Thus the length of T^i_j is $L(T^i_j) = 2(n-i+1) + 7$ steps. The longest path in this maximally parallel task system is:

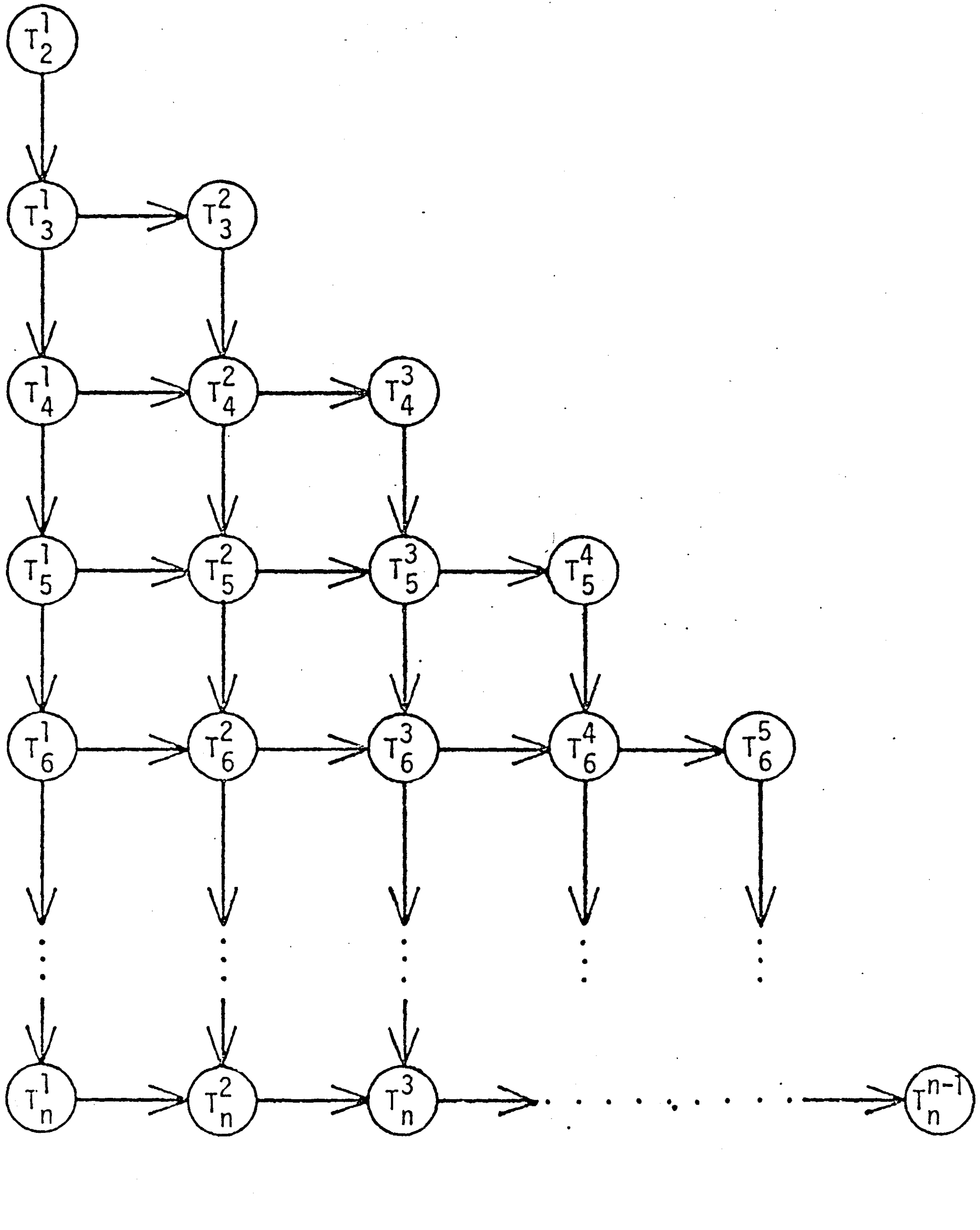


Figure 7: Maximally Parallel Task System C'

$$S_1 = \{T_2^1, T_3^1, \ldots, T_n^1, T_n^2, \ldots, T_n^{n-1}\},$$

and the total length of S_1 is

$$\begin{aligned} L(S_1) &= (4n+7)(n-1) + (4(n-1)+7) + (4(n-2)+7) + \ldots (4\cdot 2+7) \\ &\doteq 6n^2 + 8n - 25 \text{ operations.} \end{aligned}$$

To execute our task system with $p = \frac{n-1}{2}$ processors we have selected a scheduling scheme called ZIGZAG, shown in Fig. 8. According to this scheme the processors P_k, $k = 1, 2, \ldots, \frac{n-1}{2}$ are assigned to the tasks as follows:

P_1 executes: $\{T_2^1, T_3^1, T_3^2, T_4^2, \ldots, T_{n-1}^{n-2}, T_n^{n-2}, T_n^{n-1}\}$

P_2 executes: $\{T_4^1, T_5^1, T_5^2, T_6^2, \ldots, T_{n-1}^{n-4}, T_n^{n-4}, T_n^{n-3}\}$

$\vdots$

P_j executes: $\{T_{2j}^1, T_{2j+1}^1, T_{2j+1}^2, T_{2j+2}^2, \ldots, T_n^{n-2j+1}\}$

$\vdots$

$P_{\frac{n-1}{2}}$ executes: $\{T_{n-1}^1, T_n^1, T_n^2\}$.

For this schedule the speedup and efficiency are:

$$S_p = \frac{T_1}{T_p} = \frac{\frac{4}{3}\, n^3 + O(n^2)}{6n^2 + O(n)} \cong \frac{\frac{4}{3}\cdot n^3}{6n^2} = \frac{2n}{9}$$

$$E_p = \frac{S_p}{p} = \frac{2n}{9} \cdot \frac{2}{n-1} = \frac{4}{9}\,\frac{n}{n-1}$$

and for sufficiently large values of n

$$E_p = \frac{4}{9} = .444\ldots$$

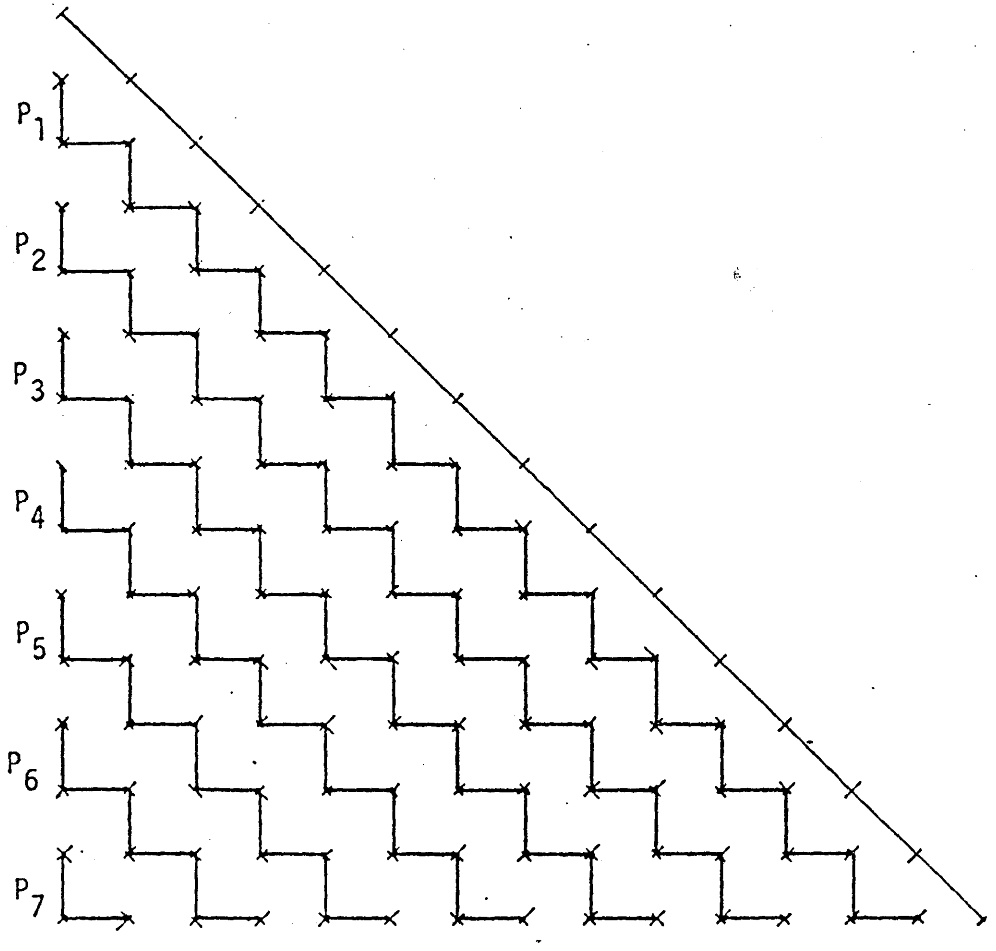


Figure 8: Parallel Zigzag Scheme for $n = 15$ $p = \frac{n-1}{2} = 7$

Computational Results

The ZIGZAG scheme for orthogonal tringularization shown in Fig. 8 was programmed and executed on the HEP parallel computer. Due to the present memory limitations the program was run for the values of n not exceeding n=29. Since for this machine $1 \leq p \leq 8$, and we assumed that $p = \frac{n-1}{2}$, the obtained numerical results up to n=17 are useful to compare. The actual and predicted speedups and efficiencies of the algorithm for different values of n are shown in Table 2. The differences between the predicted and actual values of S_p and E_p are due to several factors: machine overhead, approximate count of arithmetic operations involved in Givens rotations and data dependent number of scaling operations in the GIVENS routine which are not included in the operations count.

n	P	T_1	T_p	S_p	E_p	
5	2	.0036	.0025	1.44	.72	A
				1.40	.70	P
7	3	.0087	.0045	1.93	.64	A
				1.83	.61	P
9	4	.0168	.0072	2.33	.58	A
				2.27	.57	P
10	5	.0222	.0087	2.55	.51	A
				2.50	.50	P
11	5	.0286	.0105	2.72	.54	A
				2.72	.54	P
13	6	.0448	.0146	3.07	.51	A
				3.16	.52	P
15	7	.0660	.0194	3.34	.47	A
				3.61	.51	P
17	8	.0927	.0256	3.62	.45	A
				4.01	.50	P

Table 2. Actual and predicted speedup and efficiency.

References

1. Coffman, Jr., E. G., and Denning, P. J., Operating Systems Theory, (Prentice Hall, Englewood Cliffs, NJ, 1973).

2. Gentleman, W. M., "Least Squares Computation by Givens Transformations without Square Roots," J. Inst. Math. Applic. 12, 329-336 (1973).

3. Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, 20, 740-777 (1978).

4. Kowalik, J. S., Kumar, S. P., and Kamgnia, E. R., "An Implementation of the Fast Givens Transformations on a MIMD Computer," Washington State University, Dept. of Computer Science, Pullman, WA 99164, unpublished manuscript.

5. Sameh, A. H., and Kuck, D. J., "On Stable Linear System Solvers," J. ACM 25, 31-91 (1978).

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

SCHEDULING RECURRENCE EQUATIONS FOR PARALLEL COMPUTATION

BY DR. R. E. LORD

ABSTRACT

The problem which is investigated is that of scheduling the calculation of recurrence equations as typified by the numerical solution of differential equations. These calculations are represented by means of a cyclic precedence graph and an algorithm is presented which determines the minimum period during which these calculations can be performed. The algorithm then extracts an acyclic precedence graph whose longest path has a length equal to this minimum period. We show, by example, that this minimum period can be considerably shorter than the scheduling period determined by scheduling just the calculations of the inner loop. Next, we provide an improvement to the known lower bound on schedule length given a fixed number of processors. This improvement is also shown to improve the effectiveness of the critical path scheduling method which we employ. Finally, an algorithm for the actual scheduling is described which uses limited backtracking. On the basis of randomly generated test cases the schedule length produced can be expected to be no more than .4% longer than an optimal schedule. All of the algorithms used have a time complexity which is polynomial in the number of tasks.

INDEX TERMS

Scheduling, recurrence equations, critical path list scheduling, bounds on schedule length, limited backtracking.

1. INTRODUCTION

The general problem we are interested in is the scheduling of computation on a parallel computer, but more specifically, the scheduling of repetitious calculations as is the case, for example, with the numerical solution of differential equations. The parallel computer model we consider is MIMD [15] type where we are interested in parallelism all the way down to a per instruction basis.

Specifically, the problem which we investigate is that of representing the computations involved in the solution of a recurrence equation by a cyclic precedence graph and of then determining the minimum period during which all of the calculations could be performed once, while still perserving the precedence constraints. We then extract from the cyclic graph an acyclic one whose longest path is equal to this mimimum period and investigate methods of efficiently scheduling this system. We consider both schedules whose length is equal to the minimum period using as few identical processors as possible, as well as schedules using a fixed number of processors and having as short a length as possible.

Prior to any formal definitions, we present an example which motivated our concern with the minimum solution period for recurrence equations. Consider the Van der Pol equation written as two first order equations:

$$\dot{x}_1 = x_2$$

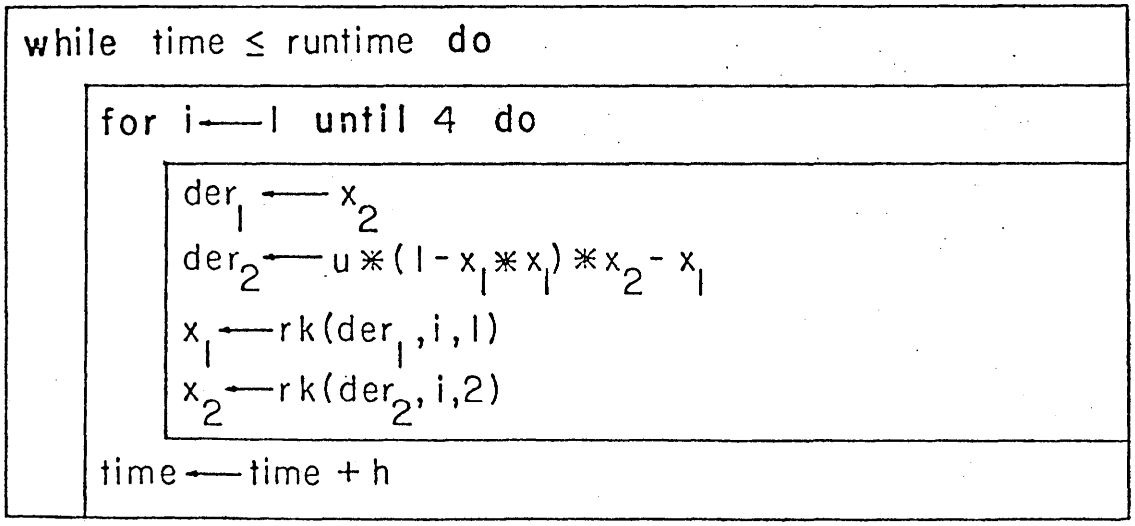
$$\dot{x}_2 = u(1 - x_1^2) x_2 - x_1$$

By using some suitable integration method, eg. 4-th order Runge Kutta, indicated by the function rk, the main part of a program for solving these equations is given in Figure 1. The calculation interior to the "for" loop can be represented by the acyclic precedence graph [6] shown in Figure 2. If we assume that each of the binary operations can be executed in one time unit and that the function rk can be evaluated in four units then the entire "for" loop can be represented by the cyclic precedence graph also shown in Figure 2, where, as is indicated, T_3 represents the calculation $u*(1 - x_1^2)$, T_4 represents $*x_2 - x_1$ and, T_1 and T_2 represent the calculation of the function rk.

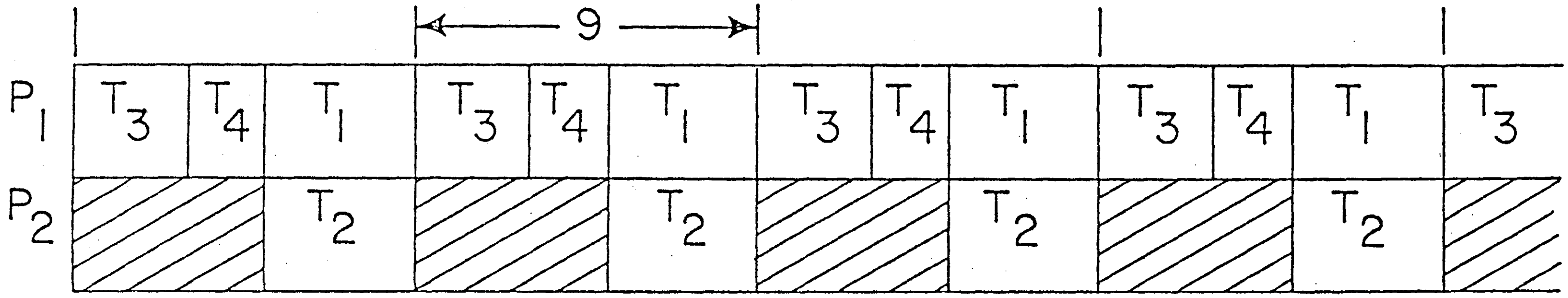
Given two parallel processors, then one way to schedule this solution is to assign the tasks interior to the "for" loop to processors in such a way as to preserve the precedence relations and yet complete all tasks as quickly as possible. The solution to the problem is then the repeated execution of this schedule. Such an assignment is shown by means of a Gantt chart in Figure 3. We note that this assignment is as good as possible since the precedence graph has a maximum path length equal to the assignment period.

The second Gantt chart of Figure 3 shows the assignments made if we assume initial values for x_1 and x_2 and then assign the tasks from the cyclic precedence graph while still maintaining all precedence constraints. This assignment has a repetition period of 7 units as compared with the 9 units for assigning the acyclic precedence graph. This shorter schedule is the motivation for examining recurrence equations to determine their minimum solution period and then to find methods to schedule them in that minimum period with as few processors as possible.

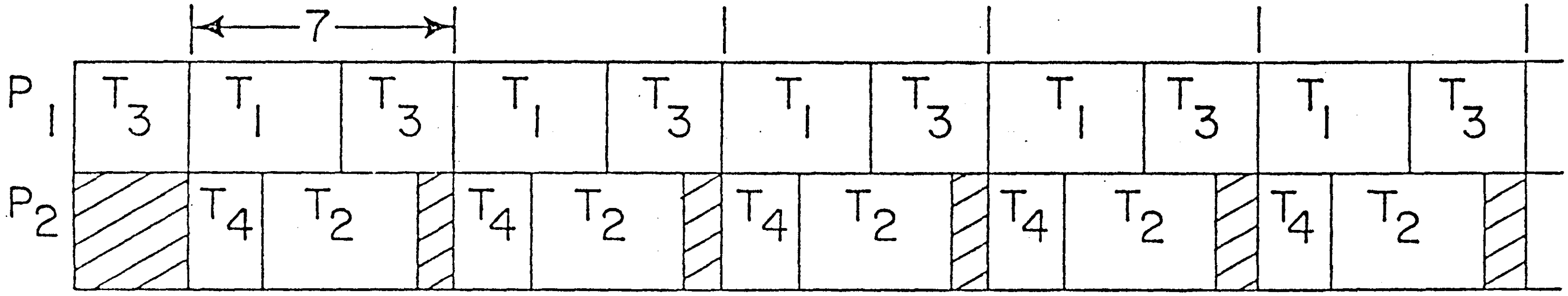
Recurrence equations were studied by Karp, Miller and Winograd [21] in the form of uniform recurrence equations which modeled the numerical



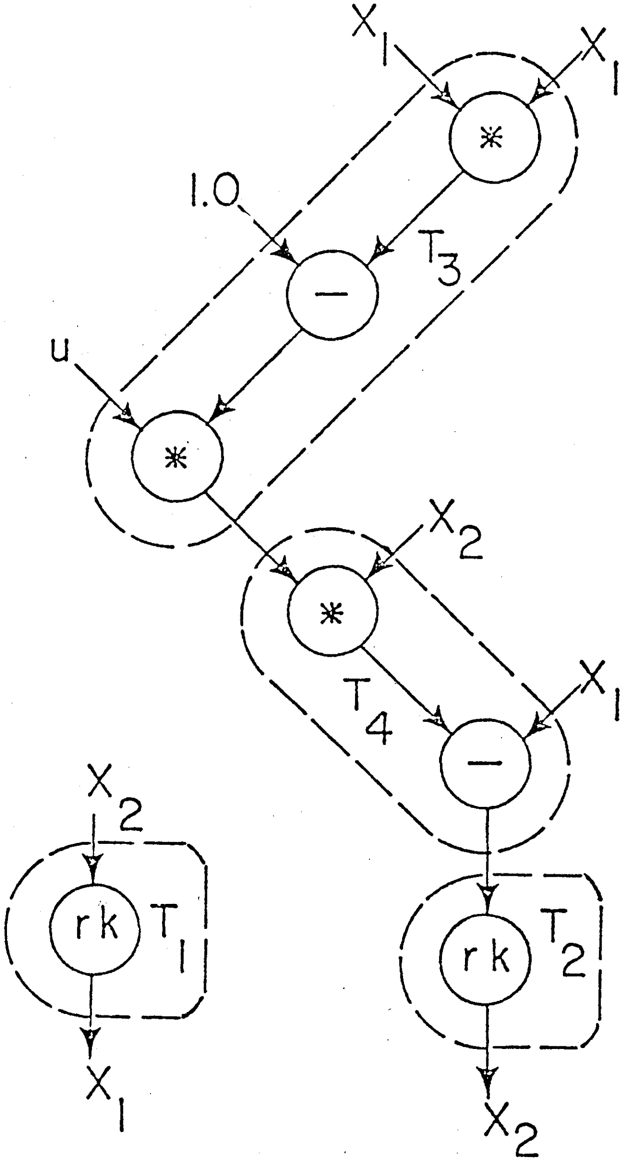
```
while $time \leq runtime$ do
    for $i \leftarrow 1$ until 4 do
        $der_1 \leftarrow x_2$
        $der_2 \leftarrow u * (1 - x_1 * x_1) * x_2 - x_1$
        $x_1 \leftarrow rk(der_1, i, 1)$
        $x_2 \leftarrow rk(der_2, i, 2)$
    $time \leftarrow time + h$
```

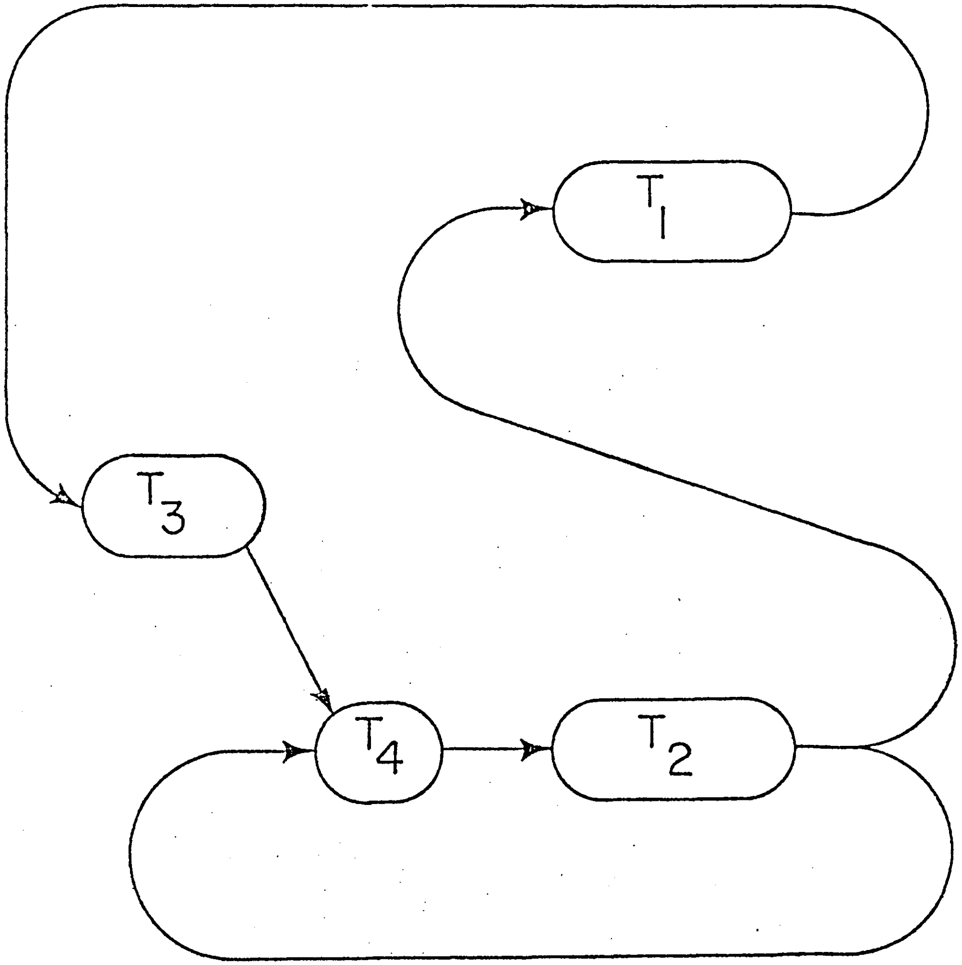


Assignments for Acyclic Precedence Graph



Assignments for Cyclic Precedence Graph





Cyclic Precedence Graph

solution of partial differential equations. More recently, Kogge [22] has studied them in a restricted form for solution on SIMD type computers. Essentially following Kogge we define the solution of a j-th order recurrence problem of dimension m to be a sequence x(1), x(2), . . . x(t) of vectors of dimension m where, we are given:

1. a set of initial values [x(0), . . . , x(-j+1)], and
2. a recurrence function f such that, for $1 \leq i \leq t$
   $x(i) = f(a_i, x(i-1), \ldots, x(i-j))$ where a_i is
   a constant vector of any dimension.

Kogge studied solutions of this type problem on SIMD type computers for the case of m = 1 and with restrictions on f. However, by allowing arbitrary values for m, the definition covers a wide class of problems including the numerical solution of differential equations and many optimization problems. Our restriction on f is only that it be a computable function. In general, we will represent this function by a directed graph where the nodes represent functions out of which f is composed and the edges represent that composition. We will further restrict the graph to be acyclic, so that if any part of the representation of f is iterative, then we will represent the entire loop with a single node. With each node of the graph we will associate a weight which represents the expected computation time of the associated function. We can now represent the recurrence equations by allowing terminal nodes in the representation of f (called recurrence nodes) to have outgoing edges, thereby creating a cyclic graph. Throughout we employ only standard graph terminology using [11] as a reference. The cyclic precedence graphs which are used as examples are usually drawn with the width of each node proportional to its ececution time. This convention was used in drawing the cyclic precedence graph in Figure 2.

