ED 210 037 IR 009 856

AUTHOR	Lord, Robert E.; And Others
TITLE	Multiple-Instruction, Multiple-Data Path Computers: Parallel Processing Impact on Flight Simulation Software. Final Report.
INSTITUTION	Denelcor, Inc., Denver, Colo.; Washington State Univ., Pullman.
SPONS AGENCY	Air Force Human Resources Lab., Brooks AFB, Texas.
REPORT NO	AFHRL-TR-80-64
PUB DATE	Aug 81
CONTRACT	F33615-79-C-0009
NOTE	104p.
EDRS PRICE	MF01/PC05 Plus Postage.
DESCRIPTORS	*Computer Programs; *Computers; *Flight Training; Mathematical Formulas; Mathematical Models; *Programing; *Simulation
IDENTIFIERS	*Computer Architecture

ABSTRACT

The purpose of this study was to evaluate the parallel processing impact of multiple-instruction multiple-data path (MIMD) computers on flight simulation software. Basic mathematical functions and arithmetic expressions from typical flight simulation software were selected and run on an MIMD computer to evaluate the improvement in execution time that results from the parallel architecture of this type of computer. Recommendations as to the types of tasks which are optimally suitable for this computer architecture are made, together with the improvement in execution speed to be expected. Twenty-six references are listed. (Author/LLS)

AFHRL-TR-80-64

ED210037

AIR FORCE

HUMAN RESOURCES

MULTIPLE-INSTRUCTION, MULTIPLE-DATA PATH COMPUTERS: PARALLEL PROCESSING IMPACT ON FLIGHT SIMULATION SOFTWARE

By

Robert E. Lord
Swarn Kumar
Washington State University
Pullman, Washington 99164

Rodney A. Schmidt
Denelcor, Inc.
3115 East 40th Avenue
Denver, Colorado 80205

OPERATIONS TRAINING DIVISION
Williams Air Force Base, Arizona 85224

August 1981

Final Report

Approved for public release; distribution unlimited

LABORATORY

AIR FORCE SYSTEMS COMMAND
BROOKS AIR FORCE BASE, TEXAS 78235

IR009856

NOTICE

This report has been reviewed and is approved for publication.

TERRANCE K. TEMPLETON
Contract Monitor

MILTON E. WOOD, Technical Director
Operations Training Division

RONALD W. TERRY, Colonel, USAF
Commander

PREFACE

The purpose of this study was to develop a technique to optimize software for execution on a Multiple-Instruction Multiple-Data Path (MIMD) computer and test its efficiency on existing flight simulator programs. This effort was performed in support of the Air Force Human Resources Laboratory's work on Advanced Simulator Concepts, which is, in turn, part of a larger effort (or thrust) entitled "Engagement Simulation Technology."

The work was accomplished by Denelcor, Inc., Denver, Colorado, and Washington State University (WSU) under Project 6114 sponsored by the Air Force Human Resources Laboratory, Operations Training Division, Williams Air Force Base, under contract F33615-79-C-0009.

The principal investigators and authors are Dr. Robert E. Lord of WSU, Ms. Swarn Kumar of WSU, and Dr. Rodney A. Schmidt of Denelcor, Inc. Patrick E. Price was the Air Force project engineer throughout most of this project; however, during the final stages, he was succeeded by Terrance K. Templeton.

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Public Affairs Office has reviewed this report, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This report has been reviewed and is approved for publication.

TERRANCE K. TEMPLETON
Contract Monitor

MILTON E. WOOD, Technical Director
Operations Training Division

RONALD W. TERRY, Colonel, USAF
Commander

TABLE OF CONTENTS

SECTION 1: INTRODUCTION

Real-time flight training simulators generally use several single-instruction single-data path (SISD) computers to attain the required processing capability. This is similar to the capability offered on a smaller scale by a multiple-instruction multiple-data path (MIMD) computer. Until recently, however, a practical functioning MIMD computer had not been implemented -- all predictions of increased speed and fidelity with MIMD architecture were purely theoretical. Even though an operating MIMD computer now exists, there are still problems obtaining the maximum efficiency from the software. Because the trend is toward more parallel computer processing and parallel processing configurations, the Air Force sponsored this study to develop the technology needed to take advantage of the benefits offered by MIMD architecture. The purposes of this study were to determine which software techniques are most practical to implement, and to determine the implications of using an MIMD computer in real-time simulation.

Computer Architecture

The machine used in the study was Denelcor, Inc.'s Heterogeneous Element Processor (HEP). HEP 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 may be viewed as an MIMD machine implemented by the time-multiplexing of ten process states to one functional unit.

[1]M. J. Flynn. "Very High Speed Computing Systems". Proceedings IEEE, 54 (1966), pp. 1901-1909.

In a HEP processor, two queues time-multiplex the process states. One queue provides input to a pipeline that fetches a three-address instruction, decodes it, obtains the two operands, and sends the information to one of several pipelined function units that complete the operation. If the operation is a data memory access, the process state enters a second queue. This queue provides input to a pipelined switch that interconnects several data memory modules with several processors. When the memory access is complete, the process state is returned to the first queue. Figure 1 shows the processor organization, and Figure 2 shows the system layout.

Each HEP processor supports up to 128 processes, and nominally begins executing a new instruction (on behalf of some process) every 100 nanoseconds (ns). The time required to complete an instruction is 800 ns. Thus if at least eight independent processes (processes that do not share data) are executing in one processor, the instruction execution rate is 10^7 instructions per second per processor.

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 function units, except the divider, execute one instruction every 100 ns. The divider 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 states. The TID specifies a task status word that contains base and limit

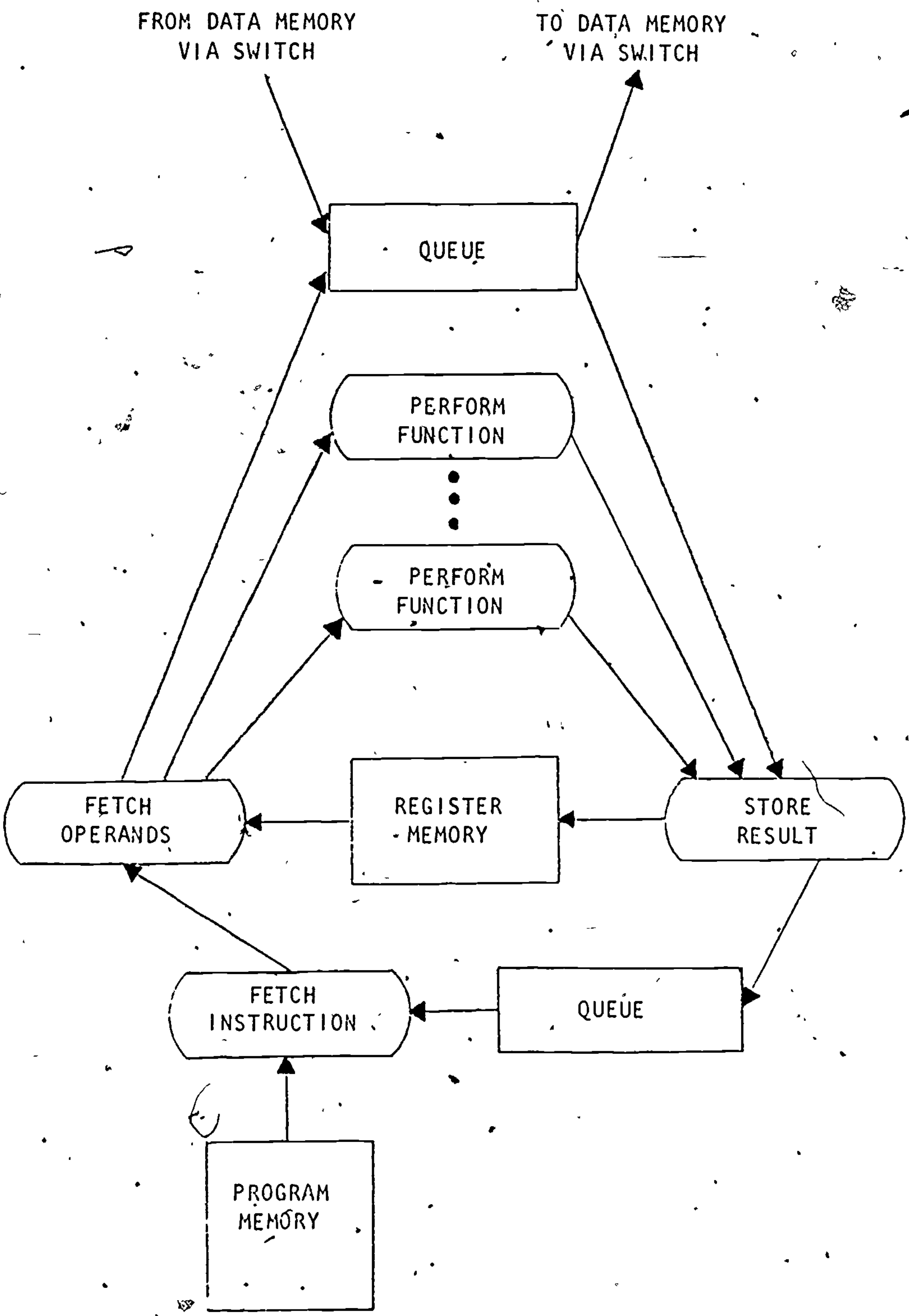


Figure 1 - Processor Organization

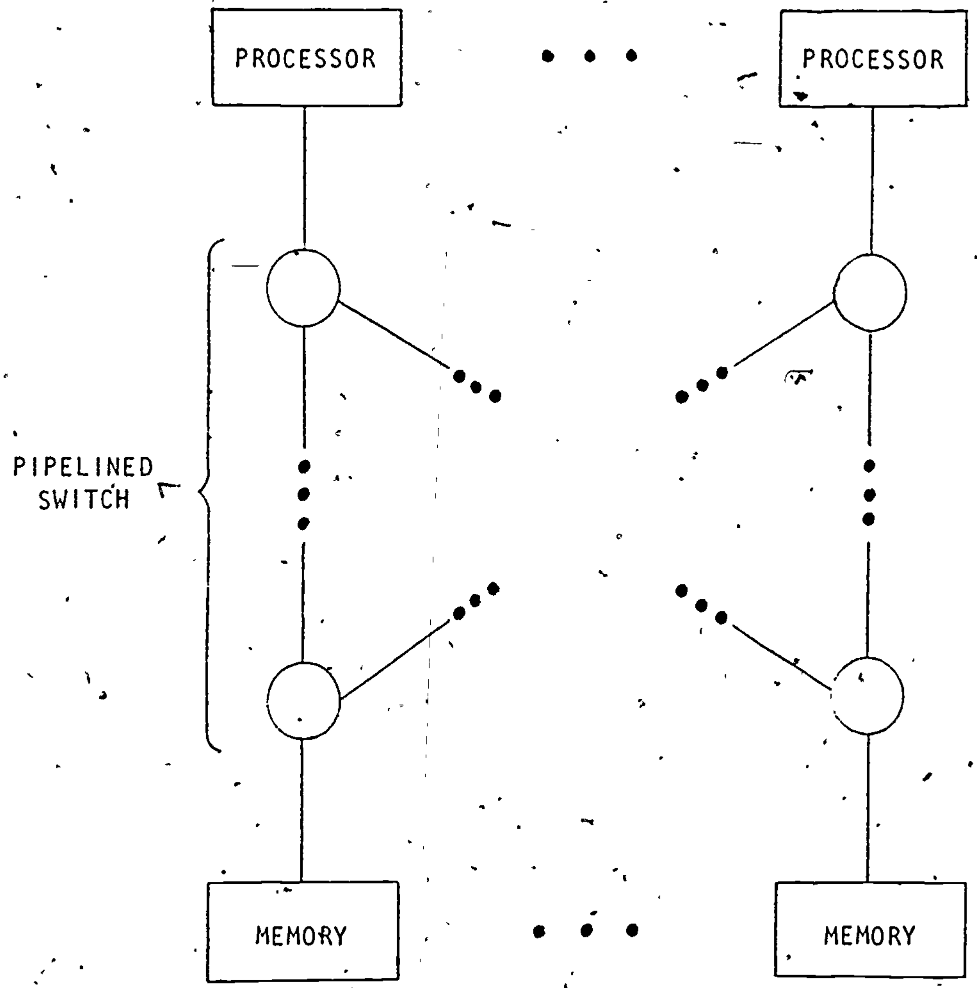


Figure 2 - HEP System Layout

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 overlapping allocations 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 uses must be specified to the system when the task is loaded. The operating system insures that the total allocation of processes to tasks does not exceed the number available. A create fault (too many processes) can occur only 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. Sixteen tasks are available in each processor; eight are user tasks and the other eight are corresponding supervisor tasks. When a process causes a trap, the entire task is made dormant to prevent further execution by any process in it. 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. This allows any create instructions in progress to complete normally, and allows for the creation of the create fault handler process. All other traps in HEP are precise in the sense that they prevent the execution of any subsequent instructions in the offending task.

Synchronization

Any register or data memory location in HEP can be used to synchronize two processes in a producer-consumer fashion. This requires three access states: a reserved state to provide for mutual exclusion, a full state, and an empty state. When an instruction executes, it tests the states of locations and modifies them indivisibly. Typically an instruction tests its sources full and its destination empty. If a test fails, the process reattempts the instruction on its next turn for servicing. If all tests succeed, the process executes the instruction and sets both sources empty and the destination reserved. The operands from the sources are sent to the function unit, and the program oounter in the process state is incremented. When the function unit eventually writes a result in the destination, it sets the destination full.

A destination may be tested full rather than empty, to preserve the state of a source or to override the state of a source or destination. A reserved state, however, may not be overridden except by certain privileged instructions.

Input-output synchronization is handled naturally by mapping I/O device registers into 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. 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 by 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 port every 100 ns. The node attempts to retransmit each message on a port that reduces the distance of that message from its recipient; for this purpose, each node has a table that maps the recipient address into the number of a port that reduces distance. If there is conflict for a port, the node routes one message correctly and the rest incorrectly. To help insure fairness, an incorrectly routed message has its priority incremented as it passes through the node. Preference is given in conflicts to the message with the highest priority.

The success or failure of the operation (based on the access state of the memory location) must be reported back to the processor so it can decide whether to reattempt the operation. Thus, the time required to complete a memory operation via the switch includes two message transmission times, one in each direction.

The propagation delay through a node and its associated wiring is 50 ns. Since a message is distributed among two or three nodes at any instant, the switch is two-colorable to avoid conflicts between the beginning of one message and the middle of another. When the switch fills up due to a high conflict rate, misrouted messages begin to "leak". Every originator is obliged to reinsert a leaking message before inserting a new message. Special measures are taken when the priority reaches its maximum value. This avoids indefinite delays for such messages. A preferable scheme would have been to establish priority by time of message creation, but this would have required too many bits.

FORTRAN Extensions

Two extensions to FORTRAN allow parallelism in source programs. First, subroutines may execute in parallel with their callers, either by being CREATEd instead of CALLed or by executing a RESUME before 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; the subroutine ADD adds the elements. There are a total of 14 processes executing as a result of running the program -- the main program itself, the INPUT and OUTPUT subroutines and 11 copies of ADD.

As a parallel computer, HEP has an advantage over SIMD machines and more loosely coupled MIMD machines in solving large systems of ordinary differential equations that simulate 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. It is quite simple with one process per instruction as in a data flow architecture.

