ICASE

A LANGUAGE COMPARISON FOR SCIENTIFIC
COMPUTING ON MIMD ARCHITECTURES

Mark T. Jones
Merrell L. Patrick
Robert G. Voigt

Contract Nos. NASI-18107, NASI-18605
January 1989

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association
A Language Comparison for Scientific Computing On MIMD Architectures

Mark T. Jones*, Merrell L. Patrick* and Robert G. Voigt†

Abstract

Choleski’s method for solving banded symmetric, positive definite systems is implemented on a multiprocessor computer using three FORTRAN based parallel programming languages, the Force, PISCES and Concurrent FORTRAN. The capabilities of the languages for expressing parallelism and their user friendliness are discussed, including readability of the code, debugging assistance offered, and expressiveness of the languages. The performance of the different implementations is compared. It is argued that PISCES, using the Force for medium-grained parallelism, is the appropriate choice for programming Choleski’s method on the multiprocessor computer, Flex/32.

*Department of Computer Science, Duke University, Durham, NC 27706
†ICASE, NASA Langley Research Center, Hampton, VA 23665.
This research was supported by the National Aeronautics and Space Administration under NASA contract Nos. NAS1-18107 and NAS1-18605 while the authors were in residence at ICASE. Additional support for the first two authors was provided by NASA grant No. NAG-1-466.
1. Introduction

Efficient programming of parallel computers to support scientific applications is of increasing importance. Although many programming environments are available on different machines, there have been relatively few comparisons of different programming paradigms on the same machine. Several factors that contribute to the useability of a language have been identified. Using these factors this paper explores the strong and weak points of three parallel languages by implementing Choleski's method for solving \( Ax = b \), where \( A \) is a banded symmetric positive definite matrix, on the Flexible Computer Corporation Flex/32 [Mat84]. The Flex/32 has twenty processors with each processor having local memory and access to a shared memory. Appendix 9 illustrates the overall architecture of the Flex/32. The architecture and three languages support both shared memory and local memory implementations of the algorithm. In addition, one language supports message passing. Thus, three programming paradigms can be considered: shared memory, message passing, and shared/local which takes advantage of the local memory. These are discussed in the next section. The three languages are all derivatives of FORTRAN and are discussed briefly in section 3. The Choleski algorithm is given in Section 4 along with a brief discussion of the implementation tradeoffs. Section 5 presents observations on the implementation of the algorithm using the various paradigms. The observations are based on factors such as expressibility of functional parallelism and data partitioning, support for communication and synchronization, runtime cost, ease of program conversion, and user friendliness. The Appendices contain the code representing the implementations.

2. Programming Paradigms

Three different parallel programming paradigms are considered: shared memory, message passing, and shared/local (henceforth referred to as local memory). Parallel architectures can also be placed in these three classes. Each paradigm can be implemented on each architecture, but the cost of implementing a paradigm on an architecture that doesn't naturally support that paradigm can be substantial.

For the purposes of this paper, a shared memory architecture is one in which each processor has equal access to a shared or common memory (architectures where processors have cache memory are placed in this category). In a hybrid architecture, each processor has a local memory and access to memory shared by all the processors. Processors in a message passing architecture only have access to local memory and must communicate via messages with other processors.

2.1 Shared Memory

When using the shared memory paradigm, the programmer can view the computer as a sequential computer with several concurrent processes running. Some of the programming issues that arise are similar to those arising in concurrent programming on a sequential machine. Since all processors are viewed as having equal access to all memory, the location of data is not important. However, contention between processors for a particular location in the shared memory or for the interconnection network between the processors and memory must be considered. The programmer is primarily concerned with dividing up the work among the processors to allow for maximum parallelism while minimizing communication and providing synchronization among the processors.
Version A

sum = sum + 1

Version B

LOCK(sumlock)

sum = sum + 1

UNLOCK(sumlock)

Figure 1: Shared memory programming bug

All communication and synchronization between processors takes place via shared memory. One of the major burdens that the shared memory paradigm places on the programmer is the necessity to synchronize references to objects that are used by more than one processor. Some objects or sections of code require that they be accessed sequentially and the programmer must ensure that this is the case while trying to keep all the processors doing useful work. This need for synchronization is often the source of "parallel bugs" in shared memory programs ("parallel bugs" are bugs that are introduced because the tasks of the program are being run simultaneously, not traditional programming bugs). This type of bug also arises when running concurrent processes on a sequential computer. Figure 1 shows an example of this type of bug. If several processors are simultaneously executing Version A, more than one processor could fetch the same value for sum, add one to it, and replace sum with the same value. In order to get the correct answer, the addition to sum must be atomic. In version B, the addition to sum is made atomic by putting an exclusive lock around it. This is an example of synchronization which the programmer must provide.