2. CYCLIC PRECEDENCE GRAPHS

As we showed in the introduction, a simultaneous set of m recurrence equations can be represented by a cyclic precedence graph of n weighted nodes. For convenience the first m nodes will be labeled with the recurrence variables and all of the nodes simply represent functions, the number of whose parameters matches the number of incoming edges. The outgoing edge represents the functional value which is an argument for another function. The cyclic graph becomes acyclic if we remove all edges going out of the recurrence nodes for, in our model, all computations which are interative are represented by a single node whose weight is the expected execution time of the loop. For all nodes, the weight of the node is assumed to be non-negative. Since there are no cycles except those that pass through recurrence nodes, we may define the m x m path matrix P where $p_{i,j}$ is the length of the longest path from recurrence node i to recurrence node j which does not pass through any recurrence node. For this case, we define the length of a path from i to j to be the sum of the weights of all the intervening nodes including node j but not including the weight of node i. In the event that there is no path between them, then define $p_{i,j}$ to be zero. We may interpret $p_{i,j}$ as the minimum execution time between completing the calculation of the t-th value for x_i and completing the calculation on the t+1-th value for x_j. Thus, if we let maxp denote the largest $p_{i,j}$ for all i and j, then we know that if, at some point we had the t-th values for all x then by maxp units later we could complete calculating the t+1-th values for all x. This, of course, assumes a sufficient number of processing units.

However, from the example given in the introduction we have seen that in some cases, the calculation can be performed with a period shorter than the length of the longest path. We will now define the procedures for determining this minimum period.

Two assumptions are made regarding the form of the minimum length solution. First, we assume that no task is assigned to more than one processor and secondly, that the time between calculating the t-th and the t+1-th value is the same for all recurrence variables and no less than the minimum solution period. If we denote the execution time for the i-th task T_i by $tlen_i$ and the minimum solution period by minsol, then on the basis of the first assumption we have

$$(1) \qquad \text{minsol} \geq \max \left\{ tlen_i \;\middle|\; i \leq n \right\}$$

and on the basis of the second assumption

$$(2) \qquad \text{minsol} \geq \max \left\{ p_{i,i} \;\middle|\; i \leq m \right\}$$

As a further bound on the minimum solution period consider any pair of recurrence nodes i and j with $p_{i,j} \neq 0$ and $p_{j,i} \neq 0$. Now by our assumption the calculation of the t-th value of x_i will preceed the calculation of the t+1-th value by exactly minsol and similarly for x_j. Thus, we have

$$(3) \qquad \text{minsol} \geq p_{i,j} - \text{minsol} + p_{j,i}$$
$$\geq \left\lceil \frac{p_{i,j} + p_{j,i}}{2} \right\rceil$$

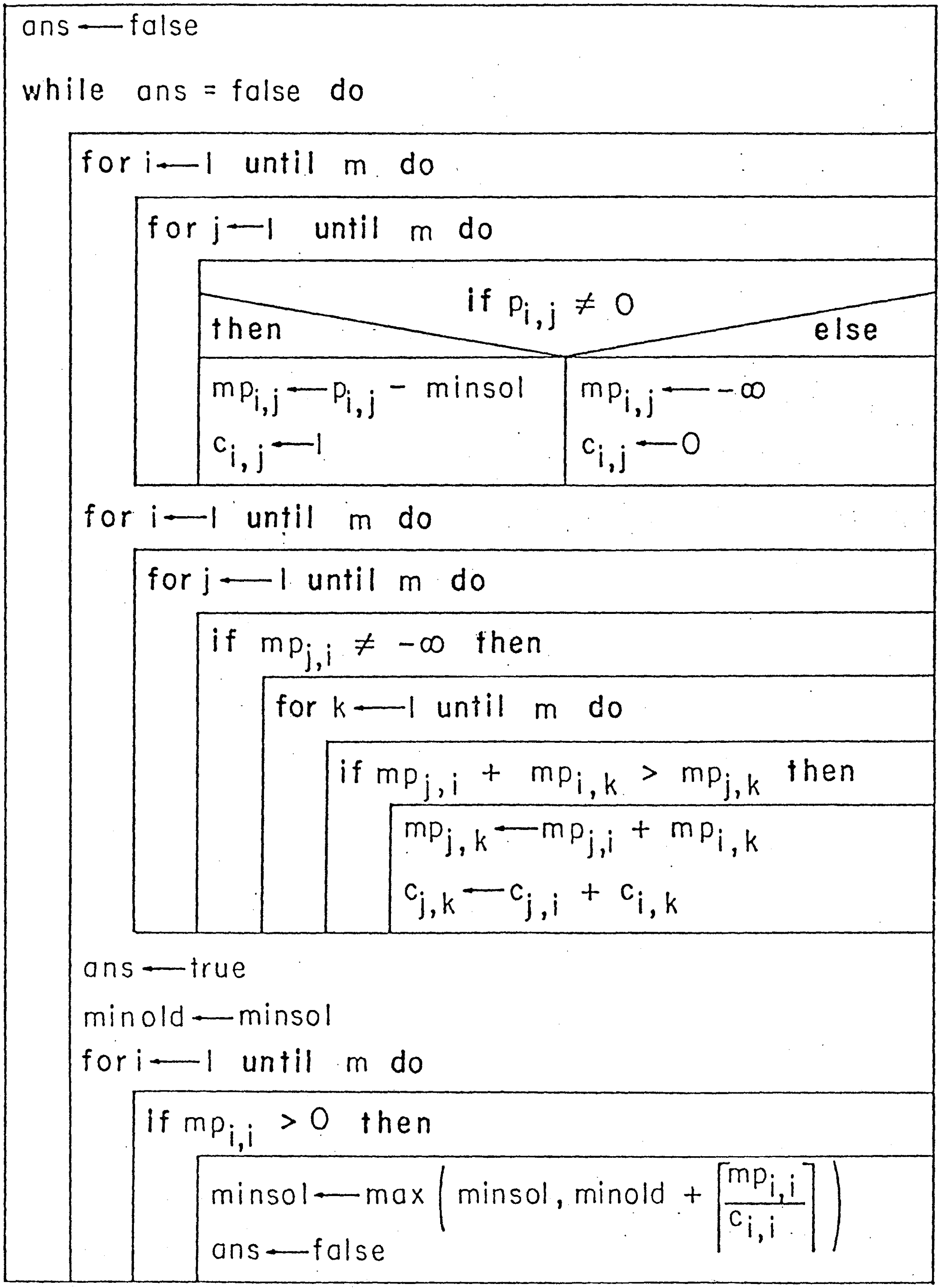
By similar reasoning, given k ordered recurrence nodes $i_1, i_2, \ldots, i_k$ with $p_{i_j,i_{j+1}} \neq 0$ for $0 < j < k$, then

$$(4) \quad \text{minsol} \geq \left\lceil \frac{p_{i_1,i_2} + p_{i_2,i_3} + \ldots + p_{i_k,i_1}}{k} \right\rceil,$$

and this bound contains bounds (2) and (3). Thus, minsol is the maximum of (1) and the maximum of (4) over all ordered sets of nodes that satisfy the condition of non-zero path lengths between them.

If we view P as representing the weights of an edge weighted directed graph of m vertices and m^2 edges, then the bound on minsol given by (4) is the per edge cost of a cycle having maximum per edge cost. Since the number of cycles in a digraph of m vertices is exponential in m, then any computational procedure based upon exhaustively examining the per edge cost of all cycles can be expected to have a very large execution time as m increases.

The computational procedure that we employ is to first estimate minsol by bounds (1) and (2) and then use this estimate as a parameter to the procedure shown in Figure 4. The computation is based upon the following: if we were to subtract the minimum solution period from each of the edge weights and then determine the maximum path length between all vertex pairs, then the maximum length path from any vertex to itself would be zero or negative. Thus, the algorithm is iterative in that we call it with an estimate of minsol, subtract this estimate from all the $p_{i,j}$ entries and then apply the Floyd-Warshal longest path algorithm [14]. If, after this calculation, all the diagonal elements of the longest path matrix are zero or negative then the estimate satisfies (4) for all cycles, and since it was initially chosen to satisfy (1), it is a correct bound. In the event that any diagonal element is positive, there is some cycle in P which has a per edge length greater than our current estimate of minsol, and hence, we must increase our estimate. The longest path algorithm proceeds by determining for each edge (i,k) whether the path length from j to k would be increased by substituting the path from j to i and thence to k. If so, the substitution is made, and the entry $mp_{j,k}$ represents a path which consists of



```
ans ← false
while ans = false do
    for i ← 1 until m do
        for j ← 1 until m do
            if $p_{i,j} \neq 0$
            then
                $mp_{i,j} \leftarrow p_{i,j} - minsol$
                $c_{i,j} \leftarrow 1$
            else
                $mp_{i,j} \leftarrow -\infty$
                $c_{i,j} \leftarrow 0$
    for i ← 1 until m do
        for j ← 1 until m do
            if $mp_{j,i} \neq -\infty$ then
                for k ← 1 until m do
                    if $mp_{j,i} + mp_{i,k} > mp_{j,k}$ then
                        $mp_{j,k} \leftarrow mp_{j,i} + mp_{i,k}$
                        $c_{j,k} \leftarrow c_{j,i} + c_{i,k}$
    ans ← true
    minold ← minsol
    for i ← 1 until m do
        if $mp_{i,i} > 0$ then
            $minsol \leftarrow max\left(minsol, minold + \left\lceil \frac{mp_{i,i}}{c_{i,i}} \right\rceil\right)$
            ans ← false
```

the number of edges in the path from j to i plus the number of edges in the path from i to k. Since, at the start of the calculation, all paths consist of a single edge, we can initialize an edge count matrix C to ones and zeros, and then, as we substitute paths, we accumulate the number of edges that make up those paths. At the completion of the calculation, if any diagonal element is positive, we increase our estimate of minsol by

$$\max \left\{ \frac{mp_{i,i}}{c_{i,i}} \;\middle|\; i \leq m \right\}$$

and then repeat the calculation. We note that this iterative precedure finishes after a finite amount of time since, on each iteration, we have either found a solution or we increase the estimate of minsol by an integral amount. Since minsol is bounded above by maxp, we are certain of reaching a solution. That this resultant value is the least value that satisfies the conditions can be seen by the fact that if, after a longest path calculation, there is a diagonal element $mp_{i,i}$ which is positive then we have discovered a cyclic path of length $mp_{i,i}$ + minsol*$c_{i,i}$. This path has a per edge length of ($mp_{i,i}$ + minsol*$c_{i,i}$)/ $c_{i,i}$, and thus,

$$\text{minsolnext} = \text{minsol} + \left\lceil \frac{mp_{i,i}}{c_{i,i}} \right\rceil$$

is the least integral estimate which will cause that cycle to be non-positive. As we previously mentioned, the number of iterations is no more than the length of the longest path, which, if we assume a fixed upper limit on task execution time, is of order n. Since $m \leq n$ then the complexity of the longest path calculation is $O(n^3)$ and hence the complexity of the entire procedure is $O(n^4)$.

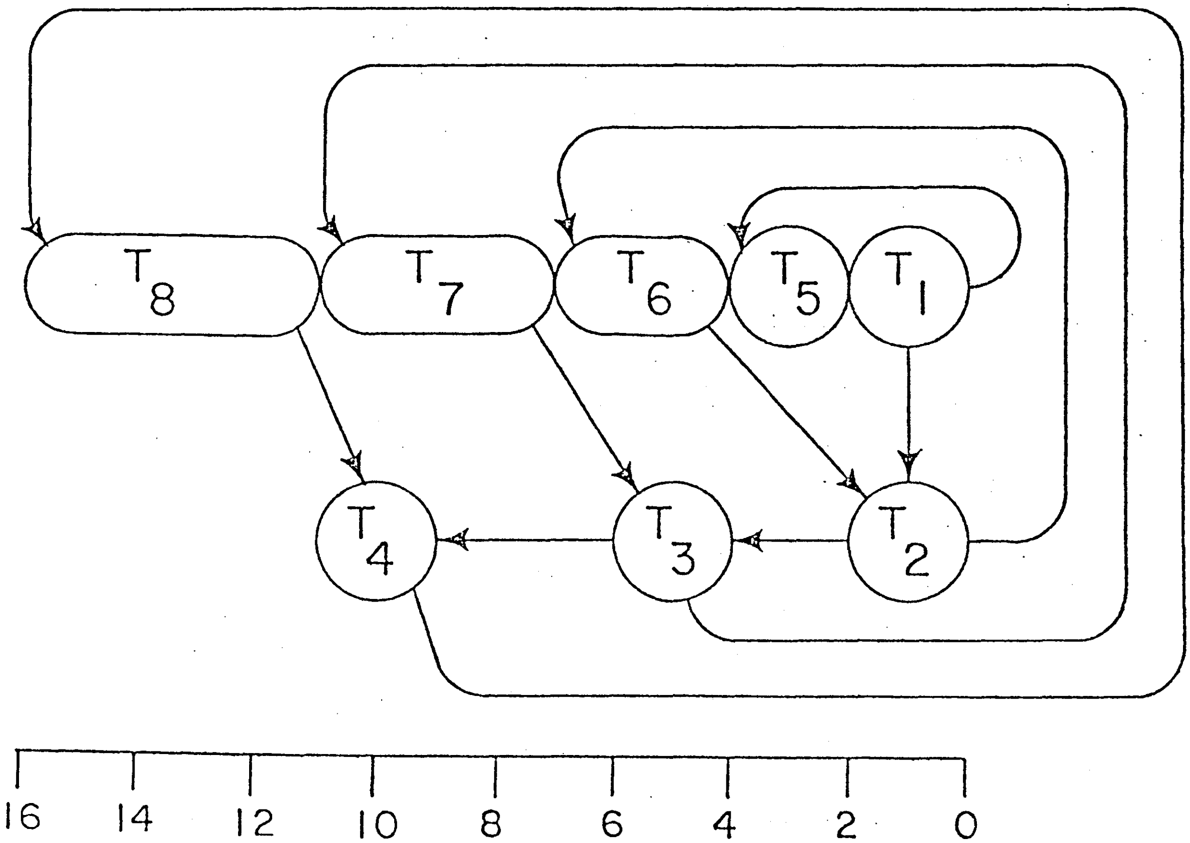
Having determined the minimum solution period, it still remains to determine the relative timing of the recurrence nodes. For example, if $p_{i,j} > \text{minsol}$ and $p_{i,j} + p_{j,i} = 2*\text{minsol}$, then the calculation of the t-th value for x_i must preceed the t-th calculation of x_j by $p_{i,j}$ - minsol units. On the other hand, if $p_{i,j}$ and $p_{j,i}$ are both less than minsol, then the t-th calculation of x_i can preceed that of x_j by as much as minsol - $p_{j,i}$ or follow it by as much as minsol - $p_{i,j}$. Hence, not all of the relative timing is unique. The procedure which we use to determine this relative location is shown in Figure 5 and is based upon the longest path matrix mp which was produced as a result of determining the minimum solution period. Now is $mp_{i,j} > 0$ for some $i \neq j$, then this means that the t-th calculation of x_i must preceed the t-th calculation of x_j by this amount. The values that the procedure determines are named lmaxp and denote the amount of time the calculation represented by this node must be started prior to the last recurrence variable being updated to its t-th value. The procedure is based upon finding the vertex j for which, for some i, $mp_{i,j}$ is maximum. We then determine the lmaxp values for all nodes that have a path to node j, but in no event do we set lmaxp to a value less than the corresponding value of tlen. For those vertices having no path to vertex j then the vertex amongst them is chosen which has the longest path into it and the above process for determining lmaxp is repeated for this set of vertices. We note that this "while" loop will be executed no more than once for each strongly connected subgraph and hence is limited to m executions. Further, the complexity of the procedure interior to the "while" loop is $O(m^2)$, and hence, the complexity of the entire procedure is $O(m^3)$.

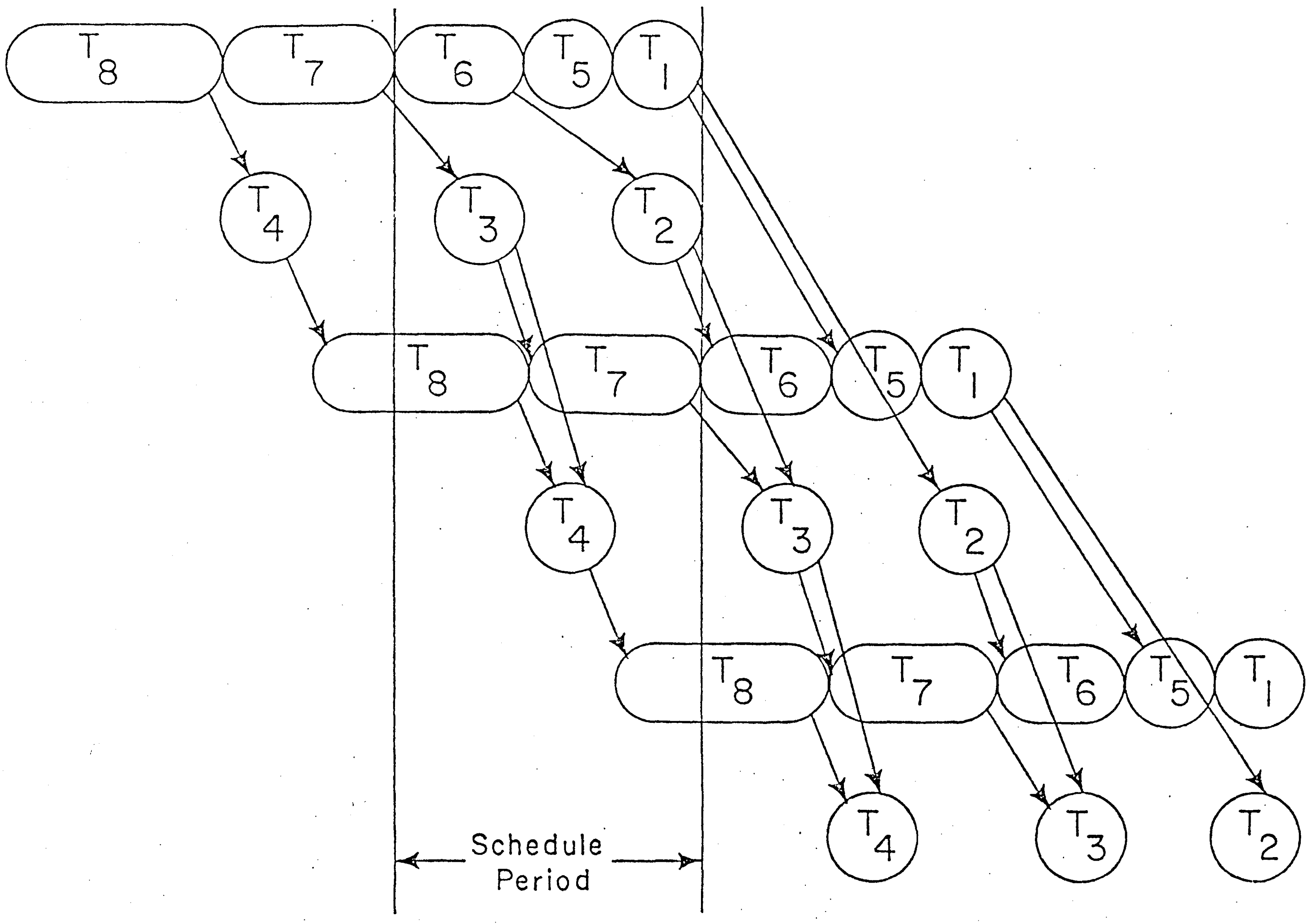
Prior to actual scheduling, it is necessary to transform the cyclic precedence graph into an acyclic one to which we can apply standard methods to determine either a schedule which solves the problem in the minimum solution period with a minimum number of processors or a schedule which

```
used ← 0
getmax (col)
while col ≠ 0 do
    for i ← 1 until m do
        if $used_i = 0$ & $mp_{i,col} \neq -\infty$ then
            if $mp_{i,col} \leq 0$
            then
                $lmax\,p_i \leftarrow tlen_i$
            else
                $lmax\,p_i \leftarrow mp_{i,col} + tlen_i$
    getmax (col)
for i ← 1 until m do
    if $used_i = 0$ then
        $lmax\,p_i \leftarrow tlen_i$
```

```
procedure getmax(col)
    col ← 0 ; maxv ← 0
    for i ← 1 until m do
        for j ← 1 until m do
            if $used_i = 0$ & $mp_{i,j} > maxv$ then
                $maxv \leftarrow mp_{i,j}$ ; col ← j
```

solves the problem as quickly as possible with a fixed number of processors. This transformation is accomplished by both deleting some edges and by splitting some of the tasks into two separate tasks with no edges between them. This method is best explained with the aid of an example. Consider the cyclic precedence graph shown in Figure 6 which consists of eight nodes, the first four of which represent recurrence variables. The diagram representing the graph is drawn relative to a time scale with the left part of each node placed in proportion to its lmaxp value and with the width of each node proportional to its tlen value. By examining the paths, we can see that the minimum solution period of seven units is determined by both the path from 4 to 4 and by the two paths $p_{3,4} = 2$ and $p_{4,3} = 11$. Now, the first order recurrence equations, the edges coming into a recurrence node represent information to be used in computing the t-th value for the node while the edges going out of the recurrence node represents the t-th value which is to be used in computing the t+1-th values. Thus, in the actual scheduling operation, the edges out of the recurrence nodes are really directed to another replica of the graph. In the example, the longest path is 18 units long and the minimum solution period is 7 units. Thus, three copies of the graph are required to illustrate the seven unit slice that is to be scheduled. This is shown in Figure 7 with the edges out of the recurrence nodes directed to the next copy of the precedence graph. We can now choose any time slice that is seven units wide and that contains all of the tasks. In our computational procedure, we choose that period which ends with the recurrence nodes having lmaxp values equal to their tlen values. Now, we are interested in scheduling the tasks within the scheduling boundaries so that repeated execution of this schedule will result in solving the recurrence


T8
T7
T6
T5
T1
T4
T3
T2
16
14
12
10
8
6
4
2
0


T_8
T_7
T_6
T_5
T_1
T_4
T_3
T_2
T_8
T_7
T_6
T_5
T_1
T_4
T_3
T_2
T_8
T_7
T_6
T_5
T_1
T_4
T_3
T_2
Schedule Period

equations. Thus, the edges that cross the scheduling boundaries can be deleted in producing the acyclic graph. In this example, the deleted edges are (1,2),(2,3),(7,3) and (7,6). In the general case, given an edge in a cyclic precedence graph which goes from vertex i to vertex j then that edge is to be deleted if

1. vertex i is not a recurrence node and

$$\left\lfloor \frac{\text{lmaxp}_i - \text{tlen}_i}{\text{minsol}} \right\rfloor \neq \left\lfloor \frac{\text{lmaxp}_j - 1}{\text{minsol}} \right\rfloor$$

2. or vertex i is a recurrence node and

$$\left\lfloor \frac{\text{lmaxp}_i - \text{tlen}_i + \text{minsol}}{\text{minsol}} \right\rfloor \neq \left\lfloor \frac{\text{lmaxp}_i - 1}{\text{minsol}} \right\rfloor$$

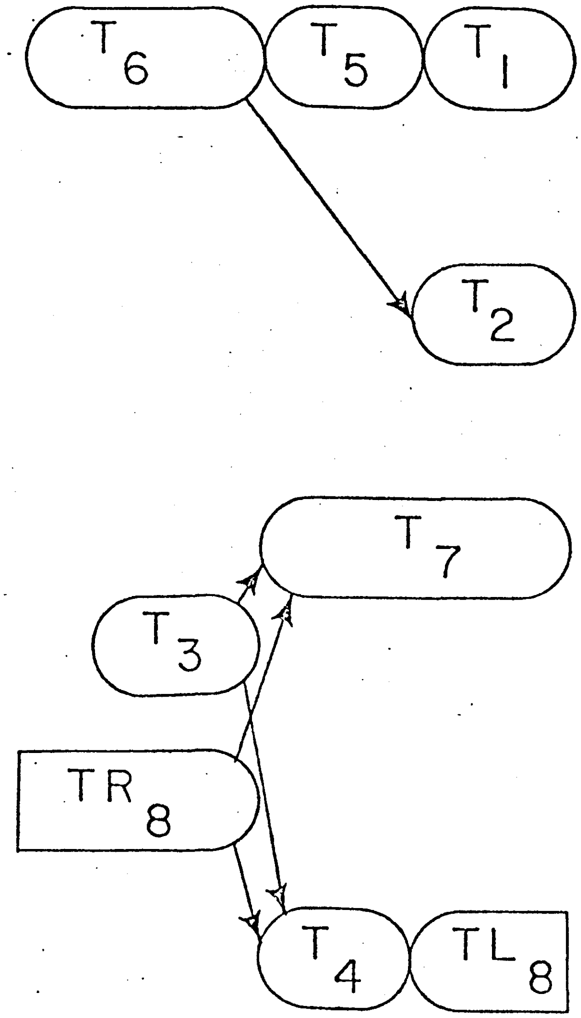
These relations simply formalize the conditions under which an edge crosses a scheduling boundary.

In addition to deleting edges, one can see that, for the example, T_8 must be split into two tasks, TL_8 which is two time units long and TR_8 which is three time units long. There is no edge between these two tasks and the edges that previously went to T_8 now go to TL_8. In the general case, task i must be split if

$$\left\lfloor \frac{\text{lmaxp}_i - \text{tlen}_i}{\text{minsol}} \right\rfloor \neq \left\lfloor \frac{\text{lmaxp}_i - 1}{\text{minsol}} \right\rfloor .$$

We also use this example to illustrate that splitting tasks may be necessary for any choice of the scheduling period. Since T_7 and T_8 are part of the path $p_{4,3}$ which, with $p_{3,4}$, require the minimum solution period, then T_7 must start immediately after T_8 completes, or at most one time unit later. Thus, any seven unit scheduling period will split either T_7 or T_8.

The resultant acyclic graph is shown in Figure 8.


T6
T5
T1
T2
T7
T3
TR8
T4
TL8

3. BOUNDS ON SCHEDULE LENGTH

The determination of a good lower bound on the number of processors required to schedule an acyclic precedence graph in a period equal to its longest path or, alternately, a lower bound on the schedule length given a fixed number of processors is valuable for two reasons. First, the scheduling method employed is goal oriented, and hence, a good estimate of the goal decreases the number of scheduling attempts with an unrealizable goal. Secondly, it is desirable to have a measure of how well the scheduling algorithm performs as compared with the best possible schedules. In this regard, Kohler [23] reports one graph of only thirty nodes that required over three minutes of computation time to determine an optimum schedule using a good branch and bound algorithm. Since we are interested in some graphs of more than one hundred nodes, the computational requirements of determining optimal schedules for all test cases is prohibitive, and thus, the measures of performance of our algorithm will have to be with comparison to good lower bounds.