Problem Selection

The contractor principal investigator and the Air Force contract monitor had a large number of programs and program segments to examine and select. These included tens of thousands of source lines provided by the Air Force contract monitor, and several programs provided by the contractor principal investigator. The contract monitor provided the simulation system for the T-38B and A-10 aircrafts. These programs are clearly most representative of current and future simulation programs. A complete program, however, was too large for the scope of this study. Futher, these programs supported a "man in the loop" and had inputs and

```
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

outputs external to the computer. Thus, the T-38B and A-10 simulation programs provided interesting code segments for analysis, but could not be executed on HEP. Four subroutines, however -- constituting the solution of the flight equations for the A-10 aircraft -- were selected for the study. The same subroutines in the T-38B simulation used more than 50% of all of the CPU cycles used by the total simulation. Thus it was felt that the results gained from studying these subroutines could be extrapolated to the entire simulation.

To include a complete program whose serial and parallel versions could be executed on HEP, the contractor furnished a program that simulates the flight characteristics of a ground-launched missile. This program is a sequential FORTRAN program of 442 source lines that solves a set of 10 nonlinear, first-order differential equations.

The code supplied by the Air Force contract monitor included a library of mathematical functions that many of the modules invoke. Thus, one elementary function and two mathematical functions were also included in the study.

SECTION 2: PARALLELISM AT THE MACHINE INSTRUCTION LEVEL

Elementary Functions

A significant computational task in any scientific computing activity is approximating elementary functions (SIN, LOG, SQRT, etc.). The extensive mathematical library in the listings supplied by the Air Force contract monitor indicates that this is the case for flight simulation.

Evaluating Polynomials

Since the very beginnings of electronic digital computing, the preferred method of approximating elementary functions has been polynomials. We have found no evidence that parallel computing alters this choice. Thus we concentrate on parallel methods of evaluating polynomials.

The evaluation of a polynomial of degree n,

$$P_n(x) = a_0 + a_1x + \ldots + a_nx^n$$

requires 2n operations[2]. Thus Horner's rule

$$P_n = a_n$$

$$P_i = (P_{i+1}x) + a_i \qquad i = n-1, n-2, \ldots, 0$$

$$P_n(x) = P_0$$

[2] F. Winograd. "On The Number of Multiplications Required to Compute Certain Functions". Proceedings, National Academy of Science USA, Vol. 58 (1967), pp. 1840-1842.

is optimal for SISD computers because it requires precisely 2n operations and 2n time steps. But we know that, given n processors, the lower bound for polynomial evaluation is $[\log_2 n]+1$ time steps[3]. From this, and future examples, it is clear that Horner's rule is no longer optimal for MIMD computing, where execution time is the criterion.

To describe techniques for evaluating polynomials, we require the following notational conventions:

(n,m) denotes a polynomial of n terms in which the smallest is multiplied by x^m, and

(0,m) denotes the variable x to the mth power.

To analyze the performance of the algorithms, we assumed:

(a) a sufficient number of processors that execute arithmetic (add, multiply) in one time step are available,

(b) results of an operation are available to all processors in the next time step,

(c) processors suspend operations until all operands are available, and

(d) there is no operational overhead in assigning a process or performing an operation.

For HEP, assumptions a, b, and c present no problem so long as "sufficient" does not exceed the number available (for elementary functions, this is the case). In general,

---

[3]Ian Munro. "Optimal Algorithms for Parallel Polynomial Evaluation". Journal of Computer and System Sciences, 1973, pp. 189-198.

assumptions will not hold; as will be seen in the code for elementary function, however, the assumptions do hold by use of certain coding practices.

A straightforward method of evaluating a polynomial $P_n(x)$ is to decompose it into two polynomials of lesser degree. This method computes $P_n(x)$ as

$$P_n(x) = Q_{n/2}(x).x^{n/2+1} + R_{n/2}(x)$$

where

$$Q_{n/2}(x) = a_n x^{n/2-1} + \dots + a_{n/2+1}$$

and

$$R_{n/2} = a_{n/2} x^{n/2} + \dots + a_0$$

and then computes $Q_{n/2}(x)$ and $R_{n/2}(x)$ similarly by binary splitting. Thus it starts by computing in parallel

$$a_1x+a_0 \ , \ a_3x+a_2 \ , \ a_5x+a_4 \ , \ \dots \ , \ a_nx+a_{n-1}$$

and then

$$(A_1x+a_0)x^2 + a_3x+a_2, (a_7x+a_6)x^2 + a_5x+a_4, \dots$$

The time required for this algorithm is approximately $2 \log_2 n$.

This algorithm can be improved by performing the binary splitting in the Fibonacci ratio instead of in halves. Let $F(i)$ denote the ith element of the Fibonacci sequence

$$1,1,2,3,5,8,13,21,\dots$$

and for a polynomial of degree n determine the least i such that $F(i) \geq n+1$. We then split the evaluation of the polynomial by:

$$(n+1,0) = [n+1-F(i-1),0]\ [0,F(i-1)] + [F(i-1),0].$$

The execution time for large n is $\alpha \cdot \log n + O(\log n)$ where

$$\alpha = 1/\log[1/2(\sqrt{5} + 1)] = 1.44.$$

An example of the use of Fibonacci splitting to evaluate a polynomial of degree 20 is shown in Figure 4.

Improvements to the Fibonacci splitting method have been reported (Munro, 1973), but the improvements appear only for very large values of n. For elementary functions where the degree of the polynomials is generally less than 20, a discussion of these improvements does not seem warranted. Table 1 presents the largest degree polynomial that may be evaluated in t steps using Fibonacci splitting versus using the best known algorithms.

t =	2	3	4	5	6	7	8	9	10	11
Fibonacci	1	2	4	7	12	20	33	54	88	143
Best Known	1	2	4	7	12	21	37	63	107	187

Table 1 - Greatest Degree of Polynomial Computable in Time t

In addition to the splitting techniques, a generalization of Horner's rule to make it amenable to parallel computing has been reported by Dorn[4]. If the execution time of an addition and a multiplication are the same, however,

[4] W. S. Dorn. "Generalization of Horner's Rule for Polynomial Evaluation". IBM Journal, April 1962, pp. 239-245.

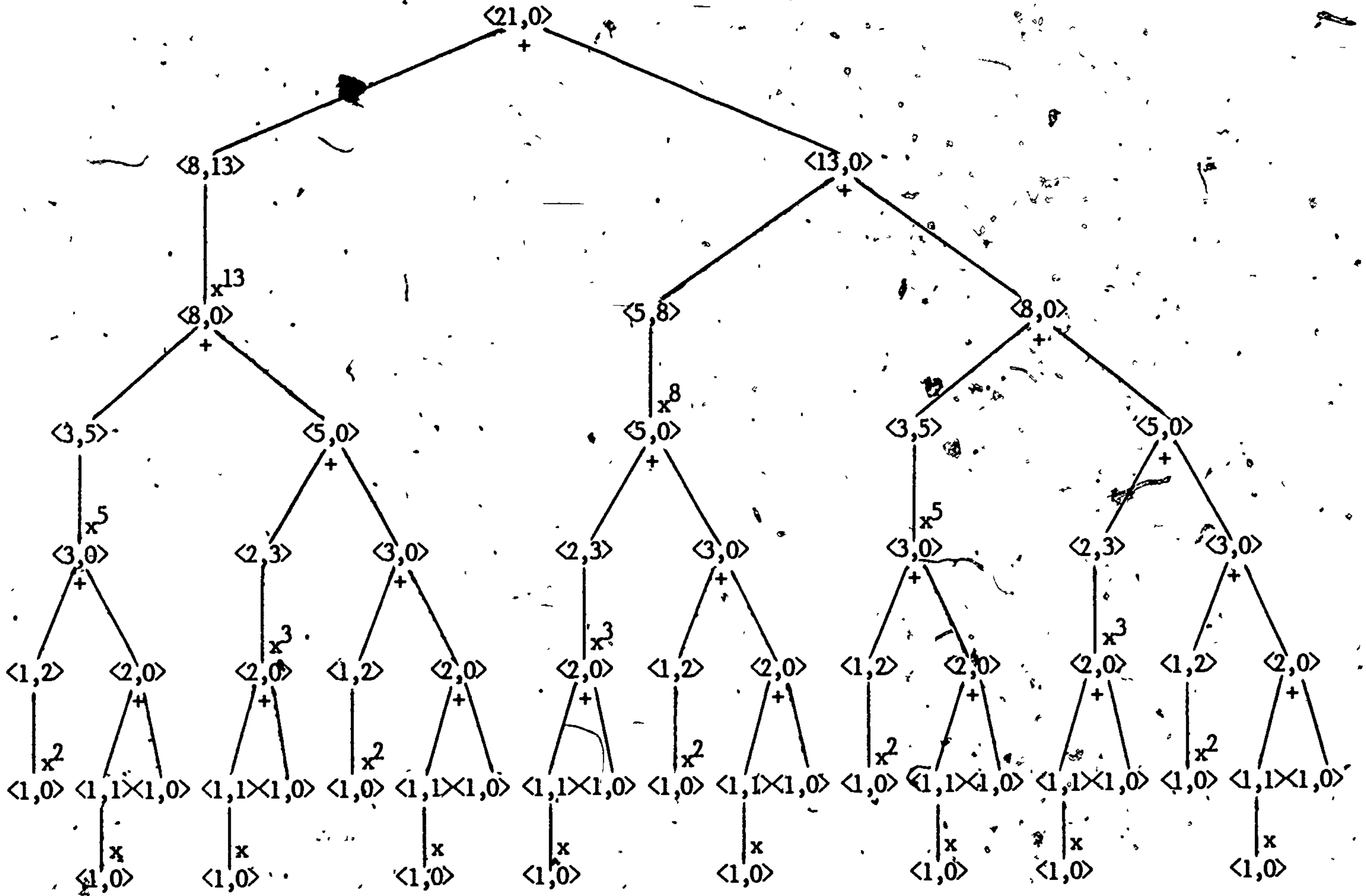


Figure 4 - Fibonacci Splitting of $P_{20}(x)$

this method requires 2 log n steps for the evaluation of an nth order polynomial.

Specific Examples

The approximation of elementary functions by an MIMD computer requires not only techniques for parallel evaluation of polynomials but also techniques for generating coefficients of "best" approximations. The latter subject has received extensive attention in the literature and is not addressed here[5].

The specific elementary functions chosen to be included in this study were the approximation of cosine and logarithm ($base_e$). The algorithms use a 64-bit floating point word with an 8-bit exponent (Radix 16) and a 56-bit normalized fraction.

Cosine

The cosine function accepts an argument (A) in the range $-16^{11} \leq A \leq 16^{11}$ and produces a result in the range $-1 \leq \cos(A) \leq 1$. The method used converts the argument into the range $0 \leq x \leq 2\pi$ by the relationships

$$\cos(x) = \cos(-x) = \cos(x+2K\pi).$$

Next, the argument is reduced to the range 0 to $\pi/2$ by the relationships pictured in Figure 5.

---

[5]The interested reader is referred to Computer Approximations by J. F. Hart, et. al. (New York: Wiley & Sons, Inc., 1968).

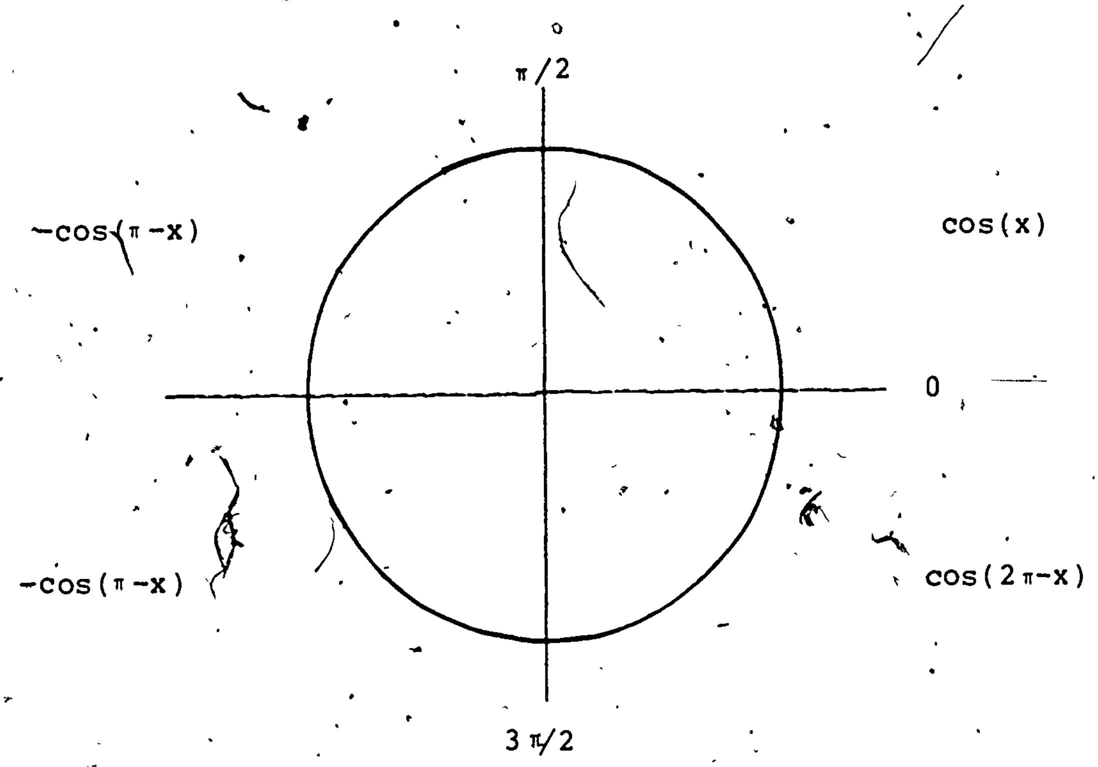


Figure 5 — Cosine Function Relationships

Finally, the function is approximated by a 9th degree polynomial in the converted argument x^2. That is:

$$\cos(x) = P_9(x^2).$$

More concisely:

$$\cos(a) = \cos(\delta b) \quad \delta \in (-1,1) \quad b \geq 0$$

$$\cos(b) = \cos(c + 2k\pi) \quad k \in (0,1,2,\ldots,10^{12}) \quad 0 \leq c \leq 2.$$

$$\cos(c) = \delta \cos(y) \quad \delta \in (-1,1) \quad 0 \leq y \leq \ /2$$

δ, y defined $\quad 0 \leq c \leq \ /2 \quad y = c \quad \delta = 1$

$\pi/2 \leq c \leq \pi \quad y = \pi - c$
$\delta = -1$

$\pi \leq c < 3\pi/2 \quad y = c - \pi$
$\delta = -1$

$3\pi/2 \leq c < 2\pi \quad y = 2\pi - c$
$\delta = 1$

$\cos(y) = P_9(Y^2)$

with coefficients

P_0 =	+.9999 9999 9999 9999 9999 3632 9000	E+0
P_1 =	-.4999 9999 9999 9999 9948 3628 4300	E+0
P_2 =	+.4166 6666 6666 6665 9756 7005 4000	E-1
P_3 =	-.1388 8888 8888 8853 0208 2298	E-2
P_4 =	+.2480 1587 3014 9274 6422 2970	E-4
P_5 =	-.2755 7319 2096 6674 8555	E-6
P_6 =	+.2087 6755 6674 2345 8605	E-8
P_7 =	-.1147 0670 1991 7777 7011	E-10
P_8 =	+.4776 8729 8095 7170	E-13
P_9 =	-.1511 9893 7468 8700	E-15

This polynomial approximation has an absolute accuracy of 20.19 digits_{10} (16.77 digits_{16}). Scaling the argument into the range $[0, 2\pi]$ causes a loss of $\lceil \log_{16}(A/2\pi) \rceil$ digits_{16}. Therefore, the machine word size of 14 digits_{16} should determine accuracy.

The approximation was programmed, and its accuracy tested, with 50 uniformly distributed argument values in the range 0 to 2. The results were compared with 112 bit routines. Statistically the results were as shown in Table 2.

	magnitude	base 2 log
maximum absolute error	1.01×10^{-15}	-49.8
maximum relative error	1.99×10^{-15}	-48.8
average relative error	4.01×10^{-16}	-51.2
Std. deviation of relative error	4.01×10^{-16}	-----

Table 2 - Accuracy of Cosine Approximation

The algorithm comprises the following tasks:

T_1	- Remove sign from argument
T_2	- Scale magnitude of argument into 0 to 2
T_3	- Select quadrant reduction
T_4	- Perform reduction and save quadrant sign
T_5	- Evaluate approximation
T_6	- Combine approx. value and quadrant sign
T_7	- Empty multiple last uses variables

The tasks have the following precedence graph:

$$\begin{tikzcd}
 & T_2 \arrow[dr] & & T_5 \arrow[r] \arrow[dr] & T_7 \\
T_1 \arrow[ur] \arrow[dr] & & T_4 \arrow[ur] \arrow[rr] & & T_6 \\
 & T_3 \arrow[ur] & & &
\end{tikzcd}$$

All tasks except T_5 have no internal parallelism or are more efficiently processed sequentially. T_5," Evaluate approximation", has the computational tree shown in Figure 6.

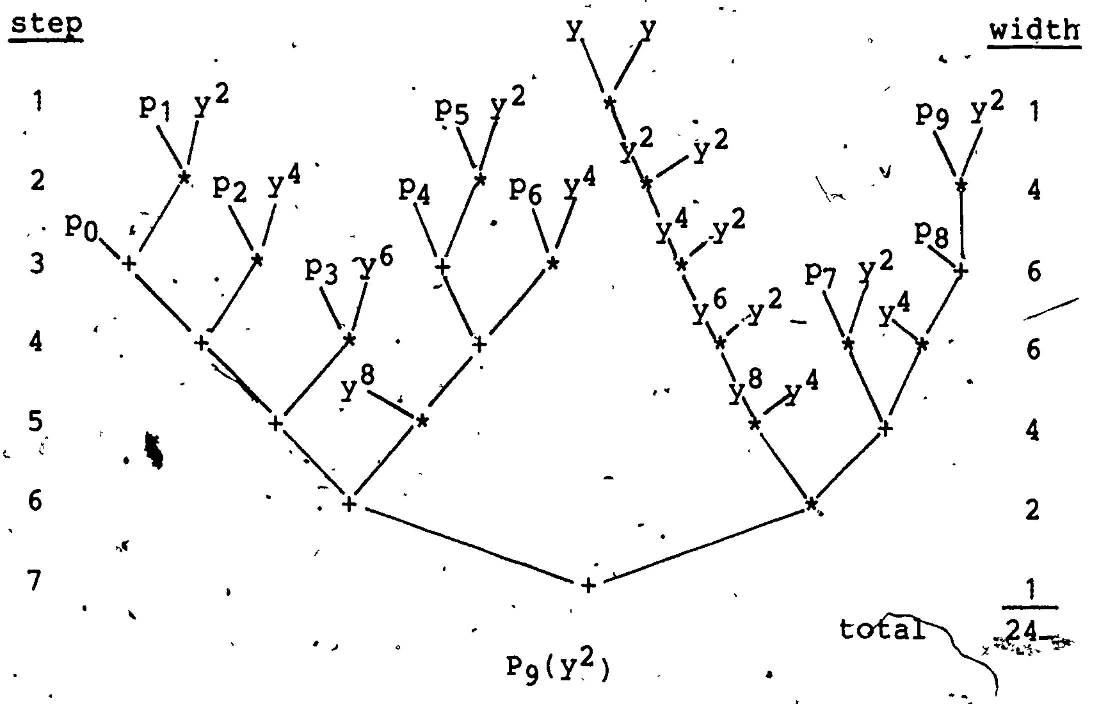


Figure 6: Cosine Task 5 Computational Tree

This routine was programmed for HEP and resulted in the following performance:

Total number of instructions executed	60
Number of instruction cycles used	24
Maximum number of concurrent processes	6
Average number of concurrent processes	2.50
Planned number of waved off instructions	1
Storage:	
Total words of:	
Program Memory	69
Register Memory	25
Constant Memory	29

Thus we have achieved a speed-up of 2.5 in evaluating the cosine function.

Logarithm

The second elementary function examined was the approximation of logarithm base e. The method breaks the input range of the argument into two different ranges. Range I is $(2)^{-1/2} \leq a \leq (2)^{1/2}$, where the function uses a direct rational approximation of the form:

$$\log(a) \approx z[P_3(z^2)/Q_3(z^2)]$$

$$z = \frac{a-1}{a+1}$$

For Range II, $(2)^{-1/2} < a$ or $(2)^{1/2} < a$, we can use the following relationships to convert to base 2 logarithm and extract a bounded value to approximate:

$$\log_e(a) = \log_e(2) \cdot \log_2(a)$$
$$\log_2(a) = \log_2(f \cdot 2^2) = n + \log_2(f), \quad 1/2 \leq f < 1$$

We now approximate $\log_2(f)$, the result of which we combine with n, then multiply by $\log_e(2)$ for the final result.

$$\log_2(f) \approx yR_6(y^2) - 1/2$$

$$y = 1/2(1 - [(2)^{1/2}/(f + (2)^{-1/2})])$$

More concisely:

Range I
for $(2)^{-1/2} \leq a \leq (2)^{1/2}$

$$\log_e(a) \approx z[P_3(z^2)/Q_3(z^2)]$$

$$z = (a-1)/(a+1)$$

Range II
for $a < (2)^{-1/2}$ or $(2)^{1/2} < a$

$$\log_e(a) = \log_e(2) \cdot \log_2(a)$$

$a = 16^n \cdot 2^{-N} \cdot f \quad n \epsilon \{-64,\ldots,63\}$

$N \epsilon \{0,1,2,3\}$

$1/2 \leq f < 1$

$\log_2(a) = (4n - N - 1/2(+ (\log_2(f) + 1/2)$

$(\log_2(f) + 1/2) \quad yR_6(y^2)$

$y = 1/2 - (2)^{1/2} / (2 \cdot f + (2)^{1/2})$

Execution time decreases by

$\log_e(a) = \log_e(2) \cdot (4n - N - 1/2) + yR'_6(y^2)$

where coef of R' = $\log_e(2)$ times, coef of R:

$\log_e(a) \approx z(P_3(z^2)/Q_3(z^2))$

P_0	=	-24.01 3917 9559 2105 10E+0
P_1	=	+30.95 7292 8215 3765 01E+0
P_2	=	-9.637 6909 3368 6865 93E+0
P_3	=	+.4210 8737 1217 9797 15E+0
g_0	=	-12.00 6958 9779 6052 55E+0
g_1	=	+19.48 0966 0700 8897 31E+0
g_2	=	-8.911 1090 2793 7831 23E+0
G_3	=	+1.0 E+0

$\log_2(f) \cdot \log_e(2) \approx yR'_6(y^2)$

r_0	=	+4.000 0000 0000 0000 67E+0
r_1	=	+5.333 3333 3332 4188 96+0
r_2	=	+12.80 0000 0198 2788 68E+0
r_3	=	+36.57 1412 4660 5914 90E+0
r_4	=	+113.7 8399 8715 0066 37E+0
r_5	=	+371.1 3591 8715 6528 26E+0
r_6	=	+1379. 3999 4910 9060 60E+0

These approximations provide 19.38 digits_{10} (16.09 digits_{16}) of absolute accuracy for Range I, and 17.18 digits_{10} (14.27 digits_{16}) of relative accuracy for Range II.

As was the case with the approximation of cosine, this approximation was split into several tasks to facilitate parallelism. The tasks are as follows (tasks within brackets [] apply only to Range II):

T_1	-	Select range
$[T_2]$	-	Extract fraction (1/16 to 1)
$[T_3]$	-	Extract exponent
$[T_4]$	-	Select fraction and exponent adjustment values
$[T_5]$	-	Adjust fraction (1/2 to 1)
T_6	-	Form approx. argument
T_7	-	Evaluate approximation
$[T_8]$	-	Form result exponent
$[T_9]$	-	Combine exponent and approx. value
T_{10}	-	Empty mutliple last use variables

This set of tasks has the following precedence graph:

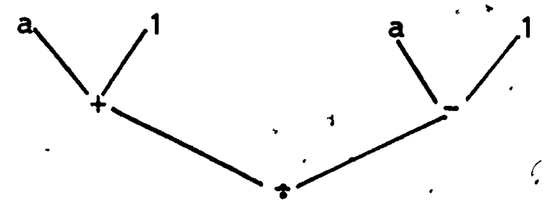
$$\begin{tikzcd}
 & {[T_3]} \arrow[rd] & & & & \\
T_1 \arrow[ru] \arrow[r] \arrow[rdd] & {[T_4]} \arrow[r] \arrow[rd] & {[T_8]} \arrow[rrr] & & & {[T_9]} \\
 & & {[T_5]} \arrow[r] & T_6 \arrow[r] & T_7 \arrow[ru] \arrow[rd] & \\
 & {[T_2]} \arrow[ru] & & & & T_{10}
\end{tikzcd}$$

T_1, "Select range", has the following parallelism:

$$\begin{tikzcd}
a \leq 2^{1/2} \arrow[dr, dash] & & 2^{1/2} \leq a \arrow[dl, dash] \\
 & \& \arrow[dl, dash] \arrow[dr, dash] & \\
T_6 & & T_2, T_3, T_4
\end{tikzcd}$$

(Range I) (Range II)

T_6, "Form approximate argument", has the following parallelism:



T_7, "Evaluate approximation", has the computational trees shown in Figure 7.

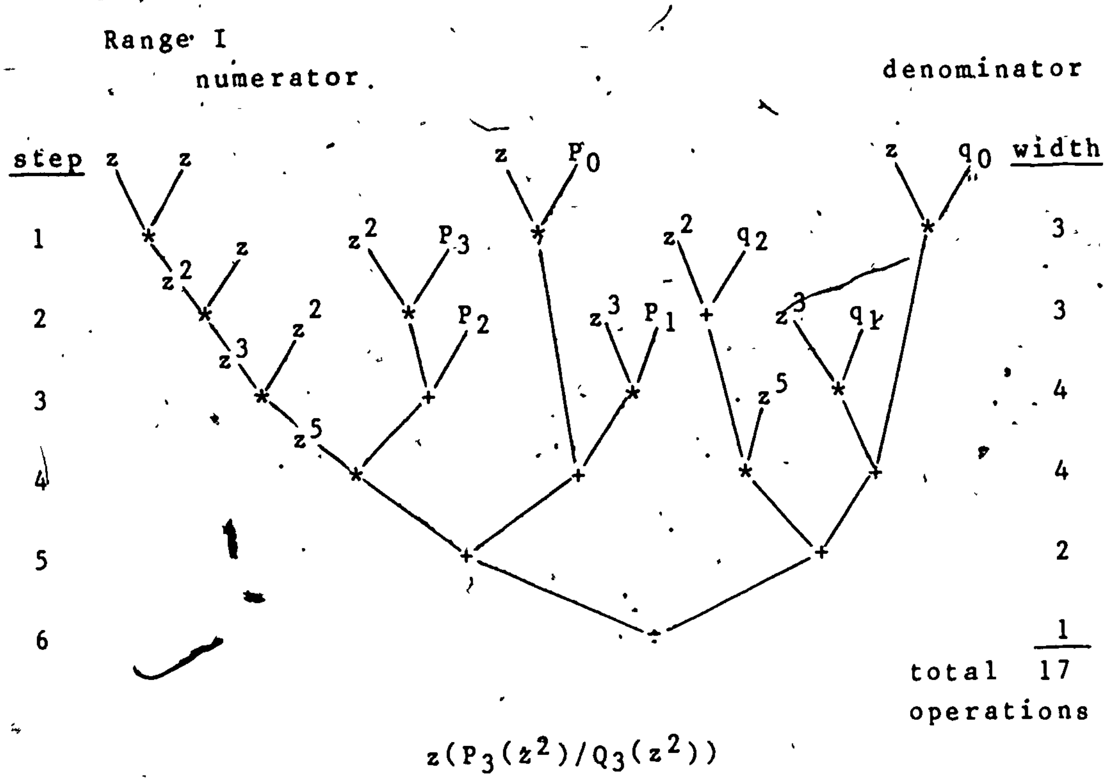
The remaining tasks have no internal parallelism or are more efficiently processed sequentially.

The logarithm approximation was tested using two sample sets of 100 uniformly distributed values in the range 0 to 2 and 0 to 10^6. The results were compared against 112 bit routines; the statistics on the accuracy obtained are given in Table 3.

Range (0,2)	magnitude	base 2 log
maximum absolute error	7.61×10^{-16}	-50.2
maximum relative error	8.35×10^{-16}	-50.1
average relative error	3.20×10^{-16}	-51.5
Std. deviation of relative error	1.84×10^{-16}	-----

Range ($0,10^6$)	magnitude	base 2 log
maximum absolute error	6.55×10^{-16}	-50.4
maximum relative error	4.52×10^{-17}	-54.3
average relative error	3.15×10^{-17}	-54.8
Std. deviation of relative error	7.48×10^{-18}	-----

Table 3 - Accuracy of Logarithm Approximation



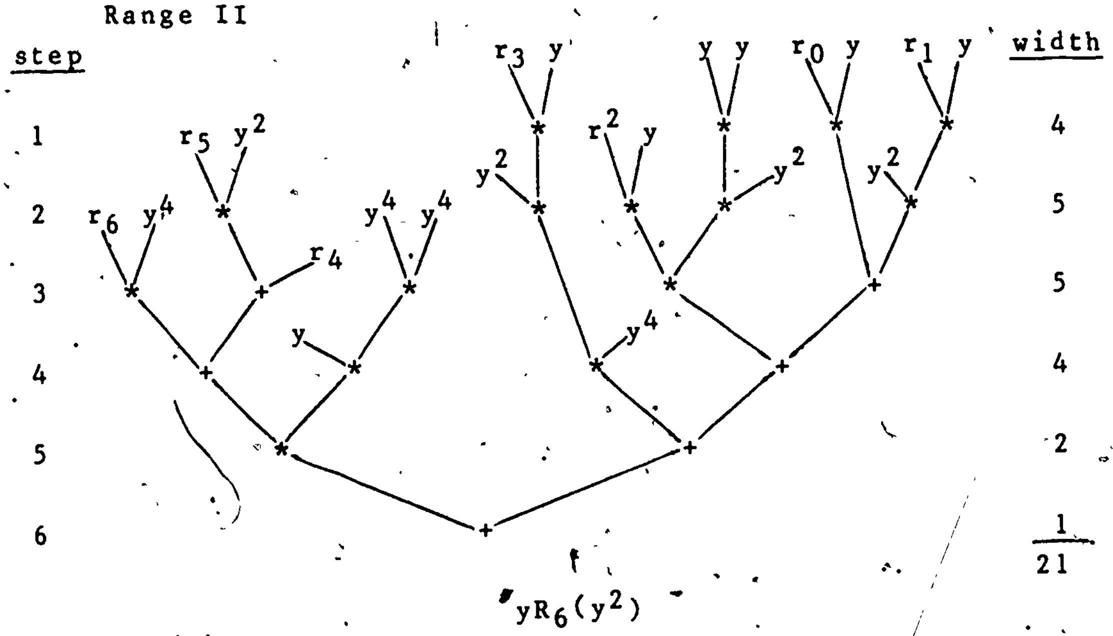


Figure 7 - Logarithm Task 7 Computational Trees

The approximations for the two ranges performed as follows:

Range I:

Total number of instructions executed	47
Number of instruction cycles used	30
Maximum number of concurrent processes	4
Average number of concurrent processes	1.97
Planned number of waved off instructions	6

Range II:

Total number of instructions executed	66
Number of instructions executed	32
Maximum number of concurrent processes	5
Average number of concurrent processes	2.28
Planned number of waved off instructions	4

Storage:

Total words of:

Program Memory	103
Register Memory	26
Constant Memory	69

Assuming that arguments in the two ranges are equally probable, the speed-up of this algorithm is 2.125.

General Code Sequences

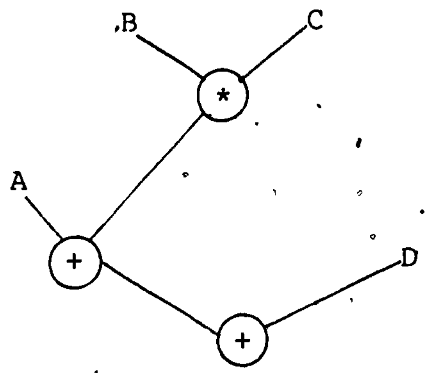
This section examines the problems of generating parallelism at the machine instruction level for general code sequences. The basic techniques consist of tree height reduction methods. In some cases optimal algorithms exist, in others only heuristic methods apply.

Tree Height Reduction Techniques

The basic entity to which tree height reduction applies is expression evaluating. From simple fan-in arguments it is clear that given an expression of n distinct atoms and involving the binary operations of addition, subtraction, multiplication and division, a lower bound on the tree height is $\lceil \log_2 n \rceil$. In many cases, however, the tree produced by an ordinary compiler does not achieve this lower bound. Thus we consider associativity, commutativity, and distributivity to reduce tree height.

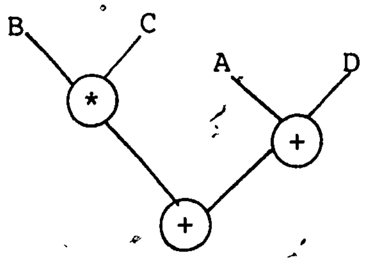
Consider the following expression and its tree representation:

(A + (B * C)) + D



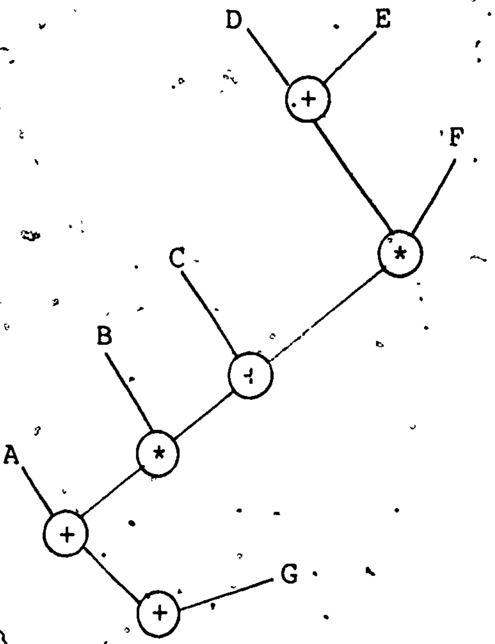
By associativity and commutativity, we can reduce the expression and its tree height to:

(B * C) + (A + D)



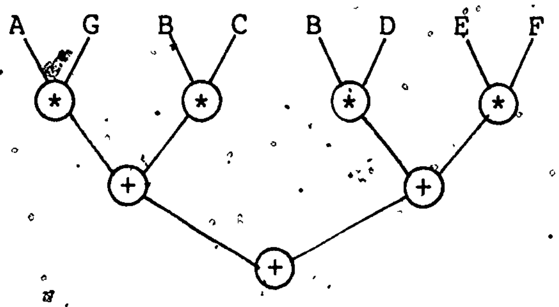
The original expression could use only one processor for its evaluation and required three time steps, whereas the transformed expression can be evaluated by two processors in only two time steps. If we restrict ourselves to associativity and commutativity, algorithms presented by Baer and Bovet[6] have been shown to be optimal. But distributivity can also reduce tree heights. Consider:

A + B (C + D * E * F) + G



This expression has a tree height of 6 and can be reduced by associativity and commutativity to a tree height of 5. By also using the laws of distributivity, however, we produce:

A + G + B * C + B * D * E * F



---

[6]J. L. Baer and D. P. Bovet. "Compilation of Arithmetic Expressions for Parallel Computation". Information Processing '68, North Holland Publishing Company, Amsterdam, 1969.

This has a tree height of 3. Using four processors would result in a speed-up of 2 over the original expression. Unfortunately, we cannot just distribute a multiplication across a parenthesis and reduce tree height. For example,

A * B * (C + D)

has a tree height of 2. Using the distributive law, we get

A * B * C + A * B * D

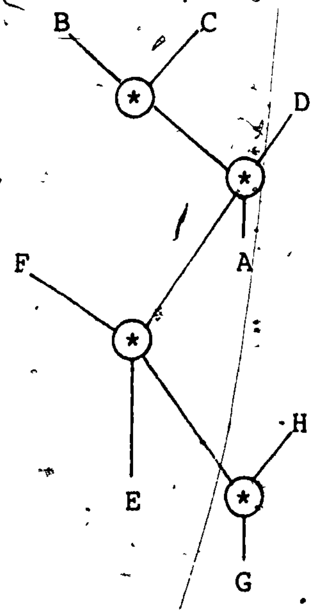
which has a tree height of 3. There are good algorithms that reduce tree height using distributivity, associativity, and commutativity, but they are not necessarily optimal.

We now consider multiple expressions, as would be the case with a set of assignment statements. Consider:

A = B * C * D

E = F * A

G = E + H



This block of assignment statements has a tree height of 4, which may be reduced to 3 by back substitution:

A = B * C * D
E = F * B * C * D
G = F * B * C * C + H

Observe, however, that considerably more operations have been introduced to achieve this reduction.

In addition to arithmetic expression, linear recurrences offer a possibility for significant speed-ups. Consider the linear recurrence represented by the following nested DO loops:

```
      DO 3 I = 1, 10
      DO 3 J = 2, 10
    3 A (I,J) = A (I,J-1) + B(J)
```

The outer loops can be done simultaneously as:

```
      DO 31 J = 2, 10
   31 A(1,J) = A(1,J-1) + B(J)
      DO 32 J = 2, 10
   32 A(2,J) = A92,J-1) + B(J)
              .
              .
              .
```

Further, the interior of each loop is just:

```
      A(I,J) = A(I,1) + B(2) +···· + B(J)
```

This expression can be evaluated in logarithmic speed. Hence the total speed-up could be as high as 25. But this is at a cost of considerably more code. Also, the efficiency in this example is only 50%.

Another area that has been studied is conditional branches represented by IF statements. For an isolated IF statement, all instruction streams must be funneled through the branch, as with a JOIN statement. But a section of code with several IF statements and some assignment statements may be expressed as:

(a) a set of assignment statements all of which may be executed simultaneously,

(b) a set of Boolean functions all of which may be evaluated simultaneously,

(c) a binary decision tree through which one path will be followed for each execution of the program segment, and

(d) a collection of sets of assignment statements with a single variable on the right where each set is associated with each path through the tree.

These techniques have been written as a PL/1 program and applied to a set of 86 FORTRAN programs[7]. Averaged over the 86 programs, these techniques could use 35 processors, resulting in a speed-up of 9.2 and an efficiency of 33%.

Specific Examples

To determine the applicability of these techniques, we examined the code of both the ground-launched missile and the A-10 flight simulation. In neither case could we find significant amounts of code where back substitution could be applied. Further, the code contained no DO loops that constituted linear recurrence equations. Nor were there significant IF blocks that would benefit from reorganization. As would be expected, however, arithmetic expression evaluation provided extensive parallelism. For example, consider the expression for the variable QDOT -- typical of expressions from both programs.

