Parallel and Vector Problems on the FLEX/32

H S. McFaddin

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
87-661
PARALLEL AND VECTOR PROBLEMS ON THE FLEX/32

H. S. McFaddin
J. R. Rice

CSD-TR-661
March 1987
PARALLEL AND VECTOR PROBLEMS ON THE FLEX/32

H.S. McFaddin
J.R. Rice

Department of Computer Science
Purdue University
West Lafayette, Indiana 47907
CSD-TR-661

March 2, 1987

ABSTRACT

Earlier we proposed [Rice, 1985] sixteen problems to test the effectiveness of languages in expressing parallel and vector computations. These problems were presented in ordinary notation (mathematics and English) plus four algorithmic forms: A) Fortran 77, B) Fortran 77 with extensions (resembling Fortran 8X), C) PROTRAN with extensions and D) Cyber 205 Fortran. We now present these problems programmed for the FLEX/32 multiprocessor in a Concurrent Fortran. Our objectives are twofold: 1) to show how these problems appear in this language (which is similar to those on several other multiprocessors), 2) to show the parallel efficiency achieved for these problems.

* This work was supported by IMSL, Inc.
** This work supported in part by ARO grant DAALO3-86-K-0106
1. INTRODUCTION AND SUMMARY

The purpose of this report is to explore two properties of the FLEX/32 multiprocessor described in detail in [Flexible, 1985] (see Section 3 for a brief description). First, we wish to evaluate its effectiveness as a parallel computer (see [Houstis, et al., 1987] for some earlier work). Second, we wish to evaluate the programming methodology required by its Concurrent Fortran language. We have programmed sixteen problems on the FLEX/32 taken from [Rice, 1985] which are designed to test the effectiveness of programming languages for parallel and vector applications. These problems are also well suited to test the speedup obtained with parallel and vector computers.

The sixteen problems are summarized in the next section and the FLEX/32 and its Concurrent Fortran programming language are briefly described in Section 3. In Section 4 we present speedup curves for these problems using 2, 3, 4 and 5 processors of the FLEX/32. Each problem is parametrized so that both the amount of parallelism and the size of the computation can be increased. The data is in the form of efficiency

\[ E = \frac{\text{Time with one processor}}{N \cdot (\text{Time with } N \text{ processors})} \]

versus computer time with one processor. The range of times are from 200 to 2200 ticks (or 4 to 44 seconds), depending on the problem. A tick on the FLEX/32 clock is 1/50 second.

Appendix One contains the sixteen programs for the problems. These are complete programs just as used for the speedup evaluations. Each program starts with code for interactive input which is very similar from program to program.

The design of the FLEX/32 is best suited for one or a small number of applications that run for a long time. Such applications occur commonly in real-time control. The initiation of a set of parallel processes is a substantial activity: programs are loaded into the local memories of the processors, data are placed in common memories, tables are set up for parallel synchronization, etc. It is somewhat analogous to the link/load step in a sequential computation and takes at least one second of real time. The effect of this activity is magnified by the fact that some parts are sequential in the number of processors used.

This situation has a direct impact on synchronizations that must occur in some of the sixteen problems. The programs initiate the number of parallel processes to be used and then synchronization is carried out by traditional techniques (semaphores, critical variables, etc.) without using the parallel constructs of Concurrent Fortran.

An examination of the efficiency curves in Section 4 shows that they all start off quite low and one must have a 10 to 20 ticks sequential job for any of the programs to reach efficiencies of 80 or 90 percent. This is due to the start-up times for the parallel computations. Appendix Two contains the data on timing and efficiency.

A visual examination of the efficiency curves suggests that, with five processors, no program reaches 95% efficiency for a job less than 800-1000 ticks (about 15-20 seconds). We summarize that observations for the 14 problems (excluding numbers 8 and 16), the job sizes are measured in ticks on a single FLEX/32 processor.
Job size to reach 75% efficiency

<table>
<thead>
<tr>
<th></th>
<th>2 processors</th>
<th>5 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>Typical</td>
<td>15-20</td>
<td>200-300</td>
</tr>
<tr>
<td>Maximum</td>
<td>100</td>
<td>3000</td>
</tr>
</tbody>
</table>

Job size to reach 95% efficiency

<table>
<thead>
<tr>
<th></th>
<th>2 processors</th>
<th>5 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>100</td>
<td>500</td>
</tr>
<tr>
<td>Typical</td>
<td>250</td>
<td>2000-5000</td>
</tr>
<tr>
<td>Maximum</td>
<td>&gt;2000</td>
<td>?</td>
</tr>
</tbody>
</table>

The reason that efficiencies of 95% are difficult to achieve is that most of the problems have some small "sequential" part (such as forming a sum) or have some synchronization code which is unneeded in a sequential computation.

Keep in mind that we have programmed most of these problems in the style that we think is typical of general programming: 1) some initial thought is given to the computations, 2) an approach is chosen, 3) a code is written and, 4) some testing for correctness is made. We did not invest the effort to obtain highly efficient codes for these problems.

Two problems, #8 (compute a divided difference table) and #16 (multiple linear equation solutions), have low efficiencies for all job sizes. Our program for problem #8 is essentially sequential in behavior, we have not used an appropriate parallelization here. High parallel efficiency is only possible here when the problem size is very large compared to the number of processors. Our program for problem #16 also does not use an appropriate parallelization, a comparison with problem #10 shows this.

Writing efficient programs for the FLEX/32 requires one to become familiar with the details of the machine and to learn various synchronization techniques for parallel computations. The principal support that Concurrent Fortran provides for synchronization is shared critical variables, the explicit synchronization facilities are too inefficient to be used often. We made a few experiments of the effect of coarse grain verses small grain parallelism. For example, a computation with 3 processors with $i$ going from 1 to 300 can be divided into 3 parts in two obvious ways:

\[ 1 \leq i_1 \leq 100, \ 101 \leq i_2 \leq 200, \ 201 \leq i_3 \leq 300, \]

or

\[ i_1 = 1, 4, 7, ..., \ i_2 = 2, 5, 8, ..., \ i_3 = 3, 6, 9, ... . \]

If no further synchronization were required, then these two approaches give essentially equal efficiency. Of course, the coarse grain approach is more efficient if there is much synchronization required at all.
2. THE SIXTEEN PROBLEMS

Problem 1: Evaluate the trapezoidal rule estimate of an integral of \( f(x) \):

\[
T_N = h^* \left( \frac{1}{2} f(a) + \sum_{i=1}^{N-1} f(a + i h) + \frac{1}{2} f(b) \right)
\]

Problem 2: Compute the value of

\[
e^* = \sum_{i=1}^{n} \prod_{j=1}^{m} (1. + e^{(-1 + j)})
\]

Problem 3: Compute the value of \( S = \sum_{j=1}^{m} \prod_{i=1}^{n} a_{ij} \)

Problem 4: Compute the value of \( R = \sum_{i=1}^{N} \frac{1}{x_i} \)

Problem 5: One has a table of the \( i \)-th student's score on the \( u \)-th test. One is to

(a) list the top score for each student = \( \text{top}_i \)
(b) give the number of scores above the average = \( \text{NABOVE} \)
(c) increase all the above average scores by 10 percent
(d) give the lowest score that is above average = \( \text{BLOW ABOVE} \)
(e) say whether any student has all scores above average = \( \text{GENIUS} \)

Problem 6: Solve the tridiagonal system \( Tx = y \) by the special, vector oriented algorithm of [Jordan, 1979]. The matrix \( T \) is represented by \( L, D \) and \( U \), its lower diagonal, main diagonal and upper diagonal.

Problem 7: Compute polynomial interpolant values of \( f(x) \) at five points using Lagrange interpolation formulas:

\[
p(x) = \sum_{i=1}^{N} f(x_i) l_i(x) \quad l_i(x) = \prod_{j=1, j \neq i}^{N} \frac{x - x_j}{x_i - x_j}
\]
Problem 8: The divided difference table for a set of data \( x_i, y_i = f(x_i) \) is defined by the formulas

\[
f[x_i] = y_i
\]

\[
f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k}]}{x_{i+k} - x_i}
\]

The problem is to compute the first \( M \) columns of the divided difference table

\[
D_{ik} = f[x_i, x_{i+1}, \ldots, x_{i+k-1}]
\]

Problem 9: One has an array \( u_{ij} \) of values on an \( N \) by \( M \) grid and wants to replace each value by the average of its value plus those of all its neighbors. This is expressed by

\[
u_{ij} = \left( \sum_{Neighbours} u_{ij} \right) / \text{(Number of neighbors)}
\]

This computation is typical of what one does in solving partial differential equations, image processing and geometric modeling.

Problem 10: LU factorization of the \( N \) by \( N \) matrix \( A = a_{ij} \) using Gauss elimination with pivoting.

Problem 11: Read sets of data \( d_i, i = 1, \ldots, N \), trim the negative values to zero and large values to 1000, do a logarithmic transformation \( d_i = \log(1 + d_i) \) and compute the first four Fourier moments \( \sum_{i=1}^{N} d_i \cos(\pi i/(N + 1)) \) then save these moments and the data ID in a data base.

Problem 12: Given the \( m \) by \( m \) matrix \( A \), the \( 1 \) by \( m \) vector \( R \), the \( m \) by \( 1 \) vector \( C \) and a number \( A \), construct the array

\[
\text{ABIG} = \begin{bmatrix} A & C \\ R & \alpha \end{bmatrix}
\]
Problem 13: For given vector $a$, $b$, $c$, $d$ compute the new vector

$$a_i = a_i \sin b^i$$

If $a_i < \cos(c_i)$ then $a_i = a_i + c_i$
else $a_i = a_i - d_i$

and compute

$$e = \sum_{j=1}^{n_{\text{dim}}} a_i^2$$

Problem 13H: Modify Problem 13 for a machine that wants to have the computation split into 20 processes (e.g., such as the HEP).

Problem 14: Carry out a test of four methods to integrate three different functions with 10 different levels of accuracy each. Print out a table with all the results including the number of function evaluations and in each integration. This problem comes from [Rice, 1983], page 204.

Problem 15: Carry out a comparison of two types of interpolation points (equispaced and chebyshev spaced) for Hermite interpolation using piece-wise cubic polynomials. The interpolant’s value $v$ at $y$ can be expressed as

$$v(y) = \sum_{j=1}^{N} f(x_j)h_{1j}(y) + f'(x_j)h_{2j}(y)$$

where $h_{1j}(x)$ and $h_{2j}(x)$ are suitable basis functions that depend on the $N$ interpolation points $x_j$. This problem comes from [Rice, 1983], pages 93, 98 and 380-381.

Problem 16: Solve a matrix equation $Ax = B$ where $A$ is an $N$ by $N$ Hilbert matrix and $B$ is an $N$ by 4 matrix. The matrix order $N$ takes on the values 4, 8, 12, 16 and 20 and the $B$ column-vectors are, respectively, the first column of the identity matrix, all 1’s, a 0.01 random perturbation of all 1’s, and alternating +1, -1.
3. THE FLEX/32 MULTICOMPUTER AND ITS CONCURRENT FORTRAN

3.1. FLEX/32 Architecture

The FLEX/32 is a MIMD (Multiple Instruction Stream Multiple Data Stream) computing device which may be configured to operate up to 20 independent processing units. Each processor may access a shared common memory via a common bus, as well as its own local memory. The programs included in this report primarily use the shared memory—local memories are used only to store intermediate computations. An effort is made to reduce shared memory accesses, although the report [Houstis et al., 1987] indicates that the effect of memory contention is negligible. Figure 1 shows a block diagram of the structure of the FLEX/32, see [Flex, 1985] for more details.

The FLEX/32 operated by the Computer Science Department is currently configured with 7 processors. One processor runs multi-user UNIX for program development and has 4 Mbytes of local memory. The remaining six operate in batch-parallel mode under the MMOS operating system, each has 1 Mbyte of local memory. There are six shared memory modules with 512 Kbytes of memory each.

3.2. FLEX/32 Concurrent Fortran

3.2.1 Implementing Parallelism on the FLEX/32

FLEX/32 Concurrent Fortran is an example of a parallel language constructed on top of an older one. The new features range in functionality from high level (block structures, conditional waits, etc.) to low level (explicit process forks).

A parallel program in Concurrent Fortran may be viewed as a main procedure which has been downloaded from the UNIX development computer onto one of the MMOS computers. Eventually, the main program may fork into a collection of independent processes executing concurrently on one or more of the MMOS computers. The machine is then operating in parallel as illustrated in Figure 2.

The forking operation has been observed to be expensive, requiring as much as one second of real time. Thus, it is a bad idea to write programs which repeatedly fork and join (i.e., return to MAIN) as a form of interprocess synchronization. For example, consider a program which performs Gaussian Elimination on a matrix in shared memory. We might choose to compute each submatrix in parallel, and thus approach the problem as shown in Figure 3. This is a bad approach. Instead, the programmer should use a single fork (or a constant number of them) per program run, and relegate synchronization duties to the processes spawned. In the programs of this report, for example, synchronization is implemented by incrementing and inspecting integer variables (semaphores) in the shared memory.