The simplest bound on the number of processors required to schedule a graph in a fixed amount of time t was first defined by McNaughton [24] and is given by

$$\text{kmin} = \left\lceil \frac{\sum \text{tlen}_i}{t} \right\rceil$$

Several refinements of this bound have been proposed, the most complete being the one given by Fernandez and Bussel in [13]. Their bound is determined by considering all sub-intervals (t_1,t_2) in the scheduling interval $(0,t)$ and determining the minimum number of processors to complete the amount of work required in that interval. To make this definition more precise, given a schedule length t, then for each task define te to be the earliest that the task could be started and define tl to be the latest that the task could be started while still making a schedule of length t. We note that if i is an initial task, then $te_i = 0$, and if i is a final task, than $tl_i = t - tlen_i$. Given an interval (t_1,t_2), where $0 \leq t_1 < t_2 \leq t$, then for a task i, if it were started at its earliest time,

$$we_i = \min(\max(te_i + tlen_i,t_1),t_2) - \min(\max(te_i,t_1),t_2)$$

is, for this task, the number of time units that lie within the interval (t_1,t_2). Similarly, one may define wl_i as the number of units that this task would use in the interval if it were started as late as possible. Then $w_i = \min(we_i,wl_i)$ is the minimum number of execution time units that will lie in the interval, and consequently, for this interval

$$k_{t_1,t_2} = \left\lceil \frac{\sum w_i}{t_2 - t_1} \right\rceil$$

is the minimum number of processors required. Hence for the entire interval the minimum number of processors required is given by

$$kmin = \max\left\{k_{t_1,t_2} \middle| (t_1,t_2) \in (0,t)\right\}.$$

The major drawback to the above bound is its complexity which is $O(n*t^2)$ where n is the total number of tasks. Fernandez and Bussel recognized this drawback and suggested as an alternate the bound given by

$$kmin = \max(\max\left\{k_{0,t_1} \middle| (0,t_1) \in (0,t)\right\}, \max\left\{k_{t_1,t} \middle| (t_1,t) \in (0,t)\right\}).$$

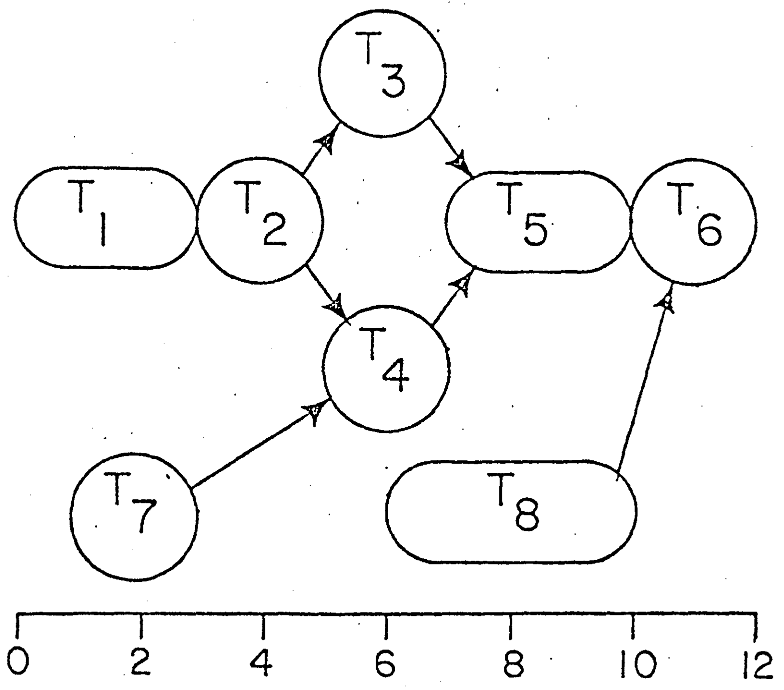
By use of two data structures of size n, these calculations have a complexity of O(max(t,n)). In [26], Ramamoorthy et al defined essential tasks for the interval (t_1,t_1+1) as those tasks that must be in process during the interval. Since the essential tasks must be processed, then the number of essential tasks is a bound on the number of processors required, and

$$\text{kess} = \max\left\{k_{t_1,t_1+1} \mid (t_1,t_1+1) \varepsilon (0,t)\right\}$$

could be combined with kmin to produce a bound which, although not as sharp as the Fernandez - Bussel bound, has a complexity of only O(max(n,t)).

An improvement in the bounds discussed above can be made by considering what we define as essential task interference. As an example, consider the graph of Figure 9 which consists of eight tasks. Application of any of the previously discussed bounds determines that, for a schedule length of twelve, the minimum number of processors is two. However, if we look at the number of essential tasks at each interval, we see that at period five and period six two processors are required for essential tasks and thus none are available for other tasks. As a result of this, task eight cannot start as late as time six as we determined from the precedence relations but must, instead, be started by time one so that it won't interfere with the essential tasks. If we now apply either the Fernandez - Bussel bound or the alternate, we find that three processors are required. We note that if task eight were only three units long and task seven were also three units, then although the graph would schedule in twelve units with two processors, the earliest that task eight could start would be time seven, and thus, rather than decreasing tl we would increase te. Now since changing either te or tl could add to the essential tasks at some interval, then whenever te or tl is changes, the essential tasks should be recomputed

TASK	TLEN	TE	TL
1	3	0	0
2	2	3	3
3	2	5	5
4	2	5	5
5	3	7	7
6	2	10	10
7	2	0	3
8	4	0	6



and any new interferences dealt with. A program for computing essential tasks and removing task interferences is shown in Figure 10. The procedures changete and changetl are resursive routines that change the te or tl value for that node and all of its sucessors or predecessors. The worst case complexity of the procedures for determining essential task interferences is $O(n*t^2)$.

```
procedure esstinf(k,t)
    done ← false
    while done = false do
        esswork ← 0
        done ← true
        for i ← 1 until n do
            for j ← $tl_i$ until $te_i + tlen_i$ do
                $esswork_j$ ← $esswork_j + 1$
        for i ← t-1 step -1 until 0 do
            if $esswork_i > k$ then
                write "Essential tasks exceed
                       processors"
            else if $esswork_i = k$ then
                for j ← 1 until n do
                    if $tl_j > i$ & $te_j \le i$ & $te_j + tlen_j > i$
                    then
                        changete(j,i+1)
                        done ← false
                    if $tl_j + tlen_j > i$ & $te_j + tlen_j \le i$
                      & $tl_j \le i$ then
                        changetl(j,i)
                        done ← false
```

4. SCHEDULING

In the previous sections, we have reduced the problem of solving recurrence equations in a minimum amount of time with fixed resources to the familiar scheduling problem. This problem of scheduling processors so as to minimize the total execution time of a set of tasks has received considerable attention and is the subject of at least three recent books [3,4,9]. The most recent of these [4] provides a thorough discussion of the problem together with notation and terminology for its representation. We depart from this notation only in that we consider the restriction to k identical processors amongst which data transfer imposes no penalty. Also, we do not consider deferral costs. Thus, we define a task system to be a three-tuple $(S,\prec,w)$ where

1. $S = \{ T_1,T_2, \ldots ,T_n \}$ is a set of n tasks,
2. $\prec$ is an irreflexive partial order defined on S which specifies the precedence constraints, and
3. $w : S \rightarrow N$ is a map which associates with each task a non-negative integer representing its execution time.

We note that as in previous sections we represent the tasks as nodes in a directed graph where an edge goes from T_i to T_j iff $T_i \prec T_j$ and we will write the value $w (T_i)$ as $tlen_i$.

Given k identical processors, a schedule of a task system has a schedule length of t is a total function $f : S \rightarrow \{ 0,1, \ldots ,t-1 \}$ subject to the conditions:

1. if $T_i \prec T_j$ then $f(T_i) + tlen_i \leq f(T_j)$,
2. for all $i \leq n$, $f(T_i) + tlen_i \leq t$, and

3. for each i, $0 \leq i < t$, there are at most k elements T_j for which $f(T_j) \leq i < f(T_j) + tlen_j$.

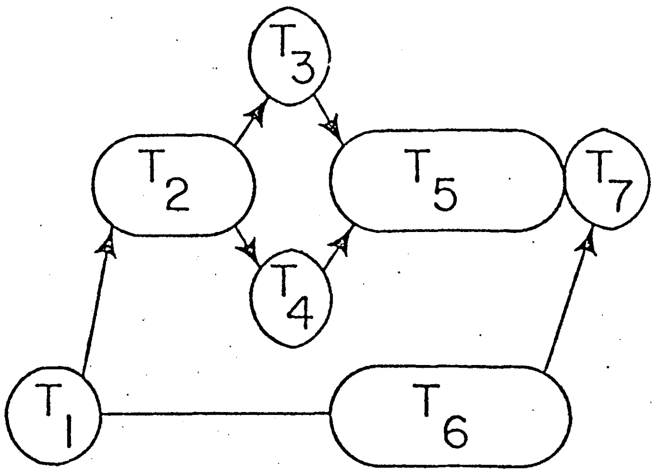
Since the processors are identical, the definition of a schedule does not distinquish to which processor a particular task is assigned. In presenting schedules, we will often make this assignment explicit by showing the schedule as a Gantt chart.

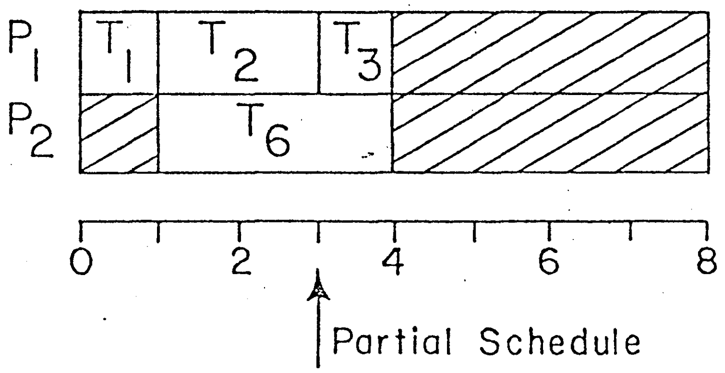
Since we desire efficient algorithms for scheduling, we turn to methods which, although not optimal, will produce "good" schedules. The most common method is termed list scheduling. In this type of scheduling we assume an ordered list of all of the tasks which is called the priority list. The sequence by which tasks are assigned to processors is then determined by scanning this list each time a processor is available and assigning to that processor the first unexecuted task all of whose predecessors have completed. This method of list scheduling forms the basis of many approximate methods as well as the algorithms that efficiently produce optimal schedules for some restricted cases. We use a special form of list scheduling termed critical path scheduling. In critical path scheduling the order of a task in the list is based upon the length of the longest path from that task to any final task. The further a task is from any final task, the earlier it appears in the list. Since critical path scheduling produces optimal schedules under suitable restrictions, it has been an attractive candidate for many scheduling problems where efficient methods are desirable. In [1] Adams et al. compared the performance of several list scheduling methods including critical path and found that critical path scheduling was significantly better than the other methods tested. Kohler [23] has also examined critical path scheduling and found that in randomly generated tests it produces optimal schedules in approximately 75% of the cases. In spite of this good expected performance, Graham [4, p 190] has shown that critical path scheduling can be as bad as the worst list. Thus, our choice of critical path as the basic scheduling method is based solely on the expectation that the schedules produced will be near optimal.

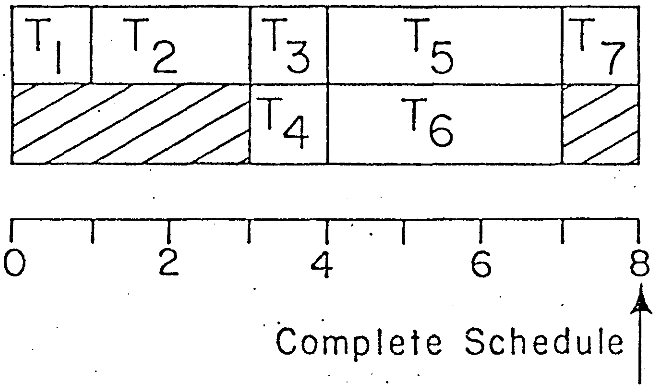
The scheduling method that we employ is a goal oriented critical path method with two further refinements. By goal oriented we mean that the scheduling algorithm, when presented with a task system, is also given a goal of the number of processors and the schedule length. The algorithm either produces a schedule with those constraints or it reports that it is unable to do so, in which case another request can be made with either a longer schedule length or more processors.

The refinements to the basic scheduling method are in two forms. First, the priority list may be modified as a result of essential task interference and second, as the scheduling progresses, limited backtracking may be employed whenever continuance of a given assignment could not meet the scheduling goal. We will discuss the modifications to the priority list first. Recall that, in Section 3, tl_i was defined for task i as the latest time that task i could be started and still meet the scheduling goals. Now if these values are not modified by the algorithm that computes essential task interference, then a list arranged in increasing order of tl would be exactly a critical path list. Instead of using only a critical path list, we use a list ordered on the value tl even if that value was changed due to essential task interference. This is justified in that the value tl can only be decreased if the original value was not consistent with any possible schedule. Thus, the resultant values for tl reflect the latest time the tasks can be started both because of precedence constraints and because of later interference with essential tasks. Also, the assignment of a task to a processor is never made prior to its te value. Initially, te represents the earliest that a task could be available for scheduling simply because of the time required for its predecessor tasks to finish. Following the determination of essential task interference, the values te also reflect the additional time, if any, that the task initiation must be delayed to avoid interference with essential tasks.

To illustrate the benefits of not scheduling a task prior to its te value, consider the task system shown in Figure 11. This system was given by Kohler [23] as an example of a system which no list scheduling method could schedule optimally. The task system has a longest path of length eight and the sum of the length of all tasks is twelve, thus a scheduling goal of eight time units and two processors appears reasonable. If we use the critical path method then the priority list is $T_1, T_2, T_3, T_4, T_5, T_6, T_7$ and the partial Gantt chart shows the assignments made until time three where it is determined that T_4 must be scheduled, if the goal is to be met, and yet no processor is available. However, if we examine te_6, we find that it was initially 1 but that, because of two essential tasks at time three, the original value would cause interference with these essential tasks, and te_6 was changed to 4. The second Gantt chart in Figure 11 shows the complete schedule generated by not scheduling T_6 prior to its te value. This example has shown how consideration of essential task interference amounts to adding a nonproductive task to the list of tasks to be scheduled. The next example shows how consideration of essential task interference can generate a better scheduling list than the critical path list. Consider the task system shown in Figure 12 where we depart from our normal method

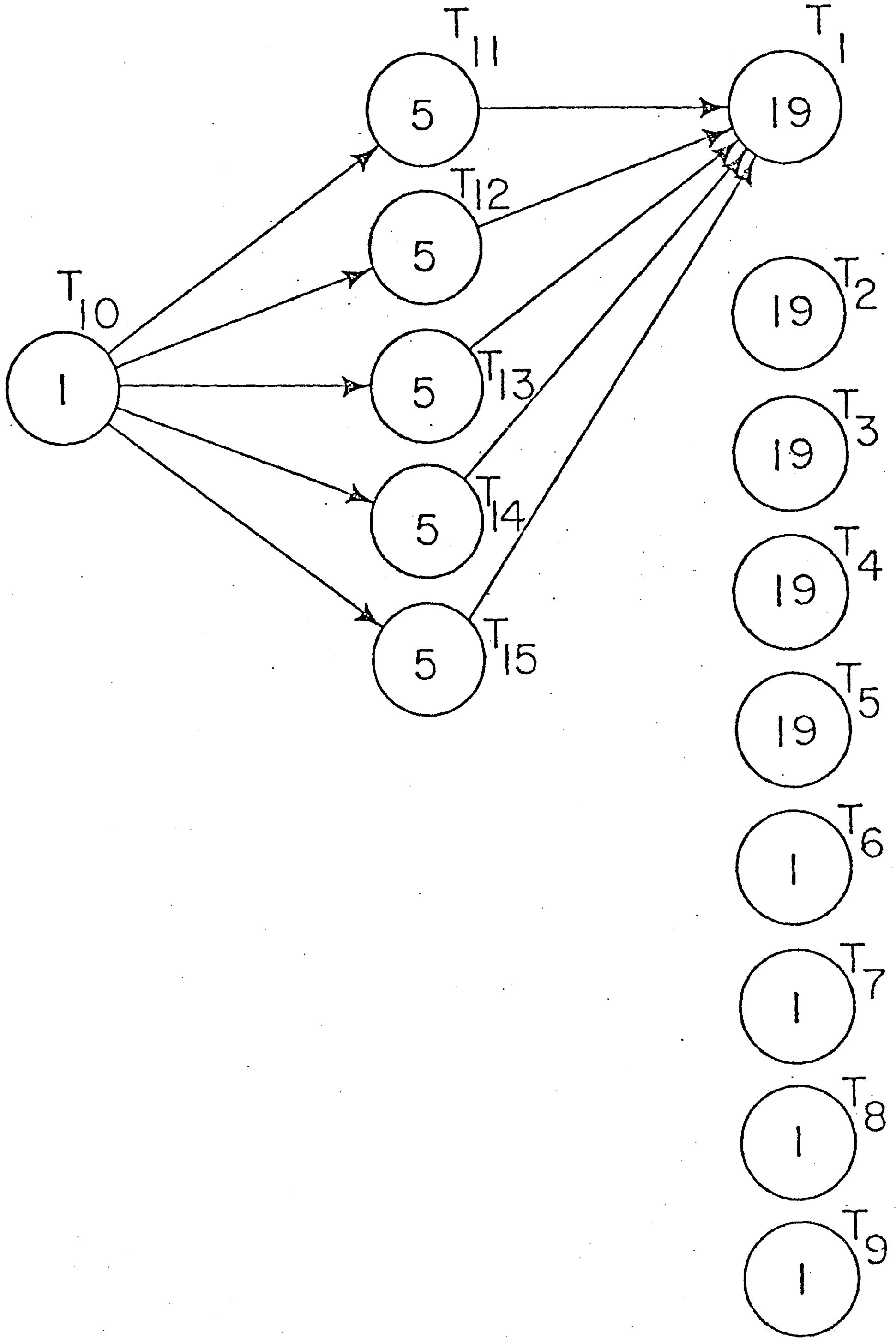

T_3
T_2
T_5
T_7
T_4
T_1
T_6


P_1
P_2
T_1
T_2
T_3
T_6
0
2
4
6
8
Partial Schedule

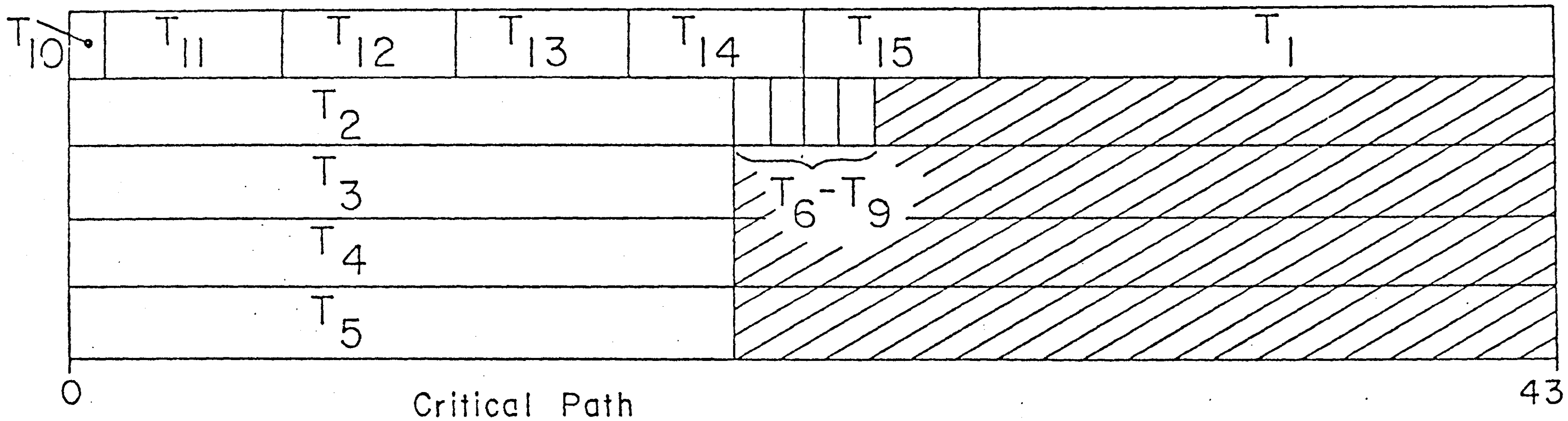

T_1
T_2
T_3
T_5
T_7
T_4
T_6
0
2
4
6
8
Complete Schedule

of showing the execution time of the tasks. Here the execution time of the task is shown interior to the node and its label is adjacent to it. This task system is an instance of the system which Graham [4,p 193] cites as an example of a system for which critical path produces a scheduling list which is as bad as any list can be. The critical path list is $T_{10}, T_{11}-T_{15}, T_1-T_9$ which will produce a schedule of length 43 using five processors. However, if we apply essential task interference methods with a goal of five processors and a schedule length of 25, we find that tasks $T_{11}-T_{15}$ are essential during time period one through five, and since there are five of them, then te for tasks T_2-T_5 must be increased from 0 to 6 so that they do not interfere. Recomputing the essential tasks, we now find that there are five essential tasks from time period 1 through 24, and hence, tl for tasks T_6-T_9 must be decreased from 24 to 0. The resultant list, ordered on non-decreasing values tl, is $T_6-T_{10}, T_{11}-T_{15}, T_1-T_5$ which will produce an optimal schedule shown in Figure 13. As a final example, we remark without showing the details that the task system attributed to G. S. Graham and given in [4,p 190] can be scheduled optimally if we remove essential task interference prior to determining the scheduling list. This system is given as an example to show that even if the partial order is a tree, the ratio of critical path schedule length to an optimal schedule can still be very close to 2.

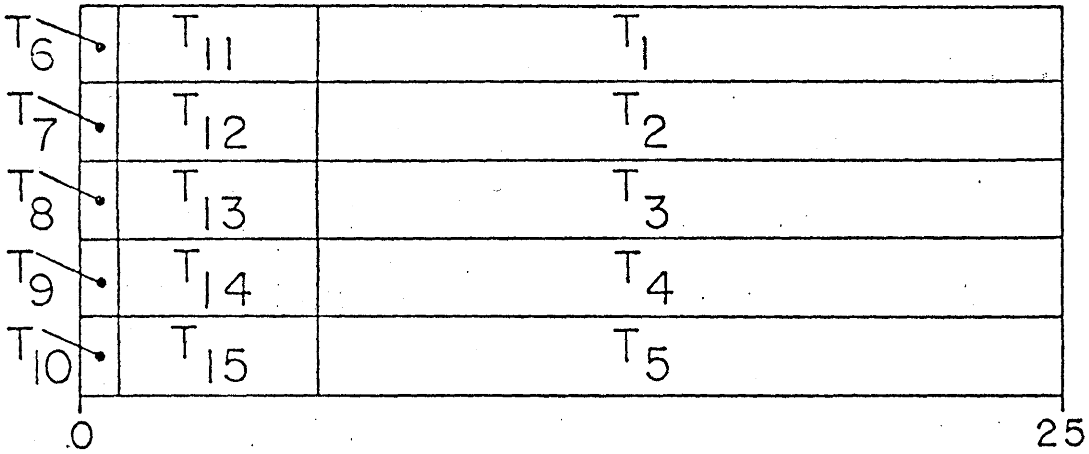
The removal of essential task interference, as we have used it here, applies only to the case where there are exactly k essential tasks during some time period. If there are k-1 essential tasks, then one other task can be processed concurrently but not two. Some instances of this case can be handled by using a limited backtracking algorithm. Consider the task system shown in Figure 14 which has a maximum path length of eight.



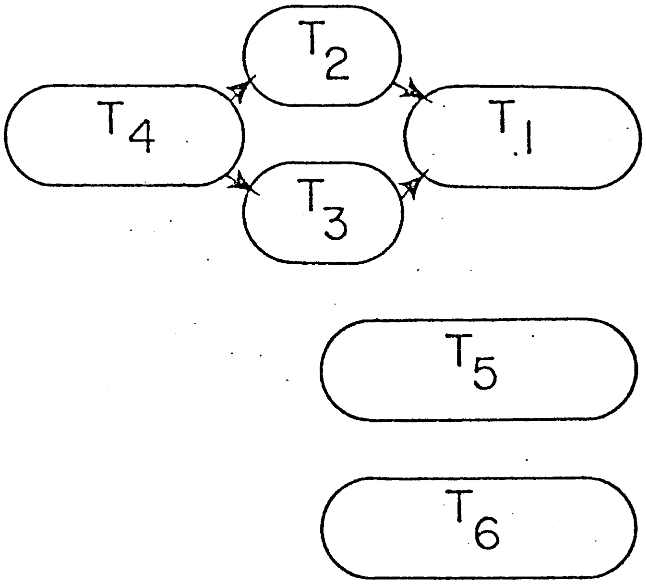
T_{10}
1
T_{11}
5
T_{12}
5
T_{13}
5
T_{14}
5
T_{15}
5
T_1
19
T_2
19
T_3
19
T_4
19
T_5
19
T_6
1
T_7
1
T_8
1
T_9
1