```
QDOT = IYS * ((RHO/2) * (WS + WZ) * US * 4.83912 * CMA
         - US * 8.86989 * RHO/4 * CMQ * QS
         - 21.1 * RS * PS - LC * FTZ)
```

[7]Kuck, David J. "Multioperation Machine Computational Complexity". Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, Ed. (New York: Academic Press, 1973) pp. 17-48.

Figure 8 shows a standard parse tree for this assignment. It contains 18 arithmetic operations and could be evaluated in six time steps using five processors. This would result in an efficiency of 60%. Figure 9 shows the modified parse tree after associativity, commutativity, and distributivity have reduced the tree height. This tree has a height of only 5 and consists of 19 operations. The modified tree could be executed in five time steps using six processors. This also results in an efficiency of 60%.

During this phase of the study, it became apparent that applying these techniques to a significant section of code was beyond the capabilities of manual techniques. The number of operations required makes it immensely time consuming, and the probabilities of error would be so high as to make the results suspect. The only alternative is to construct programs to automatically analyze the code segments and produce parallel instruction sequences. This, too, is a significant task beyond the scope of the work. Section 5 discusses the impact and value of these techniques.

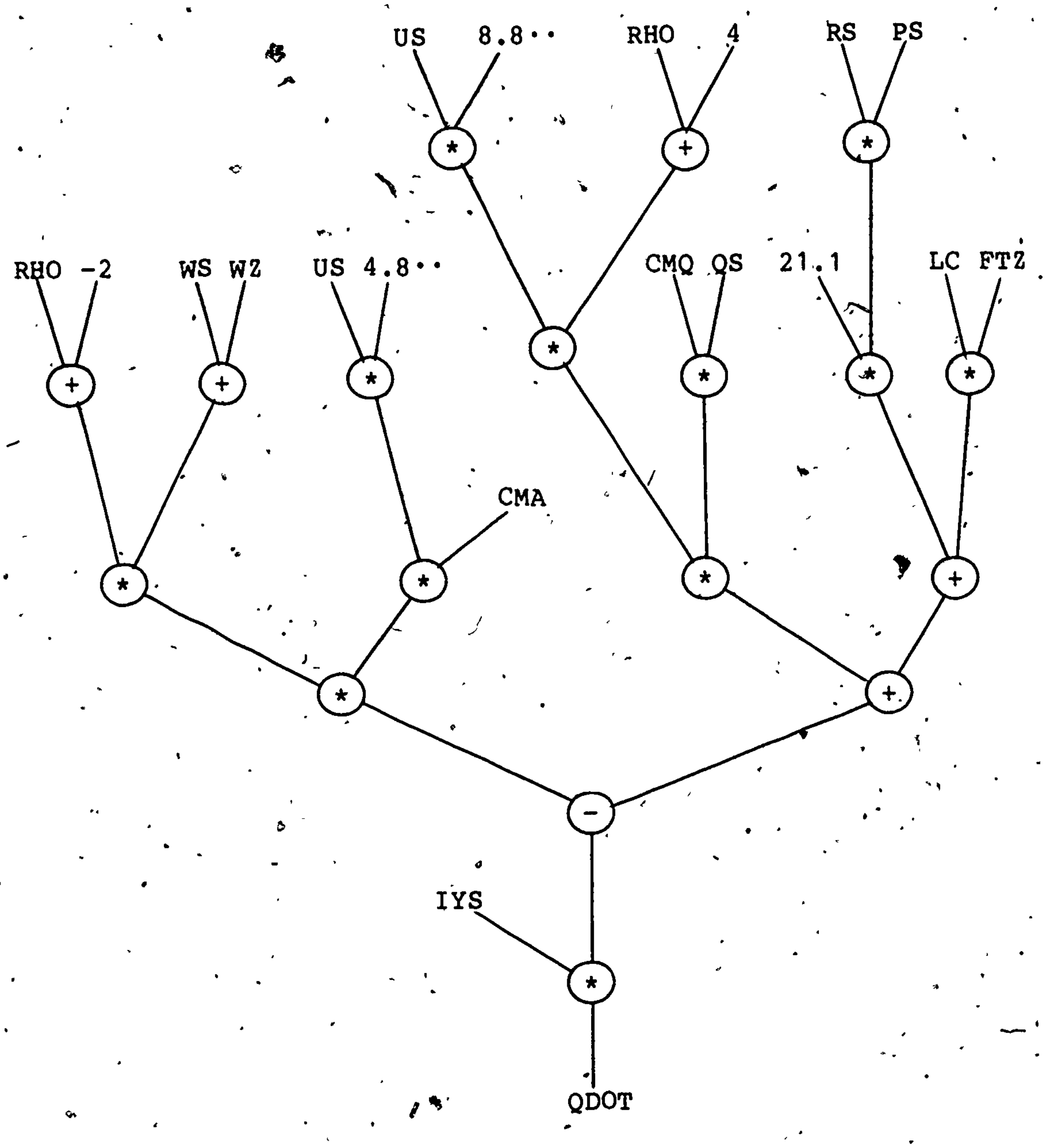


Figure 8. - Parse Tree of Expression for QDOT

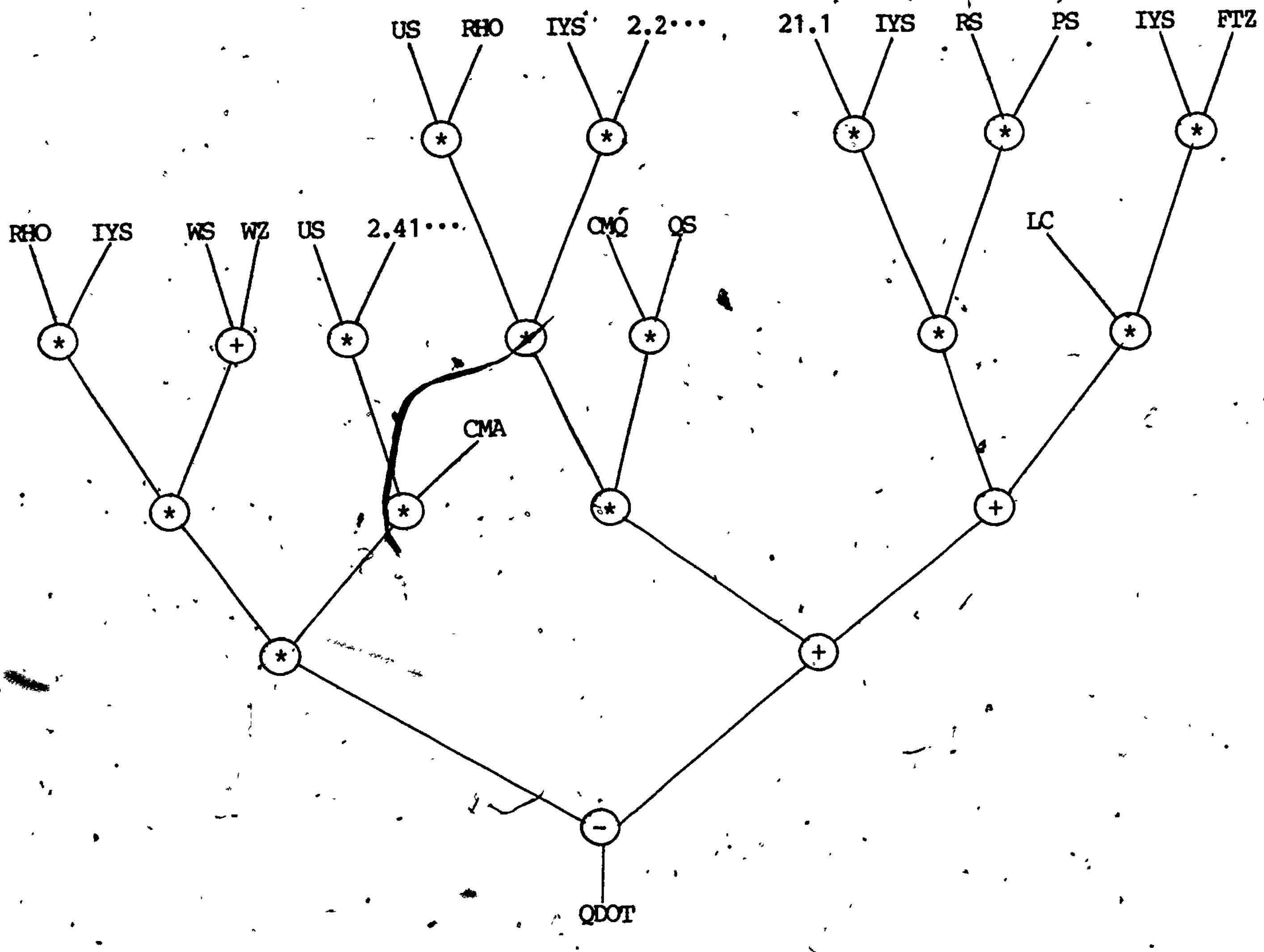


Figure 9 - Modified Parse Tree for QDOT

SECTION 3: PARALLELISM AT THE TASK LEVEL

One of the most common methods of producing a parallel program is to take a sequential program and "parallelize" it. This involves identifying tasks within the sequential program and recognizing that those tasks, together with the implied flow of control, represent a task system. When such a division is possible, standard techniques are available to produce parallel code.

Task Systems

We define a task as a unit of computational activity specified in terms of the input variables it requires, the output variables it generates, and its execution time. The specific transformation that it imposes on its input to produce its output is not part of the specification of a task. Thus the tasks may be considered uninterpreted. Let $J = (T_1, T_2, \ldots T_n)$ be a set of tasks and $<\cdot$ an irreflexive partial order (precedence relation) defined on J. Then $C = (J, <\cdot)$ is called a task system. The precedence relation means that if $T <\cdot T'$ then T must complete execution before T'.

From this definition we introduce a graphical representation, called a precedence graph, for task systems. This consists of a directed graph whose vertices (nodes) are the tasks J and which has an edge from T to T' if $T <\cdot T'$. A T'' such that $T <\cdot T'' <\cdot T'$ does not exist. Thus the set of edges in the precedence graph represents the smallest relation whose transitive closure is $<\cdot$.

Many sequential programs and program segments can be viewed as precedence graphs. Figure 10 shows an example of a program segment and its related precedence graph. Since the relation $<\cdot$ is irreflexive, antisymmetric and transitive, the precedence graph is acyclic -- it represents only