From the viewpoint of the programmer, then, parallelism in Concurrent Fortran should be coarse grained—the chunks of code parallelized are on the order of entire programs, instead of a few lines. If parallelization is synchronized and done at the “few lines of code” level, then it
Boards with processors and local memory may be connected to shared memory modules. The schematic shows up to 10 shared memory modules (SM1, SM2, ...) and up to 20 processor boards with a processor (P1, P2, ...) and local memory (LM1, LM2, ...).

Figure 1. Schematic of the FLEX/32 architecture. There may be up to 10 shared memory modules (SM1, SM2, ...) and up to 20 processor boards with a processor (P1, P2, ..., ) and local memory (LM1, LM2, ..., ).
must be synchronized using shared variables which adds a significant but not overwhelming additional computation. This reflects the design intentions of the FLEXI32, as a general purpose industrial machine capable of concurrently operating several dissimilar processes over long periods of time.

3.2.2 Declaring Shared Memory

Shared memory must be made visible to every process wishing to access and/or change it. This is done by declaring shared variables. The syntax is to precede a normal FORTRAN declaration by the word shared.

Example:

```fortran
shared real / r1 / a(100,100) pivot
shared real / i1 / isynch, ipvt, row(100)
```

Shared blocks should be named (if there is more than one) and shared variable declarations may not involve parameter values.
Figure 3. Schematic of an inefficient approach to Gauss Elimination because of excessive forking, which is very expensive in the FLEX/32.

3.2.3 Creating Processes

To create a new process, a parent process, such as MAIN, must supply an integer into which the process id of the child is to be written, provide code to begin executing (in the form of a subroutine call) and specify the computer on which to operate. The syntax of the creation statement is

\[
\text{process ( proc_id, subroutine_name, computer_number )}
\]

The code to be mapped to the specified computer is determined by mimicking the given subroutine call - addresses (I-values of argument expressions) are placed in stacks, etc. However, the
spawned process does not begin execution immediately. Rather, the parent process is free to execute other code, possibly spawning other processes, before initiating actual execution.

Example:

```fortran
  do 50 i = 3,6
      arg(i) = i
      process ( pid(i), code(arg(i)), i )
  50 continue

  -- now start above processes --
```

Here, the programmer must be careful. Since addresses (call by reference) are passed to subroutines in Fortran, the programmer must bear in mind that the values at those addresses are determined at the time execution begins, but the location references are established at the time of the process statement. To force the evaluation of arguments at the time of the process statement, an intermediate structure should be used, such as the array "arg" above. Had we specified code(i) instead of code(arg(i)) as the subroutine call, then the process would access the value in the integer i, not at the time of the process statement, but later, at the time process execution is initiated. This would give i = 6 for all processes.

The process construct interferes with programmer abstraction on two counts. First, the programmer must specify explicitly which computer is to execute the new process. This requires knowledge of the current configuration of the machine. Secondly, the programmer must chronologically separate the determination of the address of a subroutine argument (the l-value of the argument expression), occurring at the time of the process statement, and the actual value (r-value) of the argument, determined later, at process initiation. The child process cannot be viewed as a subroutine call at the point of the process statement, for the r-value is not determined. Neither can it be viewed as a subroutine call at the point of process initiation, for the l-value was determined earlier.

3.2.4 Block Structures for Process Generation

As explained above, the programmer may choose to defer initiation of one or more spawned processes, beginning them all simultaneously. This is accomplished by enclosing the process statements in a COBEGIN block:

```
  COBEGIN
      -- process statements and other code --
  END COBEGIN
```

Initiation of all processes within the block is deferred until the bottom of the block is reached. The parent process may continue executing at that point.

Another alternative is to have spawned process begin executing immediately, but force the parent process to wait until all children have returned before proceeding. This construct is the COEND block:
The COBLOCK construct combines the above two, deferring initiation of child processes to the bottom of the block, and forcing the parent process to wait until all child processes in the block have terminated.

3.2.5 Conditional Waiting

Concurrent Fortran provides a facility to have a process wait until a condition is satisfied:

```
WHEN(cond)statements
```
or

```
WHEN(cond)statements
  -- block of statements --
END WHEN
```

This is spin waiting.

3.2.6 Critical Access

When several processes wish to update a shared memory location, it is often necessary that only one process have access at a time. Suppose, for example, several processes \( P_i \) have each generated a local value \( X_i \) and we wish to form the sum \( S = \sum X_i \). Then while some process \( P_i \) executes \( S = S + X \) no other process should read \( S \). Process \( P_i \) must have exclusive access to \( S \).

Concurrent Fortran provides two facilities to ensure exclusive access to shared memory constructs.

The first is via the WHEN structure above. All shared variables in the conditional part of a WHEN statement are locked while the conditional is being evaluated and, if successful, while the code within the WHEN block is executing. Thus, it is possible to implicitly generate a block of code having exclusive access to portions of shared memory. Notice in this example that process \( P_i \) does not really want to wait on a condition — it simply wants exclusive access to \( S \) while it executes the (short) block of code

\[
S = S + X
\]

We must use a conditional which always evaluates to .TRUE. and contains the variable \( S \). The typical technique, then, is to use a block such as

```
WHEN ( S .EQ. S ) THEN
  S = S + X
END WHEN
```
The WHEN block implements exclusive access by calling the MMOS system routines CFlock and CFulck, which lock and unlock, respectively, shared variables. The programmer may wish to explicitly perform locking by calling the routines himself:

call CFlock (ICFret, # of variable names, list of variable names )
and

call CFulck (ICFret, # of variable names, list of variable names )

(ICFret is an integer through which error codes are returned) In our example, we would use the code

\[
\begin{align*}
\text{call CFlock}( & \text{ICFret, 1, 'S'}) \\
S & = S + X \\
\text{call CFulck}( & \text{ICFret, 1, 'S'})
\end{align*}
\]

We make two observations regarding this locking scheme. First, the programmer must bear in mind that any shared variables mentioned in the conditional of a WHEN statement are locked for the duration of the block, not just for the conditional evaluation. Thus, the WHEN block should be as small as possible, so that the process does not tie up the shared variable any longer than necessary. In fact, we suggest that the block form of the WHEN structure be used only when the programmer desires BOTH the "wait" functionality and the exclusive access functionality of the WHEN block. Otherwise, the simpler form of the WHEN

WHEN( condition ) continue

should be used when only waiting is required, and explicit lock calls when only exclusive access is required. Thus, we would classify the first locking scheme given above as bad.

The second observation is that the locking routines tend to lock "too much". Since the locking routine uses the variable name as a key, the programmer is not allowed to lock only portions of arrays. Here, caution is advised, for the programmer could inadvertently create a deadlock situation by assuming only part of an array was locked. This is especially likely if the programmer is in the habit of using WHEN structures to implicitly lock arrays.

3.2.7 Synchronization

Concurrent Fortran provides STATIC communication channels which allow one process to trigger exceptions in another process. In the programs of this report, we have chosen a more dynamic, and higher level, approach to synchronization, by implementing semaphores in shared memory.

Suppose we have a section of code C which we wish to be executed simultaneously on each of N processors and we wish all processors to halt together before continuing. An example of such a code fragment would be a single submatrix reduction in Gaussian elimination — no processor should proceed to the next submatrix reduction until all have finished the current one. Assume there is an integer variable ISYNCH in shared memory (and thus visible to all N processes) which had value S before any of the processes began executing code C. To synchronize the processes we require that each increment the semaphore ISYNCH and then wait until the other processors have done so. A typical code fragment would be
C (the code fragment being synchronized)

```
call.CFlock(lCFret,l,'isynch')
ISYNCH = ISYNCH + 1
call CFunlock(lCFret,l,'isynch')
WHEN (ISYNCH .GE. S + N) CONTINUE
```

The system calls provide the locking functionality needed to correctly increment the semaphore, while the WHEN statement provides the waiting functionality.

4. THE PARALLEL EFFICIENCY CURVES

We plot efficiency E versus the job size (ticks for a sequential computation) for each of the sixteen problems. Four curves are given for each problem corresponding to using 2, 3, 4 or 5 processors. As one would expect, the efficiency is generally decreasing as one increases the number of processors. The timings are made using the standard FLEX/32 mechanism (one counts "ticks" which are 20 milliseconds each). It has been found to be reliable (repeatable) and consistent with other timing data for these NS32032 processors. Data from which these curves are derived is given in Appendix Two.

5. REFERENCES


J.R. Rice, Problems to test parallel and vector languages, CSD-TR 516, Department of Computer Science, Purdue University, (May, 1985), 95 pages.
Problem 1
Integration by Trapezoidal rule

Number of evaluation points

<table>
<thead>
<tr>
<th>Number of Evaluation Points</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.8</td>
</tr>
<tr>
<td>5000</td>
<td>0.6</td>
</tr>
<tr>
<td>10000</td>
<td>0.4</td>
</tr>
<tr>
<td>25000</td>
<td>0.2</td>
</tr>
<tr>
<td>30000</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Single processor time (in ticks = 1/50 sec.)

Efficiency = \frac{(Single processor time)}{N \times (N processor time)}

Logarithmic plot of integration error as a function of single processor time (in ticks = 1/50 sec.)
Problem 2
E-STAR

Dimension of data array (assumed square)

Efficiency = \frac{(Single processor time)}{N \times (N processor time)}

Single processor time (in ticks = 1/50 sec.)

overflow in single precision computation of exponential prohibits larger problem size
Problem 3
Generalized E-STAR

Value of N and M (here, equal)

Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}
Problem 4
Sum of inverses

Number of vector elements

Efficiency = \frac{\text{(Single processor time)}}{N \times (N \text{ processor time})}

Single processor time (in ticks = 1/50 sec.)
Problem 5
Grader program

Number of students and tests (here, equal)

Efficiency = \frac{(Single processor time)}{N \times (N \text{ processor time})}

Single processor time (in ticks = 1150 sec.)
Problem 6
Tridiagonal solver

Matrix size

Efficiency

Single processor time (in ticks = 1/50 sec.)

Efficiency = \frac{\text{Single processor time}}{N \times (N \text{ processor time})}
Problem 7
Polynomial interpolation

Number of evaluation points

Single processor time (in ticks = 1/50 sec.)

Efficiency = \frac{(Single processor time)}{N \times (N \text{ processor time})}
Problem 8

Divided Difference Table

Number of evaluation nodes

<table>
<thead>
<tr>
<th>Number of Evaluation Nodes</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficiency</td>
<td>0.8</td>
<td>0.6</td>
<td>0.4</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Single processor time (in ticks = 1/50 sec.)

Efficiency = \frac{(Single processor time)}{N \times (N processor time)}
Problem 9
Image Refinement
10 passes

![Diagram showing grid size vs. efficiency](image)

Efficiency = \( \frac{\text{Single processor time}}{N \times \text{(N processor time)}} \)
Problem 10
Gaussian Elimination

Matrix size

Efficiency

Single processor time (in ticks = 1/50 sec.)

\[ \text{Efficiency} = \frac{\text{Single processor time}}{N \times (N \text{ processor time})} \]
Problem 11
Data Filtering

Vector size

Efficiency

Single processor time (in ticks = 1/50 sec.)

Efficiency = \frac{(Single processor time)}{N \times (N processor time)}
Problem 12  
Data Movement

Matrix Dimension  
(Here, assumed square.)

Efficiency = \frac{\text{Single processor time}}{N \times (N \text{ processor time})}
Problem 13
Vector Manipulation

Vector size

Efficiency

Single processor time (in ticks = 1/50 sec.)