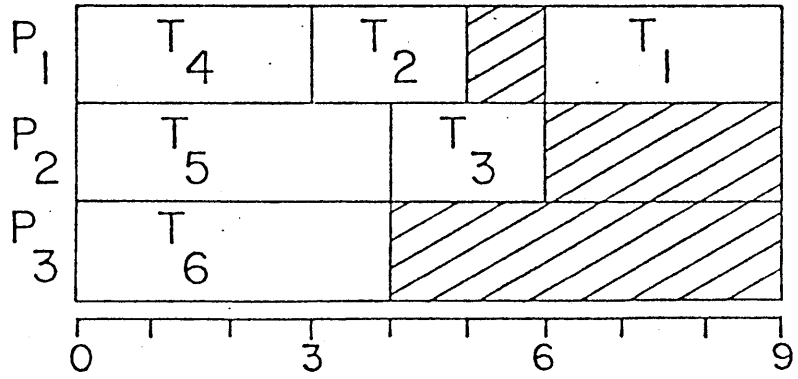


Critical Path

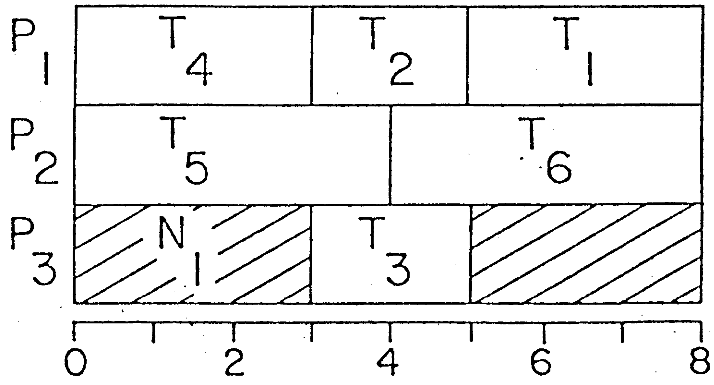


Critical Path With Essential - Task Interference





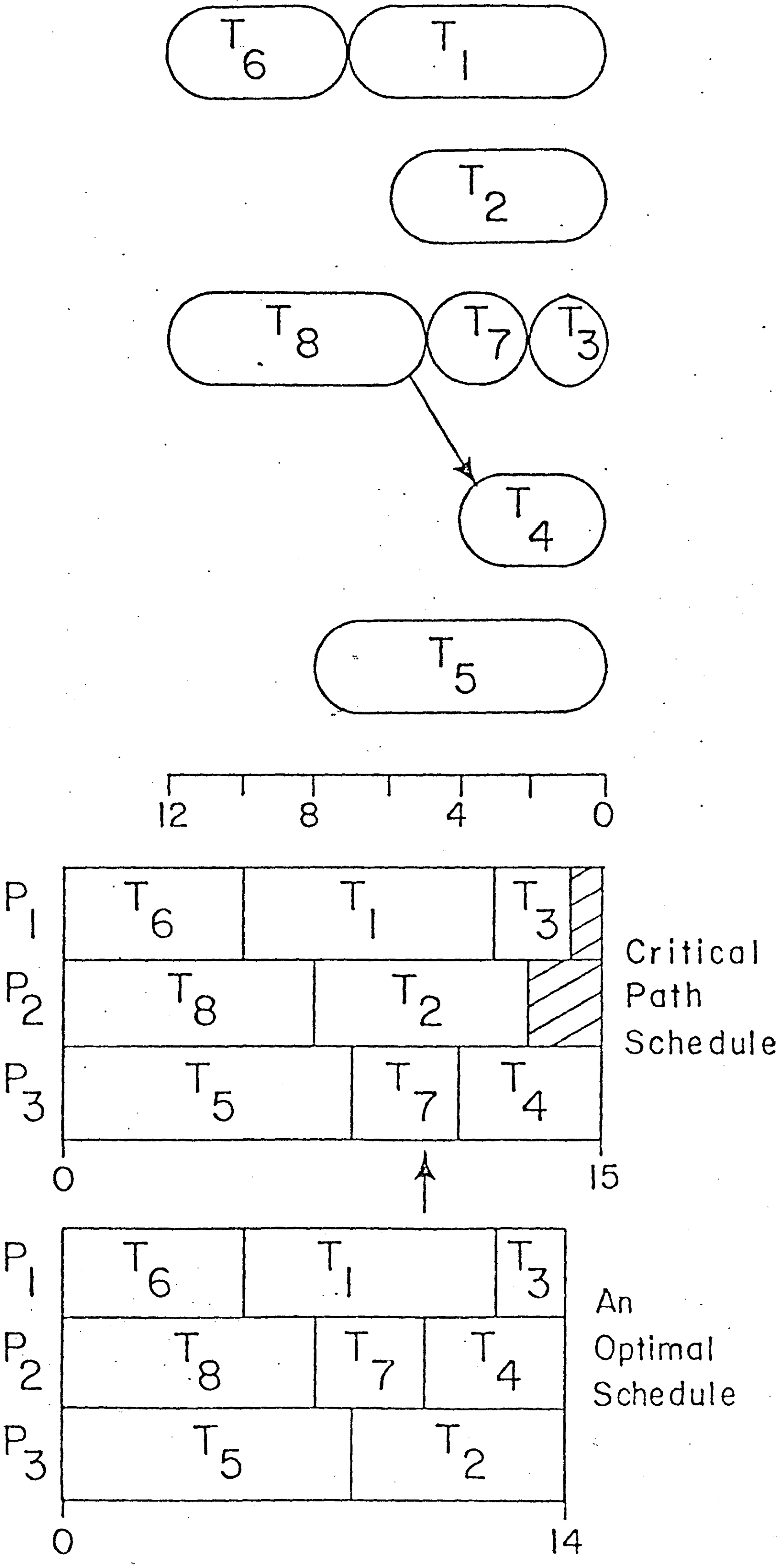
Critical Path Schedule



An Optimal Schedule

With a scheduling goal of three processors and length eight, it has only two essential tasks at time three and four. The critical path schedule illustrates the problem with assigning all of the tasks that can be assigned. At time zero we assign tasks T_4,T_5 and T_6 but then at time three, when two essential tasks should be assigned, only one processor is available. By backtracking at this point we can rescind the assignment of T_6 to processor P_3 since its start time can be delayed as late as time four. We can now assign T_3 and continue developing the schedule. In terms of list scheduling, this example of limited backtracking is another case of creating a nonproductive task N_1 which consumes a part of the resources. In this case, the original critical path list was T_4,T_2,T_3,T_5,T_6,T_1, and N_1, which is of length 3, inserted between T_5 and T_6. The part of the algorithm that inserts these extra tasks chooses the smallest one possible, in that, although the backtrack does not start until it is determined that a task must be scheduled and no processor is available, if a task can be rescinded then the critical task is assigned as early as possible. Further, in the event that there are more than one task currently assigned whose assignment could be rescinded, then we choose that one which results in the smallest nonproductive task. Additional refinement in the choice is made, when necessary, by rescinding the task which is rightmost in the list.

Another form of limited backtracking involves exchanging the position of two tasks in the list. This is illustrated by the task system shown in Figure 15. The task system consists of eight tasks with the sum of the task lengths equal to 42. A critical path schedule with three processors is shown and has a length of 15. However, if we apply any of the bounds of Section 3 we find that a schedule length of 14 is a reasonable goal. Using the critical path list, we find at time ten that T_4 must be assigned but


T_6
T_1
T_2
T_8
T_7
T_3
T_4
T_5
12
8
4
0
P_1
P_2
P_3
T_6
T_1
T_3
T_8
T_2
T_5
T_7
T_4
0
15
Critical Path Schedule
P_1
P_2
P_3
T_6
T_1
T_3
T_8
T_7
T_4
T_5
T_2
0
14
An Optimal Schedule

that there is no processor available. In this case we note that had we interchanged the position of T_2 and T_7 in the list then T_7 would be assigned to P_2 at time 7 and would have completed at time 10 so that T_4 could be assigned. The search for tasks whose interchange would allow the scheduling goals to be met is initiated only when a task must be assigned and no processor is available. At that time, a search is made of all currently assigned tasks that have not completed to determine if interchanging any two of them would allow scheduling to continue. The interchange of two tasks may also be accompanied by the creation of another nonproductive task. In the event that there are more than one pair of tasks that could be interchanged, we choose that pair which results in the smallest nonproductive task being created. If further refinement of the choice is necessary, we choose that pair with a task which is rightmost in the list.

The inclusion of the limited backtracking which we describe makes the worst case complexity of the scheduling procedure $O(n^3)$, however, in actual test cases run, the expected complexity was less than $O(n^2)$.

5. PERFORMANCE

In order to determine the effectiveness of the algorithms previously described, they were programmed using PL/I Level F, and test cases were run using an IBM System 360 Model 67.

Generation of Test Cases

The majority of the tests were made using randomly generated cyclic precedence graphs. Input to the program was the number of recurrence nodes and the total number of nodes. Based upon these numbers, the program then generated the cyclic precedence graphs where the nodes predominantly represented binary functions although unary functions were present. Since these nodes may represent a sequence of calculations with the edges representing only precedence constraint and not data flow, the task execution times were chosen from a uniform distribution over a fixed interval. The algorithms described in Section 2 were then applied, and an acyclic precedence graph having a maximum path length equal to the minimum solution period was extracted. Next, the scheduling goal for the number of processors is determined based upon

$$k = \left\lceil \frac{\sum \mathrm{tlen}_i}{\mathrm{minsol}} \right\rceil$$

This choice represents the best bound available on the number of processors required for any acyclic precedence graph extracted from the cycle one. Recall that the acyclic graph which we extract from the cyclic one is not the only one possible. It is easy to generate examples where the graph which we extract will not schedule with the goal of k processors and length minsol and yet, another acyclic graph can be extracted which will schedule with these goals. We next apply the algorithm for determining essential task interference and the alternate Fernandez - Bussel bound to determine the shortest scheduling period possible given k processors. The scheduling is then performed using the methods described in Section 4.

Test Results

Two series of 50 cases each were run with m, the number of recurrence nodes, in the range from 4 to 12 and n, the number of tasks, in the range from 16 to 144. The results of these two test series are shown in Table 1 and Table 2. The entry P-time is the lower bound on schedule length and S-time is the actual scheduling time. No entry is made in either of these columns if these numbers are the same as minsol. The column labeled ratio is the ratio of minsol to the length of the longest path from any recurrence node to any other. Density represents the utilization of the processors for this schedule. Finally, whenever the schedule produced was not possible with critical path scheduling alone, a "no" is placed in the column headed CP.

To summarize the test results with regard to scheduling, 24 cases out of 100 would not schedule optimally using only critical path scheduling which is consistent with the figures reported by Kohler [23] where he found 9 cases out of 40. We note however, that of these 24 cases, 17 (71%) of them were optimally scheduled using the one level of backtracking provided in our algorithm. By examining the ratio of the schedule length produced to the shortest bound we find that we can expect to schedule a task system with a schedule length which is no more than .158% longer than an optimal length. This figure is comparable to the .22% reported by Adams et al., [1] for similar size graphs. Certain differences however, make a close comparison difficult in that a) we have the additional scheduling problem of split tasks where the task must be scheduled both at the

TABLE 1.--Performance Figures for Test A

m	n	k	minsol	P-time	S-time	Ratio	Density	CP
4	16	3	30	-	-	.968	.922	-
		3	30	-	-	1.000	.867	-
		3	43	-	-	1.000	.767	-
		3	37	-	-	.925	.685	-
		3	42	-	-	.955	.683	-
		3	29	-	-	.569	.920	-
		3	34	-	-	.829	.912	-
		3	39	41	41	.765	.951	no
		2	53	-	-	1.000	.925	-
		3	33	33	34	.750	.951	-
6	36	5	39	-	-	.750	.872	no
		5	42	-	-	.778	.933	-
		5	56	-	-	.651	.832	-
		4	59	-	-	1.000	.784	-
		4	50	52	52	1.000	.851	-
		4	68	-	-	1.000	.754	-
		4	58	-	-	.935	.961	no
		6	41	-	-	.683	.850	-
		4	54	-	-	.818	.917	no
		4	56	-	-	.903	.875	-
8	64	6	56	57	57	1.000	.980	no
		4	94	-	-	1.000	.886	-
		5	78	78	79	.929	.965	-
		7	59	-	-	.908	.869	-
		6	66	-	-	.943	.894	-
		6	62	64	64	.849	.951	-
		6	62	-	-	.705	.944	no
		6	63	-	-	.875	.934	no
		7	50	-	-	.714	.874	-
		5	66	-	-	.930	.945	-

TABLE 1.--Continued

m	n	k	minsol	P-time	S-time	Ratio	Density	CP
10	100	7	93	-	-	1.000	.929	-
		6	108	-	-	1.000	.855	-
		9	59	-	-	.797	.962	-
		9	58	-	-	.906	.987	no
		7	85	-	-	.977	.909	-
		7	81	-	-	.976	.975	no
		7	77	77	79	.928	.975	-
		8	76	-	-	1.000	.896	-
		9	69	-	-	.945	.992	no
		8	70	-	-	.933	.984	-
12	144	10	82	-	-	1.000	.946	-
		12	64	65	66	.842	.944	-
		7	128	-	-	.985	.917	-
		9	90	-	-	.938	.917	-
		9	99	-	-	1.000	.919	-
		9	101	-	-	.990	.922	no
		10	84	-	-	1.000	.904	-
		7	119	-	-	1.000	.959	-
		13	72	-	-	.911	.931	no
		7	119	-	-	.960	.962	no
				Average		.904	.904	

TABLE 2.--Performance Figures for Test B

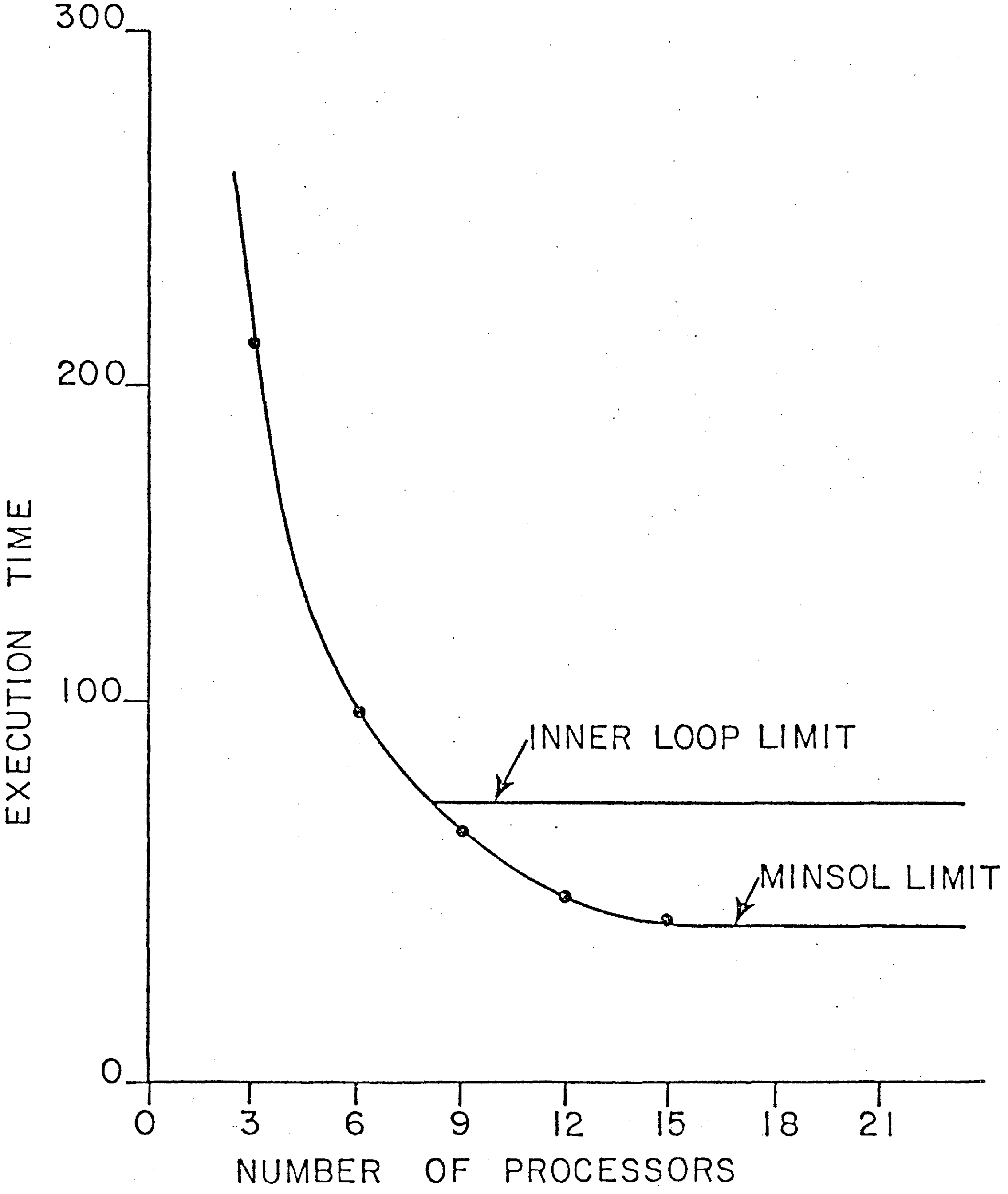
m	n	k	minsol	P-time	S-time	Ratio	Density	CP
4	16	3	30	-	-	1.000	.767	-
		3	34	-	-	.971	.853	-
		3	35	-	-	.795	.914	-
		3	33	34	34	.708	.951	-
		2	60	-	-	1.000	.842	-
		4	27	-	-	.900	.907	no
		3	41	-	-	1.000	.740	-
		4	25	-	-	.806	.880	-
		3	30	-	-	1.000	.867	-
		3	29	-	-	.935	.816	-
6	36	6	37	-	-	.881	.869	-
		5	53	-	-	.803	.830	-
		4	68	-	-	1.000	.816	-
		5	50	-	-	.893	.832	-
		5	49	-	-	.961	.894	-
		5	49	-	-	.907	.802	-
		5	41	-	-	.953	.863	-
		3	89	-	-	1.000	.831	-
		4	70	-	-	1.000	.843	-
		4	53	-	-	.828	.892	-
8	64	5	82	-	-	.932	.910	no
		5	85	-	-	.810	.915	no
		9	48	-	-	.814	.896	-
		6	79	-	-	1.000	.844	-
		5	70	-	-	.921	.891	-
		6	72	-	-	1.000	.847	-
		6	63	-	-	.926	.865	-
		5	87	-	-	1.000	.860	-
		6	53	53	55	.902	.912	no
		4	90	-	-	1.000	.950	-

TABLE 2.--Continued

m	n	k	minsol	P-time	S-time	Ratio	Density	CP
10	100	8	73	-	-	.912	.957	-
		7	84	-	-	.966	.952	-
		7	87	-	-	.897	.869	-
		10	53	55	56	.812	.927	-
		7	83	-	-	.883	.935	-
		8	76	-	-	1.000	.885	-
		8	76	-	-	.884	.880	-
		9	64	-	-	.928	.950	no
		8	83	-	-	.912	.973	-
		8	72	-	-	.960	.976	-
12	144	9	95	-	-	.969	.904	-
		9	85	-	-	.944	.950	no
		8	103	-	-	.981	.984	-
		9	92	-	-	.911	.940	-
		9	89	-	-	.864	.961	-
		8	112	-	-	.836	.926	-
		9	85	-	-	.876	.946	-
		9	104	-	-	.912	.909	no
		10	82	-	-	.901	.916	-
		11	80	-	-	1.000	.914	-
				Average		.920	.891	

beginning and at the end of the same processor, and b) we choose the number of processors such that no fewer number could still produce a schedule of the same length. This later condition may not have been met by all the schedules reported in [1].

With regard to the ratio of the schedule length to minsol, which is the best ratio we can claim for recurrence equations, we note that 89 out of 100 test cases were scheduled optimally and the expected schedule length is no more than .366% longer than an optimal schedule.


300
200
100
0
EXECUTION TIME
0
3
6
9
12
15
18
21
NUMBER OF PROCESSORS
INNER LOOP LIMIT
MINSOL LIMIT

6. CONCLUSIONS

From a practical standpoint, one of the more important areas of future research is to examine various changes to our assumption that data transfer between processors is without penalty. One such change could be that at each interval of time only some fixed number of data transfers could occur corresponding to some fixed number of data channels. Another change to this assumption would be to define a distance between processors, and to impose a time penalty on data transfers as a function of the distance between the two processors.

Another area for future research would be to examine various heuristics for extracting the acyclic precedence graph from the cyclic one. This extraction is not unique and we can show examples where one method of extraction will result in a task system that can be scheduled with a fewer number of processors than one extracted by some other method. One reason for this is that one method of extraction may result in more split tasks than some other method and by examining some of the test cases we have found that the split tasks can cause scheduling problems which will result in a longer schedule.

In terms of the benefits of using the algorithms presented, further study would be required to determine the exact characteristics of representative recurrence equations.

As compared with simply scheduling the acyclic inner loop, the random test cases of Section 5 shows that our method can be expected to produce schedules nearly 10% shorter. In many instances the savings can be considerably more. In one such case, we examined the scheduling of the solution to a set of eleven first order differential equations which represent the equations of motion of a ground launched missile. By considering just the inner loop, we found a minimum schedule period of 80 units and this was realized with 8 processors. However, by using the methods presented, a minimum solution period of 44 units was found and this was schedulable using 16 processors. The execution time verses the number of processors is shown in Figure 16.

7. ACKNOWLEDGEMENT

This work was performed while the author was at Washington State University. The support provided by the Computer Science Department is gratefully acknowledged. The author would also like to thank Professor Ottis Rechard and Professor Janusz Kowalik for their many helpful suggestions.

REFERENCES

[1] Adams, Thomas L., Chandy, K. M., and Dickson, J. R. A comparison of list schedules for parallel processing systems. Comm. ACM 17, 12 (Dec. 1974), 685-690.

[2] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[3] Baker, K. Introduction to Sequencing and Scheduling, John Wiley & Sons, New York, 1974.

[4] Coffman, E. G., Jr. Computer and Job Shop Scheduling Theory, John Wiley & Sons, New York, 1976.

[5] Coffman, E. G., Jr. A survey of mathematical results in flow-time scheduling for computer systems, Proceedings, GI 73, Hamburg, Springer-Verlag, 1973.

[6] Coffman, E. G., Jr., and Denning, P. J. Operating Systems Theory, Prentice-Hall, Englewood Cliffs, N. J., 1973.

[7] Coffman, E. G., Jr., and Graham, R. L. Optimal scheduling for two processor systems, Acta Informatica 1, 3(1972), 200-213.

[8] Cohen, Ellias. Symmetric multi-mini-processors: a better way?, Comput. Decis. 5, 1(Jan. 1973), 16-20.

[9] Conway, R. W., Maxwell, W. L., and Miller, L. W. Theory of Scheduling, Addison-Wesley, Reading, Mass., 1967.

[10] Cook, S. A. The complexity of theorem proving procedures, Proceedings 3rd ACM Symposium on Theory of Computing, 1971, 151-158.

[11] Deo, Narsingh. Graph Theory with Application to Engineering and Computer Science, Prentice-Hall, Englewood Cliffs, N. J., 1974.

[12] Enslow, P. H., Jr. Multiprocessors and Parallel Processing, John Wiley & Sons, New York, 1974.

[13] Fernandez, Eduardo B., and Bussell, Bertram. Bounds on the number of processors and time for multiprocessor optimal schedules, IEEE Trans. Comp. C-22, 8(Aug. 1973), 745-751.

[14] Floyd, R. W. Algorithm 97: shortest path, Comm. ACM 5, 6(June 1962), 345.

[15] Flynn, M. J. Some computer organizations and their effectiveness, IEEE Trans. Comp. C-21, 9(Sept. 1972), 948-960.

[16] Kujii, M., Kasami, T., and Ninomiya, K. Optimal sequencing of two equivalent processors, SIAM Journal on Applied Mathematics 17, 3(1969), 784-789.

[17] Gilliland, M. C. The hetrogeneous element processor, Proceedings, Special Symposium on Advanced Hybrid Computing, San Francisco, CA, 1975.

[18] Graham, R. L. Bounds on multi-processing anomalies and related packing algorithms, Proceedings, AFIPS Conference 40, (1972), 205-217.

[19] Hu, T. C. Parallel sequencing and assembly line problems, Operations Research 9, 6(1961), 841-848.

[20] Karp, R. M. Reducibility amoung combinatorial problems, Complexity of Computer Computation, Miller, R. E., and Thatcher, J. W. (Eds.), Plenum Press, New York, 1972, 85-104.

[21] Karp, R. M., Miller, R. E., and Winograd, S. The organization of computation for uniform recurrence equations, JACM 14, 3(July 1967), 563-590.

[22] Kogge, P. The Parallel Solution of Recurrence Problems, Ph. D. Thesis, Stanford University, 1972.

[23] Kohler, Walter H. Preliminary evaluation of the critical path method for scheduling tasks on a multi processor system. IEEE Trans. Comp. C-24, 12(Dec. 1975), 1235-1238.

[24] Krone, M. Heuristic Programming Applied to Scheduling Problems, Ph. D. Thesis, Princeton University, 1970.

[25] McNaughton, R. Scheduling with deadlines and loss functions, Management Sci. 6, 1(Oct. 1959), 1-12.

[26] Muntz, R. R. Scheduling of Computations on Multiprocessor Systems; The Preemptive Discipline, Ph. D. Thesis, Princeton University, 1969.

[27] Ramamoorthy, C. V., Chandy, K. M., and Gonzalez, Mario j., Jr. Optimal scheduling strategies in a multiprocessor system, IEEE Trans. Comp. C-21, 2(Feb. 1972), 137-146.

[28] Rumbaugh, J. E. A Parallel Asynchronous Computer Architecture for Data Flow Programs, TR-150, Project MAC, M. I. T., Cambridge, Mass., 1975.

[29] Schindler, Sigram, and Ludtke, Harald. An approach to restricted scheduling problem for multiprocessor systems. Proceedings, Sagamore Computer Conference, 1973, 121-129.

[30] Ullman, J. D. Polynomial complete scheduling problems, Operating Systems Review 7, 4(1973), 96-101.

Footnotes

[1] Montana State University, Bozeman, Montana

Captions

Fig. 1 Program for solution of Van der Pol equations

Fig. 2 Acyclic and cyclic precedence graphs for Van der Pol equations

Fig. 3 Schedules for Van der Pol equations

Fig. 4 Program to determine minimum solution period

Fig. 5 Determining relative times for recurrence nodes

Fig. 6 Cyclic precedence graph

Fig. 7 Interconnected precedence graph

Fig. 8 Resultant acyclic graph

Fig. 9 Essential task interference example

Fig. 10 Determining essential task interference

Fig. 11 Example of not scheduling early

Fig. 12 System which is worst case for critical path

Fig. 13 Critical path and optimal schedules

Fig. 14 Example for rescinding assignments

Fig. 15 Example for interchanging assignments

Fig. 16 Execution time for missile problem

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

TWO PARALLEL ALGORITHMS FOR SHORTEST PATH PROBLEMS

BY DR. N. DEO
DR. C. Y. PANG
DR. R. E. LORD

TWO PARALLEL ALGORITHMS FOR SHORTEST PATH PROBLEMS*

Narsingh Deo
C. Y. Pang, and
R. E. Lord

Computer Science Department
Washington State University
Pullman, WA 99164

March 1980

for

1980 International Conference on

Parallel Processing

*This work was supported by U.S. Department of Transportation contract no. DOT-RC-92042 and by NSF grant no. MCS78-25851.

ABSTRACT

After examining several dozen serial algorithms and their variations for various shortest-path problems, two algorithms were selected as good candidates for parallelization on an MIMD-type processor. These are: (1) Pape-D'Esopo version of the Moore's algorithm for finding shortest paths from one node to all others, and (2) Warshall-Floyd algorithm for finding shortest paths between all pairs of nodes. The techniques used in designing the two parallel algorithms are fundamentally different--one involves parallel processing with a queue and is suited for sparse networks while the other employs matrix methods and is suited for dense networks. The correctness of these algorithms is proved. Execution times are analyzed and compared with actual execution times on the HEP computer (an MIMD machine).

1. INTRODUCTION

Shortest-path problems are by far the most fundamental and also the most commonly encountered problems in the study of transportation and communication networks. Often the repeated determination of shortest paths and distances form the core (inner loop) in many transportation planning and utilization packages. Therefore, the search for faster and faster shortest-path procedures continues. After reviewing over 200 papers on shortest-path algorithms and after classifying and analyzing several dozen existing algorithms [5], two points became evident to us (among other things): (1) the shortest-path problems have almost reached their theoretical bounds of speed if conventional serial computers are to be used; and (2) certain algorithms (which may be most suited for serial mode) cannot be "parallelized" as readily as others. For example, Dijkstra's algorithm [4, 7, 18] for finding a shortest path between two nodes is not as well suited for parallelization as the Bellman-Moore [5, 14, 21] algorithm is.

We have selected two algorithms (for solving two different shortest-path problems), which appear to us as the best candidates for parallelization, for a detailed presentation in this paper. These are: (1) Pape-D'Esopo version of the Moore's algorithm for finding shortest paths from one node to all others [14, 15] and (2) Warshall-Floyd [4, 10, 18] algorithm for finding shortest paths between all pairs of nodes. The techniques used in designing the two parallel algorithms are fundamentally different--one involves parallel processing with a queue and is suited for sparse networks while the other employs matrix methods and is suited for dense networks.

We designed parallel versions of these two algorithms, suited for an MIMD (multiple instruction multiple data stream) [11] machine--keeping an eye, in fact, on the characteristics of the specific MIMD machine on which the designed parallel programs were actually to be executed. For example, on this machine the time required in creating a process is greater than the time needed to lock or unlock a resource.