```
C*** TASK 15  COMPUTE THRUST
      NDX=ITHRUST-1
      IF(TIME .LT. THRUSTIME(ITHRUST)) GO TO 151
      NDX=ITHRUST
      IF (ITHRUST .LT. LTHRUST) ITHRUST=ITHRUST+1
 151  THRUST=THRUSTAB(NDX)+THRUSTSL(NDX)
     :       *(TIME-THRUSTIME(NDX))
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
C*** TASK 29  COMPUTE CMQS
      NDX=ICMQS-1
      IF(TIME .LT. CMQSTIME(ICMQS)) GO TO 291
      NDX=ICMQS
      IF(ICMQS .LT. LCMQS) ICMQS=ICMQS+1
 291  CMQS=CMQSTAB(NDX)+(TIME-CMQSTIME(NDX))*CMQSSL(NDX)
C
C*** TASK 11  COMPUTE TIME
      CALL RK11(STEP)
C
C*** TASK 5  COMPUTE QS
      QDOT=$T(70)
      CALL RK5(STEP)
```

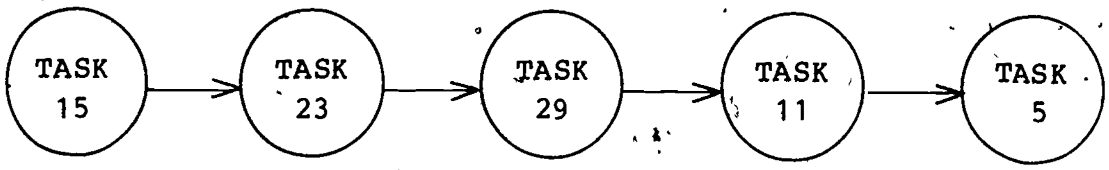


Figure 10 - Program Segment and Related Precedence Graph

straight line code (or code that can be viewed as straight line). We can deal with data-dependent branches that fall entirely within a task, but not conditional branches to other tasks. Further, many loops can be "unrolled" (viewed as straight line code) and handled in an acyclic manner. In one instance, discussed later, we can deal with specific kinds of cyclic graphs.

With each task T we associate two events: initiation and termination. An execution sequence of an n-task system $C = (J, <\cdot)$ is any string $\delta = \alpha_1, \alpha_2, \ldots, \alpha_{2n}$ of task events satisfying the precedence relation and consisting of exactly one initiation and one termination event for each task. A task system that represents a sequential program has only one execution sequence; for other task systems (perhaps equivalent to the sequential task system) there may be several.

To discuss determinant task systems, we must define an ordered set of <u>memory cells</u> $M = (M_1, M_2, \ldots, M_m)$ that represents the physical system on which task systems execute. With each task T in a system C we associate two, possibly overlapping, ordered subsets of M: the <u>domain</u> D_T and the <u>range</u> R_T. When T is initiated it reads the values stored in its domain cells; when it terminates it writes values into its range cells. Given an execution sequence δ for a task system, we can define the value sequence $V(M_i, \delta)$ as the sequence of values written by terminating tasks in δ for which $M_i \in R_T$. Before the first event in any execution sequence, we expect the memory cells to contain values. We refer to that set of values as the <u>initial state</u> P_0.

We can now define more rigorously the intuitive concept of determinant task systems:

A task system C is determinant if for any given initial state P_0, $V(M_i, \delta) = V(M_i, \delta')$, $1 \leq i \leq m$, for all execution sequences δ and δ'.

From this definition, it is clear that a task system that represents a sequential program is determinant since there is only one execution sequence. Given two task systems both consisting of the same tasks, they are said to be equivalent if they are determinant and, for the same initial state, produce the same value sequences.

Our goal now is to define a method by which, given a determinant task system (i.e. one representing a sequential program) we can derive another determinant task system equivalent to the first which has in some sense more parallelism. In fact, our method will derive one with maximum parallelism subject to the constraint that we have no knowledge of the internal transformations performed by the tasks. We begin with the following definition:

> Given a task system C, then tasks T and T' are noninterfering if either
>
> T <· T' or T' <· T O"
>
> -or-
>
> $R_T \cap R_{T'} = R_T \cap D_{T'} = R_{T'} \cap D_T = \emptyset$

We now state, without formal proof[8], a fundamental Theorem regarding noninterfering tasks and determinancy:

> Task systems consisting of mutually noninterfering tasks are determinant.

The final development falls naturally from the Theorem. Given a determinant task system C = (J, <·) we construct

[8]Interested readers may consult Operating Systems Theory by Edward G. Coffman, Jr. and P. J. Denning (Englewood Cliffs, NJ: Prentice Hall, 1973).

another task system $C' = (J, <\cdot')$ that is equivalent to C but whose precedence relation is constructed from $<\cdot$ on the basis that $(T,T') \in <\cdot'$ only if it is necessary to insure that T and T' are noninterfering. The resulting task system is, by the Theorem, determinant. Further, it is maximally parallel in that any further reduction of the precedence relation results in nondeterminancy. Finally, since $<\cdot' \in <\cdot$, every execution sequence of C is an execution sequence of C' and, since C' is determinant, every execution sequence of C' produces the same value sequence. Therefore C' is equivalent to C. This is formally stated in the following Theorem:

> From a given determinant task system $C = (J, <\cdot)$ construct a new system $C' = (J, <\cdot')$ where $<\cdot'$ is the transitive closure of the relation:
>
> $$X = \{(T,T') <\cdot \mid (R_T \cap R_{T'}) \cup (R_T \cap D_{T'}) \cup (R_{T'} \cap D_T) \neq 0\}$$
>
> Then C' is the unique, maximally parallel task system equivalent to C.

Scheduling

Standard Task Systems

Given a determinant task system and the execution time of each task, the problem remains of assigning the tasks to p processors. More formally, we define the scheduling problem to be the following: we are given

(1) a set of tasks $J = \{T_1, T_2, \ldots, T_n\}$
(2) an irreflexive partial order $<\cdot$ on J,
(3) a weighting function W from S to the positive integers, representing the execution time of each of the tasks, and
(4) the number of processors p.

We may be executing as many as p tasks at any point in time. If task T is first executed at time t using processor K, then it is executed only at times t, t+1,..., t+W(T)-1 using processor K each time. It is also required, for any task T' such that T' <· T, that T' complete execution at time t' when t' $\leq$ t. A <u>schedule</u> is an assignment of tasks to processors that satisfies the above conditions and has length tmax, where tmax is the maximum, over all tasks, of the times at which the termination events occur. The <u>scheduling problem</u>, then, is to determine an assignment that minimizes tmax. This problem is NP-complete[9] and can be considered intractable. There are, however, polynomial time bound algorithms that produce good schedules. One such algorithm is <u>critical path list scheduling</u>.

The algorithm is defined as follows:

(1) Given a task system and a list that orders the tasks, we require a scheduling strategy that assigns (to a free processor) the first unassigned task in the list whose precedence constraints have been met. Such a strategy is called <u>demand list scheduling</u>.

(2) The <u>critical time</u> of a task is the execution time of that task plus the maximum critical times of any successor tasks.

(3) If the tasks are ordered on nonincreasing critical time, then the resulting list schedule is called <u>critical path list scheduling</u>.

---

[9] J. D. Ullman. "Polynomial Complete Scheduling Problems". <u>Operating Systems Review</u>, Vol. 7 No. 4 (1973), pp. 96-101.

Kohler[10] reports a preliminary evaluation in which 20 task systems, scheduled using critical path list scheduling, produced 17 optimal schedules. The worst-case schedule was only 3.4% longer than optimal. Using only limited backtracking with a critical path list scheduler, Lord[11] found that in 100 randomly generated cases, 89 were scheduled optimally. He further found that for all cases the schedules had an expected time of only .36% longer than optimal. The worst-case time was 5.6% longer. Thus we conclude that critical path list scheduling is an acceptable technique for practical application.

Cyclic Task Systems

As we have observed before, the standard task system represents an acyclic computational method. This method applies to repetitious calculations such as flight simulation problems by treating the calculation of derivatives and the updating of the state variable as a task system, scheduling those tasks, and then repeatedly executing that schedule. In some cases, however, shorter solution times can result if we represent the problem as a cyclic task system. For example, consider the Van der Pol equation written as two first-order equations:

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

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

10 W. H. Kohler. "Preliminary Evaluation of The Critical Path Method for Scheduling Tasks on a Multiprocessor System". IEEE Transactions on Computing, Vol. C24 No. 12 (December 1975), pp. 1235-1238.

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

By using some suitable integration method (for example, 4th order Runge-Kutta as indicated by the function rk), the main part of a program for solving these equations is as follows:

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

The calculation interior to the "for" loop can be represented by the acyclic precedence graph shown in Figure 11. Assuming that each binary operation can be executed in one time unit and that the function rk can be evaluated in four units, the entire "for" loop can be represented by the cyclic precedence graph shown in Figure 12. T3 calculates $u*(1-x_1*x_1)$, T4 calculates $*x_2-x_1$, and T1 and T2 calculate the function rk.

Given two parallel processors, one way to schedule this solution is to assign the tasks interior to the "for" loop to processors. This should be done 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 the repeated execution of this schedule. Such an assignment is shown by the Gantt chart in Figure 11. Note that this assignment is as good as possible -- the precedence graph has a maximum path length equal to the assignment period.

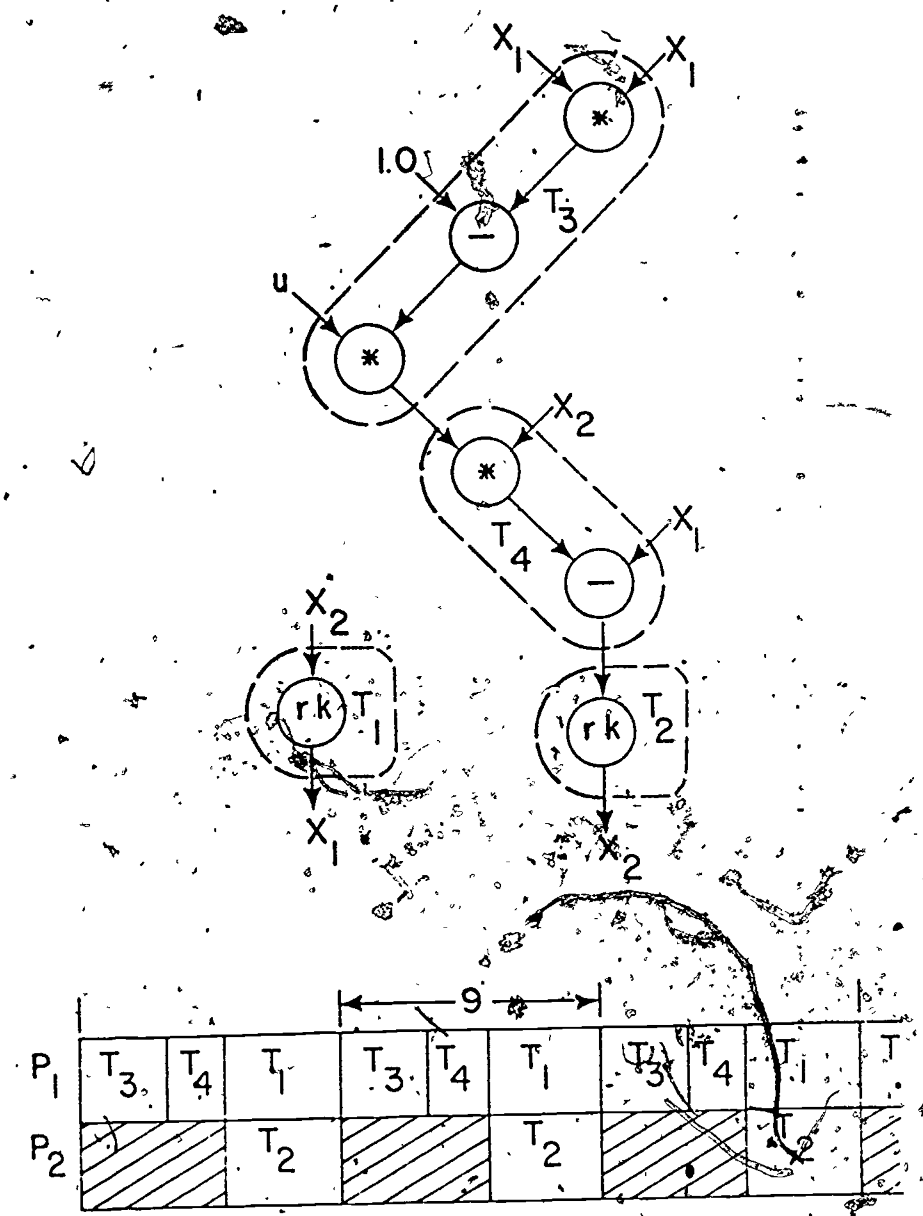
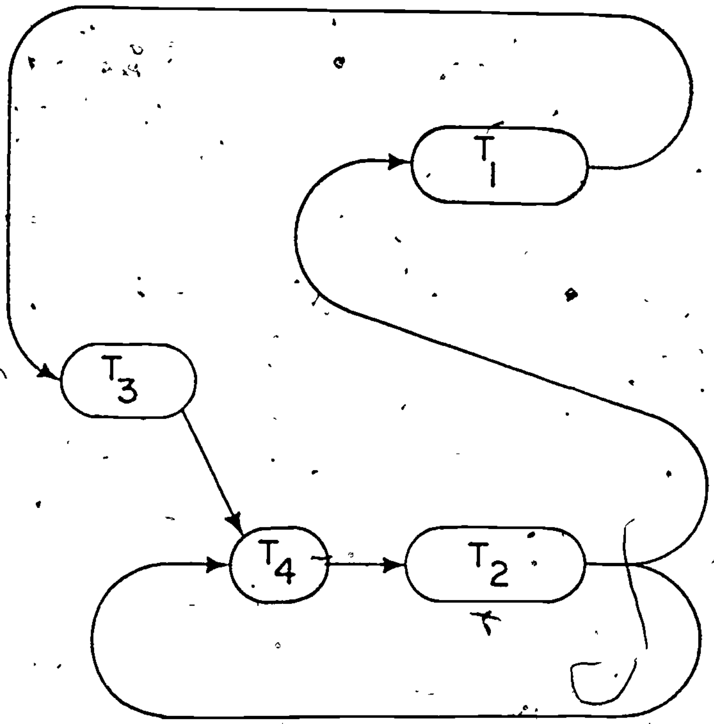


Figure 11 - Acyclic Precedence Graph and Schedule



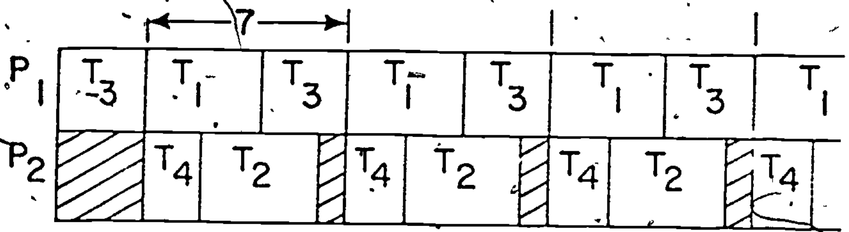


Figure 12 - Cyclic Representation and Schedule

The Gantt chart in Figure 12 shows the assignments made if we assume some initial values for X_1 and X_2 and then assign the tasks from the cyclic precedence graph while maintaining all precedence constraints. This assignment has a repetition period of seven units, as compared with nine units for assigning the acyclic precedence graph. This shorter schedule is the motivation for examining flight simulation equations to determine their minimum solution period and to schedule them in that minimum period with as few processors as possible.

The method used constructs a task system representing the solution to the flight simulation equations, where the tasks that update the state variables are flagged. The precedence graph of the task system is allowed to be cyclic so long as each cycle traverses at least one flagged task. The minimum solution period is then determined by examining all cycles in the graph.

Let the cycles be denoted by $C_1, C_2, \ldots, C_m$. For each cycle let $L(C_i)$ denote its length and $\#(C_i)$ the number of flagged tasks in the cycle. Then the minimum solution period t_{min} is:

$$t_{min} = \text{Max} \{ \lceil L(C_i)/\#(C_i) \rceil \mid 1 \leq i \leq m \}$$

Once the minimum solution period is determined, a critical path list scheduler can, with only slight modifications, produce an efficient schedule whose repeated execution solves the flight simulation problem.

Synchronization

Once a schedule has been determined, there must be some way to insure that the schedule is followed. A general assumption made regarding MIMD computing is that the precise execution rate of individual processors cannot be used to prove the correctness of a program. This assumption applies to HEP; although we know that execution rates of processes are generally the same, detailed knowledge of the progress of each process is beyond the scope of normal analysis of programs. Thus, having determined a schedule for computing the tasks, it now remains to implement it.

Much of the work on scheduling assures, at least implicitly, that some mechanism external to the processors assigns the tasks to the processors. But our execution times are estimates only, so the scheduling mechanism would have to monitor the progress of all processors. Instead we seek a mechanism whereby all the tasks to be executed by a single processor are presented as a sequential program. Synchronization primitives, operating on semaphores, coordinate those tasks.

Dijkstra[12] introduced the primitives P and V, which operate uninterruptably on an event variable termed a semaphore, to control resource allocation among concurrent processes. For our purposes we may define P and V as:

```
P (E): if $E \geq 1$                  V (E):
       then $E \leftarrow E-1$                 $E \leftarrow E+1$
       else wait
```

P is normally used before a process uses a nonsharable resource; V is executed when the use of the resource is completed.

Denning[13] shows that these primitives can synchronize concurrent tasks. As an example, consider the task system

---

[12]E. W. Dijkstra. "Cooperating Sequential Proesses". Programming Languages, F. Genuys, Ed. (New York: Academic Press, 1968) pp. 43-112.

[13]P. J. Denning. "Third Generation Computer Systems". Computing Surveys, Vol. 3 No. 4 (1971) pp. 175-216.

and concurrent program shown in Figure 13. The program uses P and V operating on the suitably initialized semaphore X23. Clearly, the program correctly executes the task system. Since we are using task systems to represent computations, precedence constraints arise because one task computes data elements used by another task. If we were to consider a task system that represents a calculation loop, such as shown in Figure 14, we find that the first program still represents a valid solution to the problem. This is because it is implied that both stream 1 and stream 2 complete execution before beginning the second execution of these streams.

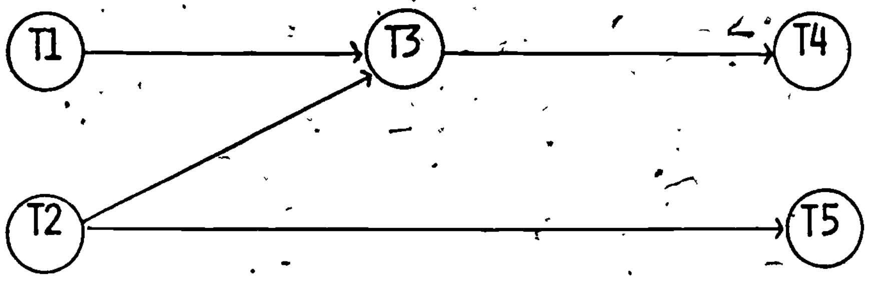
Such methods of computation have been previously proposed for handling looped and conditional execution using constructs named "fork" and "join". But if the alternate program executes the task system, then the P and V operations are no longer valid. This is so because if S2 runs more quickly than S1, at some point T2 completes the calculation of the data element that causes the precedence constraint before T3 has consumed the previous value. Even if we assume a queue for this data element, in any real implementation the queue would be of finite size and hence subject to overflow. To overcome this difficulty, we use two state semaphores associated with each data element or variable, as indicated by:

```
1   VAR
      2   VALUE
      2   SEMAPHORE  ['E','F']
```

where 'E' indicates empty and 'F' indicates full. We now define the P and V operations as:

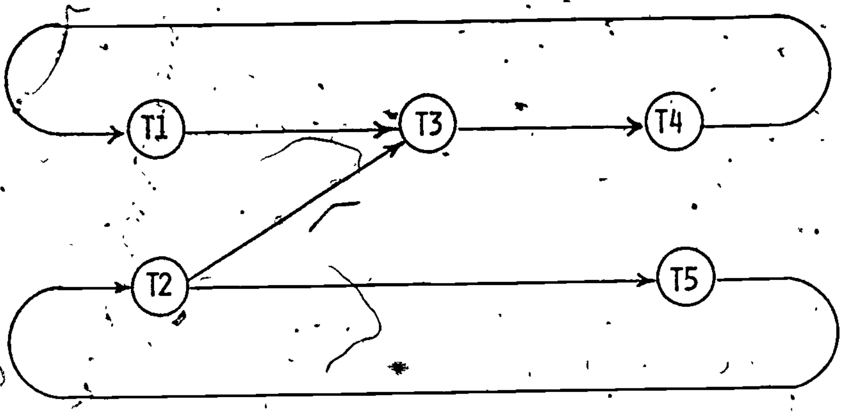
```
P(VAR):
    IF VAR.SEMAPHORE = 'F'
    THEN VAR.SEMAPHORE <- 'E'
    ELSE WAIT
```

$$\begin{tikzcd}
T1 \arrow[r] & T3 \arrow[r] & T4 \\
T2 \arrow[ur] \arrow[rr] & & T5
\end{tikzcd}$$



```
PARBEGIN
    S1: T1; P(X23); T3; T4
    S2: T2; V(X23); T5
PAREND
```

Figure 13 - Task System and Concurrent Program



```
REPEAT N TIMES
    PARBEGIN
        S1: T1; P(X23); T3; T4
        S2: T2; V(X23); T5
    PAREND

ALTERNATE
    PARBEGIN
        S1: REPEAT N TIMES
                T1; P(X23); T3; T4
            END
        S2: REPEAT N TIMES
                T2; V(X23); T5
            END
    PAREND
```

Figure 14 - Task System for Repeated Execution

```
V(VAR):
        IF VAR.SEMAPHORE = 'E'
           THEN VAR.SEMAPHORE <- 'F'
           ELSE WAIT
```

Then if we let X23 represent the variable responsible for the precedence constraints from T2 to T3, the alternate program correctly executes the task system.

To further simplify the programming aspects of such a synchronizing method, we note that, in a language involving assignment statements, context determines whether the operation is P or V. That is, any synchronizing operation on the left of the assignment symbol denotes a V operation. All others denote a P operation. In HEP FORTRAN, $ represents both P and V; context denotes which operation is implied. If some task T1 computes a value used by two other tasks, T2 and T3 (each in separate instruction streams), then the coordination problem between T1 and T2 is separate from the coordination problem between T1 and T3. Hence, two copies of the variable are required so that two separate semaphores are available.

Automated Techniques

During the course of this study, we developed and used programs to automate many of the steps involved in preparing a problem for parallel solution. We believe that sufficient knowledge is available to construct a CSSL-type language translator[14] that would produce efficient parallel code for the types of problems we have studied. But the class of problems so far studied is relatively small; desirable

[14] See "Continuous System Simulation Language" by J. C. Strauss, et. al. Simulation, Vol. 6 No. 12, December, 1967.

extensions to such a language may be poorly understood. Thus we feel that a practical approach for the immediate future is to use a set of utility programs that will significantly aid programmers in constructing parallel programs yet not impede them in the methodologies they use. In the remainder of this section we discuss the various utility programs which we feel would be useful and the source language restrictions they would impose on the programmer.

We assume that the definition of a flight simulation problem will be extended to a sequential FORTRAN program that defines the derivatives in terms of the state variables and updates the state variables by whatever technique is desired. In general, we assume that updating each state variable is a separate program segment, so they can be designated individual tasks where desirable. Since each program segment must be simple enough to be represented by a cyclic task system, restrictions on conditional branches will be required. Selecting code segments for tasks is not unique -- we can give only guidelines as to what constitutes a good selection. Thus we require the programmer to specify which pieces of code constitute tasks. No branches can occur from one task to another. In practice we have found that this restriction is not at all severe.

There are a variety of methods the programmer could use to indicate what constitutes a task. We have used a comment card of the form

```
C***TASK n [,SV]
```

to indicate that the following statements constitute task n; the option SV indicates that the task updates a state or recurrence variable. This directive is terminated by another task comment card or by an END card. Since the final maximally parallel task system equivalent to this sequential program is derived from the ranges and domains of the tasks, and since range and domain determination is not always possible in FORTRAN, further extensions of the comment cards are required. Specifically, if the statement