2.2 Message Passing

When programming in the message-passing paradigm, one of the programmer's major concerns is the distribution of data. Since one processor cannot access another processor's memory, performance is improved if the data a processor needs is allocated to its memory. Data exchange and communication between processors is achieved via messages sent explicitly from one processor to another. Thus the programmer is responsible for movement of data and the division of work among processors. The movement of data is achieved by the explicit sending and receiving of messages that contain the data to be moved. Synchronization is implicit in the message passing because a processor does not send data until the data is ready and a processor does not receive data until it is ready to receive it. Thus, the programmer doesn't have to be concerned with the synchronization problem of the shared memory paradigm, but is faced with the new problem of moving data from processor to processor and partitioning this data efficiently across the processors. The programmer must really view this paradigm as a group of isolated processes executing simultaneously that can communicate only by messages, somewhat akin to the communicating sequential processes model of Hoare [Hoa78]. Programs tend to be more difficult to write, but once written, do not have the synchronization bugs that occur in shared memory programs. The code in Figure 1 in the message passing paradigm might look like the code in Figure 2. In this code, each worktask sends the value that is to be added to sum to sumtask which holds sum and is responsible for updating sum. Thus, no explicit synchronization is necessary, just the sending of messages.
sumtask  worktask

  do 10 i=1,P  send (val) to sumtask
    receive(val)
    sum = sum + val
  10  continue

Figure 2: Equivalent message passing code for sum problem

2.3 Shared/Local

Programming in the local paradigm is very similar to programming in the shared memory paradigm, with the exception that in order to obtain peak performance, locality of data must be considered. A hybrid architecture can be programmed as a shared memory architecture, but performance may not be optimal because the use of local memory may not be optimal (local references are faster than shared memory references and there is less possibility of contention). The shared/local paradigm lets the programmer make use of this memory hierarchy by allowing the programmer to specify where memory is allocated. After the allocation is done, the program looks the same as a shared memory program. The programmer may also want to make local copies of shared data that a processor accesses many times in order to make fewer shared memory references. The bugs for the shared/local paradigm seem to be the same as for the shared memory paradigm and aside from memory allocation, the code tends to look the same.

3. Languages and Their Use

Languages compared in this study are restricted to FORTRAN based languages that have been implemented on the Flex/32.

3.1 The Force

The Force is a parallel language for shared memory multiprocessors [Jor87]. It consists of extensions to FORTRAN that include constructs for both medium and coarse grained parallelism. A Force is a set of simultaneously initiated processes which run concurrently on different processors. Force members communicate through shared variables and synchronize through barriers and critical regions. Loop iterations are partitioned among Force members by prescheduling or self-scheduling. The Force is currently implemented as a preprocessor to the ConCurrent FORTRAN preprocessor.

3.2 ConCurrent FORTRAN

ConCurrent FORTRAN [Cor86] is a parallel language for the Flex/32 computer implemented by Flexible Computer Corporation. The language assumes a shared memory model of computation with some limited message-passing capabilities for synchronization. The user is responsible for
explicit process management. ConCurrent FORTRAN is implemented as a preprocessor to the FORTRAN compiler.

3.3 PISCES

PISCES is a parallel language and environment for scientific computation [Pra87]. It can support both message-passing based programming and shared memory programming, or a mix of the two. For the purposes of this comparison, the two aspects of PISCES are treated as two separate languages. PISCES is currently implemented as a preprocessor to the FORTRAN compiler and includes a menu-driven environment for configuration of the machine, running the program, and obtaining debugging information. The message-passing portion of PISCES provides facilities for explicit generation of processes and for process identification. It also provides message sending constructs and “handlers” that accept and process messages. The shared memory portion of PISCES is actually the Force language with some minor syntactic differences. All the constructs, including shared variables, of the Force can be used within a PISCES process.