Efficiency = \frac{\text{Single processor time}}{N \times (N \text{ processor time})}
Problem 14
Integration tests

Number of evaluation points

Efficiency = \frac{\text{Single processor time}}{N \times (N \text{ processor time})}
Problem 15
Interpolation

Number of interpolation points

![Graph showing efficiency vs. single processor time]

Efficiency = \frac{\text{(Single processor time)}}{N \times \text{ (N processor time)}}
Problem 16
Hilbert Matrix System

Matrix size

Efficiency = \frac{\text{(Single processor time)}}{N \times (N \text{ processor time})}

Efficiency vs. matrix size and single processor time (in ticks = 1/50 sec.)
Problem 1

Trapezoidal rule estimate of an integral

parameter ( kases = 8 )

Shared variables . available to all processors
shared real /reals/ num, h, a, b
shared integer /ints/ id(32), Nfunc, Nprocs, N

external adder
integer procid(32)
integer max(32)

integer Npts( kases )
integer clock(32,kases)
integer maxPEs

Dialog with user : Set up experiments by getting a set of problem
sizes and computer card numbers on which to carry out the experiments:

1 write(6,*) 'Maximum number of processors ? 0 to stop.'
read(5,*) maxPEs
if( maxPEs .le. 0 ) stop

do 10 i = 1,maxPEs
write(6,7) i
7 format( 'Computer #', i2, ' = ?' )
read(5,*) map(i)
10 continue

write(6,*) 'a = ?'
read(5,*) a
write(6,*) 'b = ?'
read(5,*) b

20 write(6,*) 'Number of cases to try = ?'
read(5,*) Nkases
if( Nkases .gt. kases ) then
write(6,*) 'Too many. Try again.'
go to 20
endif

do 30 i = 1, Nkases
write(6,27) i
27 format( 'How many evaluation points for case', i3, '?' )
read(5,*) Npts(i)
30 continue

Choice of three integrands:
write(6,*) 'Function number = ?'
read(5,*) Nfunc
Main experiment loop:
do 900 nm = 1, Ncases
   N = Npts(nm)
   h = ( b - a ) / float(N)
do 800 Nprocs = 1, maxPEs

call CFrtle(istart)
sum = 0.0

COBLOCK
   do 50 i = 1, Nprocs
      id(i) = i
      izmp = map(i)
      process( procid(i), adder(id(i)), itemp )
   continue
END COBLOCK

   sum = h * ( 0.5f(a,nfunc) + sum + 0.5f(b,nfunc) )
call CFrtc(isup)
clock(Nproces,nm) = isup - istart
write(6,*)'Sum = ',sum

800 continue

Print table giving performance results so far:
write(6,312) nfunc, ( Npts(n), i = 1, nm )
812 format( 'PROBLEM 1: TRAPEZOIDAL RULE ESTIMATE ',
1 'OF INTEGRAL Function number ', i3, ' ', ',', ';
2 'Number of -> --> Number of evaluation points -->',
3 '; processors'; 1068 )
write(6,813)
813 format( ' ', ' ')

831 format( i, 2f10.5 )
do 350 i = 1, maxPEs
write(6,357) i, ( clock(i,j), j = 1, nm )
837 format( ' ', i3, 4x, 7e10.5 )
write(6,847) ( float(clock(1,j))/float(i*clock(1,j)),j=1, nm)
847 format( ' ', 1', 108.5 )
write(6,848)
848 continue
850 continue

continue
go to 1
end
One copy of this subroutine is mapped to each processor designated in the experiment:

subroutine adder(myclid)

shared real /reals/ sum, h, a, b
shared integer /ints/ id(32), Nfunc, Nprocs, N

Local variable:
real temp, x

temp = 0.0
do 80 i = myid, N-1, Nprocs
   x = a + i * h
   temp = temp + f(x, Nfunc)
80 continue

Sum is accumulated locally, so only one common memory access needed.
Note the need to insure exclusive access to global sum.

call CFlLock(1, CFret, 1, sum)
sum = sum + temp
call CFlUnlock(1, CFret, 1, sum)

return
end

real function f(x,n)
if ( n .le. 0 ) then
   f = x**2
else
   if ( n .eq. 1 ) then
      f = sin(x)
   else
      f = exp(x)
endif
endif
return
end
Problems 2 and 3. Generalized E-STAR: the sum of the products of the elements in each row of a matrix. Strategy: divide the rows up as evenly as possible among the processors, each processor computing the product for an entire row of the matrix.

parameter ( kases = 8 )

shared real reals, sum
shared integer ints, id(32), Nfunc, Nprocs, N,M
external worker

A process id and computer assignment for each spawned process.
integer procid(32)
integer map(32)

integer Nsize( kases )
integer Msize( kases )
integer clock(32,kases)
integer maxPEs

=================================
User input portion of program -- Specify the experiments to run.

1 write(6,*) 'Maximum number of processors? 0 to stop.'
read(5,*) maxPEs
if( maxPEs .le. 0 ) stop

do 10 i = 1,maxPEs
   write(6,7) i
   format( 'Computer #', i2, '=' 99 )
   read(5,*) map(i)
10 continue

20 write(6,*) 'Number of cases to try = ?'
read(5,*) Nkases
if( Nkases .gt. kases ) then
   write(6,*) 'Too many. Try again.'
go to 20
endif

20 do 30 i = 1, Nkases
   write(6,27) i
   format( 'Enter N and M for case', i3, ':' )
   read(5,*) Nsize(i), Msize(i)
30 continue

write(6,*) 'Function number = ?'
read(5,*) Nfunc

=================================
Loop over all experiments, collecting timing data:

do 900 nn = 1, Nkases
    N = Nsize(nn)
    M = Msize(nn)

do 800 Nprocs = 1, maxPEs

    call CFrtle(istart)

      sum = 0.0

      COBLOCK
      do 50 i = 1, Nprocs
          id(i) = i
          istemp = map(i)
          process(procid(i), worker(id(i)), istemp)
      continue
      END COBLOCK

    call CFrtle(istop)
    clock(Nprocs, nn) = istop - istart
    write(6, *) 'Sum = ', sum

  800 continue

Print table of performance results so far:

write(6, 812) Nfunc(Nsize(i), i=1, nn)

     format('PROBLEMS 2 and 3, SUM OF PRODUCTS', 2)
     format('Function number', 13, ', ', /,
     format('# of \ N:', i6, 9(1x), '')

     write(6, 813) ( Msize(i), i = 1, M )

     write(6, 814)
     format(' |', 19, 4x, 1, ' |','

     do 850 i = 1, maxPEs
      write(6, 837) i, ( clock(i, j), j = 1, nn )
     format(' |', i3, 4x, 1, ' |','
      write(6, 847) (float(clock(1, j))/float(*clock(1, j)), j=1, nn)
     format(' |', 1, 10f8.5 ' |','
      write(6, 848)

     848 format(' |', i7)
     850 continue

  c

  900 continue
go to 1
end
subroutine worker(myid)

shared real /reals / sum
shared integer /ints/ id(32), Nfunc, Nprocs, N,M

real temp

temp = 0.0
do 80 i = myid, N, Nprocs
   prod = 1.0
   do 70 j = 1, M
      prod = prod*f(i,j,Nfunc)
   continue
   temp = temp + prod
80 continue

c c Accumulate sum locally, then do only one write to common memory.
WHEN construct insures exclusive access.

when( sum .eq. sum ) then
sum = sum + temp
end when

return
end

real function f(i,j,n)

if( n .le. 1 ) then
   f = 1.0 + exp(-1.0 * float(abs(i-j)) )
else
   f = 1.0001
endif

return
end
Problem 4 .. Sum of inverses of elements in a vector.

Strategy: Divide the elements up among the processors (as evenly as possible). Each processor will invert and sum his elements, and add this to the global sum.

parameter ( kases = 8 )

shared real / reals / sum, x(200000)
shared integer / nums / id(32), Nfunc, Nprocs, N
external adder

integer prcid(32)
integer map(32)

integer Ndim( kases )
integer clock(32,kases)
integer maxPEs

User input section .. set up experiments on various vector lengths and processor counts:

1 write(6,*) 'Maximum number of processors ? 0 to stop.'
read(5,*) maxPEs
if( maxPEs <= 0 ) stop

do 10 i = 1,maxPEs
   write(6,7) i
   format( 'Computer #', i2, ' = ?' )
   read(5,*) map(i)
10 continue

write(6,*) 'Number of cases to try = ?'
read(3,*) Nkases
if( Nkases > kases ) then
   write(6,*) 'Too many. Try again.'
   go to 20
endif

do 30 i = 1, Nkases
   write(6,27) i
   format( 'How many vector elements for case', i3, ' ?' )
   read(5,*) Ndim(i)
30 continue
LOOP over experiments:

do 900 nm = 1, Nkases
   N = Ndim(nm)

do 800 Nproc = 1, maxPEs

Reset vector:
do 40 i = 1, N
   \(x(i) = 1.0/(1.0 + \text{float}(i))\)
40 continue
   \(x(11) = 0.0\)

call CFRtlc(istart)
sum = 0.0
COBLOCK
   do 50 i = 1, Nproc
      id(i) = i
      itemp = map(i)
      process( procid(i), adder(id(i)), itemp )
50 continue
END COBLOCK
call CFRtlc(istop)
clock(Nprocs,nn) = istop - istart
if(clock(Nprocs,nn) le 0) clock(Nprocs,nn) = 1

write(6,*) 'Sum = ', sum

Print table of current performance results.

write(6,812) ( Ndim(i), i = 1, nn )
812 format( 'PROBLEM 4: MASKED SUMMATION:', 15, 2, 15, 2, 15, 2, 15, 2 )
   'Number of vector elements ---> ', 1, 2
   'procs ---> ', 10.8 )
write(6,813)
813 format( '----==============::-1-1 '- ' )
do 850 i = 1, maxPEs
   write(6,837) ( clock(i,j), j = 1, nn )
837 format( ' ', 1x, 15, 1x, 1d8.8 )
   write(6,847) ( float(clock(i,j)) / float(10.8), j = 1, nn )
847 format( 'v ', 15, 1d8.8 )
   write(6,848)
848 format( ' v ')
850 continue

Print table of current performance results.

write(6,881) ( Ndim(i), i = 1, nn )
881 format( 'PROBLEM 4: MASKED SUMMATION:', 15, 2, 15, 2, 15, 2 )
   'Number of vector elements ---> ', 1, 2
   'procs ---> ', 10.8 )
write(6,882)
882 format( '----==============::-1-1 '- ' )
do 885 i = 1, maxPEs
   write(6,887) ( clock(i,j), j = 1, nn )
887 format( ' ', 1x, 15, 1x, 1d8.8 )
   write(6,897) ( float(clock(i,j)) / float(10.8), j = 1, nn )
897 format( 'v ', 15, 1d8.8 )
   write(6,898)
898 format( ' v ')
885 continue

Print table of current performance results.

write(6,901) ( Ndim(i), i = 1, nn )
901 format( 'PROBLEM 4: MASKED SUMMATION:', 15, 2, 15, 2, 15, 2 )
   'Number of vector elements ---> ', 1, 2
   'procs ---> ', 10.8 )
write(6,902)
902 format( '----==============::-1-1 '- ' )
do 905 i = 1, maxPEs
   write(6,907) ( clock(i,j), j = 1, nn )
907 format( ' ', 1x, 15, 1x, 1d8.8 )
   write(6,917) ( float(clock(i,j)) / float(10.8), j = 1, nn )
917 format( 'v ', 15, 1d8.8 )
   write(6,918)
918 format( ' v ')
905 continue

Print table of current performance results.

write(6,921) ( Ndim(i), i = 1, nn )
921 format( 'PROBLEM 4: MASKED SUMMATION:', 15, 2, 15, 2, 15, 2 )
   'Number of vector elements ---> ', 1, 2
   'procs ---> ', 10.8 )
write(6,922)
922 format( '----==============::-1-1 '- ' )
do 925 i = 1, maxPEs
   write(6,927) ( clock(i,j), j = 1, nn )
927 format( ' ', 1x, 15, 1x, 1d8.8 )
   write(6,937) ( float(clock(i,j)) / float(10.8), j = 1, nn )
937 format( 'v ', 15, 1d8.8 )
   write(6,938)
938 format( ' v ')
925 continue

Print table of current performance results.

write(6,941) ( Ndim(i), i = 1, nn )
941 format( 'PROBLEM 4: MASKED SUMMATION:', 15, 2, 15, 2, 15, 2 )
   'Number of vector elements ---> ', 1, 2
   'procs ---> ', 10.8 )
write(6,942)
942 format( '----==============::-1-1 '- ' )
do 945 i = 1, maxPEs
   write(6,947) ( clock(i,j), j = 1, nn )
947 format( ' ', 1x, 15, 1x, 1d8.8 )
   write(6,957) ( float(clock(i,j)) / float(10.8), j = 1, nn )
957 format( 'v ', 15, 1d8.8 )
   write(6,958)
958 format( ' v ')
945 continue

Print table of current performance results.

write(6,961) ( Ndim(i), i = 1, nn )
961 format( 'PROBLEM 4: MASKED SUMMATION:', 15, 2, 15, 2, 15, 2 )
   'Number of vector elements ---> ', 1, 2
   'procs ---> ', 10.8 )
write(6,962)
962 format( '----==============::-1-1 '- ' )
do 965 i = 1, maxPEs
   write(6,967) ( clock(i,j), j = 1, nn )
967 format( ' ', 1x, 15, 1x, 1d8.8 )
   write(6,977) ( float(clock(i,j)) / float(10.8), j = 1, nn )
977 format( 'v ', 15, 1d8.8 )
   write(6,978)
978 format( ' v ')
965 continue

Print table of current performance results.
subroutine adder(myid)

shared real /reals/ sum, x(100000)
shared integer /ints/ id(32), Nfunc, Nprocs, N

real temp

temp = 0.0
do 80 i = myid, N, Nprocs
   if (x(i) .ne. 0.0 ) temp = temp + 1.0/x(i)
80 continue

call CFlock(ICFret,1,'sum')
sum = sum + temp
call CFulck(ICFret,1,'sum')

return
end

real function f(x,n)

if( n .le. 0 ) then
   f = x**2
else
   if ( n .eq. 1 ) then
      f = sin(x)
   else
      f = exp(x)
   endif
endif

return
end
Problem 5. Compute statistics on a table of grades

Problem definition phase: Consult with user to set up experiments.