In recent years, MIMD machines are not only being built experimentally in university laboratories, but they are being built in private industries. The Heterogeneous Element Processor (HEP) of DENELCOR Inc. [20], and the SMS 201 of Siemens AG [12] are two examples of commercial MIMD machines. Since the HEP was available to us, we coded and executed our programs on the HEP and performed the timing study on it.

Although a number of theoretical studies have been reported on parallel processing of graphs [1, 8, 9, 13, 17, 19], very few of them have considered the specific problems of shortest path problems and none have actually designed, coded and executed a parallel shortest-path algorithm on a real parallel computer (particularly on an MIMD computer) to the best of our knowledge. This study considers many of the real nuts-and-bolts issues of parallelization of existing algorithms, data structures, efficiencies and speed-gains over the serial implementations.

In Section 2, we will give definitions relevant to shortest paths on a network. In Section 3, we design a parallel algorithm for finding shortest paths from one specified node to all other nodes in a given network. The proof of correctness of the algorithm and the details of our model of computation are also given in Section 3. In Section 4, we present the second algorithm--for finding shortest paths between all pairs of nodes in a given network. The proof of its correctness and some empirical results on execution time are also presented in Section 4.

2. SOME DEFINITIONS

The following are the definitions of some of the important graph-theoretic terms used in this paper. Definitions for the rest of the terms can be found in any textbook on graph algorithms or networks [4, 18]. A *directed graph* $G = (V, E)$ is an ordered pair of finite sets: V of *nodes*, and E of arcs. We will use NODES to denote the number of nodes in V. We will also use $\{1, 2, \ldots, \text{NODES}\}$ to denote the elements of V. An *arc* a in E is an ordered pair, (u, v), of nodes. An arc $a = (u, v)$ is said to *start* at u and *end* at v. A *network* is a directed graph, G, together with a real valued function, ℓ, on the set of arcs. For any arc a, $\ell(a)$ is the *arc length* of a. An *arc length matrix* has its $(u, v)^{th}$ entry as $\ell(u, v)$ if the arc (u, v) exists. The entry is ∞ if (u, v) does not exist. A *path* P is a finite sequence of arcs $P = (a_1, a_2, \ldots, a_k)$, such that a_i starts where a_{i-1} ends, for $i = 2, \ldots, k$. The *length d(P) of a path P* is defined to be $d(P) = \ell(a_1) + \ldots + \ell(a_k)$. If $a_i = (u_{i-1}, u_i)$, we will, in addition, use $(u_0, u_1, \ldots, u_k)$ to denote P, and P is called a path from u_0 to u_k. A path that starts and ends at the same node is called a *cycle*. A cycle with negative path length is called a *negative cycle*. P is a *shortest path* from u to v if $d(P)$ is minimum over the length of all paths from u to v; the *shortest distance* from u to v is then $d(P)$. The *one-to-all shortest path problem* is the problem of finding the shortest paths from a given node, called the *source*, to all the other nodes, the *destinations*. The *all-to-all shortest path problem* is the problem of finding a shortest path for every pair of nodes in the network.

3. A PARALLEL ALGORITHM FOR THE ONE-TO-ALL SHORTEST-PATH PROBLEM

A modification of Moore's algorithm [14] by D'Esopo as reported in [16] was further developed by Pape [15] into two very efficient codes for finding shortest paths from a specified source node to all other nodes in the given network. This Pape-D'Esopo-Moore algorithm, which we will refer to as PDM algorithm, may be described in an Algol-like language as follows:

Algorithm PDM

```
for all $u \neq$ SOURCE do
  $D[u] := \infty$;
D[SOURCE] := 0;
initialize Q to contain SOURCE only;
while Q is not empty do
begin
  delete Q's head node u;
  for each arc (u, v) that starts at u do
    if $D[v] > D[u] + \ell(u, v)$ then
    begin
      $P[v] := u$;
      $D[v] := D[u] + \ell(u, v)$;
      if v was never in Q then
        insert v at the tail of Q;
      if v was in Q, but is not currently in Q then
        insert v at the head of Q
    end
end
```

During the execution of Algorithm PDM, the *label* D[u] is always updated to be the currently known shortest distance from SOURCE to u, and P[u] is always updated to be the predecessor node of u on the currently known shortest path from SOURCE to u. Since each insertion of a node u into Q is preceded by a decrement of D[u], this algorithm is guaranteed to terminate provided the input network has no negative cycles.

To see that the D[u]'s do indeed converge to the shortest distances, we first note that at termination $D[v] \leq D[u] + \ell(u, v)$ holds for every arc (u, v). Suppose the node sequence (SOURCE $\equiv u_0, u_1, \ldots, u_k \equiv u$) is a path from SOURCE to u, then its path length is given by

$$\begin{aligned} &\ell(u_0, u_1) + \dots + \ell(u_{k-1}, u_k) \\ &\geq (-D[u_0] + D[u_1]) + \dots + (-D[u_{k-1}] + D[u_k]) \\ &= -D[\text{SOURCE}] + D[u] = D[u]. \end{aligned}$$

Thus, D[u] is the shortest distance from SOURCE to u, and the node sequence,

$$(\text{SOURCE} \equiv P[\dots P[u]\dots], \dots, P[P[u]], P[u], u),$$

is the shortest path from SOURCE to use as obtained by Algorithm PDM.

The experiments of Denardo and Fox [2], Dial, Glover, Karney and Klingman [3], Pape [8], and Vliet [11] show that on the average Algorithm PDM is faster than almost every other shortest-path algorithm, if the input network has a low arcs to nodes ratio. We will, therefore, base our parallel algorithm on Algorithm PDM.

Let us fix our model of parallel computation before developing parallel algorithms. We will assume that our computer can simultaneously execute up to K processes. The communication between the processes is done via a common memory. The computer supports the operations: *create*, *lock*, and *unlock* [pp. 77-78 of Ref. 2]. When a process P_1 executes the statement "*create* process P_2," P_2 will start execution and P_1 will continue. For a memory X, after process P_1 executes "*lock* X," any other process that attempts to read, write, or *lock* X will have to wait until P_1 executes an "*unlock* X." Our model of computation is a realistic one; for the HEP computer can simultaneously execute processes, it has a common memory for all the processes, and it supports the operations *create*, *lock*, and *unlock* efficiently.

For practical reasons, we will assume that *create*, *lock*, and *unlock* take non-zero units of time to execute. In designing our algorithm, we also assume that *create* requires a longer execution time than *lock* and *unlock*. This assumption is also realistic, because *create* in the HEP machine using the

FORTRAN language is implemented with four instructions, whereas only one machine instruction is required for implementing _lock_ or _unlock_.

An obvious way to utilize the concurrent processing in Algorithm PDM would be to execute the inner _for_ loop (statements 8 to 17) simultaneously. But this approach is unprofitable because the overhead for a _create_ is high compared to the execution of one pass of the loop. Moreover, in this approach the maximum number of concurrent processes utilized would be about four, if the input is a typical road network (with outdegree* = 4). Therefore, we will avoid breaking the inner _for_ loop into different processes; instead we will distribute the passes of the _while_ loop (statements 5 to 18) to different processes. This will avoid excessive use of _create_'s.

We will use only K-1 _create_'s to obtain a total of K concurrent processes at the beginning of the algorithm, and use _lock_'s and _unlock_'s to take care of the rest of the synchronization. During the execution of the algorithm, the K processes--one called MASTER and the others called WORKERs--share the computation load, as long as there are known tasks to be performed. Each process takes approximately $\frac{1}{K}$ of the work load in the initialization step. In the path-finding step, each process repeatedly deletes a node, u, from Q, and updates P[v]'s and D[v]'s for the successors, v's, of u. In addition to a WORKER's tasks, the MASTER is responsible for finishing the initialization step, and for synchronizing the initiation and termination of the path-finding step. Our parallel algorithm, which we will refer to as PPDM, is as follows:

*The _outdegree_ of a node is the number of arcs coming out from that node.

Algorithm PPDM (Parallel Pape-D'Esopo-Moore)

Process MASTER

```
        MSYN := "yes"; WAIT := 0; DONE := 0;
        for i := 2 step 1 until K do
          create process WORKER(i);
        for u := 1 step K until NODES do
          D[u] := ∞;
    L1: if WAIT < K - 1 then goto L1;
        D[SOURCE] := 0;
        initialize Q to contain SOURCE only;
    L2: lock Q;
        if Q is empty then goto L3;
        delete Q's head node u;
        unlock Q;
        MSYN := "no";
        reach successor nodes of u (Block B);
        MSYN := "yes";
        goto L2;
    L3: if WAIT = K - 1 then goto L4;
        unlock Q;
        goto L2;
    L4: DONE := 1;
        unlock Q;
    L5: if DONE < K then goto L5
```

Process WORKER(i)

```
        for u := i step K until NODES do
          D[u] := ∞;
    L1: if MSYN := "yes" then goto L3;
        lock Q;
        if Q is empty then goto L2;
        delete Q's head node u;
        unlock Q;
        reach successor nodes of u (Block B);
        goto L1;
    L2: unlock Q;
        goto L1;
    L3: lock WAIT; WAIT := WAIT + 1; unlock WAIT;
    L4: if DONE > 0 then goto L5;
        if MSYN = "yes" then goto L4:
        lock WAIT; WAIT := WAIT - 1; unlock WAIT;
        goto L1;
    L5: lock DONE; DONE := DONE + 1; unlock DONE
```

Block B

```
 1     for each arc (u, v) that starts at u do
 2     begin
 3       newdv := D[u] + ℓ(u, v);
 4       lock D[v];
 5       if D[v] ≤ newdv then
 6         unlock D[v]
 7       else begin
 8         P[v] := u;
 9         D[v] := newdv;
10         unlock D[v];
11         lock Q;
12         if v was never in Q then
13           insert v at the tail of Q;
14         if v was in Q, but is not currently in Q then
15           insert v at the head of Q;
16         unlock Q
17       end
18     end
```

Note: For Block B of the MASTER process, Statement 11 should be changed to:

```
11         MSYN := "yes"; lock Q; MSYN := "no";  .
```

In Algorithm PPDM, the local variables are written in lower case letters, they are i, u, v, and newdv. The variables MSYN, WAIT, and DONE are the communication links between the MASTER and the WORKERs. MSYN = "yes" signals the WORKERs to let the MASTER check the Q first. WAIT is the number of WORKERs waiting for further command from the MASTER (i.e. WAIT is the number of WORKER processes which are executing statements 13 and 14). DONE is used by the MASTER to broadcase the termination signal. This algorithm requires the processes to keep on processing Block B until Q is empty. Block B is equivalent to statements 8 to 17 of Algorithm PDM. The locking and unlocking of D[v] and Q are added in Block B to ensure that Algorithm PPDM computes correctly.

Proof of correctness

We will now informally prove the correctness of this algorithm. It is easy to see that the initialization step is correct. For the path-finding step, we will first state and prove six remarks to show that the algorithm terminates for all networks which have no negative cycles.

Remark 1: For any node v, D[v] is nondecreasing with time.

Remark 2: Each finite D[v] represents the length of a path from SOURCE to v.

Remark 3: Only a finite number of insertions are made into Q.

Remark 4: Every execution of Block B always terminates.

Remark 5: There exists a time, t_1, such that the MASTER process will not execute Block B and MSYN = "yes" for all time after t_1.

Remark 6: Algorithm PPDM terminates.

To see that D[v] is nondecreasing, one simply observes that D[v] only changes when it is locked, and the changes are always decrements. To see that each finite entry D[v] represents a path length, we use induction on the time sequence of the change on the array D[•]. Let t_1 be the time immediately after D[SOURCE] is initialized to zero, and let t_{i+1} be the time immediately after the first change (or changes) in D[•] after t_i, for i = 1, 2, ... At time t_1, D[SOURCE] = 0 is the only entry of D[•] with a finite value, and 0 is the path length of the null path from SOURCE to SOURCE. Suppose for all time $t \leq t_i$, each finite D[v] represents a path length from source to v, and suppose D[v] is changed immediately before t_{i+1}. Assume that the change in D[v] is caused as we fan out from u, and that the value of D[u], at the time of its reading statement 3 in Block B, is the path length of (SOURCE $\equiv u_0$, u_1, ..., $u_j \equiv u$). At time t_{i+1}, D[v] is the path length of (u_0, u_1, ..., u_j, v). Thus, Remark 2 follows by induction.

To see that Remark 3 holds, we first notice that each D[v] is bounded from below, because the D[v]'s represent path lengths and the input network has no negative cycles. Secondly, we notice that there are only finitely many decrements to the D[v]'s, because each decrement decreases a D[v] by at least the minimum length difference between two loopless paths. Thus Remark 3 follows, since each insertion into Q implies a previous decrement of a D[v].

We will prove Remarks 4 and 5 together. To prove Remark 4, it suffices to show that no indefinite waits occur at Block B's statements 3, 4, and 11. By Remark 3, we see that Block B can be executed for only finitely many times. Thus every waiting at statements 3 and 4 takes a finite time. Because Q can be locked outside Block B, more arguments are needed to show that no indefinite wait occurs at Block B's "lock Q" statement (statement 11). We will prove a stronger result that no indefinite wait can occur at any "lock Q" statement in Algorithm PPDM. The MASTER always sets MSYN to "yes" before it executes "lock Q", and when MSYN is "yes" all WORKERs will be blocked from entering statements 4 to 11 and Block B. Thus the MASTER has no indefinite wait at "lock Q", and that its executions of Block B take finite time. Before we prove similar results for the WORKERs, we first prove Remark 5. It is easy to see that the loop of the MASTER's statements 9 to 16 has no indefinite wait. We claim that the loop of statements 9, 10, 17, 18, and 19 has no indefinite wait also, for if the MASTER is waiting at statement 17, then MSYN would have the value "yes", and consequently, only finitely many short lockings of WAIT can occur at the WORKERs' statement 12. Since indefinite wait does not occur at the MASTER process, and there are only finitely many insertions into Q, we conclude that eventually the MASTER will never enter Block B. We have just proved Remark 5. To finish the proof of Remark 4, we assert that the WORKERs have finite waiting time for executing the "lock Q" statements. Suppose the converse is true, and j WORKERs are waiting indefinitely at the "lock Q" statements (i.e. WORKER's statement 4 or Block B's statement 11). By Remark 5, the MASTER will eventually be looping at statements 9, 10, 17, 18, and 19. Each time the MASTER executes "unlock Q", statement 18, one of the j waiting WORKERs is allowed to finish executing "lock Q", which is a contradiction.

To prove Remark 6, we first recall that every execution of "lock Q" takes a finite waiting time. From Remark 3, we see that Q will eventually be empty and WORKER will not execute statements 6 to 9. By Remark 5, MSYN eventually has the value "yes", therefore all WORKERs are directed to the loop of statements 14 and 15. Consequently, Algorithm PPDM terminates.

Now we prove the correctness of the outputs, D[$\cdot$], and P[$\cdot$]. We use $D_t[u]$ and $P_t[u]$ to denote the values of D[u] and P[u] at time t, and use z to denote the termination time. We first claim that $D_z[v] \leq D_z[u] + \ell(u, v)$, for each arc (u, v). Suppose (u_1, v_1) is an arc of the input network. Let a be the time of the last deletion of u_1 from Q. Consequently, Block B is executed for u_1 after time a. The processing of the arc (u_1, v_1) includes the execution of either statements 5 and 6, or statements 5, 8, 9, and 10. Let b be the time of the execution of "unlock $D[v_1]$", at statement 6 or 10. Since the last deletion of u_1 occurs at a, it is easy to see that $D[u_1]$ stays constant after time a. Consequently, $D_z[v_1] \leq D_b[v_1] \leq D_z[u_1] + \ell(u_1, v_1)$. Having proved $D[v] \leq D[u] + \ell(u, v)$ for all arcs (u, v), we conclude that the D[u]'s are the shortest distances by the same argument that was used for the proof of correctness of Algorithm PDM.

To prove that for each u,

$$(\text{SOURCE} \equiv P_z[\ldots P_z[u]\ldots], \ldots, P_z[u], u)$$

is a shortest path, it suffices to show that for each v_1, if $u_1 = P_z[v_1]$ then $D_z[v_1] = D_z[u_1] + \ell(u_1, v_1)$, for it says that a shortest path from SOURCE to u_1 concatenated with (u_1, v_1) forms a shortest path from SOURCE to v_1. Let time a and time b be defined as before. It is easy to see that D[v] is decreased in that execution of Block B, and so $D_b[v_1] = D_z[u_1] + \ell(u_1, v_1)$. Finally, we see that $D_z[v_1] = D_b[v_1]$, because any change of $D[v_1]$ after time b implies a change in $P_b[v_1] = u_1$. This completes the proof of correctness of Algorithm PPDM.

Algorithm PDM and Algorithm PPDM were coded to run on the HEP computer. The programs use linked queue, which is used in Pape [15], and Dial, Glover, Karner, and Klingman [6]. The input network is stored in a linked list structure called the forward star form, used also in [6]. Timing experiments were performed with randomly generated connected networks. Following the characteristics of the Eastern Washington Highway Network, the generated networks were assigned exponentially distributed arc lengths and have approximately 35% of nodes outdegree of one, 9% of nodes outdegree of two, 40% of nodes outdegree of three, and 16% of nodes an outdegree of four. Highway networks usually have all two-way roads, and so do generated networks.* For each NODES = 10, 25, 50, 75, 100, we generated two networks. For each network, we picked five source nodes. Each of these 100 problems are solved with the sequential Algorithm PDM, and the parallel version, Algorithm PPDM, with the number of processors K = 1 to 8. Let T_S denote the solution time for the sequential algorithm, and T_K denote the solution time with the K-processor, parallel algorithm. For each problem, the speed-up, $S_K = \frac{T_S}{T_K}$, and the efficiencies, $E_K = \frac{S_K}{K}$, are computed. For fixed NODES and K, the averages of S_K's and E_K's are plotted in Figure 1 and Figure 2, respectively. For NODES = 75 and 100, we see that a speed-up of approximately three is achieved with five processors, and thus an approximate efficiency of 60%. However, regardless of the number of processors used, we expect that Algorithm PPDM has a constant upper bound on its speed-up, because every process demands private use of the Q.

*A two-way road is represented by a pair of arcs, (u, v) and (v, u), such that $\ell(u, v) = \ell(v, u)$.

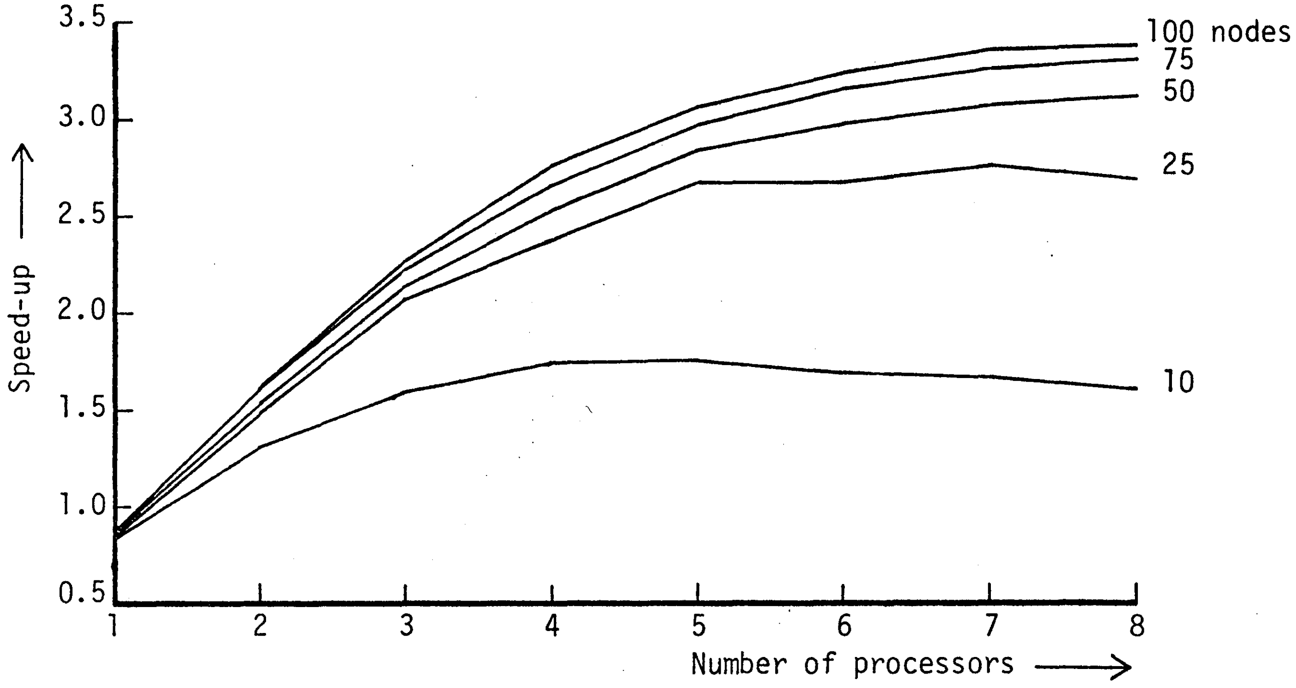


Fig. 1. Average Speed-up

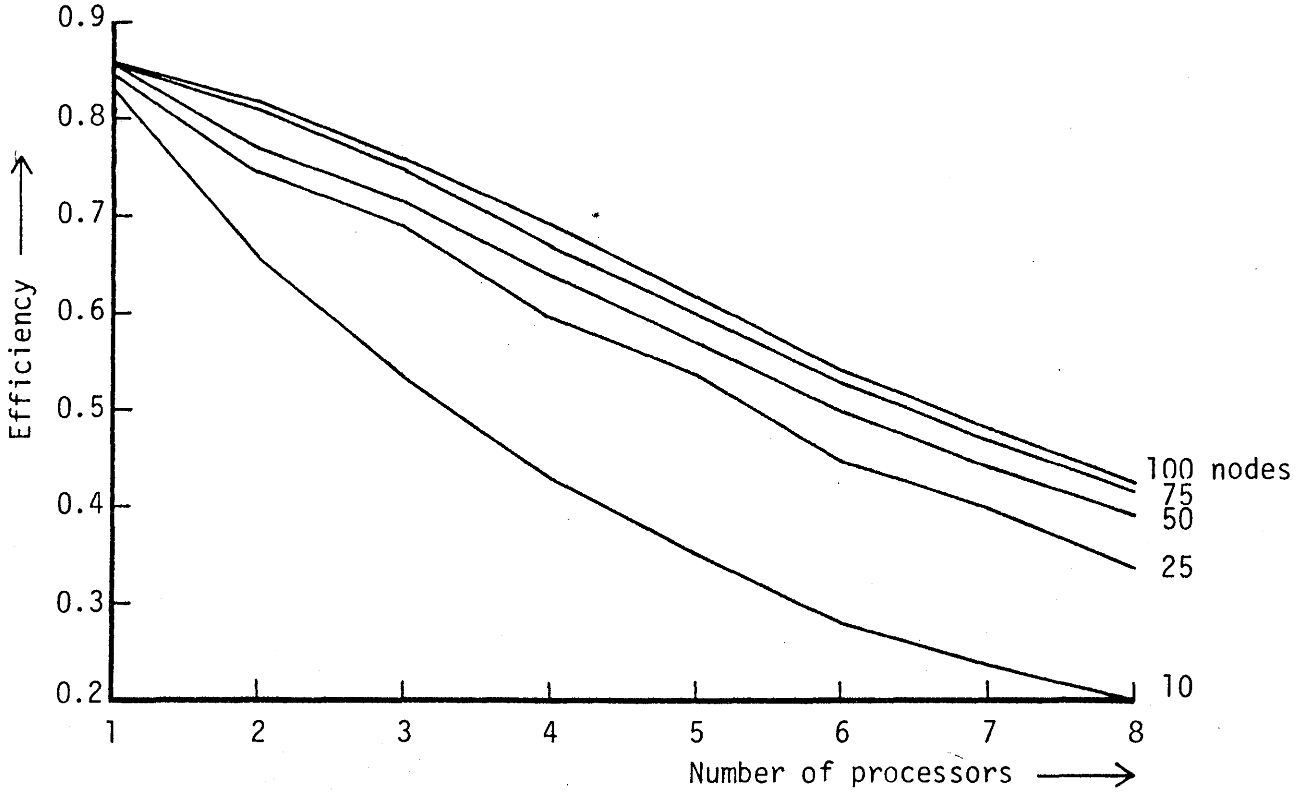


Fig. 2. Average Efficiency

4. A PARALLEL ALGORITHM FOR THE ALL-TO-ALL SHORTEST PATH PROBLEM

The best known algorithm for determining shortest paths between all pairs of nodes is due to Floyd [10], which in turn is based on an earlier algorithm for transitive closure proposed by Warshall [4].

The basic idea of the algorithm may be expressed as follows*:

Algorithm F

```
 1    for k := 1 step 1 until NODES do
        for i := 1 step 1 until NDOES do
 3        for j := 1 step 1 until NODES do
 4          if D[i, j] > D[i, k] + D[k, j] then
*5            D[i, j] := D[i, k] + D[k, j]
```

The matrix, D[•], is initialized to be the arc length matrix. If the input network contains no negative cycle element D[i, j] at the termination is the shortest distance from u to v; because at the end of the k^{th} iteration, D[i, j] is updated to be the shortest distance from i to j via paths that have intermediate nodes which are contained in {1, 2, ..., k}. We will show that the inner loops of Floyd's algorithm may be computed in parallel as follows:

Algorithm PF (Parallel Floyd)

```
1    for k := 1 step 1 until NODES do
2      for 1 ≤ i, j ≤ NODES do simultaneously
3        if D[i, j] > D[i, k] + D[k, j] then
4          D[i, j] := D[i, k] + D[k, j]
```

To prove that Algorithm PF is correct, we use the theory developed for controlling concurrent processes in operating systems. In particular, we use the definition and results in Chapter 2 of [2].

*If the actual shortest paths are desired (in addition to the shortest distances), then statement 5 should be replaced by "begin P[i, j] := P[k, j]; D[i, j] := D[i, k] + D[k, j] end." P[i, j] should be initialized to i if the arc (k, j) exists, and P[i, j] need not be initialized if arc (i, j) does not exist.

We first informally review some definitions. A *task system* $C = (\tau, \lessdot)$ is a set of tasks, $\tau = \{T_1, T_2, \ldots, T_n\}$, together with a precedence relation, $\lessdot$, where $T \lessdot T'$ means that T must be completed before T' begins. Any execution sequence of C must obey the precedence relation. Each task T is associated with two subsets, the *domain* D_T and the *range* R_T, of the memory cells. When T starts it reads values from its domain, and when T terminates it writes values into its range. T and T' are *noninterfering* if either $T \lessdot T'$, or $T' \lessdot T$, or $R_T \cap R_{T'} = R_T \cap D_{T'} = D_T \cap R_{T'} = \emptyset$. Tasks $\{T_1, \ldots, T_n\}$ are *mutually noninterfering* if every pair of tasks T_i and T_j $(i \neq j)$ are noninterfering. We will use the following theorem which is stated and proved in [2], pp. 39-40.