```
          CALL SUBR (A,B,C,)
```

is within a task, there is no way to determine if A, B, and C are in the range of this task, the domain of the task, or both. Also, it may not be worth the effort to analyze all of the equivalence statements. Thus we use comment cards of the form

```
          C *** RANGE (list)
     and
          C *** DOMAIN (list)
```

to indicate that all variables within the list are in the range or domain of the current task.

Another piece of information required by automated analysis is an estimate of the execution time of each task. We have chosen the units of this measure to be the number of instructions executed within the task. If the code constituting the task is straight line code, the number of instructions is known at compilation time. But if the task contains conditional branches or invokes external subprograms, the execution time of the task is not usually determinable; the programmer must supply an estimate. To this end we use a comment card of the form

```
          C *** TIME n
```

to specify the execution time as n machine instructions. Note that specifications of range, domain and time are required only if the form of the code precludes the utility from determining the values.

A final comment card reduces analysis time by listing local variables that are to be excluded from range-domain analysis. This card has the form

```
          C *** LOCAL list
```

and indicates that, for this task, variables in the list are to be excluded from both the range and domain of the task.

This is used mostly for variables that are first in the range and then in the domain, as would be the case for DO loop control variables.

A PASCAL program was constructed for this study to determine range and domain. It soon became evident, however, that this program must perform lexical and syntactic analysis of FORTRAN source code just as the FORTRAN compiler must do. Therefore, compiler output was used to determine the task system. This requires the following:

(1) A source code image. This allows the extent of the tasks to be determined, and comment cards pertaining to range, domain and time to be examined.

(2) An image of the generated machine code. This allows execution time to be estimated for those tasks that consist of only straight line code.

(3) A cross reference listing. This allows ranges and domains of the tasks to be determined.

If suitable compile options are invoked, all this information is in the compiler output listing. The output of this program is the cyclic task system required for a scheduler, and the names of variables involved in intertask communication. An additional output is a file of the source program which is used to construct the parallel program.

The second utility program is the scheduler. The inputs are a cyclic task system, an estimate of the execution time, and a specification of the number of processes. The output is a schedule that is not necessarily optimal but has good efficiency. In test runs, the schedules produced for 100 randomly generated cyclic task systems resulted in 93 schedules that were optimal. The expected schedule length was no more than .158% longer than optimal.

The third utility program uses the output of the previous two programs to determine the synchronization required to insure that the schedule is not violated. The basic algorithm examines, for each pair of tasks to be executed in different processes, which variables are in the range of one and the domain of the other. For those variables, asynchronous (semaphored) variables must be used.

For each variable V_i it must be the case that $V_i \in R_T$ for some task T. The possibilities for each variable are:

(a) For all tasks T' such that $V_i \in D_{T'}$, T' has been assigned to the same processes as T, in which case no synchronization is required.

(b) There is only one task T' with $V_i \in D_{T'}$ and it has been assigned to another processor. Further, if the variable name has only one instance in both T and T', then the variable name is prefixed with a $ in both T and T' and the asynchronous variable is placed in COMMON.

(c) There are multiple tasks T' such that $V_i \in D_{T'}$ and some of these tasks have been assigned to different processors than T. In this case, for each T' which has been assigned to a different processor we associate a new asynchronous variable $W. This variable is placed in COMMON. The assignment statement $W = V_i is placed at the end of the code for T, and the assignment statement V_i = $W is placed at the beginning of the code for T'.

Another utility program that would be useful is one that actually constructs the code sequences for the various processes based upon the preceding analysis. This would be particularly useful in a system with a flexible text editor. This would allow the programmer to add output statements or exception-handling code. This utility has no firm technical requirements; it is more a convenience feature for the programmer.

Results

The methods of the previous section were applied to the flight simultion of a ground-launched missile. The methodology employed in programming the flight simulation equations for an MIMD computer can be divided into several categories. These include equation segmentation, scheduling and synchronization.

Equation segementation takes a representation of the problem, in our case a sequential FORTRAN Program, and identifies the tasks. These tasks are considered to be individual computational activities, and could range from individual machine instructions to groups of FORTRAN statements. We chose individual statements or small groups of statements, where any branching took place entirely within the group of statements identified as a task. An example of this task selection is shown in Figure 15, which shows a portion of the sequential code and indications of some specific tasks. In this case a total of 40 tasks were identified. Ten of them update the state variables by the chosen integration method, and one updates the independent variable time. The remaining 29 tasks are associated with evaluating the derivatives.

The next step was to estimate the execution time of each task. Since the HEP computer executes all instructions in the same time, this involved compiling the program and counting the number of machine instructions generated by each task. The number of instructions per task ranged from 2 to 88, with an average of 34.6. We next determined a maximally parallel task system equivalent to the set of tasks selected, and the sequential program for those tasks.

```
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 15 - Task Selection

Figure 16 shows the task system for the 40 tasks comprising the problem solution. The task number and execution time (in machine instructions) are within the nodes. All arcs go left to right. Observe that the three tasks highlighted in Figure 15 (tasks 18, 19 and 20) can all be executed in parallel.

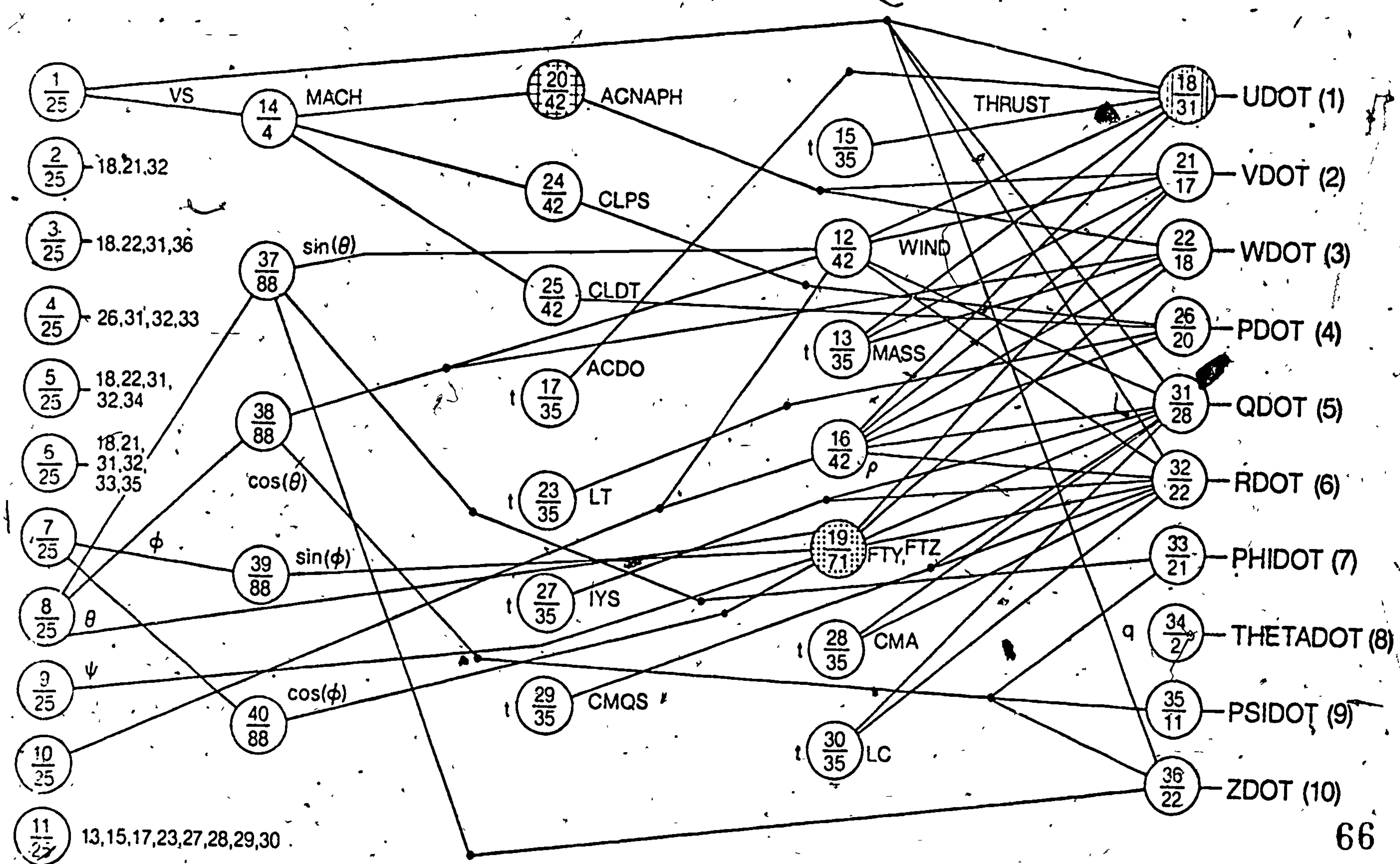
Before scheduling, we make a transformation on the parallel task system to shorten the solution time. In Figure 16 we see that the longest path traverses nodes 7, 39, 19 and 31, and has a length of 212 units. This path does not determine minimum execution time, however, because there is no path from node 31 to node 7. The minimum time is instead determined by the cycle traversing 7, 39, 19, 32, 6 and 3, which has a length of 252. This yields a minimum execution time (for n iterations) of n * 126 + constant.

The next steps in our methodology were scheduling the transformed task system for execution on p processors and synchronizing. Figure 17 shows the resulting schedule.

The schedules for the flight simulation problem were programmed for HEP FORTRAN and were executed on the HEP. Equation segmentation, in conjunction with the fourth-order Runge-Kutta formula given by (RK4), was used for the eight-processor schedule shown in Figure 17. The computations of the integration formula were also done as parallel tasks. This scheme was also programmed using six processors; the speed-up was 3.98. The speed-up and efficiency of the eight processor program, along with the computational results, are shown in Table 4.

PROGRAM	P	T_1	T_p	S_p	E_p
RK	8	28.18	4.87	5.78	72.3%

Table 4. Speed-up & Efficiency, Eight Processors



65

66

Figure 16 - Maximally Parallel Task System

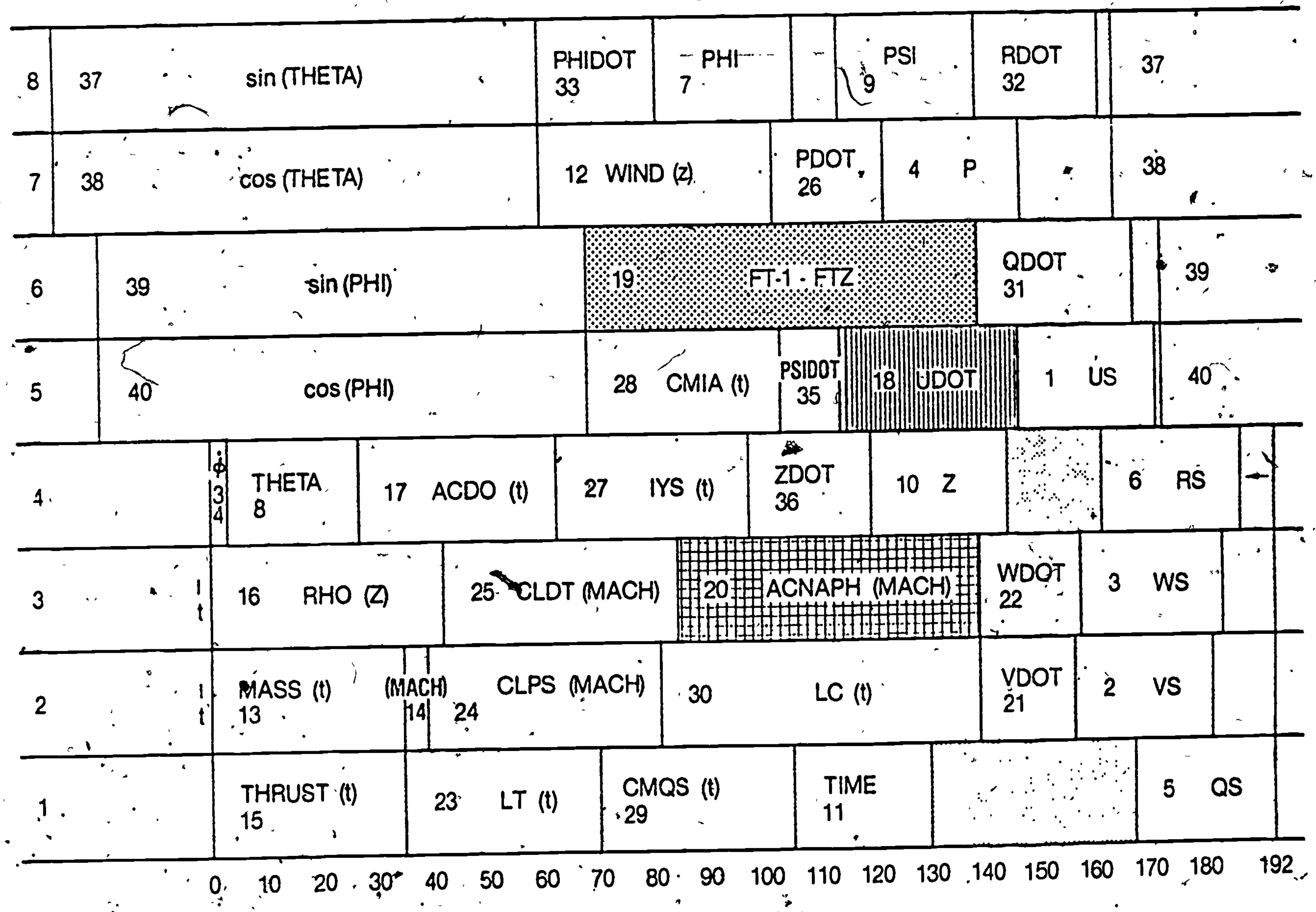


Figure 17 - A Schedule Using Eight Processors

Subsequent analysis has shown that the speed-up shown in Table 4 can be increased to 7.0. This is accomplished by reducing the amount of synchronization. The following analysis indicates the efficiency losses in the solution.

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 and 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

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

The predicted solution time is given by

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

Compare this to the actual solution time, given by Table 4, of 4.87 seconds.

Although execution of the A-10 code was impossible, as discussed in Section 1, some analysis was made of the four subroutines ZM6SF101, ZM6SF102, ZM6AF103, and ZM6SF10E. These subroutines were treated as a single code segment and were divided into 50 tasks. The total execution time of the tasks was 2803 machine instructions, or 2.242 milliseconds.

on HEP, using a single processing stream. Since the execution speed on HEP considerably exceeds the requirement[15], large amounts of parallelism did not seem required. We did, however, schedule the task system for maximum speed-up, which for this system resulted in a solution time of 671 machine instructions or .537 milliseconds using five processors. The efficiency of this solution is 83.5%, but does not include synchronization requirements.

[15] 33.3 milliseconds (30 times per second), as specified by comment lines in the subroutine ZM6SF101.

SECTION 4. REORGANIZATION

This section deals with problem-solving by alternate methods that are either inherently parallel or lend themselves to parallelization. Specifically, we cover the problem of solving ordinary differential equations and examine two examples in the general area of mathematical library functions.

Parallel Techniques for Ordinary Differential Equations

General Methods

This section deals with parallel methods for solving a set of n ODEs denoted by

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

where

$$t_0, \ t \in R, \ y_0 \in R^n, \ y : R \to R^n, \ f : R \times R^n \to R^n$$

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

Interpolation Method

Nievergelt[16] proposed a parallel form of a serial integration method to solve a differential equation, in which the algorithm is divided into subtasks that can be computed independently. The method is as follows:

(1) Divide the integration interval [a,b] into N equal subintervals $[t_{i-1}, t_i]$, $t_0 = a$, $t_N = b$, $i = 1, 2, 3, \ldots, N$

(2) Make a rough prediction y_i^0 of the solution $y(t_i)$

(3) Select a certain number M_i of values y_{ij}, $j = 1, \ldots, M_i$ in the vicinity of y_i^0

(4) Integrate simultaneously (with an accurate integration method M) all the system

$$y' = f(t, y),\ y(t_0) = y_0,\ t_0 \le t \le t_1$$

$$y' = f(t, y),\ y(t_i) = y_{ij},\ t_i \le t \le t_{i+1}$$

$$j = 1,\ldots,M_i,\ i=1,\ldots,N-1$$

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. These branches are connected 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

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

---

[16]J. Nievergelt. "Parallel Methods for Integrating Ordinary Differential Equations". Journal of Computer and System Sciences, 1973, pp. 189-198.

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. 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.

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

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

$$K_i = h_n f(t_n + a_i h_n, y_n + b_{ij} K_j)$$

$$y_{n+1} = y_n + R_i K_i$$

with appropriate values of a's, b's, and R's. A classical four-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[17] considered Runge-Kutta formulas that can be used in a parallel mode. They introduced the concept of a computational front for allowing parallelism. Their parallel second and third order RK formulas are derived by a modification of Kopal's results[18]. The parallel schemes have the structure:

first order: $K_1 = h_n f(t_n, y_n^1)$ (RK1)

$$y_{n+1}^1 = y_n^1 + K_1$$

second order: $K_1^2 = K_1 = h_n f(t_n, y_n^1)$

$$K_2 = h_n f(t_n + ah_n, y_n^1 + bK_1^2) \quad \text{(RK2)}$$

$$y_{n+1}^2 = R_1^2 K_1^2 + R_2^2 K_2$$

third order: $K_1^3 = K_1$

$$K_2^3 = K_2$$

$$K_3 = h_n f(t_n + ah_n, y_n^2 + bK_1^3 + cK_2^3) \quad \text{(RK3)}$$

$$y_{n+1}^3 = R_1^3 K_1^3 + R_2^3 K_2^3 + R_3^3 K_3$$

---

[17] N. L. Miranker and W. M. Liniger. "Parallel Methods for Numerical Integration of Ordinary Differential Equations". Mathematical Computation, Vol. 21 (1967), pp. 303-320.