3.4 Using the Languages

Each processor of the Flex 32 Multiprocessor Computer has its own local memory as well as access to a shared memory. This classifies it as a hybrid of distributed and shared memory architectures. Given this hybrid nature and implementations of the three languages which support it, algorithms can have strictly shared memory implementations or local memory implementations which use shared memory for communication amongst processors. In addition, one language, PISCES, supports strictly message passing implementations of the algorithms. Therefore, in our study a total of seven different implementations of Choleski’s method were possible on the Flex/32. This makes it a particularly interesting architecture on which to compare the various paradigms for programming parallel computers. In the following sections the terms shared memory, local memory and message passing will be used to distinguish between the different implementations.

4. Choleski’s Method and its Parallel Implementation

The solution of $Ax = b$

where $A$ is symmetric positive definite and banded with semi-bandwidth $\beta$ is carried out in three phases:

1) Factor $A$ into $LL^T$,
2) forward solve $Ly = b$ for $y$, and
3) backward solve $L^Ty = x$ for $x$.

There are different ways of organizing each of these phases of computation as described by Dongarra, et al. [DGK84]. For the factorization phase, the “kji” form used by Cleary, et al. [CHO86] has been chosen, namely:
for $k = 1$ to $N$
  $l_{kk} = a_{kk}^{1/2}$
  for $s = k + 1$ to $\min(k + \beta, N)$
    $l_{sk} = a_{sk} / l_{kk}$
  for $j = k + 1$ to $\min(k + \beta, N)$
    for $i = j$ to $\min(k + \beta, N)$
      $a_{ij} = a_{ij} - l_{ik} l_{jk}$

$kji$ Choleski Factorization

This form of Choleski factorization is column oriented, so columns are used to define the granularity of parallelism. Hence, individual processors are assigned sets of columns which they operate upon one at a time. The column wrapped assignment is chosen, which means processor $i$ is assigned columns $i$, $i + p$, $i + 2p$, ..., assuming, of course, there are $p$ processors. In the shared memory versions, each processor operates on its columns which are all stored in shared memory, whereas in the local memory versions a processor's columns are copied to its local memory and operated upon there. In the latter case, data shared by all the processors, e.g., a pivot column, are written to shared memory and accessed there.

For the forward and backward solve phases the inner product $(ij)$ algorithm [RO88] and the column sweep algorithm [GH86] are considered. These are given below.

for $i = 1$ to $n$
  for $j = \max(i - \beta, 1)$ to $i - 1$
    $b_i = b_i - l_{ij} y_j$
    $y_i = b_i / l_{ii}$

The Inner Product $(ij)$ Algorithm for $Ly = b$

for $j = n$ to $1$
  $x_j = y_j / l_{jj}$
  for $i = j - 1$ to $\max(j - \beta, 1)$
    $y_i = y_i - l_{ij} x_j$

Column Sweep $(ji)$ Algorithm for $L^T x = y$

For the shared memory versions of the forward and backward substitutions, the column sweep algorithm is used in both cases. The inner product algorithm could have been equally as effective. After the factorization phase in the local versions, the columns of $L$ are stored in the local memories in wrapped column form. In this case, the inner product $(ij)$ algorithm for $Ly = b$ and the column sweep $(ji)$ algorithm for $L^T x = y$ yielded the more efficient implementation. Note that here the hybrid nature of the architecture affected the choice of algorithm used. To optimize use of local memory, the matrix is stored by columns. To take advantage of this storage, the inner product
algorithm followed by the column sweep algorithm must be used, rather than using the column sweep algorithm in the both cases as we did for the shared memory version.

5. Comparisons

In the process of carrying out this study several factors contributing to the useability of a language were identified. These include expressibibility of functional parallelism and data partitioning, support for communication and synchronization, ease of learning the language, ease of converting existing programs, readibilty of the code, debugging and syntax checking, and user friendliness.