How many cases? 0 to stop

If (kases > isizes) then
   TOO MANY. Must be <= isizes
   go to 12
endif

If (kases <= 0 ) stop

do 15 i = 1, kases
   write(6,14) i
   format(9. Enter # tests and # students for case',3,
         ' Must be <= 300.' )
   read(5,*) itests(i), iswds(i)
   if ( (iswds(i) > 300) .or. (itest(i) > 300) ) then
      TOO LARGE. Try again.
   endif
continue

Maximum number of processors for this experiment:
read(5,*) nowPEs

do 20 i = 1, nowPEs
   write(6,17) i
   format(9. Enter computer # for process',3)
   read(5,*) map(i)
continue

====================================================================
Loop over experiments:

```c
do 1000 m = 1, kases
   Ntests = iTests(nn)
   Nsuids = iSuids(m)
```

do 800 Nprocs = 1, nowPEs

**INITIALIZE SCORE TABLE:**

do 40 i = 1, Ntests
   do 30 j = 1, Nsuids
      score(i,j) = 100.0 * sin( float(i+j) )
   30 continue
   40 continue

First parallel phase: compute average of each row:

call CFrtic(t1)
COBLOCK
do 50 i = 1, Nprocs
   id(i) = i
   j = map(i)
   process( P(i), avcalc(id(i)), j)
50 continue
END COBLOCK

Second parallel phase:
Distribute computations for each student among processors:

genius = .false.
COBLOCK
do 100 i = 1, Nprocs
   id(i) = i
   j = map(i)
   process( P(i), staus(id(i)), j )
100 continue
END COBLOCK

call CFrtic(2)
clock(Nprocs,nn) = t2 - t1
if( clock(Nprocs,nn) < 0 ) clock(Nprocs,nn) = 1

Print table of results:
— Code deleted for brevity —
subroutine avealn( istart )

shared real /r/ score(300,300), avg(300), top(300), lowabv(300)
shared integer /i1/ id(32), Nprocs, nabove(300)
shared integer /i2/ Ntests, Nsmds
shared logical /logvar/ genius

real temp
do 250 i = istart, Ntests, Nprocs
   temp = 0.0
   do 200 j = 1, Nsmds
      temp = temp + score(i,j)
      continue
   temp = temp / float(Nsmds)
   avg(i) = temp
250 continue
return
end

subroutine stats(jstart)

shared real /r/ score(300,300), avg(300), top(300), lowabv(300)
shared integer /i1/ id(32), Nprocs, nabove(300)
shared integer /i2/ Ntests, Nsmds
shared logical /logvar/ genius

real temp, ltemp, stemp
integer ntemp
logical anygen, allabv

anygen = .false.
do 350 j = jstart, Nsmds, Nprocs
   ntemp = 0
   temp = 0.0
   ltemp = 50000.0
   allabv = .true.
   do 300 i = 1, Ntests
      stemp = score(i,j)
      if ( stemp .gt. temp ) temp = stemp
      if ( stemp .gt. avg(i) ) then
         if ( stemp .lt. ltemp ) ltemp = stemp
         ntemp = ntemp + 1
         score(i,j) = 1.1*stemp
      else
         allabv = .false.
      endif
   300 continue
   anygen = anygen .or. allabv
   top(j) = ntemp
   lowabv(j) = ltemp
   nabove(j) = ntemp
350 continue
   genius = genius .or. anygen
return
end
Problem 6

Tridiagonal solver using iterative method of Jordan.

parameter ( cases = 8 )

shared real /reals1 / d((10000,2)), x((10000,2)), limit
shared real /reals2 / d((10000,2)), y((10000,2))

shared integer /ints / id(32), Nprocs, N, isync
external worker

integer procid(32)
integer map(32)

integer Nsize( cases )
integer clock(32, cases)
integer maxPES

==================================================================
User dialog phase. Get computer numbers and problem sizes for the various cases.
-- Code deleted for brevity --
==================================================================

LOOP OVER EXPERIMENTS:
do 900 i = 1, Ncases
   N = Nsize(i)
   limit = 1.44269504 * alog( float( N ) ) + 1

mination vectors:
do 40 i = 1, N
   u(i,1) = 1.0
   v(i,1) = 1.0
   y(i,1) = 15.0
   y(1,1) = 10.0
   y(N,1) = 10.0
call CFrtc(idstart)

COBLOCK
   isync = 0
   do 50 i = 1, Nprocs
      id(i) = i
      itemp = map(i)
      process( procid(i), worker(id(i)), itemp )
   continue
END COBLOCK

call CFrtc(idstop)
clock(Nprocs, N) = idstop - idstart

IF ( ( Nprocs .eq. 1 ) .and. ( N .le. 50 ) ) then
do 80 i = 1, N
   write(6,*) 'x(i,1) = ', x(i), y(i,1), d(i,1)
continue
endif

80 continue
800 continue

==================================================================
Print table of performance results. Code omitted.
==================================================================
900 continue
go to 1
end
Each computer executes a copy of this routine. Note that since this is an iterative method, synchronization is necessary to ensure that all processors are on the same iteration. Global integer values serve as synchronization semaphores.

subroutine worker(myid)

shared real /reals1/ u(10000,2),l(10000,2), limit
shared real /reals2/ d(10000,2),x(10000),y(10000,2)
shared integer /ints/ id(32),Nprocs,N,isynch

TOP OF ITERATION LOOP:
continue

usual update is performed on a subset of vector elements:
do 50 i = myid,N,Nprocs    
if( d(i,1) > 0.0 ) then
  l(i,2) = l(i,1)/d(i,1)
  u(i,2) = u(i,1)/d(i,1)
  y(i,2) = y(i,1)/d(i,1)
endif
50 continue

SYNCHRONIZATION: Wait here for fellow processors.
call CFWlock(ICFret,1, isynch)
isynch = isynch + 1
call CFWlock(ICFret,1, isynch)
when(isynch .ge. (2*mycnr-1)*Nprocs ) continue

SYNCHRONIZATION: Again wait for fellows.
call CFWlock(ICFret,1, isynch)
isynch = isynch + 1
call CFWlock(ICFret,1, isynch)
when(isynch .ge. 2*Nprocs*mycnr ) continue

mycnr = mycnr + 1
k = 2*k
if( mycnr .gt. limit ) then
do 150 i = myid,N,Nprocs
  x(i) = y(i,1)/d(i,1)
150 continue
return
endif

continue go to 10
end
Problem 7. Compute polynomial interpolation values at 5 points using the Lagrange interpolation formulas.

```
parameter ( kases = 5 )

shared real / reals / sum(5), x(5), node(10000)
shared integer / ints / id(32), Nprocs, N
external adder

integer procid(32)
integer map(32)

integer Ndim( cases )
integer clock(32,kases)
integer maxPEs

================================================================================================

Query user to establish experimental parameters. Set up x(1 .. 5). Code omitted.

================================================================================================

LOOP OVER EXPERIMENTS:
do 900 nm = 1, Nkases
    N = Ndim(nm)
do 40 i = 1, N
    node(i) = .08 * i
    continue

do 800 Nprocs = 1, maxPEs
    call CFrtic(istart)
do 45 i = 1,5
    sum(i) = 0.0
    continue

COBLOCK
do 50 i = 1, Nprocs
    id(i) = i
    itemp = map(i)
    process( procid(i), adder(id(i)), itemp )
    continue
END COBLOCK

call CFrtic(istop)
clock(Nprocs,nm) = istop - istart
if(clock(Nprocs,nm) .le. 0 ) clock(Nprocs,nm) = 1

800 continue

write(6,55) maxPEs, Ndim(nm)
55 format(1,'-------- Answer computed by ', i3, ' procs for ', i5, ' nodes.' )
do 60 i = 1, 5
    write(5,58) i,sum(i), f(x(i)), abs( (f(x(i))-sum(i))/f(x(i)) )
58 format( 'sum(',i2,') = ',e12.5, ' actual = ',e12.5,
      ' rel err = ',e12.5)
60 continue

================================================================================================

Print out performance results for the experiments done so far. Code deleted for brevity

================================================================================================

900 continue
go to 1
end
```

Mar 21 05 1987
Each computer executes a copy of this. The job is to compute an equal number of the terms in the Lagrange sum, and add these into a global variable.

```c
subroutine adder(myid)
  
  shared real /reals/ sum(5), x(5), node(10000)
  shared integer /ints/ id(32), Nprocs, N
  
  LOOP over the 5 summations.
  do 300 k = 1,5
      temp = 0.0
      
      do 200 i = myid,N,Nprocs
          prod = 1.0
          do 100 j = 1,N
              if (j .ne. i) prod = prod*(x(k)-node(j))/(node(i)-node(j))
          continue
          temp = temp + f(node(i)) * prod
      continue
      
      Write result into shared memory, using WHEN for exclusive access
      call CFlock(FCFSr(1), 'sum')
      sum(k) = sum(k) + temp
      call CFunlock(FCFSr(1), 'sum')
  
  300 continue
  return
end
```

real function f(x)
  f = exp(x)
  return
end
Problem 8. Construct divided difference table.

parameter ( kases = 8 )

shared real / reals/ x(300),d(300,300)
shared integer / ints/ id(32), Nprocs, N, mypos(32)
external worker

integer proid(32)
integer map(32)

integer Ndum( kases )
integer clock(32,kases)
integer maxPEs

LOOP OVER EXPERIMENTS:
do 900 n = 1, Nkases
N = Ndum(n)

do 40 i = 1N
x(i) = 2 * i + .01 * cos(float(i))
d(i,1) = f(x(i))
40 continue

do 800 Nprocs = 1, maxPEs

do 45 i = 1, Nprocs
mypos(i) = 0
45 continue
mypos(Nprocs) = N

call CFrstic(isr.op)

COBLOCK
do 50 i = 1, Nprocs
id(i) = i
itemp = map(i)
process( proid(i), worker(id(i)), itemp )
50 continue
END COBLOCK

call CFrstic(isr.op)
clock(Nprocs,n) = istop - start
if(clock(Nprocs,n) .le. 0 ) clock(Nprocs,n) = 1

continue

if( N .le. 10) then

do 80 i = 1N
write(6,77) x(i),(d(l,k),k=1,N-1-k+1)
77 format(11f13)
80 continue
endif

Display performance results which have been collected so far. Code omitted.
Columns are divided among processors, each processor constructing the whole column. Each processor is synchronized with the processor working to his left, working one row higher (because of the data dependencies). Synchronization overhead is extremely high. Each entry in the table is associated with a number, specifically, \((i-1)N + i\) for the \((i,j)\) entry. Each processor records the number of the entry he most recently computed in an array in shared memory called "mypos", so that processors to his right may synchronize on him.

```
subroutine worker(myid)
  shared real / reals / x(300), d(300,300) 
  shared integer / units / id(32), Nprocs, N, mypos(32) 

  This is the proc. id. of the processor operating to my immediate left. 
  integer master 
  master = Nprocs - mod(Nprocs-myid+1, Nprocs) 

  do 200 k = myid+1,N,Nprocs 
      when(mypos(master) .ge. (k-2)*N + 2) continue 
      d(1,k) = ( d(2,k-1)-d(1,k-1) ) / (x(1+k)-x(1)) 
      mypos(myid) = N*(k-1) + 1 
      do 100 i = 2,N-k+1 
          when(mypos(master) .ge. mypos(myid)-N+1) continue 
          d(i,k) = ( d(i+1,k-1)-d(i,k-1) ) / (x(i+k)-x(i)) 
          mypos(myid) = mypos(myid) + 1 
      100 continue 
  200 continue 
  return 
end

real function f(x) 
  f = sin(x) 
  return 
end
```
Problem 9
Image smoothing on an N by M grid. Mild synchronization.

parameter ( cases = 8 )

shared real /reals/ (300,300.2)
shared integer /ints/ id(32), Nprocs, N, i.synch, K
external worker

integer procid(32)
integer map(32)

integer Ndim( cases )
integer reps( cases )
integer clock(32,kases)
integer maxPEs

User dialog phase. Read in experimental parameters. Code omitted

LOOP OVER EXPERIMENTS:
do 900 nn = 1, Ncases
   N = Ndim(nn)
   K = reps(nn)
   do 800 Nprocs = 1, maxPEs
      do 45 i = 1, N
         do 40 j = 1, N
            a(i,j,1) = 125.0 * sin(float(i+j))
         40 continue
      45 continue
      i.synch = 0
      call CFRule(istart)
      COBLOCK
      do 50 i = 1, Nprocs
         id(i) = i
         itemp = map(i)
         process( procid(i), worker(id(i)), itemp )
      50 continue
      END COBLOCK
      call CFRule(i.stop)
      clock(Nprocs,nn) = i.stop - istart
      if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1
   800 continue
   if( N .le. 10) then
      do 80 i = 1, N
         itog = i + mod(K,2)
         write(6,77) (a(i,j,itog),j=1,N)
      77 format(10f8.3)
   80 continue
   endif
   Print performance results so far. Code omitted.