[18] Z. Kopal. "Numerical Analysis with Emphasis on The Application of Numerical Techniques to Problems of Infinitestimal Calculus in Single Variable". Wiley, New York: Chapman & Hall London, 1955 M R 17, 1007.

The parallel character of these formulas is based on the fact that RK_i is independent of RK_j 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) the parallel third order RK formula is given by:

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

$$y^1_{n+3} = y^1_{n+2} + K_{1,n+2} \qquad \text{(PRK3)}$$

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

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

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

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

where

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

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

The main drawback of (PRK3) is that it is weakly stable. Miranker and Liniger (1967) show 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 $h \to 0$.

Predictor-Corrector (PC) Methods

The serial one-step methods of the 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; because of their one-step nature, they do not make full use of the available information, and their numerical stability does not extend 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 on y_n but also, say, on y_{n-1}, y_{n-2}, ... and f_{n-1}, f_{n-2}, ... 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 to obtain parallel schemes for such methods is reasonable.

A standard, fourth-order, Adams-Moulton serial predictor corrector (SPC) 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 computation scheme (called PECE) of one PC step to calculate y_{i+1} 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 evaluation 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 (1967) developed formulas for the PC method in which the corrector does not depend serially on the predictor, and the corrector calculations can be performed simultaneously. This Parallel Predictor-Corrector (PPC) operates 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 18. A fourth order PPC is given by:

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

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

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

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

where

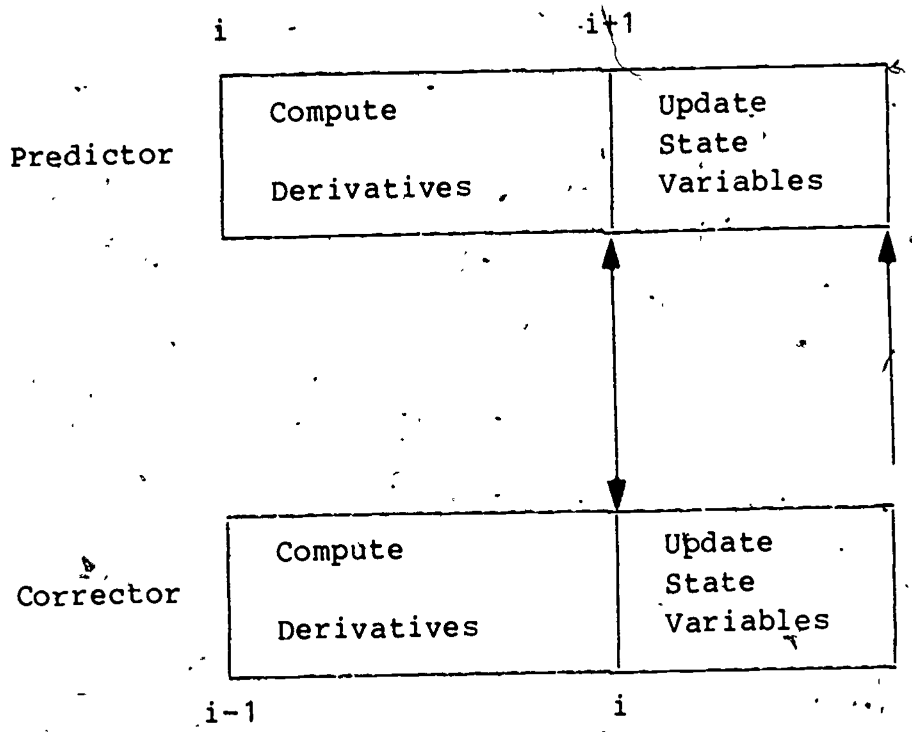


Figure 18 - Parallel PC Scheme

T_{PCE} = T_f as defined before and

T_{DC} = time taken for data communication

T_S = time taken for synchronization.

Generally, the higher accuracy and fewer function evaluations of PC methods (as compared to RK methods) are obtained at the cost of increased complexity and, sometimes, numerical instability. The parallel RK methods given by Miranker and Liniger (1967) do not inherit the stability of their serial counterparts. On the other hand, PPC methods in Miranker and Liniger, as described above, are as stable as their serial formulas. This is proved by Katz et.al.[19].

Block-Implicit Methods

Sequential block implicit methods as described by Andria et. al.[20] and Shampine and Watts[21] produce more than one approximation of y at each step of integration. Shampine and Watts and Rosser[22] discuss block implicit methods for RK and PC type schemes. A two-point, fourth-order PC given by Shampine and Watts is:

---

[19]N. Katz, M. A. Franklin and A. Sen. "Optimally Stable Parallel Predictors for Adams-Moulton Correctors". Computing and Mathematics with Applications, Vol. 3, (1977), pp. 217-233.

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

[21]L. F. Shampine and H. A. Watts. "Block Implicit One Step Methods". Mathematical Computation, Vol. 23 (1964) pp. 731-740.

[22]J. Rosser. "A Runge-Kutta for All Seasons". SIAM Review, Vol. 9 (July. 1967), pp. 417-452.

$$y^p_{i+1} = 1/3(y^c_{i-2} + y^c_{i-1} + y^c_i) + h/6(3f^c_{i-2} - 4f^c_{i-1} + 13f^c_i)$$

$$y^p_{i+2} = 1/3(y^c_{i-2} + y^c_{i-1} + y^c_i) + h/12(29f^c_{i-2} - 72f^c_{i-1} + 79f^c_i)$$

$$y^c_{i+1} = y^c_i + h/12(5f^c_i + 8f^p_{i+1} - f^p_{i+2}) \qquad \text{(BPC)}$$

$$y^c_{i+2} = y^c_i + h/3(f^c_i + 4f^p_{i+1} + f^p_{i+2})$$

Worland[23] describes the natural way to parallelize (BPC) using the number of procesors = number of block points by the schemes shown in Figure 19. The parallel time for one Block calculation given by Franklin[24] is:

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

Franklin also gives a performance comparison of (PPC) and parallel (BPC) methods in case of two procesors.

---

[23] P. B. Worland. "Parallel Methods for the Numerical Solution of Ordinary Differential Equations". IEEE Transactions on Computing", Vol. C-25 (October, 1976), pp. 1045-1048.

[24] M. A. Franklin. "Parallel Solution of Ordinary Differential Equations". IEEE Transactions on Computing, Vol. C-27 No. 5 (May, 1978).

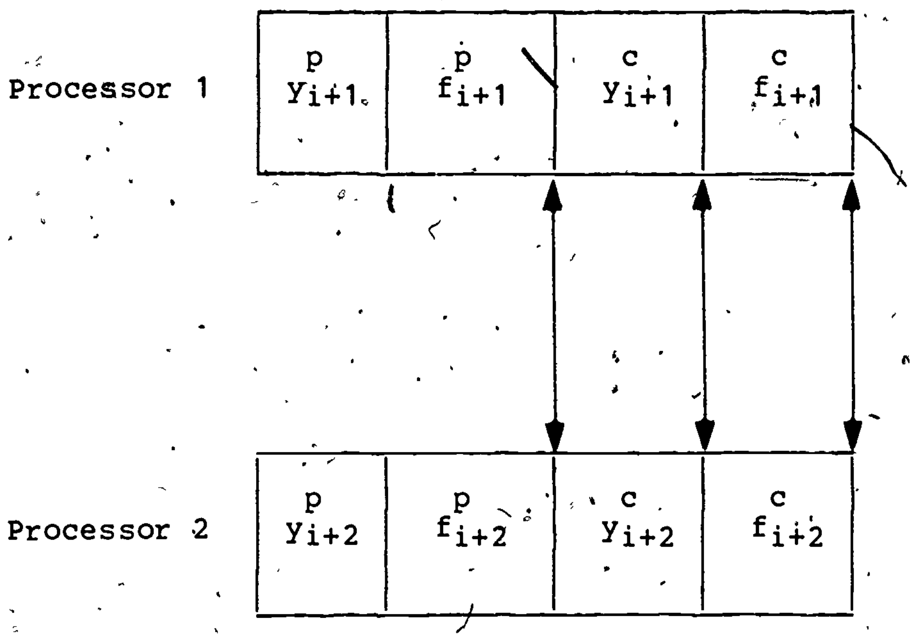


Figure 19 - Parallel Scheme for BPC

Results

For implementation, we used the parallel predictor-corrector method in conjunction with the techniques described in Section 3. For comparison, we also included the results of the Runge-Kutta solutions.

The schedules for the flight simulation problem discussed in Section 3 were programmed using HEP FORTRAN and were executed on the HEP parallel computer. The computational results are shown in Table 5. The sequential times T_1 and the parallel times T_p with p processors are given in terms of seconds. For comparison, the times for the Runge-Kutta method described in Section 3 are also included.

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 5 - Speed-up & Efficiency: Predictor-Corrector and Runge-Kutta Methods

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 5. Since the Serial PC methods are expected to be more efficient than 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 the RK formula. These calculations are done in the following analysis of the loss of the efficiencies in both 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 the 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

Cycles = A + (B - A) + C
= 173 + 19 + 23 = 215

The predicted solution time is given by

PST = Cycles x 28,000 x .8 x 10^{-6} = 4.816 seconds

where the actual solution time given by Table 5 is 4.87 seconds.

For the four-processor PC method, the values of A, B, and C are:

A = 1384/4 = 346
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

Cycles = A + (B - A) + C
= 356 + 17 + 58 = 421 cycles

This gives the predicted efficiency for the PC method

PE = 356/421 = 82%

where the actual efficiency given by Table 5 is 81%.

Mathematical Functions

In addition to the reorganization of differential equations, we have examined the reorganization of two common functions of a mathematical library. In the case of differential equations, the reorganization resulted in different algorithms being employed, whereas in the cases we are about to discuss the algorithms are identical but the programs are considerably reorganized.

Shortest Path Problem

Shortest path problems are among the most fundamental and commonly encountered problems in transportation and communication networks. We included such a problem in this study for three reasons. First, it is directly applicable to flight simulation studies as a mathematical utility function. Second, it is used to schedule algorithms for generating MIMD programs that solve ordinary differential equations. Finally, the techniques used to derive parallelism clearly show the limitations of automatic detection of parallelism. We elaborate this third point in Section 5.

The shortest path problem we examined was the all-to-all program: given n nodes (points, vertices, etc.) and given a distance (cost) between each ordered pair of points, determine the minimum distance (or cost) and path between all pairs of nodes. The distance or cost function does not require the distance from i to j to be the same as the distance from j to i. Further, the triangle inequality is not required to be satisfied. Finally, the distance values may be negative so long as there are no negative cycles. Such a problem could well be stated as: given a number of locations (latitude, longitude and altitude) and given the fuel consumption of an aircraft between all adjacent pairs of points, what is the minimum fuel consumption between some given point and any other point?

The literature contains well over 200 papers on shortest path algorithms[25]. We chose Floyd's algorithm[26], which is very general and provides the minimum path as well as the cost of that path. The nodes of the graph are represented by the integers 1, 2,..., n and the path length (cost) is represented by an n-by-n matrix W where $W_{i,j}$ is the distance from node i to node j. Node j must be adjacent to i (otherwise $W_{i,j}$ has the value ∞). The sequential algorithm is shown in Figure 20. For the algorithm to produce the paths as well as the shortest distance, we need a second n-by-n matrix Z (often referred to as the optimal-policy matrix) where $Z_{i,j}$ is initiated to j if $W_{i,j} \neq \infty$ and zero otherwise. During execution of the innermost loop, if it is found that $W_{j,i} + W_{i,k}$ is less than $W_{j,k}$ then (in addition to replacing $W_{j,k}$) the value of $A_{j,k}$ is replaced with the current value of $Z_{j,i}$. Upon completion of execution, the shortest path from vertex a to vertex b is determined by the vertex sequence:

$$V_1 = Z_{a_1,b}$$

$$V_2 = Z_{V_1,b}$$

$$V_3 = Z_{V_2,b}$$

$$\vdots$$

$$b = Z_{V_q,b}$$

---

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

[26] R. W. Floyd. "Algorithm 97: Shortest Path". Communications of the ACM, Vol. 5 (1962), p. 345.

PROGRAM MINPATH

```
READ N,W
FOR I = 1 TO N DO
    FOR J = 1 TO N DO
        FOR K = 1 TO N DO
            IF $W_{j,i} + W_{i,k} < W_{j,k}$
              THEN
                $W_{j,k} \leftarrow W_{j,i} + W_{i,k}$
WRITE W
```

Figure 20 - Minimum Path Algorithm

To determine a parallel version of this algorithm, we define the code in the most-interior loop to be a task and denote it as T_{ijk}. The algorithm requires that the execution of T_{ijk} be complete before starting execution of T_{uvw} (denoted by $T_{ijk} < T_{uvw}$) if ijk precedes uvw in the natural lexical order. We now determine the maximally parallel task system equivalent to the task system of the sequential program by examining the range and domain of the tasks T_{ijk}. We denote the range (memory locations that are written into by T_{ijk}) by R_{ijk} and the domain (memory cells read by T_{ijk}) by D_{ijk}.

Note that

$$D_{ijk} = \{ W_{j,k}, W_{j,i}, W_{i,k} \}$$

For determining the range, note that the problem requires that there be no negative weight cycles (if this were the case there would be no minimum path for any nodes within the negative cycle). Thus, if i = j then

$$W_{j,k} \leq W_{i,j} + W_{i,k}$$

and if i ≠ k then

$$W_{j,k} \leq W_{j,i} + W_{i,k}$$

Thus

$$R_{ijk} = \begin{cases} W_{j,k} & \text{if } i \neq j \text{ and } i \neq k \\ 0 & \text{otherwise} \end{cases}$$

From the above range and domain we observe that given distinct T, T' where

$$T, T' \in S_i = \{T_{ijk} \mid 1<j<n,\ 1<k<n\}$$

then

$$D_T \cap R_{T'} = 0$$

$$R_T \cap D_{T'} = 0$$

and

$$R_T \cap R_{T'} = 0$$

Consequently, for each value of i, all tasks in S_i may be executed in parallel. Thus:

```
PROGRAM PARPATH

for i <- 1 to n do
    for all 1<j<n and 1<k<n
    do concurrently
        S<- $W_{ji} + W_{ik}$
        if S < $W_{ji}$ then
            $W_{ij}$ <- S
```

is a correct program that produces the same results as the sequential program MINPATH. The parallel version can use n^2 processors resulting in a speed-up on n^2. Since n^2 processors may be larger than the number available in typical systems, we programmed this parallel algorithm for use of K processors where K is in the range of 1 to n. This program is shown in Figures 21 and 22.

The memory limitations of the prototype HEP limited us to a matrix size of 40 x 40. We ran randomly generated test cases for this size using from 1 to 14 processes. The results are shown in Table 6. The agreement between actual and predicted efficiency is good. When the number of processes (P) is a divisor of the dimension (N) of the matrix, the efficiencies are excellent.

Linear Equation Solver

Solving a set of linear equations is a problem of central importance. Nearly every mathematical library contains programs for its solution. One of the most popular

PROGRAM PARPATH

```
READ N,W

$K <- 0.

CREATE STREAM 1 (1)
CREATE STREAM 2 (2)
CREATE STREAM 3 (3)
CREATE STREAM 4 (4)
CREATE STREAM 5 (5)
CREATE STREAM 6 (6)
CREATE STREAM 7 (7)
CALL STREAM 8 (8)
WRITE W.
```

Figure 21 - Main Program for Parallel Path Using Eight Processes

PROCEDURE STREAM$_i$ (L)

```
FOR I = 1 TO N DO
    FOR J = L TO N STEP 8 DO
        FOR K = 1 TO N DO
            IF $W_{j,i} + W_{i,k} < W_{j,k}$
              THEN
                $W_{j,k} <- W_{j,i} + W_{i,k}$
$K <- $K + 1
WAIT UNTIL $K = 8 * I
```

Figure 22 - Subroutine for Parallel Path

			Efficiency	
N	P	Solution Time	Achieved	Predicted
40	1	1.3102		
40	2	.65408	1.0	1.0
40	3	.45176	.966	.952
40	4	.32808	.998	1.0
40	5	.26258	.997	1.0
40	6	.22749	.959	.952
40	7	.19565	.956	.952
40	8	.16524	.99	1.0
40	9	.18217	.898	.889
40	10	.16573	.988	1.0
40	11	.18099	.905	.909
40	12	.19492	.84	.83
40	13	.21017	.779	.769
40	14	1.7582	.931	.952

TABLE 6 – Performance of Parallel Path Algorithm

algorithms is LU decomposition using Gaussian elimination with some form of pivoting. We address only partial pivoting. Details of this algorithm can be found in any standard text on numerical analysis, such as Introduction to Numerical Analysis[27]. Figure 23 shows a serial program for LU decomposition. Our method of reorganizing is to unroll the DO loops, apply the techniques of Section 3, schedule the resulting parallel system, and finally write a number of subroutines, employing DO loops whose parallel execution is equivalent to the original program.

The tasks we have selected are indicated in Figure 23. They consist of the code segment that works on a particular column j for a particular value of k. We denote those tasks by

$$J = \{T_k^j \mid 1 \le K \le j \le n,\ K \le n-1\}$$

The precedence constraints imposed by the sequential program are

$$<\cdot = [(T_k^j, T_m^l) \mid j<l \text{ or } k<m].$$

Thus, $C = (J, <\cdot)$ is the task system that represents the sequential program. The range and domain of these tasks are:

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

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

[27]F. B. Hildebrand. (New York: McGraw Hill, 1974).

```
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]                          $T_k^k$
    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                              $T_k^j$, j>k
          A(i,j) <- A(i,j) - A(i,k)*A(k,j)
```

Figure 23 - Program for LU Decomposition

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' = (T,<·), where <· is the transitive closure on the relation

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

is a maximally parallel system equivalent to C. This system is illustrated in Figure 24.