As noted above seven different implementations of Choleski's method using the three languages on the Flex/32 are possible. We examine only six of those implementations in carrying out our comparisons below. The six are shared and local memory Force, shared and local memory ConCurrent FORTRAN, strictly message passing PISCES, and PISCES with Force. Programs for each of these implementations are included in the appendices. Note that the PISCES with Force program is just shared memory Force enclosed in a PISCES task definition statement.

5.1 Expression of Functional Parallelism and Data Partitioning

First the expression of functional and data parallelism is examined. In line 1 of the Force program in Appendix 1, a Force macro declares the start of a parallel main program, named Choleski, which will be executed by \( NP \) processes each of which will be identified by a unique identifier \( ME \). The number of processes executing the program is a parameter specified by the user at runtime. A "driver" routine creates these processes, assigns values to \( NP \) and \( ME \) and returns control to the user main program. All processes begin executing from this point on, until they are terminated by the Join statement in line 141. Segments of program which are to be executed by only one process are enclosed in a Barrier - End Barrier pair, e.g., the program segment which puts the pivot column into shared memory for everyone to access (lines 70 - 74). Without barriers each process would execute the main program (the function, in this case) in parallel.

Another example of functional parallelism is illustrated by the parallel Presched DO loop in lines 38-40 of the shared memory version of the Force in Appendix 2. Since the statements within the loop indexed by \( S \) do not depend on each other, they can be executed in parallel for different values of \( S \). Pre-scheduling partitions different values of \( S \) evenly over processes at compile time. The function being executed in parallel is the computation of the pivot column.

In ConCurrent FORTRAN, the Process statement defines a process to the executing environment and if the statement is within a COBEGIN or COBLOCK statement, it also starts execution of the process. For example, in lines 71-75 of Appendix 3, \( NP \) processes are defined where \( NP \) is the number of processors being used. Since the process statements are in a COBLOCK statement, each process will begin execution of the Choleski factorization subroutine ELCOL() at the end of the COBLOCK statement. Process with tag \( PID(J) \) will be executed by processor number PROCNUM(\( J \)) and will operate upon the set of columns assigned to it's local memory by the processes executed in the COBLOCK statements 62-66. This set of statements accomplishes the data partitioning needed for parallel execution of the Choleski factorization given in lines 152-187 (the main body of the subroutine ELCOL).

Every PISCES program is structured as a set of one or more tasks that carry out the computational work. The first statement in the PISCES program of Appendix 5 defines the main task,
chol. Within this parent task other tasks are initiated which will work in parallel to carry out the
Choleski factorization, the forward solve and backward solve. These tasks are initiated in statement
193 with statements defining the Choleski factorization phase of the tasks given in lines 263-301.
Sets of data required by the tasks are sent to them at task initiation time much as data is passed
to a FORTRAN subroutine when it is called. Subtask initiation and the passing of data to them
are illustrated in lines 82-92 of Appendix 5.

The Force constructs provide the user with the ability to do medium grain, loop-level parallelism
(using the parallel do loops) as well as coarser grain parallelism by simply calling subroutines
within the parallel do loops. These levels of parallelism are supported efficiently by starting up
processes on each processor at the beginning of the program and using constructs like the Barrier
statement to provide synchronization. With PISCES and ConCurrent FORTRAN, the user is
responsible for starting up the processes and is limited to a coarser grain granularity unless he
provides the synchronization constructs. The implementation of Choleski factorization required
loop-level parallelism. This required a high ratio of messages to computation in the case of PISCES
and the use of the WHEN and CFlock statements in ConCurrent FORTRAN to construct the
equivalent of a barrier.

5.2 Communication

Here language features and constructs which support the communication of intermediate data
between tasks or processes executing in parallel are compared.

Within the Force program of Appendix 1 and the ConCurrent FORTRAN program of Appendix
3, communication between processes is accomplished by a process assigning the values to be com-
municated into shared variables in shared memory from which they can be read by other processes
which need them. This is illustrated, e.g., within the Choleski factorization loop, given by lines
55-85 in Appendix 1 and lines 152-185 in Appendix 3, where the process owning the current pivot
column will modify it and then write it from its private local memory to a shared variable in shared
memory. This action is carried out by a simple assignment statement. The Force shared memory
program required no communication between the tasks.