900 continue
   go to 1
end
Rows are divided among the processors. Each processor handles an entire row. Synchronization is necessary at the end of each pass over the grid to insure all processors are working on the same pass.

subroutine worker(myid)

shared real *a(300,300,2)
shared integer *id(32), Nprocs, N, isynch, K

iold = 1
do 500 mycnt = 1,K
    iold = itog
    itog = itog + mod(itog, 2)
    do 300 i = myid+1,N-1,Nprocs
    do 200 j = 2,N-1
        a(i,j,iold) = ( a(i-1,j,iold) + a(i+1,j,iold) + a(i,j-1,iold) + a(i,j+1,iold) )/4.0
    300 continue
    continue
200 continue
300 continue

Notify semaphore that I'm done.
call CFlock(1,'isynch')
isynch = isynch + 1
call CFunlock(1,'isynch')

500 continue

return
end
Problem 10. LU factorization by Gaussian elimination with partial pivoting.

parameter ( kases = 8 )

shared real /reals/ a(300,300), pivot
shared integer /ints/ id(32), Nprocs, N, isynch, ipvt, row(300)

external worker

integer procid(32)
integer map(32)
integer Ndim(kases)
integer clock(32,kases)
integer maxPEs

LOOP OVER EXPERIMENTS:
do 900 nn = 1, Nkases
   N = Ndim(nn)
do 800 Nprocs = 1, maxPEs
   do 45 i = 1, N
      row(i) = i
   do 40 j = 1, N
      a(i,j) = 125.0 * sin(float(i+j))
   continue
   continue
Go ahead and (sequentially) find the first pivot.
call CFRtic(start)
ipvt = 1
pivot = 0.0
   do 47 i = 1, N
      if ( abs(a(i,1)) .gt. pivot ) then
         pivot = abs(a(i,1))
ipvt = i
      endif
   continue

isynch = Nprocs -1
GOBLOCK
   do 50 i = 1, Nprocs
      id(i) = i
      itemp = map(i)
      process( procid(i), worker(id(i)), itemp )
   continue
END GOBLOCK

call CFRtic(stop)
clock(Nprocs,nn) = stop - start
if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

DISPLAY current performance results, and factored matrix if small. Code omitted.
subroutine worker(myid)

shared real / a(300,300), pivot 
shared integer / ids(32), Nprocs, N, isynch, ipvt, row(300)

LOOP OVER SUBMATRIX STEPS: At the beginning of each such step, it is assumed
that the pivot element was determined on the previous step, as that submatrix
was computed. (This explains the initialization in the main program).

do 500 k = 1,N
  Process 1 actually makes the swap.
  IF ( myid .eq. 1 ) then
    Wait until all proc are done with last submatrix step.
    WHEN ( isynch .ge. k*Nprocs - 1 ) continue
  endif

  Signal semaphore and wait here for fellow processors.
  call CFluck(ICFlast,1,'isynch')
  isynch = isynch + 1
  call CFluck(ICFlast,1,'isynch')
  WHEN ( isynch .ge. k*Nprocs ) continue

  Rows of submatrix are distributed among processors. Compute next pivot
  by looking at the entries in the (k+1)st column as they are computed.
  do 400 i = k+myid, N, Nprocs
  a(row(i),k) = a(row(i),k) / a(row(k),k)
  do 300 j = k+1,N
    a(row(i),j) = a(row(i),j) - a(row(i),k)*a(row(k),k)
  continue
  call CFluck(ICFlast,1,'pivot')
  IF ( abs(a(row(i),k+1)) .gt. pivot ) then
    pivot = abs(a(row(i),k+1))
    ipvt = i
  endif
  call CFluck(ICFlast,1,'pivot')

  400 continue
  500 continue

return
end
Problem 11. Filter data in a vector and write to a database.

**parameter** ( kases = 8 )

shared real /reals/ data(100000), sum(4)
shared integer /ints/ N, id(32), Nprocs
external worker

Integer procid(32)
Integer map(32)

Integer Ndim( kases )
Integer clock(32,kases)
Integer maxPEs

---

Query user for experiment sizes, computer numbers, etc. Code omitted.

---

do 900 Im = 1, Nkases
   N = Ndim(Im)
do 800 Nprocs = 1, maxPEs

d0 40 i = 1, N
   data(i) = float(i)/1 + 1080.0 * sin(float(i+10))
   continue

call CFrtlc(istart)
d0 45 i = 1,4
   sum(i) = 0.0
   continue

COBLOCK
   do 50 i = 1, Nprocs
      id(i) = i
      itemp = map(i)
      process( procid(i), worker(id(i)), itemp )
   continue
END COBLOCK

c
Save data in database within timer scope:
do 50 i = 1,4
   write(*,11) sum(i)
   continue

call CFrtlc(istop)
clock(Nprocs,Im) = istop - istart
if(clock(Nprocs,Im) .le. 0 ) clock(Nprocs,Im) = 1

continue

c
Here, print table of performance results

---

continue
go to 1
end
Perform filter on select vector elements, accumulating sums, and then add this to the shared sums.

subroutine worker(ident)
    shared real /reals / data(100000), sum(4)
    shared integer /msg / N, id(32), Nprocs

    real locsum(4), pi, temp
    data pi / 3.141592654 /

    do 50 i = 1, 4
        locsum(i) = 0.0
    50 continue

    do 100 i = ident, N, Nprocs
        temp = amax1( 0.0, amin1( 1000.0, data(i) ) )
        data(i) = temp
        locsum(1) = locsum(1) + temp
        do 75 k = 2, 4
            locsum(k) = locsum(k) + cos( float(pi*k*i) / float(N+1))*temp
        75 continue
    100 continue

Note that in this block, access to SUM entries will be exclusive.

call CFlock(FCFrst,1,'sum')
    sum(1) = sum(1) + locsum(1)/N
    sum(2) = sum(2) + locsum(2)
    sum(3) = sum(3) + locsum(3)
    sum(4) = sum(4) + locsum(4)
call CFunlock(FCFrst,1,'sum')

return
end
Problem 12. Simple data movement. Move data from smaller array to form a composite larger one.

Parameter (kases = 3)

shared real r1(300,300), c(300), r(300)
shared real r2(acom, abig(300,300))
shared integer finis, id(32), Nprocs, N, M
external worker

integer procid(32)
integer map(32)

integer Nsize(kases)
integer Msize(kases)
integer clock(32,kases)
integer maxPEs

==================================================
Query user for experiment sizes, computer numbers, etc. Code omitted.
==================================================

do 900 Nkases = 1, Nkases
    N = Nsize(Nkases)
    M = Msize(Nkases)
    do 40 i = 1, N
        c(i) = float(i+1)
    do 35 j = 1, M
        a(i,j) = float(i+j)
    continue
    do 45 j = 1, M
        r(j) = float(1-j)
    continue
    acom = .5

do 800 Nprocs = 1, maxPEs
    call CFrdc(istart)
    COBLOCK
    do 50 i = 1, Nprocs
        id(i) = i
        itemp = map(i)
        process( procid(i), worker(id(i)), itemp )
    continue
    END COBLOCK
    abig(N+1,M+1) = acom
    call CFrdc(istop)
    clock(Nprocs,m) = istop - istart
    800 continue

Print table of performance results

==================================================
900 continue
go to 1
end
Move entries into select rows of the big matrix.

subroutine worker(myid)

shared real h1 / c(300,300), c(300), r(300)

shared real h2 / acom, abig(300,300)

shared integer /inst/ id(32), Nprocs, N,M

  do 100 i = myid,N,Nprocs
      abig(i,M+1) = c(i)
  do 50 j = 1,M
      abig(i,j) = a(i,j)
  50 continue

  100 continue

  do 200 j = myid,M,Nprocs
      abig(N+1,j) = r(j)
  200 continue

return

end
parameter ( kases = 8 )

shared real /reals / sum, a(50000), b(50000), c(50000), d(50000)
shared integer /ints / id(32), Nprocs, N
external adder

integer procid(32)
integer mp(32)

integer Ndim(kases)
integer clock(32, kases)
integer maxPEs

Here, read in experiment sizes, computer numbers, etc. from user. Code omitted.

do 900 nm = 1, Nkases
   N = Ndim(nm)
do 800 Nprocs = 1, maxPEs

do 40 i = 1, N
    a(i) = i/float(10) + 1.0/float(i)
    b(i) = alog(a(i)) + .02
    c(i) = (a(i) + b(i)) * sin(a(i))
    d(i) = a(i) + b(i) - 2.0*c(i)
40 continue

call CFrtic(istart)
sum = 0.0
COBLOCK
   do 50 i = 1, Nprocs
      id(i) = i
      itemp = map(i)
      process( procid(i), adder(id(i)), itemp )
50 continue
END COBLOCK
call CFrtic(iSTOP)
clock(Nprocs, nm) = istop - istart
if(clock(Nprocs, nm) .le. 0 ) clock(Nprocs, nm) = 1
write(6,*) 'Sum = ', sum
800 continue

Here, write out intermediate performance results.

c continue
900 continue
go to 1
end
subroutine adder(myid)

shared real /reals/ sum, a(50000), b(50000), c(50000), d(50000)
shared integer /ints/ id(32), Nprocs, N

real temp

temp = 0.0

do 80 i = myid, N, Nprocs

   s(i) = a(i)**2 * sin(b(i))
   if (sin(a(i)) .le. cos(c(i))) then
      s(i) = s(i) + c(i)
   else
      s(i) = s(i) - d(i)
   endif

   temp = temp + s(i)**2

80 continue

CRITICAL REGION: Exclusively read and update SUM

    call CFlock(IClock, 'sum')
    sum = sum + temp
    call CFlock(IClock, 'sum')

return
end

real function f(x,n)

if (n .le. 0) then
   f = x**2
else
   if (n .eq. 1) then
      f = sin(x)
   else
      f = exp(x)
   endif
endif

return
end

Mar 3 21:20 1987

parameter ( kases = 8 )

shared real preact1 / sum(3,3), a, b
shared real preact2 / hgauss, offset
shared Integer pInt / id(32), Nfunc, Nprocs, N, Ngauss

external adder
Integer procid(32)
Integer map(32)
real exact(3)
Integer Ndim( kases )
Integer clock(32,kases)
Integer maxPES

Here, read in parameters for experiments from user. This includes the
accuracies (#of evaluations), exact solution values with which a
relative error may be computed, the limits of integration a and b,
and the computer numbers. CODE OMITTED.

LOOP OVER EXPERIMENTS:
do 900 nm = 1, Nkases
    N = Ndim( nm )
    h = (b-a)/float(N)
    Ngauss = N/3
    hgauss = (b-a)/float(Ngauss)
    offset = .774596669241 * hgauss / 2.0

do 800 Nprocs = 1, maxPESs
call CFrtdc(istart)
do 45 i = 1, 3
do 40 j = 1, 3
    sum(i,j) = 0.0
40 continue
45 continue
COBLOCK
do 50 i = 1, Nprocs
    id(i) = i
    itemp = map(i)
    process( procid(i), adder(id(i)), itemp )
50 continue
END COBLOCK

ADJUST SUMS:
do 70 j = 1,3
    sum(1,j) = h*(sum(1,j) + f(a,j)/2.0 + f(b,j)/2.0 )
    sum(2,j) = h*(sum(3,j) + f(a,j) + f(b,j) )/3.0
    sum(3,j) = hgauss * sum(3,j)/ 18.0
70 continue

call CFrtdc(istop)
clock(Nprocs,nn) = istop - istart
if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

Print table of performance results. Code omitted.