Theorem: Task systems consisting of mutually noninterfering tasks are determinate.

The definition of determinacy of task systems requires a long development, [2], pp. 35-38, which we will not review here. For the purpose of proving the correctness of the Algorithm PF, it suffices to note that determinacy of a task system implies that for the same initial memory state, any execution sequence of the task system will end up with the same final memory state. We will define a set of task systems, and prove that each of them contains mutually noninterfering tasks. Then, we will use the above theorem to conclude that Algorithm F and Algorithm PF compute identical results.

For each $1 \le i, j, k \le$ NODES, let T_{kij} denote the task

"*for* D[i, j] > D[i, k] + D[k, j] *then* D[i, j] := D[i, k] + D[k, j]".

For each k = 1, ..., NODES, define task system $C_k = (\tau_k, \emptyset)$, where task set $\tau_k = \{T_{kij} \mid 1 \le i, j \le \text{NODES}\}$ and $\emptyset$ is the null precedence relation, i.e. no task needs to precede any other task. We will now show that each C_k contains mutually noninterfering tasks, and thus conclude that every execution sequence of C_k produces the same result as Algorithm F's execution sequence does. We will use M_{ij} to denote the memory cell for the variable D[i, j]. $M_{ij} = M_{ab}$ if and only if i = a and j = b. We will use D_{kij} and R_{kij} to denote the domain and range of task T_{kij}.

Remark 7: (a) $D_{kij} = \{M_{ij}, M_{ik}, M_{kj}\}$

(b) $R_{kij} \subset \{M_{ij}\}$

(c) If the input network has no negative cycle, then $R_{kkj} = R_{kik} = \emptyset$.

Parts (a) and (b) follow immediately from the definitions of domain and range of a task. For part (c), T_{kkj} contains the test "D[k, j] > D[k, k] + D[k, j]". Since the network has no negative cycle, D[k, k] is nonnegative.* Thus the test result is always false, and the content of M_{kj} will not be changed. $R_{kkj} = \emptyset$ follows. Similarly, $R_{kik} = \emptyset$ also follows.

Remark 8: If the input network has no negative cycle, then τ_k contains mutually noninterfering tasks.

Because there are no precedence constraints between tasks in τ_k, we need to prove that $R_{kij} \cap R_{kab} = R_{kij} \cap D_{kab} = D_{kij} \cap R_{kab} = \emptyset$, for all $(i, j) \neq (a, b)$. $R_{kij} \cap R_{kab} \subset \{M_{ij}\} \cap \{M_{ab}\} = \emptyset$, because $(i, j) \neq (a, b)$. $R_{kij} \cap D_{kab} \subset \{M_{ij}\} \cap \{M_{ab}, M_{ak}, M_{kb}\} = \emptyset$, for $(i, j) \neq (a, b)$, $j \neq k$, and $i \neq k$. Similarly $D_{kij} \cap R_{kab} = \emptyset$. It follows that τ_k contains mutually noninterfering tasks, for k = 1, ..., NODES. As noted before, this implies that Algorithm PF is correct.

Algorithm PF is programmed to run on the HEP computer. The number of processes created is minimized in order to reduce the overhead (of the create operation). The logic of our program referred to as Algorithm HEPPF (HEP parallel Floyd) is as follows:

*This fact can be proved by an induction argument on k.

Algorithm HEPPF

Process MASTER

```
1      SYN := 0;
2      for ℓ := 1 step 1 until K-1 do
3         create WORKER(ℓ);
4      execute WORKER(K)
```

Process WORKER(ℓ)

```
1      for k := 1 step 1 until NODES do
2      begin
3        for i :=   step K until NODES do
4          if D[i, k] < ∞ then
5            for j := 1 step 1 until NODES do
6              execute $T_{kij}$;
7        lock SYN; SYN := SYN + 1; unlock SYN;
8 L1:    if SYN < K * k then goto L1
9      end
```

Algorithm HEPPF was coded and run for the experimental timing study. Experiments used randomly generated 20-, 30-, and 40-node networks. NODES x NODES arc length matrices with different densities of non-infinity entries distributed uniformly from 0 to 99 were generated. The results of our timing study are shown in Table 1. Let T_K denote the experimental running time of the algorithm with K processors. Let S_K and E_K denote the speed-up, T_1/T_K, and efficiency, S_K/K, respectively. The efficiency of this algorithm for networks with 40, 30 and 20 nodes is plotted in Figures 3, 4 and 5, It is evident that the efficiency tends to be high when the number of nodes in the network is a multiple of K, the number of processors. For in such a case, each WORKER process does exactly the same amount of work,* but in the case where K does not divide NODES exactly, all WORKERs do not do the same amount of processing. For example, for each K, WORKER(1) performs $\left\lceil \frac{\text{NODES}}{K} \right\rceil$ executions of statements

*We assume that the infinity entries are uniformly distributed in the arc length matrix.

Table 1. Running time of Algorithm HEPPF (in secs).

		Density			
NODES = 40		100%	50%	25%	12.5%
No. of Processors	1	1.30478	1.24866	1.13903	0.88217
	2	0.65522	0.63133	0.58283	0.46305
	3	0.45726	0.44399	0.40812	0.32185
	4	0.32989	0.32097	0.29727	0.25366
	5	0.26484	0.25992	0.24512	0.21071
	6	0.23169	0.22906	0.21123	0.17719
	7	0.19889	0.19627	0.18433	0.15915
	8	0.16693	0.16594	0.15423	0.13571
NODES = 30		100%	75%	50%	25%
No. of Processors	1	0.55024	0.53037	0.49828	0.45644
	2	0.27684	0.27116	0.25537	0.23737
	3	0.18544	0.18088	0.17221	0.15966
	4	0.14774	0.14519	0.13785	0.12816
	5	0.11213	0.11039	0.10760	0.09756
	6	0.09417	0.09429	0.08958	0.08582
	7	0.09294	0.08973	0.08699	0.08280
	8	0.07550	0.07559	0.07361	0.06762
NODES = 20		100%	75%	50%	25%
No. of Processors	1	0.16299	0.15615	0.14249	0.11844
	2	0.08213	0.08028	0.07291	0.06457
	3	0.05753	0.05683	0.05195	0.04626
	4	0.04165	0.04086	0.03888	0.03528
	5	0.03348	0.03304	0.03118	0.02770
	6	0.03317	0.03287	0.03016	0.02767
	7	0.02533	0.02541	0.02503	0.02292
	8	0.02513	0.02479	0.02401	0.02166

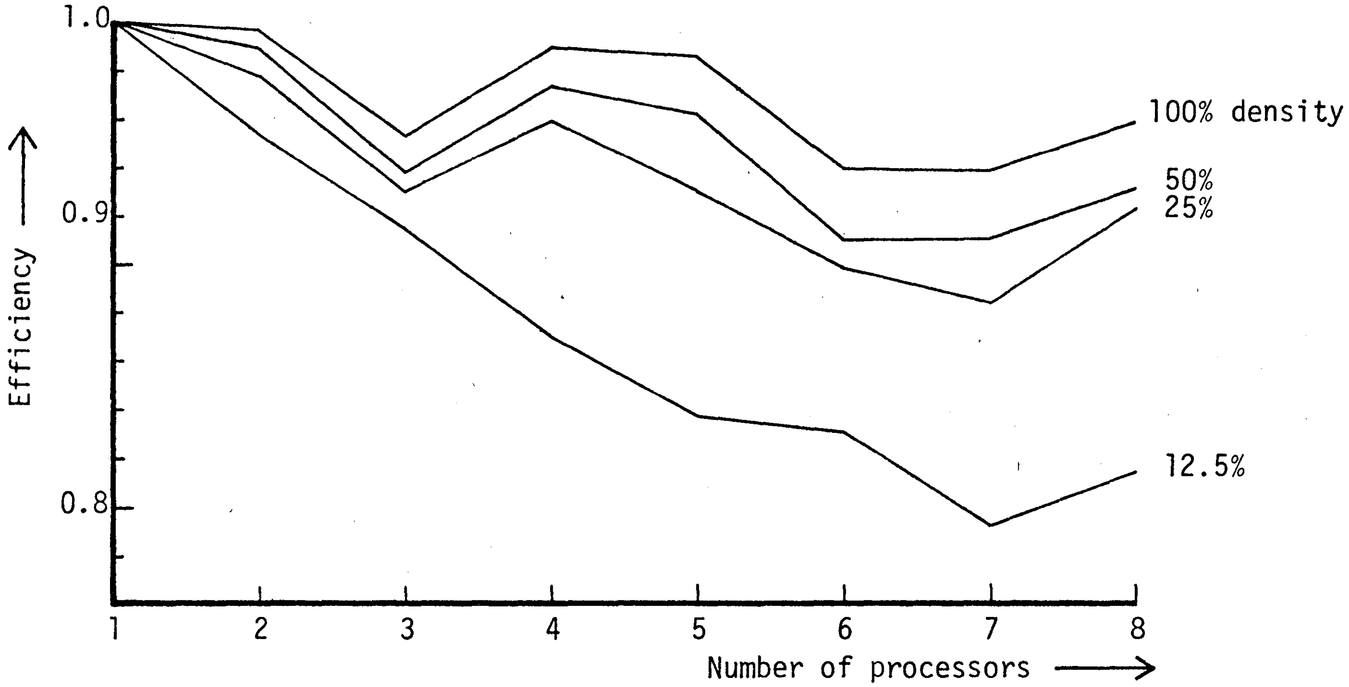


Fig. 3. Efficiency for 40-Node Networks

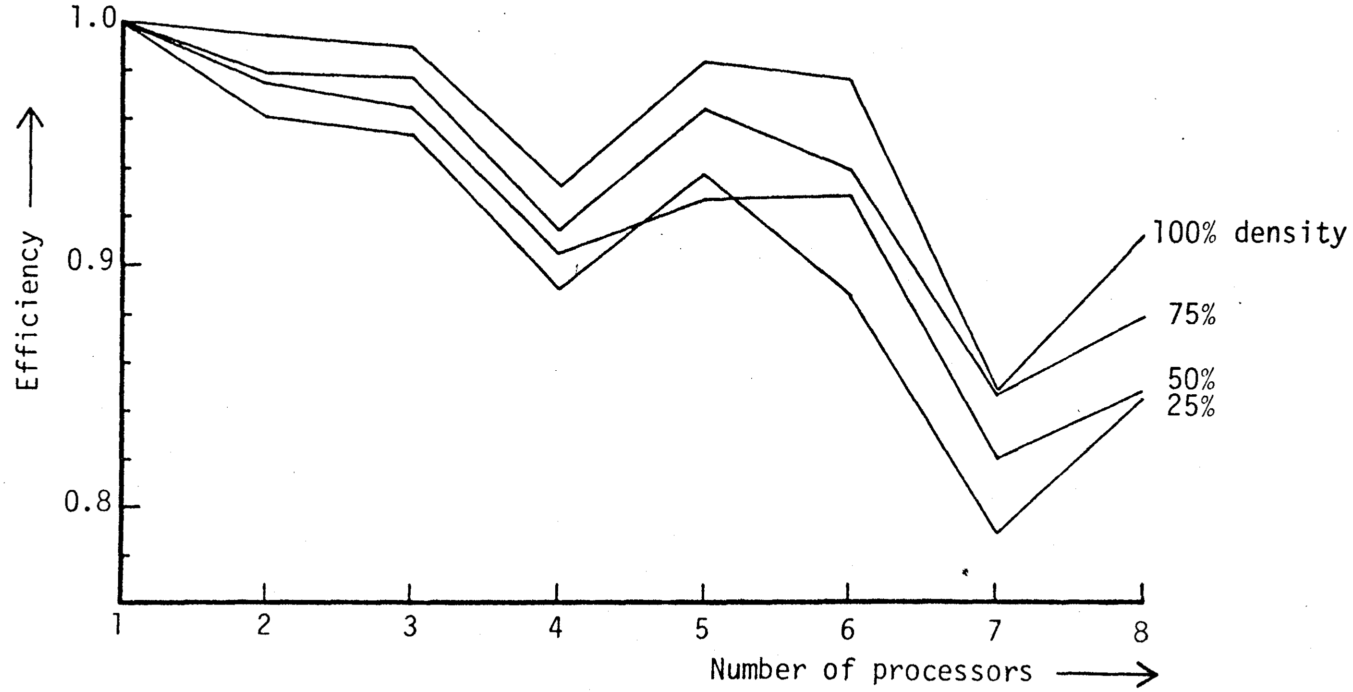


Fig. 4. Efficiency for 30-Node Networks

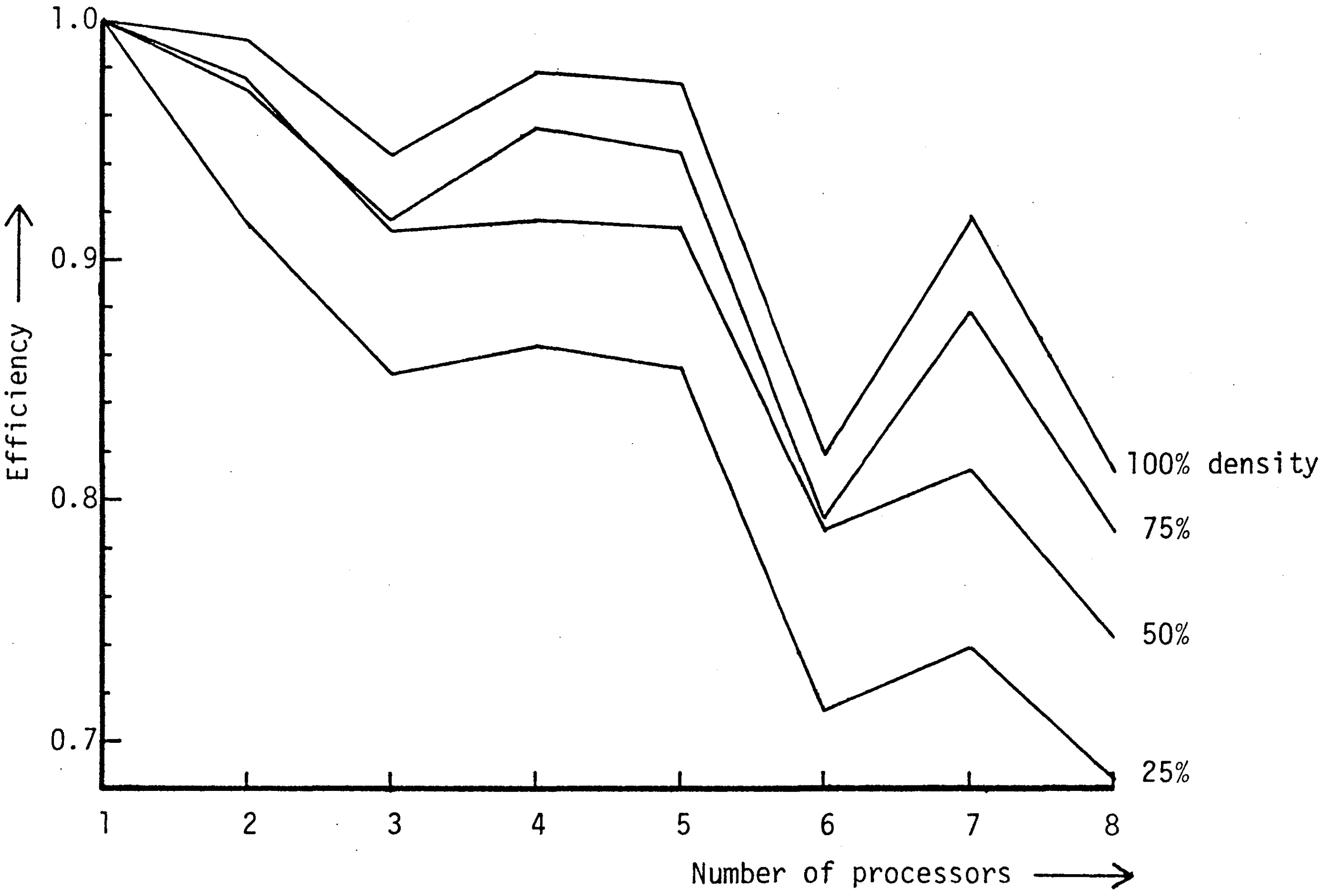


Fig. 5. Efficiency for 20-Node Networks

4 to 6, but WORKER(K) performs $\left\lfloor \frac{\text{NODES}}{K} \right\rfloor$ executions of statements 4 to 6. The WORKERs which finish their work earlier must wait for all others, before starting on the next iteration. Thus the theoretical speed-up should be approximately NODES/ $\lceil$NODES/K$\rceil$. More precisely, if we let t_1 denote the time for executing one iteration of the for loop in statement 3 of procedure of WORKER, and t_2 denote the time for executing statements 1, 8, 9, and 10 once, then the theoretical speed-up is

$$TS_K = \frac{T_1}{T_K} = \frac{(\text{NODES} \quad t_1 + t_2)\ \text{NODES}}{\left(\left\lceil \frac{\text{NODES}}{K} \right\rceil \quad t_1 + t_2\right)\text{NODES}} = \frac{\text{NODES} + \frac{t_2}{t_1}}{\left\lceil \frac{\text{NODES}}{K} \right\rceil + \frac{t_2}{t_1}} \quad .$$

For our compiled code of Algorithm HEPPF, t_2/t_1 is estimated to be approximately 1/(2NODES+1). Using this estimate, the ratio

$$\frac{\text{observed efficiency}}{\text{theoretical efficiency}} = \frac{E_K}{TS_K/K}$$

is calculated and plotted in Figure 6. From this plot we observe that the overhead for the create and the synchronization is relatively small when the input network is dense.

5. CONCLUSION

Two parallel shortest-path algorithms are designed and proved correct in this paper. They were both programmed to run on the HEP computer. For the first algorithm, i.e. Algorithm PPDM, random highway-like sparse networks were generated and used as inputs. We observed empirically a speed-up of three when five processors were employed, for networks with 75 or more nodes. For the second algorithm, i.e. Algorithm HEPPF, random arc-length matrices of order up to 40 were generated and used as inputs. We found that the efficiency is higher for larger and denser networks. Thus we have clearly demonstrated theoretically as well as empirically that parallel processing techniques can be used profitably to speed up determination of shortest paths in large networks. We have also shown how this can be accomplished.

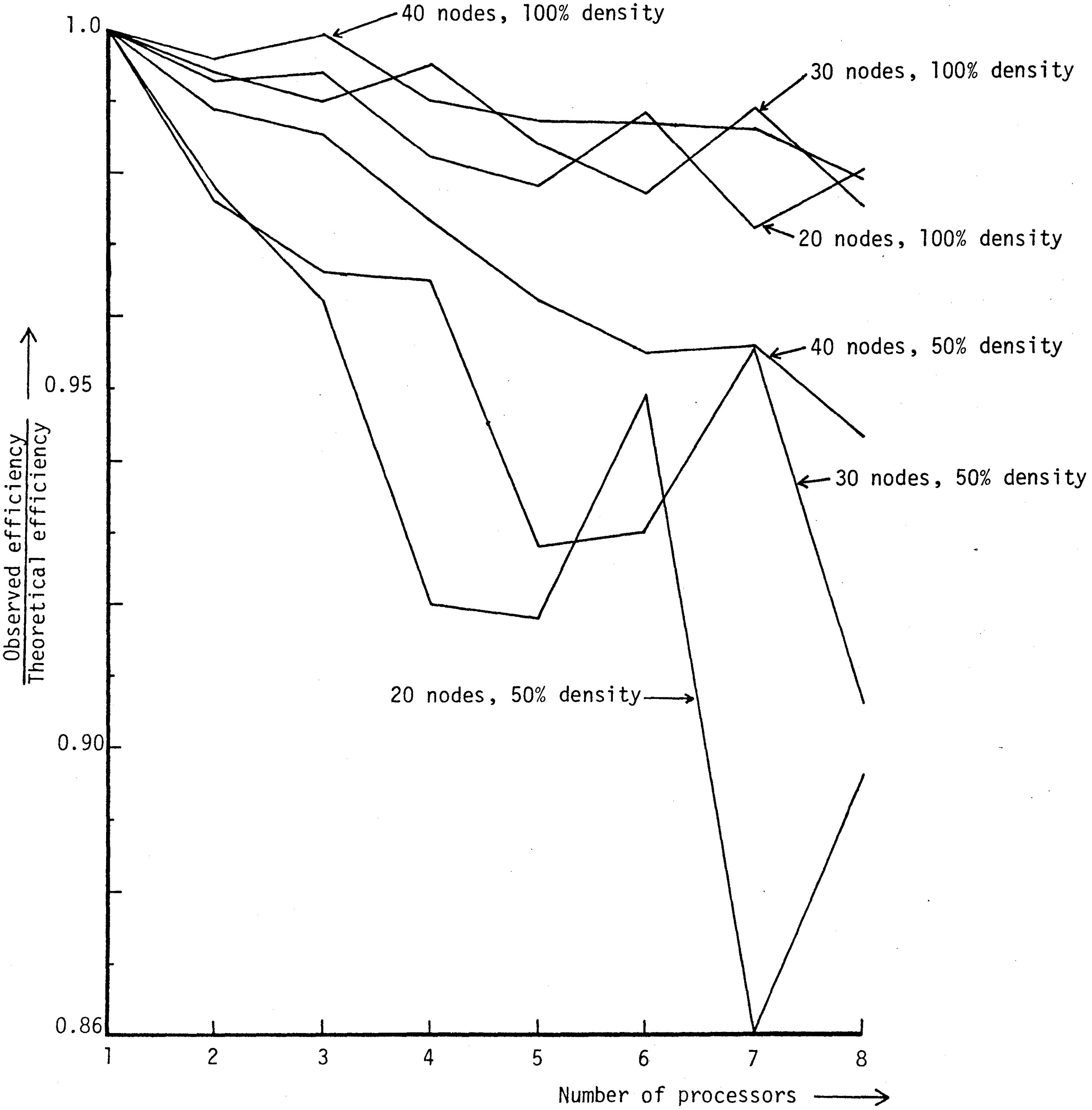


Fig. 6. Observed/Theoretical Efficiency

REFERENCES

[1] E. Arjomandi, A Study of Parallelism in Graph Theory, Doctoral thesis, Dept. of Computer Science, Univ. of Toronto, 1975 (available as Technical Report No. 86).

[2] E. G. Coffman, Jr. and P. J. Denning, Operating Systems Theory, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[3] E. V. Denardo and B. L. Fox, "Shortest-route methods: 1. reaching, pruning, and buckets," Oper. Res., 27 (1979), pp. 161-186.

[4] N. Deo, Graph Theory with Applications to Engineering and Computer Science, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[5] N. Deo and C. Y. Pang, Shortest Path Algorithms: Taxonomy and Annotation, Tech. Report No. CS-80-057, Computer Science Dept., Washington State Univ., Pullman, WA (March 1980).

[6] R. B. Dial, F. Glover, D. Karney and D. Klingman, "A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees," Networks, 9 (1979), pp. 215-248.

[7] E. Dijkstra, "A note on two problems in connexion with graphs," Numerische Mathematik, 1, (1959), pp. 269-271.

[8] D. M. Eckstein, Parallel Processing Using Depth-First and Breadth-First Search, Doctoral thesis, Dept. of Computer Science, Univ. of Iowa, Iowa City, Iowa, July 1977.

[9] D. M. Eckstein and D. A. Alton, "Parallel searching of non-sparse graphs," to appear in SIAM J. Comput.

[10] R. W. Floyd, "Algorithm 97: shortest path," Comm. ACM, 5 (1962), p. 345.

[11] M. J. Flynn, "Very high-speed computing systems," Proc. IEEE, 54 (1966), pp. 1901-1909.

[12] J. Gosch, "Computer processes multiple instruction sets, multiple data streams," Electronics, (Oct. 1979), pp. 77-78.

[13] D. S. Hirschberg, A. K. Chandra and D. V. Sarwate, "Computing connected components on parallel computers," Comm. ACM, 22 (1979), pp. 461-464.

[14] E. F. Moore, "The shortest paths through a maze," Proc. Internat. Symp. on Theory of Switching, 1957, pp. 285-292.

[15] U. Pape, "Implementation and efficiency of Moore-algorithms for the shortest route problems," Math. Programming, 7 (1974), pp. 212-222.

[16] M. Pollack and Wiebenson, "Solutions of the shortest-route problem--a review," Oper. Res., 8 (1960), pp. 224-230.

[17] E. Reghbati (Arjomandi) and D. G. Corneil, "Parallel computations in graph theory," SIAM J. Comput., 7 (May 1978), pp. 230-237.

[18] E. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[19] C. Savage, Parallel algorithms for graph theoretic problems, Ph.D. theses, Math. Dept., Univ. of Illinois at Urbana-Champaign (Aug. 1977), Report ACT-4, Coordinated Science Lab., Univ. of Illinois.

[20] B. J. Smith, "A pipelined, share resource MIMD computer," Internat. Conf. on Parallel Processing, 1978.

[21] D. Van Vliet, "Improved shortest path algorithm for transportation networks," Transportation Res., 12 (1978), pp. 7-20.

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

PARALLEL SOLUTION OF FLIGHT SIMULATION EQUATIONS

BY DR. R. E. LORD
AND DR. S. P. KUMAR

Tomorrow's Computers. . . Today

PARALLEL SOLUTION OF FLIGHT SIMULATION EQUATIONS

R. E. Lord and S. P. Kumar

CS-80-060

This work was partially supported by a grant from DENELCOR, Inc.

1. INTRODUCTION

In recent years, the increasing availability of multi-processor Computer Systems has motivated many Computer Scientists to develop methods to perform computations in parallel. The development of multi-processor computer architecture has been led by the performance, reliability, and the low cost of the digital devices such as microprocessors. One of the several applications of parallel computing is the development of parallel algorithms for continuous system simulation. Frequently a continuous system is described by a set of ordinary differential equations (ODE'S) and simulation consists of numerical integration of these equations. Enough concern has been shown for parallel methods to solve an ODE system by researchers in this field. Nievergelt [1] proposed a method in which parallelism is introduced at the expense of redundancy of computation. An introduction to parallel methods for the numerical solution of ODE systems is given by Miranker and Liniger [2] and Worland [3]. A single bus multi-processor architecture as well as time and speed up ratios between sequential and parallel algorithms, are given in a paper by Franklin [4].

In this paper we present an actual implementation of some of the parallel methods of integration, which individually and in combination were used to solve a set of ordinary differential equations describing the flight characteristics of a ground-launched missile on an MIMD Computer namely HEP (Hetrogeneous Element Processor). The HEP Computer implemented by DENELCOR, INC. is a MIMD machine of shared-resource type as described by Flynn [5].

The rest of this presentation is divided into five sections. A model of an MIMD Computer, specifically the HEP architecture is shown in section 2. In section 3 we discuss some of the integration techniques for which parallel versions have been developed, and also the methods used in our implementation. This presents the idea of algorithm decomposition in parallel computing. The method of problem decomposition applied in our program by equation segmentation is given in section 4. Section 5 contains the actual performance achieved by these programs on HEP, and an analysis of efficiency loss. Section 6 concludes this presentation by describing how the methods used in our programs can be used to design an automatic language translator based upon a continous system simulation language (CSSL [6]), which translates high level language representation of the solution of sets of differential equations into efficient parallel code.