In PISCES programs, the communication of intermediate data between executing tasks is more
explicit. This is accomplished with “send” statements and “accept” statements which use “han-
dlers” to accept the data being sent. The use of these constructs is illustrated in the Choleski
factorization tasks, lines 251-289 of Appendix 5. If a task owns the current pivot column it updates
it and uses the “to all send” statement to send it to all other tasks. The send statement also
specifies the name of a “handler” pivot in this case, which accepts the data. Statements 268-276
deal with the acceptance of the pivot column while statements 373-385 define the “handler” task.

The setup time for communication (and programming time) required by PISCES is much larger
than that of the local memory versions of Force and ConCurrent FORTRAN. In Force and ConCur-
rent FORTRAN, it is a simple matter of using an assignment statement to assign data to a variable
in shared memory and then the other processors can read this data. In PISCES, the programmer
must use a send statement to send the message to the tasks that need the data, and those tasks
must then execute a “handler” which is in effect a subroutine.
5.3 Synchronization

Next, the constructs available in the different languages for managing synchronization of processes and tasks are examined. Two types of synchronization are used within the FORCE program of Appendix 1, the barrier and critical statements. The use of the barrier statement is illustrated in the Choleski factorization loop. Statements 70 and 74 are a "Barrier" - "end Barrier" pair. This causes all processes to wait before proceeding until the process which computes the current pivot column has written it to shared memory. The use of the critical section is illustrated in lines 100-102 of Appendix 1.

In the ConCurrent FORTRAN program of Appendix 3, the WHEN statement and CFlock statements are used to accomplish synchronization. The WHEN statement appears in line 162 and prevents the process which owns the current pivot column from updating it and writing it to shared memory until all other processes have finished using the old pivot column. The WHEN statement in line 170 prevents the processes that need the current pivot column from continuing until it is available in shared memory. The CFlock-CFulck statement in lines 182-184 assures that only one process at a time will update the shared memory variable, NUMDONE.

In the PISCES program of Appendix 5, "send" and "accept" statements are used to synchronize the execution of tasks. For example, in the Choleski factorization, a task cannot update its set of columns until it has accepted the pivot column (lines 268-270) from the task which owns, updates and sends it (lines 254-260). A check is made by each task to see that the pivots it requires are being received in proper order. If not, the task resends them to itself until they are received in the proper order (lines 273-275).

When using PISCES message passing, synchronization is taken care of by the communication of data; the programmer is not responsible for it. However, in the ConCurrent FORTRAN and FORCE programs this is one of the programmer's main responsibilities. The FORCE synchronization constructs are easier to use than those in ConCurrent FORTRAN, but they are not as flexible. The Barrier statement is very useful, however it requires that all processors reach a Barrier. The programmer cannot specify that one task execute some code while the other tasks execute some other code that contains a Barrier. When the programmer needs the equivalent of a barrier statement in ConCurrent FORTRAN he must construct it himself.

5.4 Runtime Cost

Comparisons of the runtimes of the various programs were obtained by running the programs on several different data sets. Appendix 7 shows the results of this comparison on a data set generated from a structural analysis application at NASA Langley Research Center. Negative speedups occur in some of the forward and back solve cases due to the large ratio of synchronization to computation in these algorithms. From these comparisons, it is clear that ConCurrent FORTRAN becomes increasingly costly as more processors are added. The FORCE versions are faster, with the shared and local memory versions being competitive with each other. The difference in execution times of the FORCE programs and strictly message passing PISCES programs is due in part to the overhead inherent in message passing and in part to its implementation on an architecture which does not support message passing. Runtimes of FORCE and PISCES with FORCE programs are nearly identical. The high cost of ConCurrent FORTRAN is due to the costly implementation of WHEN on the Flex/32 compared to the efficient lock routines used in FORCE.
5.5 Conversion of Existing Programs

If the parallelism in an existing FORTRAN program exists in DO-loops then it is a fairly simple matter to convert FORTRAN into the Force by using pre-scheduled or self scheduled loops. Synchronization is accomplished by barrier statements and critical sections which are easy to use. In both PISCES and ConCurrent FORTRAN, a conversion of existing programs involves more restructuring of the code with PISCES requiring considerably more than ConCurrent FORTRAN. One measure of coding efficiency is the number of lines of code. By this measure, as seen in Appendix 8, the Force is clearly the language of choice of the three languages examined for conversion of existing FORTRAN code.