Page 1 of Problem14
subroutine adder(myid)

shared real /reals1/ sum(3,3), a, b
shared real /reals2/ h, hgauss, offset
shared integer /ints/ id(32), Nfunc, Nprocs, N, Ngauss

do 1000 Nfunc = 1,3

Compute temporary sum for trapezoidal rule.

        temp1 = 0.0
        do 100 i = myid,N-1,Nprocs
            x = i*h
            temp1 = temp1 + f(x,Nfunc)
        enddo 100

Compute sum for Simpson's Rule.

        temp2 = 0.0
        do 200 i = myid,N-1,Nprocs
            x = i*h
            if( mod(i,2).ge. 1 ) then
                temp2 = temp2 + 4.0*f(x,Nfunc)
            else
                temp2 = temp2 + 2.0*f(x,Nfunc)
            endif
        enddo 200

Gaussian quadrature.

        temp3 = 0.0
        do 300 i = myid,Ngauss,Nprocs
            x = ( float(i) = .5 ) * hgauss
            temp3 = temp3 + 5.0*f(x-offset,Nfunc)
            + 8.0*f(x,Nfunc) + 5.0*f(x+offset,Nfunc)
        enddo 300

Access shared memory sums, using system locks.

        call CFlock(1,CFret,1,'sum')
        sum(1,Nfunc) = sum(1,Nfunc) + temp1
        sum(2,Nfunc) = sum(2,Nfunc) + temp2
        sum(3,Nfunc) = sum(3,Nfunc) + temp3
        call CFunlock(1,CFret,1,'sum')

1000 continue
return
end

real function f(x,n)
if( n.le. 1 ) then
    f = exp(x)
else
    if( n.eq. 2 ) then
        f = sqrt( abs(x-.2345) )
    else
        f = 1.0 + x**2 + 1.0/( 1.0 + 100*x**2)
    endif
endif
return
end
parameter ( kases = 8 )
shared real / x,b,sum,x(0:50001,2),x(10),approx(10,2)
shared integer / sinu, id(32), Nprocs, N, K, isynch
external adder

integer procid(32)
integer map(32)
integer Ndim( kases )
integer clock(32,kases)
integer maxPEs

LOOP OVER EXPERIMENTS:
do 900 nm = 1, Nkases
   N = Ndim(nm)
do 800 Nprocs = 1, maxPEs
   call CFrtle(mstart)
do 40 j = 1,2
      x(0,j) = a - 1
      x(N+1,j) = b + 1
   do 38 i = 1,K
      approx(i,j) = 0.0
   continue
40
continue

isynch = 0
COBLOCK
   do 50 i = 1, Nprocs
      id(i) = i
      itemp = map(i)
      process( procid(i), adder(id(i)), itemp )
   continue
50
END COBLOCK

do 60 i = 1,K
   do 58 j = 1,2
      approx(i,j) = abs( ((f(i)) - approx(i,j))/f(i) )
   continue
60
continue

call CFrtle(mstop)
clock(Nprocs,m) = mstop - mstart
if(clock(Nprocs,m) le. 0) clock(Nprocs,m) = 1

continue
write(6,805)
format(' t point rel error rel error',
     1 ' uniform chebyshev')
do 810 i = 1, K
   write(6,808) t(i),approx(i,1),approx(i,2)
808
continue
810
continue
Here, print performance results. CODE OMITTED.

900 continue
go to 1
end
subroutine add(myid)

shared real /real/ a,b,sum,x(0:50001,2),t(10),approx(10,2)
shared integer /int/ id(32), Nprocs, N, K, isynch

real h,pi
data pi /3.141592654 /

h = (b-a)/float(n-1)

do 100 i = myid,N,Nprocs
  x(i,1) = a + (i-1)*h
  x(i,2) = a + (a-b)*cos(float((2*i-1)*pi/(2*N)) )
100 continue

SYNCHRONIZATION... Wait on fellows

call CFlock(ICFret,l,"synch")
isynch = isynch + 1

when( isynch .ge. Nprocs ) continue

do 500 kk = 1,K
  temp1 = 0.0
  temp2 = 0.0

do 400 i = myid,N,Nprocs
  temp1 = temp1 + f(x(i,1)) * herm1(t(kk),i,1) +
         fprime(x(i,1)) * herm1(t(kk),i,1)
  temp2 = temp2 + f(x(i,2)) * herm1(t(kk),i,2) +
         fprime(x(i,2)) * herm1(t(kk),i,2)
400 continue

call CFlock(ICFret,l,"approx")
  approx(kk,1) = approx(kk,1) + temp1
  approx(kk,2) = approx(kk,2) + temp2

call CFlock(ICFret,l,"approx")

500 continue
return
end
real function hermc(tpt,i,iflag)
shared real /reals/ a,b,sum,x(0:50001,2),t(10),approx(10,2)
shared integer /ints/ id(32), Nprocs, N, K, isynch

if( (tpt .le. x(i-1,iflag)) .or. (tpt .ge. x(i+1,iflag)) ) then
hermc = 0.0
return
endif
if( tpt .gt. x(i,iflag) ) then
  dt = x(i+1,iflag) - x(i,iflag)
  dx = x(i+1,iflag) - tpt
else
  dt = x(i,iflag) - x(i+1,iflag)
  dx = tpt - x(i,iflag)
endif
hermc = ( 3.0 - 2.0 *dx/dt ) * dx**2/dt**2
return
end

real function hermc1(tpt,i,iflag)
shared real /reals/ a,b,sum,x(0:50001,2),t(10),approx(10,2)
shared integer /ints/ id(32), Nprocs, N, K, isynch

if( (tpt .le. x(i-1,iflag)) .or. (tpt .ge. x(i+1,iflag)) ) then
  hermc1 = 0.0
  return
endif
dx = tpt - x(i,iflag)
if( tpt .gt. x(i,iflag) ) then
  dx2 = ( x(i,iflag) - x(i+1,iflag) )**2
  dx2 = ( tpt - x(i+1,iflag) )**2
else
  dx2 = ( x(i,iflag) - x(i-1,iflag) )**2
  dx2 = ( tpt - x(i-1,iflag) )**2
endif
hermc1 = dx2*dx/dt2
return
end

real function f(x)
f = x**2 + 25.0
return
end

real function fprime(x)
fprime = 2.0 * x
return
end
Problem 16. Factor and backsolve using a Hilbert matrix.

parameter ( kases = 8 )

shared real k1 / x(100,100),hilb(100,100),norm(4)
shared real k2 / x(100,4), x(100,4), pivot, resid(100,4)
shared integer hint / id(32), Nprocs, N, isynch,pivtrow(100)
shared logical /log / done(100,2)
external factor, solve

integer procid(32)
integer map(32)
integer Ndim( kases )
integer clock(32,kases)
integer maxPEs

Here, query user for problem sizes, computer numbers, etc.

--------------------------------------------------------------

do 35 i = 1,100
   b(i,1) = 0.0
   b(i,2) = 1.0
   b(i,3) = 1.0 + .01 * sin( float( 100*i ) )
   b(i,4) = 0.0
35 continue

do 34 j = 1,100
   hilb(i,j) = 1.0/float(i+j-1)
   b(i,4) = b(i,4) + hilb(i,j)
34 continue

LOOP OVER EXPERIMENTS:

do 900 jn = 1, Nkases
   N = Ndim(jn)
do 800 Nprocs = 1, maxPEs

do 45 i = 1,N
   row(i) = i
   done(i,1) = .false.
   done(i,2) = .false.
do 39 j = 1,4
   x(i,j) = b(i,j)
   resid(i,j) = b(i,j)
39 continue

40 continue

45 continue

call CFortc(istart)
isynch = Nprocs -1
ipvt = 1
pivot = 0.0

do 47 i = 1,N
   if( abs(a(i,1)) .gt. pivot ) then
      pivot = abs(a(i,1))
      ipvt = i
   endif
47 continue
In this COBLOK, the matrix "a" is factored in parallel.
C
COBLOK
DE 50 I = 1, Nproces
   id(i) = i
   itemp = map(i)
   process( procid(i), factor(id(i)), itemp )
50  continue
END COBLOK

In this COBLOK, backsolving and residual computation is done in parallel.
C
COBLOK
DE 60 I = 1, Nproces
   id(i) = i
   itemp = map(i)
   process( procid(i), solve(id(i)), itemp )
60  continue
END COBLOK

call DFric(idmp)
clock(Nproces,m) = istop - istart
IF(clock(Nproces,m) le 0 ) clock(Nproces,m) = 1
800  continue

Here, print table of performance results so far, and print solution vector.

900  continue
   go to 1
end

subroutine factor(myid)
shared real h1 : a(100,100),hib(100,100).momm(4)
shared real h2 : b(100,4), x(100,4), pivot, resid(100,4)
shared integer init : id(32), Nproces, N, isynch,pvrt,rsd(100)
shared logical /log : id(32), done(100,2)

SEE CODE FOR ROUTINE "WORKER" OF PROBLEM 10

Mar 4 10:24 1987
subroutine solve(myid)
  shared real a(100,100), b(100,4), x(100,4), pivot, resid(100,4)
  shared integer id(32), Nprocs, N, isynch, ipvt, row(100)
  shared logical /log/ done(100,2)
  real temp(4)

  ! FORWARD SUBSTITUTION
  do 500 i = myid,N,Nprocs
    do 400 j = 1,i-1
      when( done(j,1) ) continue
      do 40 k = 1,4
        x(row(j),k) = x(row(j),k) - a(row(j),i)*x(row(i),k)
      end
    end
    400 continue
    done(i,1) = .true.
  500 continue

  ! BACK SUBSTITUTION
  do 1000 i = myid,N,Nprocs
    i = N+1-i
    do 900 jj = 0, i-1
      j = N - jj
      when( done(j,2) ) continue
      do 800 k = 1,4
        x(row(i),k) = x(row(i),k)/a(row(i),i)
      end
    end
    900 continue
    done(i,2) = .true.
    do 950 jj = 0, i-1
      j = N - jj
      when( done(j,2) ) continue
      do 800 k = 1,4
        resid(row(j),k) = resid(row(j),k) - a(row(j),i)*x(row(i),k)
      end
    end
    950 continue
  1000 continue

  ! Use semaphore ISYNCH to signal fellow processors that your work is done, so far.
  call CFlock(ICFret,1,'isynch')
  isynch = isynch + 1
  call CFlock(ICFret,1,'isynch')

  do 1100 k = 1,4
    temp(k) = 0.0
  1100 continue

  ! Wait on fellows to get done with their work and then compute residual:
  when ( isynch .ge. Nprocs ) continue
  do 1500 i = myid,N,Nprocs
    do 1400 k = 1,4
      temp(k) = temp(k) + resid(i,k)**2
    end
  1400 continue
  1500 continue

  call CFlock(ICFret,1,'morm')
  do 2000 k = 1,4
    morm(k) = morm(k) + temp(k)
  2000 continue
  call CFlock(ICFret,1,'morm')

  return
end
APPENDIX TWO: MEASURED PERFORMANCE DATA

We present in tabular form the data sets plotted in Section 4. The entries are clock ticks (= to 1/50 of a second) and, for more than 1 processor, the resulting efficiency.

Number of Integration Points

<table>
<thead>
<tr>
<th>Processors</th>
<th>100</th>
<th>500</th>
<th>1,000</th>
<th>5,000</th>
<th>10,000</th>
<th>25,000</th>
<th>32,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>9</td>
<td>16</td>
<td>79</td>
<td>158</td>
<td>394</td>
<td>473</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>42</td>
<td>81</td>
<td>199</td>
<td>238</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>31</td>
<td>56</td>
<td>135</td>
<td>161</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>25</td>
<td>45</td>
<td>103</td>
<td>124</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>21</td>
<td>36</td>
<td>86</td>
<td>101</td>
</tr>
<tr>
<td>rel error</td>
<td>2·10⁻⁴</td>
<td>8·10⁻⁶</td>
<td>2·10⁻⁶</td>
<td>2·10⁻⁷</td>
<td>4·10⁻⁷</td>
<td>8·10⁻⁷</td>
<td>3·10⁻⁷</td>
</tr>
</tbody>
</table>

Problem 1: Trapezoidal Rule for \( f(x) = e^x \).
## Number of Indices

<table>
<thead>
<tr>
<th>Processors</th>
<th>10</th>
<th>25</th>
<th>40</th>
<th>55</th>
<th>70</th>
<th>80</th>
<th>85</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>11</td>
<td>25</td>
<td>48</td>
<td>77</td>
<td>101</td>
<td>114</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>15</td>
<td>26</td>
<td>41</td>
<td>.53</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>.25</td>
<td>.69</td>
<td>.83</td>
<td>.92</td>
<td>.94</td>
<td>.95</td>
<td>.95</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>20</td>
<td>30</td>
<td>37</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>.17</td>
<td>.52</td>
<td>.69</td>
<td>.80</td>
<td>.86</td>
<td>.91</td>
<td>.90</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>8</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>31</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>.10</td>
<td>.34</td>
<td>.52</td>
<td>.67</td>
<td>.80</td>
<td>.81</td>
<td>.86</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>20</td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>.10</td>
<td>.37</td>
<td>.56</td>
<td>.74</td>
<td>.77</td>
<td>.84</td>
<td>.84</td>
</tr>
</tbody>
</table>

Problem 2: Compute sum of products of expression.
Array Size (square)