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, constitutes a time step. Thus, neglecting any overhead for loop control, the execution time $W(T_j^k)$ for each of the tasks is given by:

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

Treating the task system C' together with $W(T_k^j)$ as a weighted graph we observe that the longest path traverses the nodes :

$$T_1^1, T_1^2, T_2^2, T_2^3, T_3^3, \ldots, T_{n-1}^{n-1}, T_{n-1}^n$$

We denote this path as S_1 and the length of the path as $L(S_1)$:

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

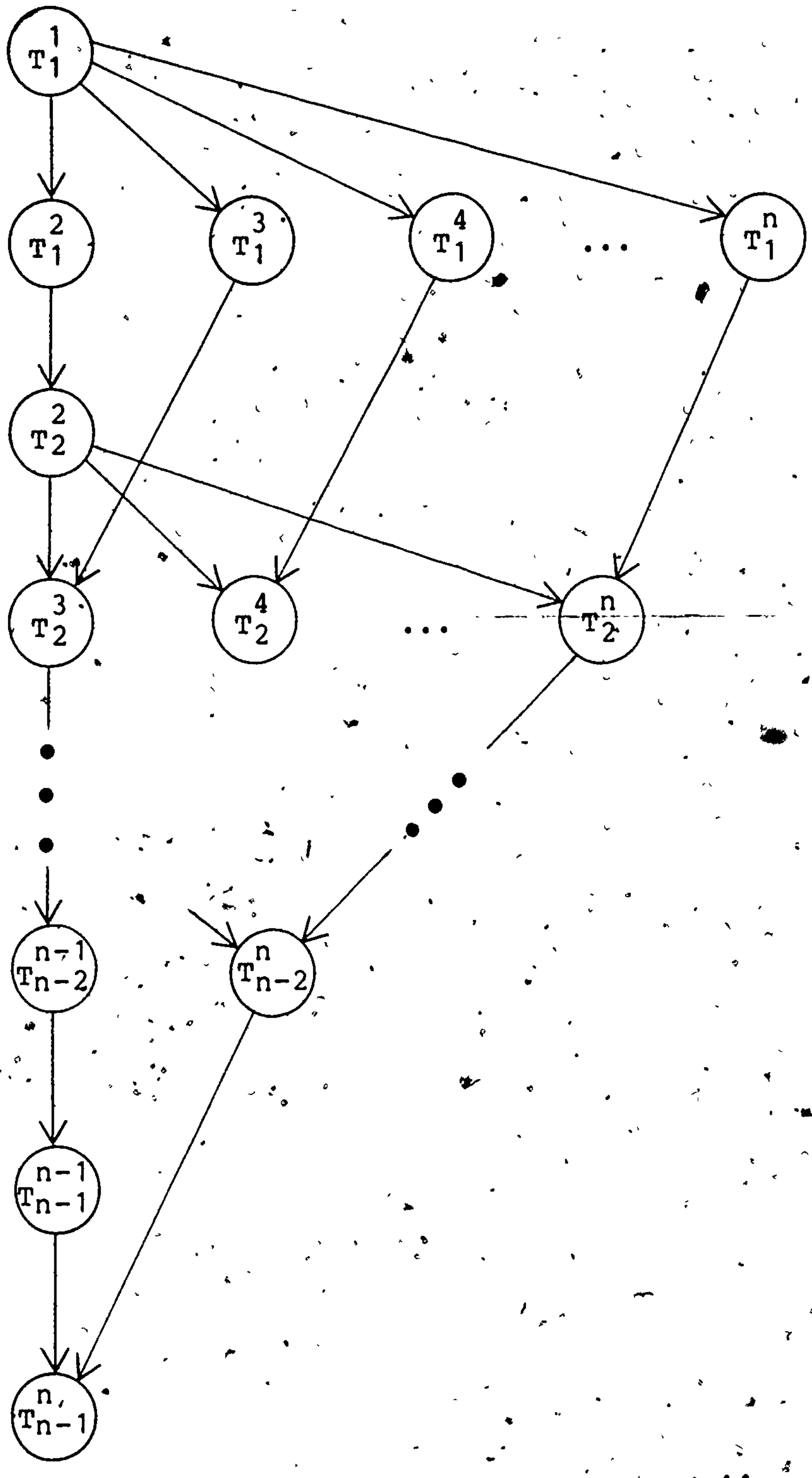


Figure 24 - Maximally Parallel Task System Equivalent to C

Since the problem cannot be solved in a 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 $[n/2] - 1$ additional processors. Processor 2 executes the tasks

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

More generally, processor j executes the tasks

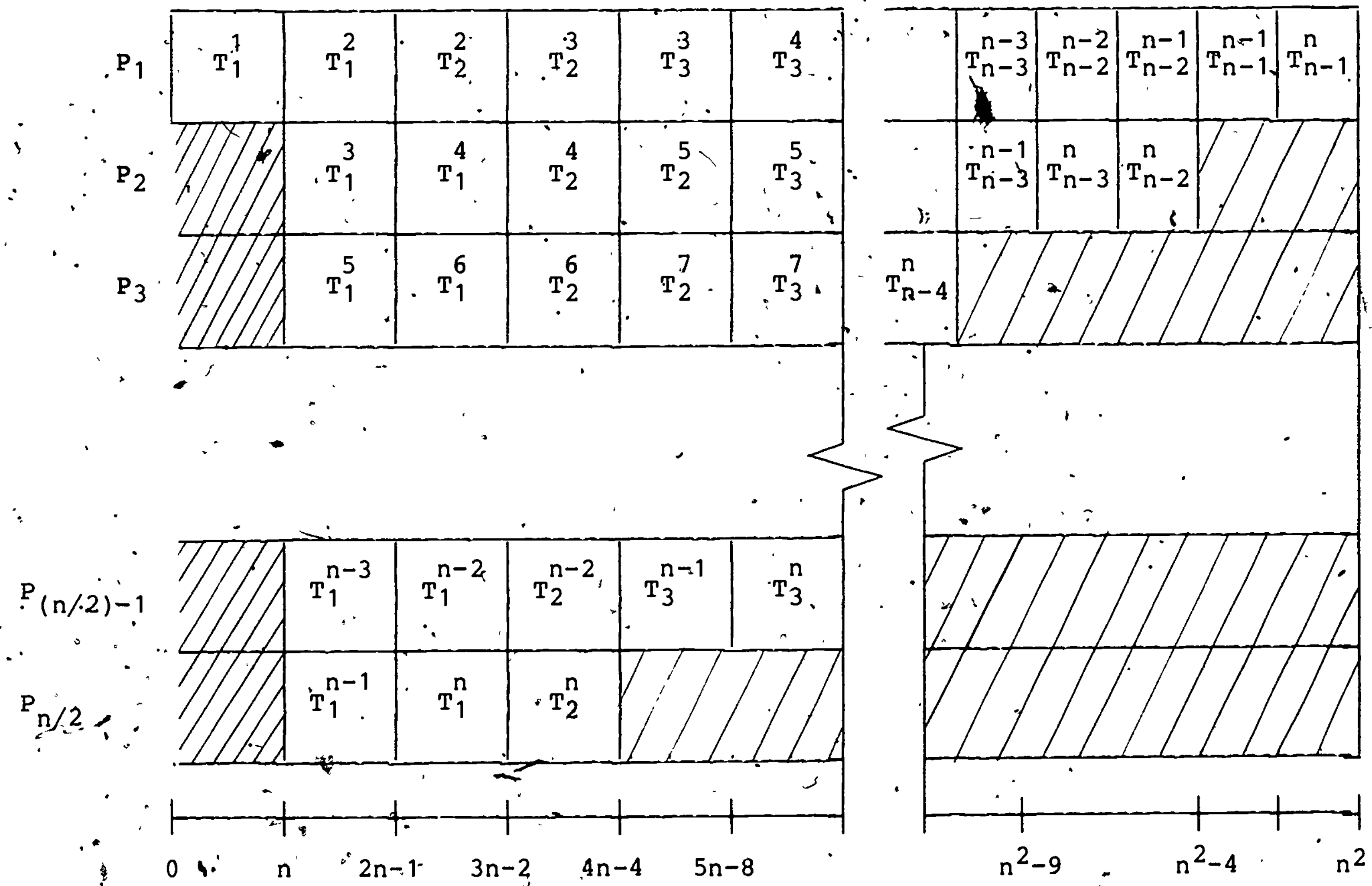
$$T_1^{2j-1},\ T_1^{2j},\ T_2^{2j},\ T_2^{2j+1},\ \ldots,\ T_{n-2(j-1)}^{n}$$

We 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 Figure 25. 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 ($n \geq 50$).

These schedules were programmed using HEP FORTRAN, and were run on the HEP parallel computer. Although the program solved a set of linear equations, we recorded timing for only the LU decomposition so that it could be compared with the predicted solution times. Table 7 gives the actual and predicted efficiencies for the number of equations ranging from 10 to 35 and the number of parallel instruction streams ranging from 2 to 8.



P_1
P_2
P_3
$P_{(n/2)-1}$
$P_{n/2}$
T_1^1 T_1^2 T_2^2 T_2^3 T_3^3 T_3^4
T_1^3 T_1^4 T_2^4 T_2^5 T_3^5
T_1^5 T_1^6 T_2^6 T_2^7 T_3^7
T_{n-3}^{n-3} T_{n-2}^{n-2} T_{n-2}^{n-1} T_{n-1}^{n-1} T_{n-1}^{n}
T_{n-3}^{n-1} T_{n-3}^{n} T_{n-2}^{n}
T_{n-4}^{n}
T_1^{n-3} T_1^{n-2} T_2^{n-2} T_3^{n-1} T_3^{n}
T_1^{n-1} T_1^{n} T_2^{n}
0 n 2n-1 3n-2 4n-4 5n-8
n^2-9 n^2-4 n^2

Figure 25 - Schedule Using n/2 Processors (n even)

no. of equations		number of processors						
		2	3	4	5	6	7	8
10	A	.833	.719	.642	.633			
	P	.852	.739	.678	.685			
15	A	.888	.794	.740	.651	.618	.625	.581
	P	.900	.815	.766	.679	.652	.681	.633
20	A	.921	.843	.774	.758	.670	.623	.605
	P	.931	.863	.798	.789	.703	.656	.640
25	A	.934	.878	.830	.763	.755	.692	.642
	P	.944	.896	.855	.739	.788	.726	.675
30	A	.942	.892	.844	.818	.757	.744	.710
	P	.949	.911	.863	.843	.783	.777	.745
35	A	.948	.901	.862	.819	.790	.747	.741
	P	.956	.918	.880	.843	.827	.779	.769

A = Actual efficiency.

P = Predicted efficiency.

Table 7 - Efficiency of LU Decomposition

SECTION 5: CONCLUSIONS

In this study we have examined programs that are all in support of flight simulation, but which can also be categorized by the types of mathematical functions or services that they supply. Categorized in this manner they are:

(1) numerical approximation of elementary functions,
(2) solution of linear algebraic equations,
(3) solution of shortest path problems on graphs, and
(4) solution of ordinary differential equations.

For problems in the first category, we examined the program at an arithmetic instruction level and produced parallel code based on this examination. These techniques produced speed-ups in the range of two to three. Since elementary function approximation involves small amounts of computation, these very modest speed-ups are perhaps to be expected. In producing parallel code for these functions, we were guided by the formal work in the area of polynomial evaluation. We do not foresee any automated approach to producing a library of elementary functions for a particular MIMD computer, but this does not adversely affect the potential of MIMD computing. Historically, elementary function libraries have been coded in machine language and highly tailored for the target machine.

To solve linear algebraic equations, we unrolled the DO loops, represented the computation as a task system, and from this produced a number of DO loops that could be executed in parallel. As we mentioned in Section 2, automatic detection of parallelism within nested DO loops is receiving considerable research interest. We believe that if future Air Force simulation requirements include flexible body representations (generally requiring solution of linear algebraic equations), either the algorithms developed here will be of benefit or automatic recognizers of parallelism within nested DO loops will be available. The speed-up available in this problem type is bounded only by the size of the problem. For example, given a set of 100 linear equations, our algorithm solves them approximately 35 times faster than a sequential equivalent, and with very good efficiency.

Producing a parallel version of the shortest path program involved unrolling the DO loops and treating the resulting code as a task system. To determine the precedence relations, however, we used information from both the code of the program and knowledge of the input data sets for which this program was correct. Thus it is difficult to see how automatic detection of parallelism within DO loops could have produced the same parallels we did. But this should not detract from the benefits of MIMD computing. In most flight ~~simulation~~ programs, minimum and maximum path algorithms are usually utility routines; their status as library function should be adequate. As was the case for the linear equations, parallelism and speed-up is bounded only by the size of the problem. For example, given a shortest or longest path problem involving 100 points, a speed-up of 100 over an equivalent sequential program is achievable with efficiencies near 100%.

For solving ordinary differential equations, a variety of techniques were investigated. Those described in Section 3 seemed most successful. Two distinct flight simulation programs were examined using these techniques, the ground-launched missile and the aerodynamics portion of the A-10 flight simulation. The A-10 aerodynamics is approximately twice as much code as the missile simulation (2803 machine instructions versus 1384) and presumably represents the approximate fidelity of simulations currently used for training purposes. On these assumptions we conclude that a MIMD computer of the power of HEP could be used in Air Force flight simulation projects in two ways:

(1) as a multiprogramming computer capable of running several concurrent simulations of the fidelity of the A-10 simulation, or

(2) as a computer capable of running one or two concurrent flight simulations of significantly more fidelity than the current A-10 programs.

Greater fidelity is possible not only in the aerodynamic section but also in computing visual cues for the trainee.

Should an MIMD computer of significantly less power than HEP be employed for flight simulation, our study indicates that there is adequate parallelism within these types of problems that lesser computing power would be adequate for single simulation programs.

REFERENCES

Andria, F. D., G. D. Byrne and D. R. Hill "Natural Spline Block Implicit Methods". BIT, Vol. 13 (1973), pp. 131-144.

Baer, J. L. and D. P. Bovet "Compilation of Arithmetic Expressions for Parallel Computation". Information Processing 68, Amsterdam: North Holland Publishing Company, 1969.

Coffman, Edward G. Jr. and P. J. Denning. Operating Systems Theory. Englewood Cliffs, NJ: Prentice Hall, 1973.

Denning, P. J. "Third Generation Computer Systems". Computing Surveys, Vol. 3 No. 4 (1971), pp. 1975-216.

Deo, N. and C. Y. Pang. Shortest Path Algorithms: Taxonomyand Annotation. Technical Report No. CS-80-057, Computer Science Department, Washington State University, Pullman, WA (March, 1980).

Dijkstra, E. W. "Cooperating Sequential Processes". Programming Languages, F. Genuys, Ed. New York: Academic Press, 1968) pp. 43-112.

Dorn, W. S. "Generalization of Horner's Rule for Polynomial Evaluation". IBM Journal, April 1962, pp. 239-245.

Floyd, R. W. "Algorithm 97: Shortest Path". Communications of the ACM, Vol. 5 (1962), p. 345.

Flynn, M. J. "Very High Speed Computing Systems". Proceedings IEEE, Vol. 54 (1966), pp. 1901-1909.

Franklin, M.A. "Parallel Solution of Ordinary Differential Equations". IEEE Transactions on Computing, Vol C-27 No. 5 (May, 1978).

Hart, J. F. et. al. Computer Approximations. New York: Wiley & Sons, Inc., 1968.

Hildebrand, F. B., Introduction to Numerical Analysis. New York: McGraw Hill, 1974.

Katz, N., M. A. Franklin and A. Sen. "Optimally Stable Parallel Predictors for Adams-Moulton Correctors". Computing and Mathematics with Applications, Vol. 3 (1977), pp. 217-233.

Kohler, W. H. "Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on a Multiprocessor System". IEEE Transactions on Computing, Vol. C24 No. 12 (December 1975), pp. 1235-1238.

Kopal, Z. Numerical Analysis with Emphasis on The Application of Numerical Techniques to Problems of Infinitesimal Calculus in Single Variable. New York: Wiley and Sons, Inc. (1955)

Kuck, David J. "Multioperation Machine Computational Complexity". Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, Ed. New York: Academic Press (1973) pp. 17-48.

Lord, R. W. Scheduling Recurrence Equations for Solution on MIMD Type Computers. PhD Dissertation, Washington State University, 1976.

Miranker, N. L. and W. M. Liniger. "Parallel Methods for the Numerical Integration of Ordinary Differential Equations". Mathematical Computation, Vol. 21 (1967), pp. 303-320.

Munro, Ian. "Optimal Algorithms for Parallel Polynomial Evaluation". Journal of Computer and System Sciences, 1973, pp. 189-198.

Nievergelt, J. "Parallel Methods for Integrating Ordinary Differential Equations". Communications of the ACM, Vol. 7 No. 12 (December 1964), pp. 731-733.

Rosser, J. "A Runge-Kutta for All Seasons". SIAM Review, Vol. 9 (July, 1967), pp. 417-452.

Shampine, L. F. and H. A. Watts. "Block Implicit One Step Methods". Mathematical Computation, Vol. 23 (1964), pp. 731-740.

Strauss, J. C., et. al. "Continuous System Simulation Language". Simulation, Vol. 6 No. 12, (December 1967).

Ullman, J. D. "Polynomial Complete Scheduling Problems". Operating Systems Review, Vol. 7 No. 4 (1973), pp. 96-101.

Winograd, F. "On The Number of Multiplications Required to Compute Certain Functions". Proceedings National Academy of Science USA, Vol. 58 (1967), pp. 1840-1842.

Worland, P. B. "Parallel Methods for The Numerical Solution of Ordinary Differential Equations". IEEE Transactions on Computing. Vol. C-25 (October, 1976), pp. 1045-1048.