5.6 Readability and Learning of the Languages

By design, the Force is like FORTRAN with a small number of constructs added. The use of these constructs is reasonably intuitive. Hence, programmers who know FORTRAN can easily learn and read the Force. This can be observed by looking at the Force program of Appendix 2. Although FORTRAN based, PISCES is harder to learn. First, the language is based on the idea of communicating tasks which is a programming paradigm quite different from that of standard languages. Because of this, the new constructs are more complex and hence more difficult to learn. They are, however, much more versatile than those in the Force and ConCurrent FORTRAN. A comparison of the Force program in Appendix 1 with the PISCES program of Appendix 5 clearly indicates different complexities of the two languages. The constructs added to FORTRAN to produce ConCurrent FORTRAN are not much more complex than those those added to the Force.

The readability of a program written in some language is, of course, related to the ease with which that language can be learned. It is not surprising then, given knowledge of FORTRAN, that a Force program is relatively easy to read. Force constructs are simple and almost self-explanatory. However, the lack of explicit process management can create difficulty in understanding the flow of program control in a Force program. For example, in the factorization portion of the Force program in Appendix 1 (lines 55-85), every processor is executing the same code and it is difficult to follow the flow of control.

Once one understands how processes are initiated and the meaning of "when" and "lock/unlock" statements, ConCurrent FORTRAN is quite readable. As PISCES is more difficult to learn, PISCES programs are more difficult to read. PISCES parallel constructs are quite complex, e.g., the message handlers of PISCES tend to hide some of the work being done in a task. This is illustrated by examining statements 257-259 of the PISCES program of Appendix 5 where the "accept" statement names a "handler" incol. One must locate the code for the "handler" incol, lines 370-382, which is not very self-explanatory.

A reasonable measure of difficulty of reading (and time taken to write) languages is comparing the number of lines of code for the same implementation in different languages. This would not always be a good measure of readability if we were comparing very different languages such as APL and FORTRAN, however, since the languages being discussed are all extensions to FORTRAN, it appears to be reasonable. Appendix 8 shows the comparison based on the lines of code. It is clear the Force is the least verbose of the languages and that local versions take more lines of code than shared versions. This is illustrated by comparing the Force local memory and shared memory versions of the programs in Appendix 1 and Appendix 2, respectively. First, one observes that the number and type of declaration statements increases. In the local memory version, additional lines
of code (44-53) are needed to distribute data to the local memories. Also extra code is needed in each of the factoring, forward solve and backward solve phases of solution, e.g., in the factoring phase of the local memory version a test is made (statement 60) to see which processor owns the pivot column; it then computes it and places it in shared memory.

5.7 Debugging and Syntax Checking

All three languages suffer from the problem that they are preprocessors, so the FORTRAN syntax errors that are detected by the FORTRAN compiler have line numbers that do not match the line numbers of the original source file. The programmer must therefore look at the output of the preprocessor to find his syntax errors. The Force preprocessor gives no information on syntax errors that involve Force constructs, it simply passes them on to the compiler. It also provides no runtime debugging support. PISCES will detect many of the syntax errors involving PISCES constructs and give the correct line numbers of the errors in the source file. PISCES also provides very good runtime debugging support, with the capability to trace all messages, process starts, etc. ConCurrent FORTRAN will detect many syntax errors involving ConCurrent constructs and will give the correct line numbers of the errors in the source file. However, it provides no runtime debugging support.

5.8 User Friendliness

To help the user, the Force provides a routine called Forcerun that will allow the user to specify the name of a program to run and the number of processes to be used in running it. This program therefore masks any of the hardware details from the user and is the same for every machine on which the Force is implemented. PISCES is more “user friendly”; it allows the user to interactively configure the machine, set trace options, and run the program. During the run it interactively allows the user to examine such things as message queues and memory being used. ConCurrent FORTRAN, on the other hand has none of the user friendly features of the other two.