<table>
<thead>
<tr>
<th>Processors</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
<th>400</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>28</td>
<td>64</td>
<td>112</td>
<td>175</td>
<td>251</td>
<td>446</td>
<td>697</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>16</td>
<td>34</td>
<td>58</td>
<td>89</td>
<td>127</td>
<td>224</td>
<td>349</td>
</tr>
<tr>
<td></td>
<td>.67</td>
<td>.88</td>
<td>.94</td>
<td>.97</td>
<td>.98</td>
<td>.99</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>13</td>
<td>25</td>
<td>41</td>
<td>63</td>
<td>87</td>
<td>153</td>
<td>235</td>
</tr>
<tr>
<td></td>
<td>.38</td>
<td>.72</td>
<td>.85</td>
<td>.91</td>
<td>.93</td>
<td>.96</td>
<td>.97</td>
<td>.99</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>13</td>
<td>21</td>
<td>33</td>
<td>49</td>
<td>68</td>
<td>117</td>
<td>179</td>
</tr>
<tr>
<td></td>
<td>.29</td>
<td>.54</td>
<td>.76</td>
<td>.85</td>
<td>.89</td>
<td>.92</td>
<td>.95</td>
<td>.97</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>13</td>
<td>17</td>
<td>29</td>
<td>39</td>
<td>56</td>
<td>96</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>.27</td>
<td>.43</td>
<td>.75</td>
<td>.77</td>
<td>.90</td>
<td>.90</td>
<td>.93</td>
<td>.96</td>
</tr>
</tbody>
</table>

Problem 3: Compute sum of products of array elements.
### Number of vector elements

<table>
<thead>
<tr>
<th>Processors</th>
<th>2,500</th>
<th>10,000</th>
<th>25,000</th>
<th>50,000</th>
<th>75,000</th>
<th>100,000</th>
<th>150,000</th>
<th>200,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>20</td>
<td>51</td>
<td>102</td>
<td>153</td>
<td>204</td>
<td>304</td>
<td>404</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>12</td>
<td>27</td>
<td>54</td>
<td>79</td>
<td>105</td>
<td>156</td>
<td>208</td>
</tr>
<tr>
<td></td>
<td>.60</td>
<td>.83</td>
<td>.94</td>
<td>.94</td>
<td>.97</td>
<td>.97</td>
<td>.97</td>
<td>.97</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>11</td>
<td>21</td>
<td>38</td>
<td>55</td>
<td>72</td>
<td>106</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>.40</td>
<td>.61</td>
<td>.81</td>
<td>.89</td>
<td>.93</td>
<td>.94</td>
<td>.96</td>
<td>.96</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>19</td>
<td>31</td>
<td>43</td>
<td>56</td>
<td>82</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>.25</td>
<td>.45</td>
<td>.67</td>
<td>.82</td>
<td>.89</td>
<td>.91</td>
<td>.93</td>
<td>.94</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>8</td>
<td>17</td>
<td>28</td>
<td>37</td>
<td>45</td>
<td>69</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>.24</td>
<td>.50</td>
<td>.60</td>
<td>.73</td>
<td>.83</td>
<td>.91</td>
<td>.88</td>
<td>.91</td>
</tr>
</tbody>
</table>

Problem 4: Compute sum of reciprocals of non-zero elements.
### Number of Students and Grades

<table>
<thead>
<tr>
<th>Processors</th>
<th>25×25</th>
<th>50×50</th>
<th>100×100</th>
<th>150×150</th>
<th>200×200</th>
<th>225×225</th>
<th>250×250</th>
<th>300×300</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>12</td>
<td>48</td>
<td>107</td>
<td>188</td>
<td>238</td>
<td>293</td>
<td>422</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>11</td>
<td>28</td>
<td>.58</td>
<td>.99</td>
<td>124</td>
<td>151</td>
<td>215</td>
</tr>
<tr>
<td></td>
<td>.33</td>
<td>.55</td>
<td>.86</td>
<td>.92</td>
<td>.95</td>
<td>.96</td>
<td>.97</td>
<td>.98</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>12</td>
<td>24</td>
<td>43</td>
<td>70</td>
<td>87</td>
<td>106</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>.17</td>
<td>.33</td>
<td>.67</td>
<td>.83</td>
<td>.90</td>
<td>.91</td>
<td>.92</td>
<td>.95</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>20</td>
<td>36</td>
<td>57</td>
<td>69</td>
<td>83</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>.13</td>
<td>.25</td>
<td>.60</td>
<td>.74</td>
<td>.83</td>
<td>.86</td>
<td>.88</td>
<td>.93</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>11</td>
<td>18</td>
<td>35</td>
<td>48</td>
<td>58</td>
<td>70</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>.09</td>
<td>.22</td>
<td>.53</td>
<td>.61</td>
<td>.78</td>
<td>.82</td>
<td>.84</td>
<td>.88</td>
</tr>
</tbody>
</table>

Problem 5: Grading Program – Pessimistic Version: Times include creating processes.
Number of Students and Grades

<table>
<thead>
<tr>
<th>Processors</th>
<th>25×25</th>
<th>50×50</th>
<th>100×100</th>
<th>150×150</th>
<th>200×200</th>
<th>225×225</th>
<th>250×250</th>
<th>300×300</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>12</td>
<td>47</td>
<td>106</td>
<td>186</td>
<td>236</td>
<td>291</td>
<td>416</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
<td>27</td>
<td>55</td>
<td>96</td>
<td>122</td>
<td>148</td>
<td>212</td>
</tr>
<tr>
<td></td>
<td>.4</td>
<td>.67</td>
<td>.87</td>
<td>.96</td>
<td>.97</td>
<td>.97</td>
<td>.98</td>
<td>.98</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
<td>20</td>
<td>41</td>
<td>68</td>
<td>84</td>
<td>103</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td>.44</td>
<td>.5</td>
<td>.78</td>
<td>.86</td>
<td>.91</td>
<td>.94</td>
<td>.94</td>
<td>.96</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>18</td>
<td>33</td>
<td>51</td>
<td>62</td>
<td>80</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>.25</td>
<td>.5</td>
<td>.65</td>
<td>.80</td>
<td>.91</td>
<td>.95</td>
<td>.91</td>
<td>.94</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>15</td>
<td>26</td>
<td>40</td>
<td>50</td>
<td>61</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>.27</td>
<td>.48</td>
<td>.63</td>
<td>.82</td>
<td>.93</td>
<td>.94</td>
<td>.95</td>
<td>.94</td>
</tr>
</tbody>
</table>

Problem 5: Grading Program – Optimistic Version: Does not include creating processes.
## Matrix Order

<table>
<thead>
<tr>
<th>Processors</th>
<th>50</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>1,000</th>
<th>1,500</th>
<th>2,000</th>
<th>2,500</th>
<th>5,000</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>21</td>
<td>55</td>
<td>113</td>
<td>241</td>
<td>356</td>
<td>516</td>
<td>642</td>
<td>1384</td>
<td>2975</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>17</td>
<td>36</td>
<td>67</td>
<td>134</td>
<td>194</td>
<td>280</td>
<td>345</td>
<td>735</td>
<td>1571</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>16</td>
<td>31</td>
<td>52</td>
<td>96</td>
<td>137</td>
<td>193</td>
<td>238</td>
<td>496</td>
<td>1056</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>18</td>
<td>31</td>
<td>47</td>
<td>82</td>
<td>112</td>
<td>153</td>
<td>187</td>
<td>384</td>
<td>805</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>20</td>
<td>32</td>
<td>46</td>
<td>73</td>
<td>97</td>
<td>133</td>
<td>159</td>
<td>318</td>
<td>655</td>
</tr>
</tbody>
</table>

Problem 6: Tridiagonal Matrix Solver
Number of evaluation points

<table>
<thead>
<tr>
<th>Processors</th>
<th>25</th>
<th>50</th>
<th>100</th>
<th>1,000</th>
<th>2,000</th>
<th>4,000</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>47</td>
<td>116</td>
<td>134</td>
<td>241</td>
<td>435</td>
<td>975</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>27</td>
<td>60</td>
<td>73</td>
<td>127</td>
<td>226</td>
<td>.513</td>
</tr>
<tr>
<td></td>
<td>.78</td>
<td>.87</td>
<td>.97</td>
<td>.92</td>
<td>.95</td>
<td>.96</td>
<td>.95</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>20</td>
<td>44</td>
<td>52</td>
<td>88</td>
<td>157</td>
<td>348</td>
</tr>
<tr>
<td></td>
<td>.52</td>
<td>.78</td>
<td>.88</td>
<td>.86</td>
<td>.91</td>
<td>.92</td>
<td>.93</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>19</td>
<td>35</td>
<td>44</td>
<td>73</td>
<td>126</td>
<td>271</td>
</tr>
<tr>
<td></td>
<td>.39</td>
<td>.62</td>
<td>.83</td>
<td>.76</td>
<td>.83</td>
<td>.86</td>
<td>.90</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>17</td>
<td>30</td>
<td>36</td>
<td>60</td>
<td>104</td>
<td>229</td>
</tr>
<tr>
<td></td>
<td>.28</td>
<td>.55</td>
<td>.77</td>
<td>.74</td>
<td>.81</td>
<td>.84</td>
<td>.85</td>
</tr>
</tbody>
</table>

Problem 7: LaGrange polynomial interpolation.
Number of Data Points

<table>
<thead>
<tr>
<th>Processors</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>70</th>
<th>100</th>
<th>150</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>57</td>
<td>233</td>
<td>459</td>
<td>940</td>
<td>2122</td>
<td>3778</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>.53</td>
<td>219</td>
<td>404</td>
<td>863</td>
<td>1933</td>
<td>3732</td>
</tr>
<tr>
<td></td>
<td>.45</td>
<td>.54</td>
<td>.54</td>
<td>.57</td>
<td>.54</td>
<td>.55</td>
<td>.51</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>58</td>
<td>210</td>
<td>402</td>
<td>816</td>
<td>1831</td>
<td>3225</td>
</tr>
<tr>
<td></td>
<td>.27</td>
<td>.33</td>
<td>.37</td>
<td>.38</td>
<td>.38</td>
<td>.39</td>
<td>.39</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>58</td>
<td>197</td>
<td>401</td>
<td>817</td>
<td>1746</td>
<td>3322</td>
</tr>
<tr>
<td></td>
<td>.15</td>
<td>.25</td>
<td>.30</td>
<td>.29</td>
<td>.29</td>
<td>.30</td>
<td>.28</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>68</td>
<td>217</td>
<td>399</td>
<td>823</td>
<td>1784</td>
<td>3053</td>
</tr>
<tr>
<td></td>
<td>.11</td>
<td>.17</td>
<td>.21</td>
<td>.23</td>
<td>.23</td>
<td>.24</td>
<td>.25</td>
</tr>
</tbody>
</table>

Problem 8: Divided difference table - tightly synchronized version.
## Number of Data Points

<table>
<thead>
<tr>
<th>Processors</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td>15</td>
<td>51</td>
<td>103</td>
<td>175</td>
<td>265</td>
<td>371</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>20</td>
<td>56</td>
<td>110</td>
<td>181</td>
<td>276</td>
<td>383</td>
</tr>
<tr>
<td></td>
<td>.25</td>
<td>.38</td>
<td>.38</td>
<td>.46</td>
<td>.47</td>
<td>.48</td>
<td>.48</td>
<td>.48</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>11</td>
<td>21</td>
<td>59</td>
<td>115</td>
<td>189</td>
<td>281</td>
<td>393</td>
</tr>
<tr>
<td></td>
<td>.13</td>
<td>.18</td>
<td>.24</td>
<td>.29</td>
<td>.30</td>
<td>.31</td>
<td>.31</td>
<td>.31</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>12</td>
<td>23</td>
<td>62</td>
<td>119</td>
<td>191</td>
<td>291</td>
<td>404</td>
</tr>
<tr>
<td></td>
<td>.12</td>
<td>.16</td>
<td>.21</td>
<td>.22</td>
<td>.23</td>
<td>.23</td>
<td>.23</td>
<td>.23</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>11</td>
<td>25</td>
<td>64</td>
<td>121</td>
<td>202</td>
<td>298</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td>.08</td>
<td>.11</td>
<td>.12</td>
<td>.16</td>
<td>.17</td>
<td>.17</td>
<td>.18</td>
<td>.18</td>
</tr>
</tbody>
</table>

### Problem 8: Divided difference table - naive version which results in almost sequential execution.

Note that asymptotic efficiency for $N$ processors is approximately $1/N$. 
Size of 2D Array (square)