2. MODEL OF MIMD COMPUTER

The HEP computer [7] manufactured by DENELCOR, Inc. was used to verify our algorithms and methodology for solving flight simulation equations. Although this MIMD computer has many interesting architectural features, it is beyond the scope of this paper to present them and instead we present an abstract model which contains the features which we used. A block diagram of how the computer may be viewed by a user is presented in Figure 1. We used the HEP FORTRAN language for programing. This FORTRAN is slightly extended to allow the programmer access to some of the unique features of an MIMD machine and we will describe the model of the computer in terms of these language extensions. Upon commencing execution of a FORTRAN

Abstract

This paper reports on solving a set of differential equations describing the flight characteristics of a missile by use of an MIMD type computer. Two techniques were used: the first, equation segmentation and the second, a combination of equation segmentation and a parallel predictor corrector. Achieved performance is presented and as the result of the work, an outline of the design of a translator for a CCSL type language which produces parallel code is presented.

Keywords: MIMD, ODE's, parallel predictor corrector.

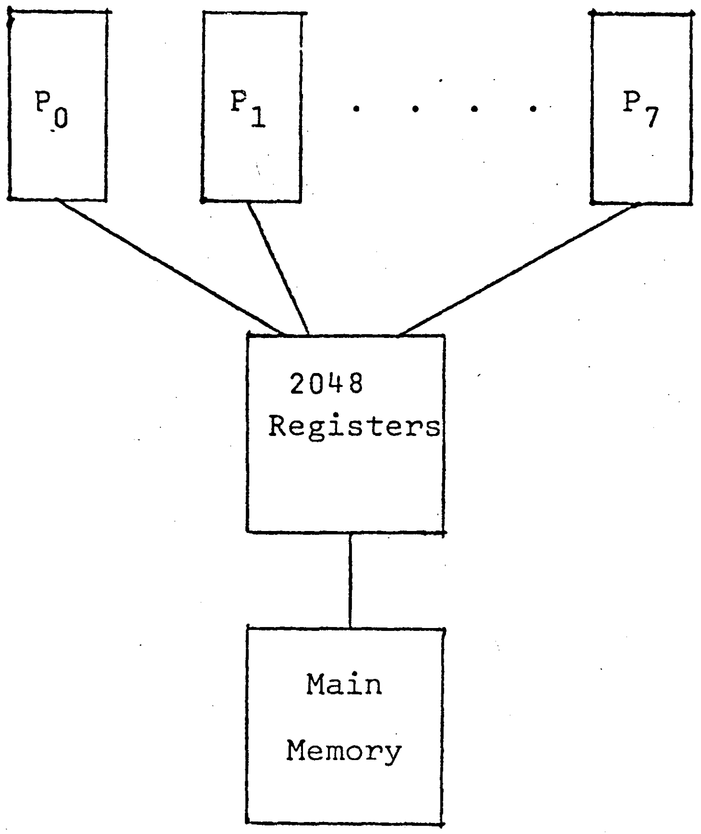


FIGURE 1. Logical representation of MIMD Computer.

program, our MIMD computer model behaves exactly like an SISD computer. That is, a single instruction stream is sequentially executed by one of the processors (CPU) shown in the block diagram. The method of achieving parallel execution is to write a subroutine which can be executed in parallel with the calling program and to then CREATE that subroutine rather than calling it. At that point the instruction stream of the CREATEd subroutine is executing on another processor (CPU) in parallel with the program segment which invoked it. The HEP FORTRAN language has the extension CREATE which may be used at any point where a call statement could be used.

Another extension of the HEP FORTRAN language is in the area of synchronization. Since subroutines may be executing in parallel, they may produce or consume data elements in conjunction with one another. To facilitate this, HEP FORTRAN allows an extension for what is termed asynchronous variables. These variables are distinguished by a naming convention in which the first character of the name is '$'. An integral part of each asynchronous variable, in addition to its data value, is a full-empty semaphore. The appearance of an asynchronous variable on the left-hand side of an assignment statement causes that assignment to be executed only when the associated semaphore is in the empty state and when the assignment is made, the semaphore is set full. Similarly, the appearance of an asynchronous variable within an expression on the right-hand side of an assignment statement causes the expression evaluation to continue only if the associated semaphore is full, and when the expression evaluation continues, the semaphore is set empty. Since these semaphores are supported in hardware, if the required conditions are met, no additional execution time penalties are imposed.

3. INTEGRATION TECHNIQUES

In this section we will be concerned with the parallel methods for the solution of a set of n ODE's denoted by

$$y'(t) = f(t, y(t)) , \quad y(t_0) = y_0 \qquad (1)$$

where

$$t_0 , t \varepsilon R , y_0 \varepsilon R^n , y : R \rightarrow R^n , f : R \times R^n \rightarrow R^n .$$

Most of the methods to solve (1) generate approximations y_n to $y(t_n)$ on a mesh $a = t_0 < t_1 < t_2 < \dots < t_N = b$. These are called step-by-step difference methods. An r-step difference method is one which computes y_{n+1} using r earlier values $y_n, y_{n-1}, \dots, y_{n-r+1}$. This numerical integration of (1) by finite differences is a sequential calculation. Lately, the question of using some of these formulas simultaneously on a set of arithmetic processors to increase the integration speed has been addressed by many authors.

(i) Interpolation Method

Nievergelt [1] proposed a parallel form of a serial integration method to solve a differential equation, in which the algorithm is divided into several subtasks which can be computed independently. The idea is to divide the integration interval [a,b] into N equal subintervals $[t_{i-1}, t_i]$, $t_0 = a$, $t_N = b$, $i = 1, 2, 3, \ldots, N$, to make a rough prediction y_i^0 of the solution $y(t_i)$, to select a certain number M_i of values y_{ij} , $j = 1, \ldots, M_i$ in the vicinity of y_i^0 and then to integrate simultaneously with an accurate integration method M all the systems

$$y' = f(t, y) , \quad y(t_0) = y_0, \; t_0 \leq t \leq t_1$$

$$y' = f(t, y) , \quad y(t_i) = y_{ij} , \; t_i \leq t \leq t_{i+1}$$
$$j = 1, \ldots, M_i , \; i = 1, \ldots, N-1.$$

The integration interval [a,b] will be covered with lines of length (b-a)/N, which are solutions of (1) but do not join at their ends. The connection between these branches is brought by interpolating at $t_1, t_2, \ldots, t_{N-1}$, the previously found solution over the next interval to the right. The time of this computation can be represented by

$$T_{PI} = 1/N \text{ (time for serial integration)}$$
$$+ \text{ time to predict } y_i^0$$
$$+ \text{ interpolation time } + \text{ bookkeeping time}$$

Interpolation can be done in parallel. If we assume that the time consuming part is really the evaluation of f(t, y), the other contributions to the total time of computation become negligible, so that the speed up is roughly 1/N. But to compare this method with serial integration from a to b using method M, the error introduced by interpolation is important. This error depends on the problem, not on the method. For linear problems the error is proved to be bounded but for nonlinear problems it may not be. Thus, the usefulness of this method is restricted to a specific class of problems, and depends on the choice of many parameters like y_i^0, M_i, and the method M.

(ii) Runge-Kutta (RK) Methods

The general form of an r-step RK method, the integration step leading from Y_n to Y_{n+1} consists of computing

$$K_1 = h_n \, f(t_n, y_n)$$

$$K_i = h_n \, f(t_n + a_i h_n, \, y_n + \sum_{j=1}^{i-1} b_{ij} \, K_j)$$

$$y_{n+1} = y_n + \sum_{i=1}^{r} R_i K_i$$

with appropriate values of a's, b's, and R's. A classical 4-step serial RK method is

$$K_1 = h_n\ f(t_n,\ y_n)$$

$$K_2 = hf(t_n + h/2,\ y_n + (1/2)K_1)$$

$$K_3 = hf(t_n + h/2,\ y_n + (1/2)K_2) \qquad \text{(RK4)}$$

$$K_4 = hf(t_n + h\ ,\ y_n + K_3)$$

$$y_{n+1} = y_n + 1/6(K_1 + 2K_2 + 2K_3 + K_4)$$

Miranker and Liniger [2] considered Runge-Kutta formulas which can be used in a parallel mode. They introduced the concept of computational front for allowing parallelism. Their parallel second and third order RK formulas are derived by a modification of Kopal's [8] results, and the parallel schemes have the structure:

first order: $K_1 = h_n\ f(t_n,\ y^1_n)$ (RK1)

$$y^1_{n+1} = y^1_n + K_1$$

second order: $K^2_1 = K_1 = h_n\ f(t_n,\ y^{\ 1}_n)$

$$K_2 = h_n\ f(t_n + ah_n,\ y^1_n + bK^2_1) \qquad \text{(RK2)}$$

$$y^2_{n+1} = R^2_1K^2_1 + R^2_2\ K_2$$

third order: $K^3_1 = K_1$

$$K^3_2 = K_2$$

$$K_3 = h_n\ f(t_n + ah_n,\ y^2_n + bK^{\ 3}_1 + cK^3_2) \qquad \text{(RK3)}$$

$$y^3_{n+1} = R^3_1K^3_1 + R^3_2K^3_2 + R^3_3K_3.$$

The parallel character of the above formulas is based on the fact that RKi is independent of RKj if and only if i < j , i ,j = 1, 2, 3. This implies that if RK1 runs one step ahead of RK2 and RK2 runs one step ahead of RK3. Then using Kopal's values of R's, the parallel third order RK formula is given by:

$$K_{1,n+2} = hf(t_{n+2},\ y^1_{n+2})$$

$$y^1_{n+3} = y^1_{n+2} + K_{1,n+2}$$

$$K_{2,n+1} = hf(t_{n+1} + ah,\ y^1_{n+1} + aK_{1,n+1}) \qquad \text{(PRK3)}$$

$$y^2_{n+2} = y^2_{n+1} + (1-1/2a)K_{1,n+1} + (1/2a)K_{2,n+1}$$

$$K_{3,n} = hf(t_n + a_1h,\ y^2_n + (a_1 - 1/6a)K_{1,n} + (1/6a)K_{2,n})$$

$$y^3_{n+1} = y^3_n + ((2a_1 - 1)/2a)(K_{1,n} - K_{2,n}) + K_{3,n}$$

where

$$a = 2(1-3a_1^2)/(3(1-2a_1)).$$

One value of a suggested by Kopal is $a = 1$. This gives $a_1 = 1/2 + 1/2\sqrt{3}$. The above 3rd order RK formula requires 3 processors to compute the three function evaluations in parallel.

The main drawback of the (PRK3) scheme mentioned above is that it is weakly stable. It is shown in [2] that the scheme leads to an error that grows linearly with n as $n \to \infty$ and $h \to 0$ for $t_n = nh$ = constant. This problem is due to the basic nature of the one step formulas with respect to their y-entries which are the only ones that contribute to the discussion of stability for $h \to 0$.

(iii) Predictor-Corrector (PC) methods

The serial one-step methods of Runge-Kutta type are conceptually simple, easy to code, self starting, and numerically stable for a large class of problems. On the other hand, they are inefficient in that they do not make full use of the available information due to their one-step nature, which, also does not extend the numerical stability property to their parallel mode. It seems plausible that more accuracy can be obtained if the value of y_{n+1} is made to depend not only y_n but also, say, on $y_{n-1}, y_{n-2}, \ldots,$ and f_{n-1}, $f_{n-2}, \ldots$ For this reason multi-step methods have become very popular. For high accuracy they usually require less work than one-step methods. Thus, the desire of obtaining parallel schemes for such methods is reasonable.

A standard fourth order serial predictor-corrector (SPC) given by Adams - Moulton is:

$$y^p_{i+1} = y^c_i + h/24(55f^c_i - 59f^c_{i-1} + 37f^c_{i-2} - 9f^c_{i-3}) \qquad \text{(SPC)}$$

$$y^c_{i+1} = y^c_i + h/24(9f^p_{i+1} + 19f^c_i - 5f^c_{i-1} + f^c_{i-2})$$

The following computation scheme of one PC step to calculate y_{i+1} called PECE is:

1. Use the predictor equation to calculate an initial approximation to y_{i+1}.

set $i = 0$.

2. Evaluate the derivative function f^p_{i+1}.

3. Use the corrector equation to calculate a better approximation to y_{i+1}.

4. Evaluate the derivative function f^c_{i+1}.

5. Check the termination rule. If it is not time to stop, increment i, set $y_{i+1} = y^c_{i+1}$ and return to 1.

Let T_f = total time taken by function evaluations done for one step of PC.

T_{PCE} = time taken to compute predictor (corrector) equation for a single equation.

Then the time taken by one step of SPC is

$$T_1 = 2(nT_{PCE} + T_f).$$

Miranker and Liniger developed formulas for PC method in which the corrector does not depend serially upon the predictor, to that the predictor and the corrector calculations can be performed simultaneously. The Parallel Predictor-Corrector (PPC) operates also in a PECE mode, and the calculation advances s steps at a time. There are 2s processors and each processor performs either a predictor or a corrector calculation. This scheme is shown in Figure 2. A fourth order PPC is given by:

$$y^p_{i+1} = y^c_{i-1} + h/3(8f^p_i - 5f^c_{i-1} + 4f^c_{i-2} - f^c_{i-3}) \qquad \text{(PPC4)}$$

$$y^c_i = y^c_{i-1} + h/24(9f^p_i + 19f^c_{i-1} - 5f^c_{i-2} + f^c_{i-3})$$

Thus the parallel time for a single step of (PPC4) is given by

$$T_{PPC} = nT_{PCE} + T_f + 3nT_{DC} + 2T_S$$

Where

$T_{PCE} = T_f$ as defined before and

T_{DC} = time taken for data communication

T_S = time taken for synchronization.

Generally the high accuracy, less function evaluations of PC methods as compared to RK methods is obtained at the cost of increase in complexity and some times numerical instability. The Parallel RK methods given in [2] do not inherit the stability of their serial counterparts. On the other hand PPC

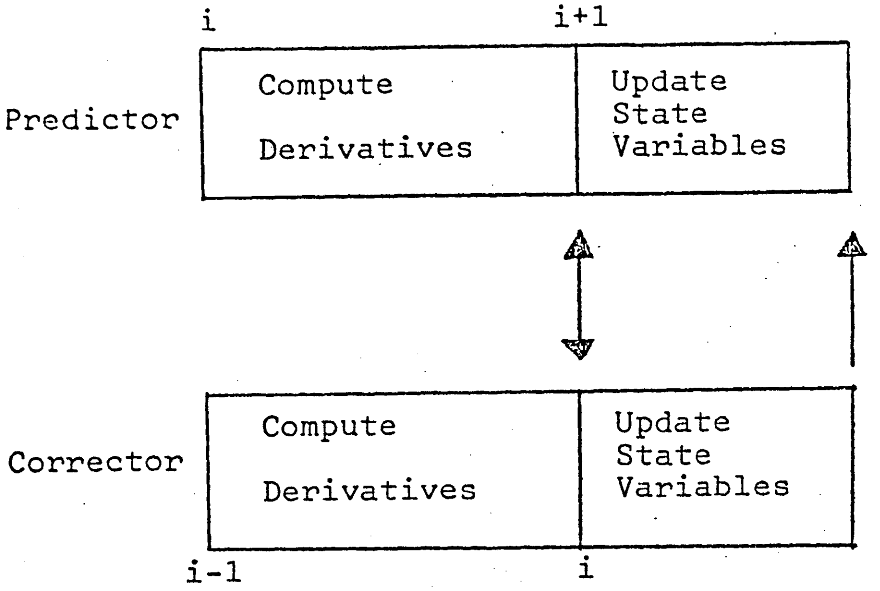


Figure 2. Parallel PC Scheme

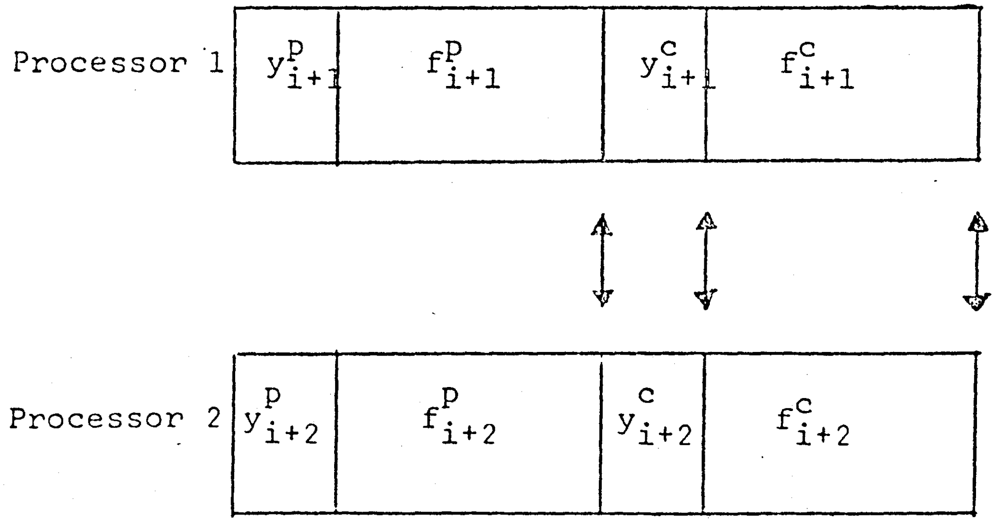


Figure 3. Parallel Scheme for BPC

methods in [2] as described above are as stable as their serial formulas. This is proved by Katz et al. [7]

(iv) Block-Implicit methods

Sequential block implicit methods as described by Andria et al. [8] and Shampine and Watts [9] produce more than one approximation of y at each step of integration. Shampine and Watts and Rosser [10] discuss block implicit methods for RK type and PC type schemes. A 2 point fourth order PC given in [9] is

$$\begin{aligned}
y^p_{i+1} &= 1/3(y^c_{i-2} + y^c_{i-1} + y^c_i) + h/6(3f^c_{i-2} - 4f^c_{i-1} + 13f^c_i) \\
y^p_{i+2} &= 1/3(y^c_{i-2} + y^c_{i-1} + y^c_i) + h/12(29f^c_{i-2} - 72f^c_{i-1} + 79f^c_i) \\
y^c_{i+1} &= y^c_i + h/12(5f^c_i + 8f^p_{i+1} - f^p_{i+2}) \qquad \text{(BPC)} \\
y^c_{i+2} &= y^c_i + h/3(f^c_i + 4f^p_{i+1} + f^p_{i+2})
\end{aligned}$$

Worland in [3] describes the natural way to parallelize (BPC) using the number of processors = number of block points by the schemes shown in Figure 3. The parallel time for one Block calculation given by Franklin [4] is

$$T_{BPC} = (2nT_{PCE} + 2T_f + 6nT_{DC} + 4T_S)/2$$

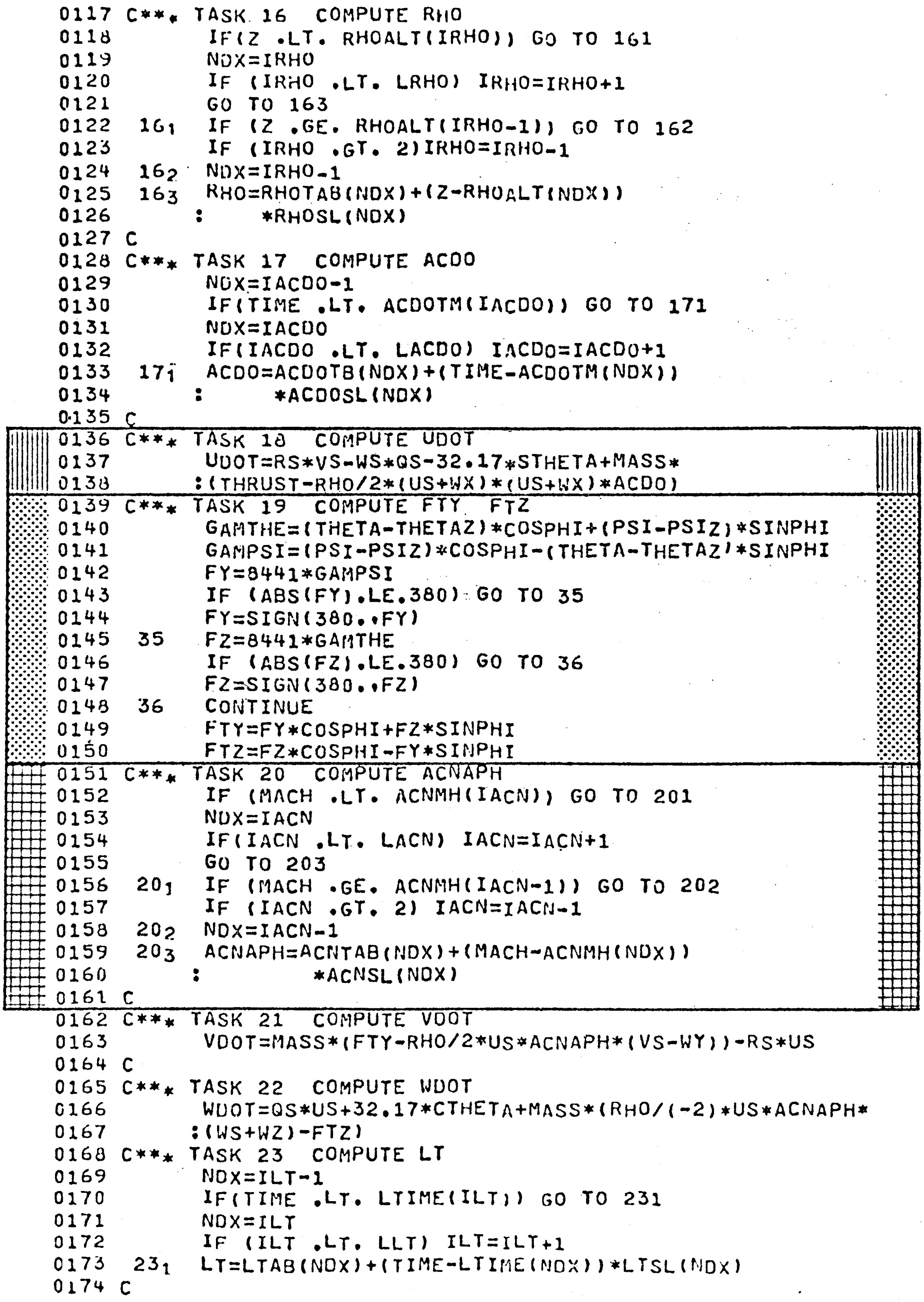
a performance comparison of (PPC) and parallel (BPC) methods is given by Franklin in [2] in case of two processors.

4 METHODOLOGY

The methodology which we employed in programming the flight simulation equations for solution on an MIMD computer can be divided into several catagories. These include equation segmentation, scheduling and synchronization. These categories will be discussed individually.

(i) Segmentation

Equation Segmentation is to take some representation of the problem, in our case a sequential FORTRAN Program, and identify the tasks [13]. These tasks are considered to be individual computational activities and could range from individual machine instruction to individual FORTRAN statements. Our choice was individual statements or small groups of statements where any branching took place entirely within the group of statements that was identified as a task. An example of this task selection is shown in Figure 4 where a portion of the sequential code is shown together with an indication of some specific tasks. In this specific case a total of 40 tasks were identified, 10 of them being the update of the state variables by means of the chosen integration method, one being the update of the independent variable time and the remaining 29 tasks associated with the evaluation of the



```
C**. TASK 16  COMPUTE RHO
      IF(Z .LT. RHOALT(IRHO)) GO TO 161
      NDX=IRHO
      IF (IRHO .LT. LRHO) IRHO=IRHO+1
      GO TO 163
 161  IF (Z .GE. RHOALT(IRHO-1)) GO TO 162
      IF (IRHO .GT. 2)IRHO=IRHO-1
 162  NDX=IRHO-1
 163  RHO=RHOTAB(NDX)+(Z-RHOALT(NDX))
     :     *RHOSL(NDX)
C
C**. TASK 17  COMPUTE ACDO
      NDX=IACDO-1
      IF(TIME .LT. ACDOTM(IACDO)) GO TO 171
      NDX=IACDO
      IF(IACDO .LT. LACDO) IACDO=IACDO+1
 171  ACDO=ACDOTB(NDX)+(TIME-ACDOTM(NDX))
     :      *ACDOSL(NDX)
C
C**. TASK 18  COMPUTE UDOT
      UDOT=RS*VS-WS*QS-32.17*STHETA+MASS*
     :(THRUST-RHO/2*(US+WX)*(US+WX)*ACDO)
C**. TASK 19  COMPUTE FTY  FTZ
      GAMTHE=(THETA-THETAZ)*COSPHI+(PSI-PSIZ)*SINPHI
      GAMPSI=(PSI-PSIZ)*COSPHI-(THETA-THETAZ'*SINPHI
      FY=8441*GAMPSI
      IF (ABS(FY).LE.380) GO TO 35
      FY=SIGN(380.,FY)
 35   FZ=8441*GAMTHE
      IF (ABS(FZ).LE.380) GO TO 36
      FZ=SIGN(380.,FZ)
 36   CONTINUE
      FTY=FY*COSPHI+FZ*SINPHI
      FTZ=FZ*COSPHI-FY*SINPHI
C**. TASK 20  COMPUTE ACNAPH
      IF (MACH .LT. ACNMH(IACN)) GO TO 201
      NDX=IACN
      IF(IACN .LT. LACN) IACN=IACN+1
      GO TO 203
 201  IF (MACH .GE. ACNMH(IACN-1)) GO TO 202
      IF (IACN .GT. 2) IACN=IACN-1
 202  NDX=IACN-1
 203  ACNAPH=ACNTAB(NDX)+(MACH-ACNMH(NDX))
     :         *ACNSL(NDX)
C
C**. TASK 21  COMPUTE VDOT
      VDOT=MASS*(FTY-RHO/2*US*ACNAPH*(VS-WY))-RS*US
C
C**. TASK 22  COMPUTE WDOT
      WDOT=QS*US+32.17*CTHETA+MASS*(RHO/(-2)*US*ACNAPH*
     :(WS+WZ)-FTZ)
C**. TASK 23  COMPUTE LT
      NDX=ILT-1
      IF(TIME .LT. LTIME(ILT)) GO TO 231
      NDX=ILT
      IF (ILT .LT. LLT) ILT=ILT+1
 231  LT=LTAB(NDX)+(TIME-LTIME(NDX))*LTSL(NDX)
C
```

Figure 4. Example of Task Selection

derivatives. The next step was to estimate the execution time of each of these tasks. Since the HEP computer executes all instructions in the same amount of time, this involved compiling the program and counting the number of machine instructions generated by each of the tasks. For our task selection the number of instructions per task ranged from 2 to 88 with an average of 34.6. We next determine a maximally parallel task system equivalent to the set of tasks selected and the sequential program for those tasks. The details of this construction are contained in chapter 2 of [13] but the essential features are presented here. Consider a numbering of the tasks in the sequential program such that for T_i and T_j then the execution of T_i occured prior to the execution of T_j only if $i < j$. We then ask for each pair of task T_i and T_j a) if the output variables for each of the tasks (variable names on the left side of the assignment statement) are distinct and b) if the variables used as input to each of the tasks is distinct from the output variables of the other task. If the answer to both a and b are true, then T_i and T_j may be executed in parallel. The resultant task system may be represented by a directed graph where the nodes represent the tasks and the arcs the precedence constraints where, if there is an arc from T_i to T_j then task T_i must complete execution prior to commencing execution of task T_j. Figure 5 shows the resultant task system for the 40 tasks comprising the problem solution. Our convention is to show the task number and execution time in machine instructions within the node and all arcs go from left to right. One can observe that the three tasks (18, 19, 20) highlighted in Figure 4 can all be executed in parallel.