6. Conclusions

The above discussion focused on comparing the Force, ConCurrent FORTRAN and PISCES as parallel programming languages. As indicated in the Appendices, the local and shared memory versions of the Force programs are very similar; there is a small difference in the performance of the two codes due to architectural characteristics of the Flex/32. It should be added that PISCES has incorporated all the features of the Force within it’s environment. Hence one is able to use the best features of both PISCES and the Force when writing programs using PISCES. Of course, resulting programs can look like nearly pure PISCES programs, nearly pure Force programs or anywhere between. The PISCES Force program is nearly the same as the Force program but is enclosed in a PISCES task which provides the richness of the PISCES environment for debugging and testing the program. Performance results given in Appendix 7 indicate that PISCES Force performs equally as well as the Force program. We therefore conclude that the best implementation of Choleski’s method on the Flex/32 is one which uses PISCES with Force constructs.

Clearly much progress is needed in the area of parallel languages for scientific computing. One approach is to construct a FORTRAN-based language that allows the easy expression of the parallelism inherent in an algorithm and provides a reasonable amount of portability across
architectures. A difficulty in this area is that many of the parallel architectures are very different from each other. There is a question of just how much portability can be achieved without an unreasonable loss in efficiency.

7. Acknowledgements

The authors thank Harry Jordan and Terry Pratt for many helpful conversations during the writing of this paper.

References


Appendix 1: Force - local memory version

1  Force Choleski of NP ident ME
2  Shared INTEGER Beta,BetaP,N
3  C  Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
4  Private INTEGER I,J,K,S
5  Private INTEGER tempmin
6  Private INTEGER tempmax
7  Shared REAL TempA(100000)
8  C  TempA is temporary holding for the matrix
9  Private REAL A(100000)
10 C  A contains the matrix
11 Private INTEGER Assign(2000)
12 C  Assign is the array of column numbers this processor owns
13 Private INTEGER NumCols
14 C  NumCols is the number of columns owned
15 Shared REAL CurL(2000)
16 C  The current pivot column
17 Shared REAL tempCurL(2000)
18 C  The temporary var holding the next pivot column
19 Shared REAL RHS(2000)
20 C  RHS is the right-hand side vector
21 Private REAL PriSUM
22 Shared LOGICAL UPDTRH
23 C  UPDTRH is used for a critical section
24 Shared REAL Y(2000)
25 C  The Y vector in the forward solve
26 Shared REAL X(2000)
27 C  X is the solution vector
28 Private INTEGER LCol, L2Col
29 End declarations
30
31 Barrier
32 C  Decide whether to read in or build the matrix
33  READ(9,525) I
34  IF (I.eq.0) THEN
35     CALL INITMAT(TempA,RHS,N,Beta,BetaP)
36  ELSE
37     CALL CRMAT(TempA,RHS,N,Beta,BetaP)
38  END IF
39  WRITE(6,600) N, Beta
40 600  FORMAT(' Order',I4,' matrix with a semi-bandwidth',14,'.')
41 525  FORMAT(I4)
42 End Barrier
43 C  Transfer the matrix from Shared memory to local memory
44 LCol = 1
45 Presched DO 700 I = 1, N
46 DO 710 J = 1, BetaP
47     A((LCol*BetaP)+J) = TempA((I*BetaP)+J)
48     Continue
49 Assign(LCol) = I
50 LCol = LCol + 1
51 700 End Presched DO
52 NumCols = LCol - 1
53
Start the Choleski factorization loop

LCol = 1

DO 100 K = 1, N

tempmin = min(K+Beta,N)

if this processor owns the pivot column then compute it and

place it in shared memory

IF (Assign(LCol).eq.K) THEN

A((LCol*BetaP)+l) = sqrt(A((LCol*BetaP)+l))

tempCurL(1) = A((LCol*BetaP)+1)

DO 110 S = K + 1, tempmin

A((LCol*BetaP)+S-K+1) = A((LCol*BetaP)+S-K+1) / A((LCol*BetaP)+1)

tempCurL(S-K+1) = A((LCol*BetaP)+S-K+1)

110 Continue

LCol = LCol + 1
END IF

Barrier

DO 115 S = K, tempmin

CurL(S-K+1) = tempCurL(S-K+1)