<table>
<thead>
<tr>
<th>Processors</th>
<th>10</th>
<th>25</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>75</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>86</td>
<td>223</td>
<td>352</td>
<td>512</td>
<td>807</td>
<td>1169</td>
<td>1449</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>49</td>
<td>116</td>
<td>181</td>
<td>261</td>
<td>414</td>
<td>589</td>
<td>729</td>
</tr>
<tr>
<td></td>
<td>.58</td>
<td>.88</td>
<td>.96</td>
<td>.97</td>
<td>.98</td>
<td>.97</td>
<td>.99</td>
<td>.99</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>38</td>
<td>85</td>
<td>126</td>
<td>184</td>
<td>285</td>
<td>407</td>
<td>496</td>
</tr>
<tr>
<td></td>
<td>.36</td>
<td>.75</td>
<td>.87</td>
<td>.93</td>
<td>.93</td>
<td>.94</td>
<td>.96</td>
<td>.97</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>34</td>
<td>70</td>
<td>99</td>
<td>143</td>
<td>221</td>
<td>303</td>
<td>381</td>
</tr>
<tr>
<td></td>
<td>.25</td>
<td>.63</td>
<td>.80</td>
<td>.88</td>
<td>.90</td>
<td>.91</td>
<td>.96</td>
<td>.95</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>34</td>
<td>63</td>
<td>88</td>
<td>119</td>
<td>180</td>
<td>254</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>.16</td>
<td>.51</td>
<td>.71</td>
<td>.80</td>
<td>.86</td>
<td>.90</td>
<td>.92</td>
<td>.93</td>
</tr>
</tbody>
</table>

Problem 9: Averaging neighbors in 2D array.
10 Passes.
Matrix Dimension

<table>
<thead>
<tr>
<th>Processors</th>
<th>15</th>
<th>25</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>55</th>
<th>65</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.56</td>
<td>158</td>
<td>327</td>
<td>441</td>
<td>577</td>
<td>919</td>
<td>1369</td>
<td>2269</td>
<td>3961</td>
</tr>
<tr>
<td>2</td>
<td>.74</td>
<td>.83</td>
<td>.88</td>
<td>.90</td>
<td>.91</td>
<td>.93</td>
<td>.94</td>
<td>.95</td>
<td>.97</td>
</tr>
<tr>
<td>3</td>
<td>.55</td>
<td>.67</td>
<td>.76</td>
<td>.79</td>
<td>.81</td>
<td>.85</td>
<td>.88</td>
<td>.90</td>
<td>.92</td>
</tr>
<tr>
<td>4</td>
<td>.41</td>
<td>.56</td>
<td>.64</td>
<td>.70</td>
<td>.72</td>
<td>.76</td>
<td>.81</td>
<td>.84</td>
<td>.88</td>
</tr>
<tr>
<td>5</td>
<td>.35</td>
<td>.43</td>
<td>.56</td>
<td>.60</td>
<td>.66</td>
<td>.68</td>
<td>.75</td>
<td>.80</td>
<td>.83</td>
</tr>
</tbody>
</table>

Problem 10: Gaussian elimination with partial pivoting.
### # of Vector Elements

<table>
<thead>
<tr>
<th>Processors</th>
<th>250</th>
<th>500</th>
<th>1,000</th>
<th>2,500</th>
<th>7,500</th>
<th>12,500</th>
<th>25,000</th>
<th>50,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>33</td>
<td>51</td>
<td>108</td>
<td>298</td>
<td>488</td>
<td>961</td>
<td>1907</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>25</td>
<td>34</td>
<td>63</td>
<td>159</td>
<td>252</td>
<td>489</td>
<td>963</td>
</tr>
<tr>
<td></td>
<td>.61</td>
<td>.66</td>
<td>.75</td>
<td>.86</td>
<td>.94</td>
<td>.97</td>
<td>.98</td>
<td>.99</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>23</td>
<td>29</td>
<td>49</td>
<td>112</td>
<td>174</td>
<td>333</td>
<td>649</td>
</tr>
<tr>
<td></td>
<td>.38</td>
<td>.48</td>
<td>.59</td>
<td>.73</td>
<td>.89</td>
<td>.93</td>
<td>.96</td>
<td>.98</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>23</td>
<td>28</td>
<td>42</td>
<td>90</td>
<td>137</td>
<td>256</td>
<td>492</td>
</tr>
<tr>
<td></td>
<td>.29</td>
<td>.36</td>
<td>.46</td>
<td>.64</td>
<td>.83</td>
<td>.89</td>
<td>.94</td>
<td>.97</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>23</td>
<td>27</td>
<td>38</td>
<td>77</td>
<td>115</td>
<td>209</td>
<td>381</td>
</tr>
<tr>
<td></td>
<td>.22</td>
<td>.29</td>
<td>.38</td>
<td>.57</td>
<td>.77</td>
<td>.85</td>
<td>.92</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Problem 11: Data filtering.**
Size of ABIG

<table>
<thead>
<tr>
<th>Processors</th>
<th>10×10</th>
<th>25×25</th>
<th>50×50</th>
<th>100×100</th>
<th>150×150</th>
<th>200×200</th>
<th>250×250</th>
<th>300×300</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>33</td>
<td>74</td>
<td>131</td>
<td>205</td>
<td>296</td>
</tr>
<tr>
<td>2</td>
<td>.25</td>
<td>.33</td>
<td>.57</td>
<td>.92</td>
<td>.95</td>
<td>.95</td>
<td>.97</td>
<td>.98</td>
</tr>
<tr>
<td>3</td>
<td>.17</td>
<td>.17</td>
<td>.44</td>
<td>.73</td>
<td>.88</td>
<td>.91</td>
<td>.95</td>
<td>.97</td>
</tr>
<tr>
<td>4</td>
<td>.08</td>
<td>.10</td>
<td>.25</td>
<td>.59</td>
<td>.77</td>
<td>.89</td>
<td>.90</td>
<td>.93</td>
</tr>
<tr>
<td>5</td>
<td>.05</td>
<td>.08</td>
<td>.23</td>
<td>.55</td>
<td>.67</td>
<td>.79</td>
<td>.91</td>
<td>.88</td>
</tr>
</tbody>
</table>

Problem 12: Construction of a big array.
Number of Vector Elements

<table>
<thead>
<tr>
<th>Processors</th>
<th>50</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>1,000</th>
<th>2,500</th>
<th>5,000</th>
<th>7,500</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>8</td>
<td>17</td>
<td>35</td>
<td>69</td>
<td>172</td>
<td>343</td>
<td>513</td>
<td>683</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>6</td>
<td>11</td>
<td>20</td>
<td>37</td>
<td>88</td>
<td>174</td>
<td>259</td>
<td>343</td>
</tr>
<tr>
<td></td>
<td>.50</td>
<td>.67</td>
<td>.77</td>
<td>.88</td>
<td>.93</td>
<td>.98</td>
<td>.99</td>
<td>.99</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>16</td>
<td>27</td>
<td>61</td>
<td>118</td>
<td>175</td>
<td>231</td>
</tr>
<tr>
<td></td>
<td>.27</td>
<td>.44</td>
<td>.63</td>
<td>.73</td>
<td>.85</td>
<td>.94</td>
<td>.97</td>
<td>.98</td>
<td>.99</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>12</td>
<td>21</td>
<td>47</td>
<td>92</td>
<td>134</td>
<td>177</td>
</tr>
<tr>
<td></td>
<td>.20</td>
<td>.40</td>
<td>.43</td>
<td>.73</td>
<td>.82</td>
<td>.91</td>
<td>.93</td>
<td>.96</td>
<td>.96</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td>11</td>
<td>18</td>
<td>41</td>
<td>76</td>
<td>110</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td>.16</td>
<td>.32</td>
<td>.31</td>
<td>.64</td>
<td>.77</td>
<td>.84</td>
<td>.90</td>
<td>.93</td>
<td>.95</td>
</tr>
</tbody>
</table>

Problem 13: Transform a vector, sum squares of elements.
### Number of Evaluation Points

<table>
<thead>
<tr>
<th>Processors</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>1,000</th>
<th>2,500</th>
<th>3,300</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>15</td>
<td>37</td>
<td>74</td>
<td>148</td>
<td>369</td>
<td>488</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>21</td>
<td>39</td>
<td>77</td>
<td>187</td>
<td>247</td>
</tr>
<tr>
<td>.33</td>
<td>.62</td>
<td>.5</td>
<td>.75</td>
<td>.88</td>
<td>.95</td>
<td>.96</td>
<td>.99</td>
<td>.99</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>17</td>
<td>29</td>
<td>53</td>
<td>128</td>
<td>166</td>
</tr>
<tr>
<td>.17</td>
<td>.33</td>
<td>.39</td>
<td>.56</td>
<td>.73</td>
<td>.85</td>
<td>.93</td>
<td>.96</td>
<td>.98</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>15</td>
<td>22</td>
<td>41</td>
<td>98</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>.13</td>
<td>.21</td>
<td>.22</td>
<td>.42</td>
<td>.62</td>
<td>.84</td>
<td>.90</td>
<td>.94</td>
<td>.95</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>14</td>
<td>22</td>
<td>38</td>
<td>81</td>
<td>106</td>
</tr>
<tr>
<td>.08</td>
<td>.20</td>
<td>.23</td>
<td>.27</td>
<td>.53</td>
<td>.67</td>
<td>.78</td>
<td>.91</td>
<td>.92</td>
<td></td>
</tr>
</tbody>
</table>

Problem 14: Test 4 integrators on 10 functions.
### Number of Evaluation Nodes

<table>
<thead>
<tr>
<th>Processors</th>
<th>50</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>1,000</th>
<th>3,500</th>
<th>5,000</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>17</td>
<td>38</td>
<td>75</td>
<td>148</td>
<td>508</td>
<td>726</td>
<td>1447</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>12</td>
<td>23</td>
<td>40</td>
<td>76</td>
<td>258</td>
<td>365</td>
<td>727</td>
</tr>
<tr>
<td></td>
<td>.63</td>
<td>.71</td>
<td>.83</td>
<td>.94</td>
<td>.97</td>
<td>.98</td>
<td>.99</td>
<td>.99</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>11</td>
<td>18</td>
<td>30</td>
<td>55</td>
<td>174</td>
<td>248</td>
<td>487</td>
</tr>
<tr>
<td></td>
<td>.37</td>
<td>.52</td>
<td>.70</td>
<td>.83</td>
<td>.90</td>
<td>.97</td>
<td>.97</td>
<td>.99</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>11</td>
<td>17</td>
<td>26</td>
<td>44</td>
<td>134</td>
<td>188</td>
<td>370</td>
</tr>
<tr>
<td></td>
<td>.28</td>
<td>.39</td>
<td>.56</td>
<td>.72</td>
<td>.84</td>
<td>.95</td>
<td>.97</td>
<td>.98</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>12</td>
<td>16</td>
<td>24</td>
<td>38</td>
<td>110</td>
<td>154</td>
<td>298</td>
</tr>
<tr>
<td></td>
<td>.18</td>
<td>.28</td>
<td>.48</td>
<td>.63</td>
<td>.78</td>
<td>.92</td>
<td>.94</td>
<td>.97</td>
</tr>
</tbody>
</table>

Problem 15: Comparison of interpolation methods.
<table>
<thead>
<tr>
<th>Processors</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>33</th>
<th>35</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>46</td>
<td>101</td>
<td>178</td>
<td>279</td>
<td>407</td>
<td>496</td>
<td>563</td>
<td>749</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>38</td>
<td>74</td>
<td>122</td>
<td>182</td>
<td>262</td>
<td>316</td>
<td>359</td>
<td>454</td>
</tr>
<tr>
<td></td>
<td>.44</td>
<td>.61</td>
<td>.68</td>
<td>.73</td>
<td>.77</td>
<td>.78</td>
<td>.78</td>
<td>.78</td>
<td>.82</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>42</td>
<td>68</td>
<td>107</td>
<td>162</td>
<td>220</td>
<td>262</td>
<td>290</td>
<td>378</td>
</tr>
<tr>
<td></td>
<td>.26</td>
<td>.37</td>
<td>.50</td>
<td>.55</td>
<td>.57</td>
<td>.62</td>
<td>.63</td>
<td>.65</td>
<td>.66</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>48</td>
<td>74</td>
<td>106</td>
<td>154</td>
<td>217</td>
<td>243</td>
<td>271</td>
<td>346</td>
</tr>
<tr>
<td></td>
<td>.15</td>
<td>.24</td>
<td>.34</td>
<td>.42</td>
<td>.45</td>
<td>.47</td>
<td>.51</td>
<td>.52</td>
<td>.54</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>48</td>
<td>74</td>
<td>112</td>
<td>164</td>
<td>202</td>
<td>241</td>
<td>270</td>
<td>339</td>
</tr>
<tr>
<td></td>
<td>.11</td>
<td>.19</td>
<td>.27</td>
<td>.32</td>
<td>.34</td>
<td>.40</td>
<td>.41</td>
<td>.42</td>
<td>.44</td>
</tr>
</tbody>
</table>

Problem 16: Solve Hilbert problem with multiple right sides.