(ii) Scheduling

Prior to actually scheduling, we make a transformation on the maximally parallel task system which may allow a shorter overall solution time. If one examines the graph of Figure 5 we see that the maximum length path traverses nodes 7, 39, 19, 31 and has a length of 212 units. However, there is no path from node 31 to node 7 and consequently, this path of length 212 times the number of iterations (time steps) does not determine the maximum execution time to solve the problem. This minimum execution time is instead determined by the cycle traversing 7, 39, 19, 32, 6, 3 length 252 which yields a minimum execution time for n iterations of n * 126 + constant. The details of the algorithims for determining the minimum execution time for n interations of a task system such as is shown in Figure 5, together with algorithims for transforming the task system are given in [14]

Scheduling the transformed task system for execution on p processors is the next step in our methodology. Ullman [15] has shown that the computation of an optimal schedule involving multiple processors and a task system such as ours is an NP-complete problem. Hence, they can be regarded as computationally intractable. However, polynomialy bound algorithms do exist which produce good schedules. An example of one such algorithm is the critical path list scheduling algorithm. Basically we define this algorithm by:

Def. 1: Given a task system and a list which orders the tasks, then a scheduling strategy which assigns to a free processor the first unassigned task

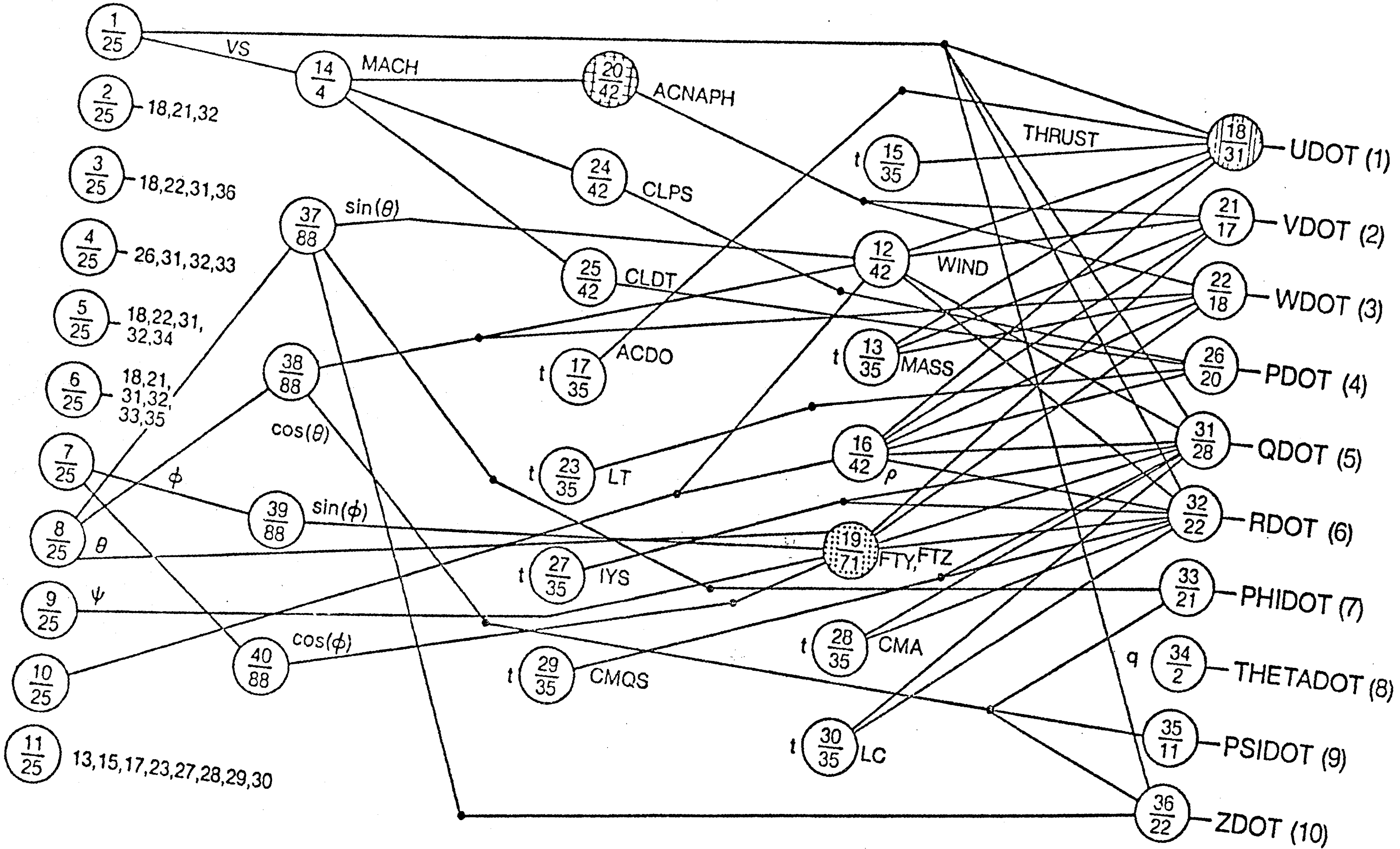


Figure 5. Maximally Parallel Task System

in the list whose precedence constraints have been met is called demand list scheduling.

Def. 2: The critical time of a task is the execution time of that task plus the maximum of the critical time of any of its successor tasks.

Def. 3: If the tasks are ordered in a list on non-increasing critical time, then a resultant list schedule is called critical path list scheduling.

Kohler [16] reported a preliminary evaluation in which 20 task systems were scheduled using critical path list scheduling which produced 17 optimal schedules. The worst case schedule was only 3.4% longer than an optimal schedule. Using only limited back-tracking with a critical path list scheduler, Lord [14] found in 100 randomly generated cases 89 were scheduled optimally. He further found that for all cases that schedules had an expected time of only .36% longer than optimal. The worst case time was 5.6% longer than optimal. Thus, we conclude that critical path list scheduling is an acceptable technique for practical applications.

Applying a limited backtrack critical path list scheduling algorithm to the task system representing the missile simulation resulted in a schedule for 8 processors as shown in Figure 6. An optimal schedule was not calculated, but this schedule is known to be no more than 9.1% longer.

(iii) Synchronization

Having determined a schedule for computing the tasks, it now remains to find means of implementing it. Much of the work on scheduling assures, at least implicitly, that some mechanism external to the processors assigns the tasks to the procssors. But since our execution times are estimates'only, the scheduling mechanism would have to monitor the progress of all of the processors. Instead, we use a mechanism whereby all of the tasks to be executed by a single processor are presented as a sequential program. The coordination of those tasks is accomplished by means of synchronization using the full/empty semaphores associated with each data location in the HEP computer. Specifically, we note from Figure 6 that task 35, the computation of PSIDOT, is executed by processor 5, whereas task 9, the update of the state variable PSI, is executed by processor 8. To insure that processor 8 does not commence executing the code of task 9 prior to task 35 having been completed by processor 5 we use an asynchronous variable common to both processors which is initially set empty and is set full upon completion of task 35. Prior to starting execution of task 9, the value of PSIDOT is read from this cell. Specifically, the code sequences would appear as:

```
Processor 5                    Processor 8
    .                              .
    .                              .
code for task 35               PSIDOT = $T
$T = PSIDOT                    code for task 9
    .                              .
```

Had PSIDOT been required by another task executing is yet another processor then a second asynchronous variable would have been required to synchronize

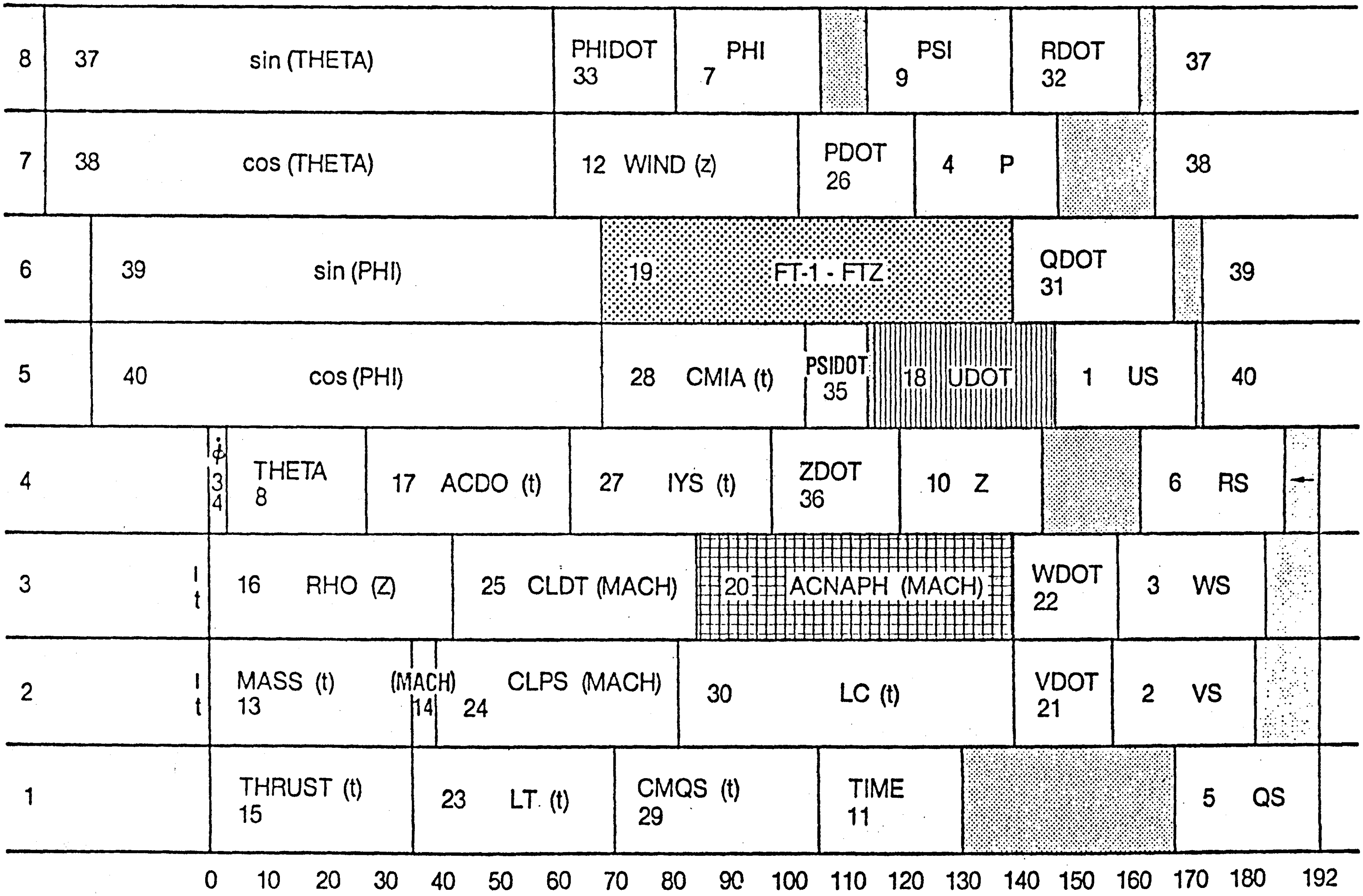


Figure 6. A Schedule Using 8 Processors

those calculations. For the 8 processor schedule, a total of 78 asynchronous variables were required to synchronize the calculations.

5. ACHIEVED PERFORMANCE

The schedules for the flight simulation problem discussed in Section 4 were programmed using HEP FORTRAN and were executed on the HEP parallel computer. The computational results are shown in Table 1. The sequential times T_1 and the parallel times T_p with p processors are given in terms of seconds.

The method of equation segmentation in conjunction with 4th order Runge-Kutta formula given by (RK4) was used for the eight-processor schedule shown in Figure 6. The computations of the integration formula were also done as parallel tasks. This scheme was also programmed using six processors and the speed up in this case was $S_6 = 3.98$. The speed up and efficiency of the 8 processors program is given by Table 1. Subsequent analysis has shown that the speed up S_8shown in Table 1 can be increased to 7.0.

The four-processor schedule was run in combination with the parallel predictor-corrector formula given by (PPC). The program created eight instruction streams in parallel, four for predictor and four for corrector iteration. The achieved speed up and efficiency in this case, as compared to the serial program is shown in Table 1. Since the Serial PC methods are expected to be more efficient than the Serial RK methods, the difference in speed up of their parallel mode is also to be expected. On the other hand, the data communication and synchronization in parallel predictor-corrector is more than the method using RK formula. These calculations are done in the following analysis of the loss of the efficiencies in both the programs.

let

A = number of cycles required by actual computation,
B = number of cycles required by the best schedule,
C = number of cycles required by synchronization.

For the eight-processor scheme with RK method the values of A, B, C are A = 1384/8 = 173 cycles; B = 192 cycles = 10.9% of A; C = (78 + 2)/8 = 19.5 average and C = 23 for worst case = 11.9% of B. The total number of cycles is then given by

$$\begin{aligned} \text{Cycles} &= A + (B - A) + C \\ &= 173 + 19 + 23 = 215, \end{aligned}$$

and the predicted solution time is given by

$$\text{PST} = \text{Cycles} \times 28{,}000 \times .8 \times 10^{-6} = 4.816 \text{ seconds}$$

where the actual solution time given by Table 1 is 4.87 seconds.

For the four-processor PC method the values of A, B, C are A = 1384/4 = 346;

PROGRAM	P	T_1	T_p	S_p	E_p
RK	8	28.18	4.87	5.78	72.3%
PC	8	21.59	3.33	6.48	81 %

TABLE 1. Actual Speedup & Efficiency .

B = 363 = 4.9% of A; C = (86 x 2)/4 + 50/8 = 55.5 average and C = 58 in worst case = 15.9% of B. This gives the total number of cycles required by the program

$$\text{Cycles} = A + (B - A) + C$$
$$= 356 + 17 + 58 = 421 \text{ cycles,}$$

which gives the predicted efficiency for PC method

$$PE = 356/421 = 82\%$$

where the actual efficiency given by Table 1 is 81%.

6. AUTOMATIC TRANSLATION

Based upon our experience with parallel solution of flight simulation equations, we conclude that, given some suitable representation of the problem, the entire procedure for generating the parallel program could be automated. We feel that a suitable representation of the problem is a CSSL-type language where our predominant focus is that the derivatives are clearly defined and the integration technique is specified but not explicitly programmed by the user. The specific steps for producing a parallel program are listed below and will be individually discussed from the standpoint of automation of translation.

1. Select tasks.
2. Determine execution time of each task.
3. Determine precedence constraints amongst tasks.
4. Transform the task system into one with minimum path length.
5. Schedule the transformed task system for execution using p processors.
6. Synchronize the resultant schedule by use of asynchronous variables.

To perform task selection we first require that the representation of the problem provides names for each of state variables and the associated name for the derivative. In our resultant translation, the specific code to update each of the state variables, based upon the integration technique specified, will each be a separate task. For the representation of the derivatives we require that there be no conditional or unconditional branches. We envision that this presents no problem to the user by postulating a library of functions which might include arbitrary function generation, limit function, switces, etc. Then in addition to the tasks for updating the state variables, we will define further tasks by isolating all function evaluation (library and user defined) as separate tasks and all assignment statements (less function evaluation) as separate tasks. Although the mechanism of doing this is an implementation detail, we will describe it as if it were performed by actual text substitution. For example, if we were given:

```
     XDOT = X + ARBFUN (Z)
        X = INT (XDOT)
```

we would translate this as :

```
C TASK 1
  TEMP = ARBFUN (Z)
C TASK 2
  XDOT = X + TEMP
C TASK 3
     X = INT (XDOT)
```

Given the selection of tasks we next need to determine the execution time of each of the tasks. Again assuming we have text for the individual tasks we could now compile this text using the HEP FORTRAN compiler. As an output of the compiler, the number of instructions for each of the tasks is available and this together with the known execution time of the library functions together with the user supplied estimates for user functions would determine the estimates of execution time for each of the tasks.

The precedence constraints for each of the tasks can be determined by computing for each task the variables which are in the tasks domain (input variables) and the variables which are in the tasks range (output variables). Based upon this and the sequential ordering given by the derivative statements, a maximally parallel system can be determined. This data together with the execution time estimates should now be available in a data structure representing a directed, weighted graph. Programs to determine the ranges and domains and produce the precedence constraints have been developed in PASCAL at Washington State University.

The next two steps, that of transforming the graph of the preceeding paragraph and producing a schedule using p processors is described in [14]. Programs to accomplish this transformation and to produce a schedule were developed at Washington State University using the PL/I language. Given this schedule we next produce p subroutines where each subroutine consists of the code for the tasks which were scheduled for that processor. Each subroutine will also contain code to determine whether the end condition of the simulation has been satisfied and if not to branch back and execute another iteration.

The final step in the translation is to add the code for synchronizing the p subroutines. This is accomplished by, for each task in the system, asking if the variables in its range are in the domain of any task assigned to another processor. For each such variable an asynchronous variable (e.g. $T(J)) is put in COMMON and assigned a value by the task which contains that variable in its domain. For the task which uses that variable, an assignment from the asynchronous variable is made prior to the code for that task. Programs to perform this synchronization do not presently exist but thier design does not seem to present any difficulty.

CONCLUSIONS

We conclude from our experience in solving flight simulation equations on the HEP parallel computer that these techniques offer a viable alternative to normal sequential computing and that the HEP computer is sufficiently fast to provide at least real time support for many cases. Further, should this type of computer become generally accepted for solving differential equations, then support in the form of high level programming languages is feasible.

REFERENCES

[1] J. Nievergelt, "Parallel Methods for Integrating Ordinary Differential Equations," CACM, vol. 7, no. 12, pp. 731-733, December, 1964.

[2] N. L. Miranker and W. M. Liniger, "Parallel Methods for the Numerical Integration of Ordinary Differential Equations," Math. Comput., vol. 21, pp. 303-320, 1967.

[3] P. B. Worland, "Parallel Methods for the Numerical Solution of Ordinary Differential Equations," IEEE Trans. Comp., vol. C-25, pp. 1045-1048, October, 1976.

[4] M. A. Franklin, "Parallel Solution of Ordinary Differential Equations," IEEE Trans. Comp., vol. C-27, no. 5, May, 1978.

[5] M. J. Flynn, "Some Computer Organizations and their Effectiveness," IEEE Trans. Comp., vol. C-21, September, 1972.

[6] J. C. Strauss et. al., "Continuous System Simulation Language," SIMULATION, vol. 6, no. 12, December, 1967.

[7] DENELCOR,Inc., "HEP Principles of Operations," June, 1979.

[8] Z. Kopal, "Numerical Analysis with Emphasis on the Application of Numerical Techniques to Problems of Infinitesimal Calculus in Single Variable," Wiley, New York; Chapman & Hall, London, 1955 MR 17,1007.

[9] N. Katz, M. A. Franklin, and A. Sen, "Optimally Stable Parallel Predictors for Adams - Moulton Correctors," Comp. & Maths. with Appls., vol. 3, pp. 217-233, 1977.

[10] F. D. Andria, G. D. Byrne, and D. R. Hill, "Natural Spline Block Implicit Methods," BIT, vol. 13, pp. 131-144, 1973.

[11] L. F. Shampine and H. A. Watts, "Block Implicit One Step Methods," Math. Comput., vol. 23, pp. 731-740, 1964.

[12] J. Rosser, "A Runge-Kutta for all Seasons," SIAM Rev., vol. 9, pp. 417-452, July, 1967.

[13] Coffman and Denning, "Operating Systems Theory," Prentice Hall, Englewood Cliffs, N.J., 1974.

[14] R. E. Lord, "Scheduling Recurrence Equations for Solution on MIMD Type Computers," PhD Dissertation, Washington State University, 1976.

[15] J. D. Ullman, "NP-Complete Scheduling Problems", Journal of Computer and System Sciences 10, pp 384-393, 1975

[16] Walter Kohler, "Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on a Multiprocessor System", IEEE Trans. Comp. C24, pp 1235-1238, 1975.

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

COMPUTER IMAGE GENERATION USING MIMD COMPUTERS

BY DR. R. A. SCHMIDT

Tomorrow's Computers. . . Today

COMPUTER IMAGE GENERATION USING MIMD COMPUTERS

I. BACKGROUND

Computer image generation (CIG) is a technique used to synthesize television images of scenes. Its primary use is in very realistic vehicle simulators. The viewable region is stored as a three-dimensional model in a computer data base, and a combination of computers and special-purpose hardware is used to select the portion of the data base to be displayed, convert it to a two-dimensional perspective view, and build raster scan lines for a TV monitor. In order to provide sufficient realism in a flight simulation, a complete frame of imagery must be calculated in roughly 1/60 second. The data base may contain several hundred objects, of which several dozen may be in view at any given time.

Conventional SISD computers cannot process data fast enough to perform the CIG function. In present systems, several SISD computers are used in a distributed-computation configuration to perform the acquisition of view angle data and the selection of viewable objects. The transformation of objects to two dimensional representation, the simulation of haze, and the conversion to scan lines are handled by hardware. In addition, present systems assume that the viewable background is static, and all scene motion is produced by motion of the simulated vehicle.

Multiple-instruction multiple-data stream (MIMD) computers executing 10**7 to 10**8 instructions/second offer the opportunity to perform complex scene generation using general-purpose (e.g. FORTRAN) computer languages with a minimum of special purpose hardware. By performing a greater portion of the CIG task in software, MIMD image generation would permit greater flexibility and greater complexity of scenes. The ability to incorporate heuristic techniques deep in the image transformation process could allow the optimal use of computing resource against the most visible portions of a scene, while minimizing computation on minor scene elements.

II. TECHNICAL DISCUSSION OF MIMD COMPUTERS

MIMD is a form of parallel computation in which multiple instructions execute simultaneously in a single computer system. It differs from distributed processing in that the multiple instruction streams may be tightly coupled and cooperate on a word-by-word basis in the solution of a single problem with very low overhead. Where in a distributed processing system, interprocess communication is a software function of the operating system, in MIMD such communication is implemented by hardware synchronization mechanisms. MIMD parallelism differs from array processors in that multiple instruction streams exist, and may be as tightly or loosely coupled as the application demands.

At the present time, the only commercially available MIMD computer is manufactured by Denelcor, Inc. of Denver, Colorado. The Denelcor processor, called HEP (for Heterogeneous Element' Processor) contains four different types of memory: program, register, constant, and data. Programs executing on the machine are allocated a "task" in which to run. Each "task" defines a contiguous region of each type of memory. The hardware restricts each user to his own region of memory, and restricts the type of access he may make to each memory type. Program memory is execute-only; constant memory is read-only; and register and data memory are read/write.

A task may contain one or several processes, which are executable code sequences. Several processes may be simultaneously executing in the HEP, unlike conventional computers. Processes are implemented by a set of hardware registers, of which there is a fixed number; thus an error condition (create fault) exists when too many processes come into existence in the processor. Since existing processes can create new processes at will, processes must be allocated to tasks and managed just as memory must be allocated and managed.

All of the sixteen hardware implemented tasks in the HEP are not equivalent. Tasks 0-7 are user tasks. In these tasks, privileged instructions are forbidden. In tasks 8-15, privileged instructions are allowed. These tasks, called "supervisors", perform system services for the user tasks. User tasks request these services with "supervisor call" (SVC) instructions. These instructions generate a "trap", creating a process in a supervisor task and suspending execution of all processes in the user. The hardware forces user traps to a particular supervisor task, for example, task 2 traps to task 10. In general, task k(k<8) traps to task k+8.

Supervisors may also generate traps. All traps from a supervisor create a process in task 8. A supervisor trap suspends the supervisor in the same way a user trap suspends the user.

The HEP operating system is organized into two main components: the Kernel and the Supervisors. The users (in tasks 1-7) make service requests (via SVC) of their corresponding supervisors. In the event of user errors, the supervisors contain error handling routines. The supervisors run in tasks 9-15, and execute privileged instructions to carry out user requests. When a user request requires I/O with the host computer, the Supervisors communicate with the Kernel (via SVC) and provide the Kernel with I/O messages for transmission to the file system or an attached host computer. The Kernel, running in task 8, controls the communications path and routes messages to the correct supervisors. The Kernel also handles error conditions arising in the supervisor code, and handles the majority of operator interface functions. In addition, since the hardware traps all create fault conditions to task 8, the Kernel handles these also. Since the task using the last process (and getting create fault) may not be the one using too many, the Kernel must find the offender with software and take appropriate action. This is the reason the create faults come to the Kernel rather than the supervisors. Supervisors have control ONLY over their associated user.

In order to support high speed MIMD computation, Denelcor is developing a high speed file storage system using a combination of I/O cache memory and commercially available disk drives to provide file I/O at sustained rates approximating one million bytes/second. The total functional capability of this file system is still under definition at this time.

III. APPLICATION OF MIMD COMPUTERS TO CIG

In order to determine the optimum application of MIMD computers to CIG, several areas should be investigated. These include the following:

a) File System Performance. Using a realistic demonstration scenario, it should be verified that the CIG data base can be accessed sufficiently rapidly to meet CIG requirements. If this is a function of data base complexity, the complexity/speed/cost tradeoffs should be identified.

b) Parallelizing of Scene Element Transformation. Techniques for parallelizing the transformation of scene elements from the data base into the two-dimensional CIG scene should be evaluated. For scenes of considerable complexity, transforming each scene object with a single process may yield sufficient parallelism, whereas for scenes with only a few relatively complex elements, parallelism may be required within element processing.

c) Evaluation of Hardware Facilities. The present Denelcor MIMD computer performs high speed high precision (64 bit) arithmetic. CIG applications do not require such high precision, and often require other operations (e.g. vector dot product) which are not in the instruction set of the present machine. An analysis should be performed to determine hardware operations which if implemented as instructions on an MIMD machine would significantly improve cost and performance.

d) Software Facilities. In order to realize the benefits of the MIMD machine as a general-purpose solution to the CIG problem, extensions to a general purpose language such as FORTRAN should be investigated. These would allow convenient use of hardware operators such as dot product from high level languages.

e) Demonstration. In order to gain confidence in the performance of MIMD on the CIG problem, a demonstration system should be built using an MIMD computer. The demonstration system, when coupled with the analysis suggested earlier, would verify that a production CIG system could be built using MIMD technology on a cost-effective basis and at low risk.

IV. SUMMARY

When applying conventional computers to CIG, limitations of computing speed and I/O rates force the use of distributed computation and special purpose hardware to meet performance requirements. The use of MIMD computers offers a significant reduction in the amount of special purpose and interconnection hardware required. In order to better quantify these benefits, several areas related to the use of MIMD in CIG should be studied. These areas include software organization and extensions to hardware capabilities to improve performance. The resulting analysis should be validated by a demonstration.