115 Continue

End Barrier

Update the rest of the columns

DO 120 L2Col = 1, NumCols

J = Assign(L2Col)

IF ((J.ge.K+1).and.(J.le.tempmin)) THEN

Do 130 I = J, tempmin

A((L2Col*BetaP)+I-J+1) = A((L2Col*BetaP)+I-J+1)

- CurL(I-K+1)*CurL(J-K+1)

130 CONTINUE

END IF

120 CONTINUE

100 CONTINUE

Forward Solve (using inner product)

LCol = 1

DO 300 I = 1, N

tempmax = max(I-Beta,1)

PriSUM = 0

Compute the amount this processor will subtract from the RHS

DO 310 L2Col = 1, NumCols

J = Assign(L2Col)

IF ((I.ge.tempmax).and.(I.le.I-1)) THEN

PriSUM = PriSUM + A((L2Col*BetaP)+I-J+1)*Y(J)

END IF

310 CONTINUE

Update the RHS

Critical UPDTRH

RHS(I) = RHS(I) - PriSUM

End Critical

IF (L.col.Assign(LCol)) THEN

CurDiv = A((BetaP*LCol)+1)

LCol = LCol + 1

END IF

Barrier

Y(I) = RHS(I) / CurDiv
End Barrier
CONTINUE

Backward Solve (using col-sweep)
LCol = NumCols
DO 400 J = N, 1, -1
If we own column J, then compute the new X
IF (J.eq.Assign(LCol)) THEN
  X(J) = Y(J) / A((BetaP*LCol)+1)
  LCol = LCol - 1
  IF (LCol.eq.0) LCol = 1
END IF
Barrier
End Barrier

tempmax = max(J-Beta,1)
Everyone update Y
DO 410 L2Col = NumCols, 1, -1
  I = Assign(L2Col)
  IF ((I.ge.J-l).and.(I.ge.tempmax)) THEN
    Y(I) = Y(I) - A((BetaP*L2Col)+J-I+1)*X(J)
  END IF
CONTINUE
410 CONTINUE
CONTINUE

Print the solution vector
Barrier
DO 500 J = 1, N
  WRITE(8,680) J, X(J)
CONTINUE
680 FORMAT(' X('T14,T6E13.6)
End Barrier

Appendix 2: Force - shared memory version

Force Choleski of NP ident ME
Shared INTEGER Beta,BetaP,N
C Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
Private INTEGER IJ,K,S
Private INTEGER tempmin
Private INTEGER tempmax
Shared REAL A(100000)
C A contains the matrix
Shared REAL RHS(2000)
C RHS is the right-hand side vector
Shared REAL Y(2000)
C The Y vector in the forward solve
Shared REAL X(2000)
C X is the solution vector
End declarations
Barrier
Decide whether to read in or build the matrix

READ(9,525) I
IF (I.eq.0) THEN
    CALL INITMAT(A,RHS,N,Beta,BetaP)
ELSE
    CALL CRMAT(A,RHS,N,Beta,BetaP)
END IF
WRITE(6,600) N, Beta
FORMAT(' Order',I4,' matrix with a semi-bandwidth',I4,'.')

End Barrier

Start the choleski factorization loop
DO 100 K = 1, N
    Compute the first element of the pivot column
    Barrier
    A((K*BetaP)+1) = sqrt(A((K*BetaP)+1))
    End Barrier
    tempmin = min(K+Beta,N)
    Compute the rest of the pivot column
    Presched DO 110 S = K+1, tempmin
        A((K*BetaP)+S-K+1) = A((K*BetaP)+S-K+1) / A((K*BetaP)+1)
    End Presched DO
    Barrier
    End Barrier
    Update the rest of the columns
    Presched DO 120 J = K+1, tempmin
        Do 130 I = J, tempmin
            A((J*BetaP)+I-J+1) = A((J*BetaP)+I-J+1) - A((K*BetaP)+1-K+1)*A((K*BetaP)+J-K+1)
        CONTINUE
    End Presched DO
    Barrier
The foward solve (using col-sweep)
DO 300 J = 1, N
    Barrier
    Y(J) = RHS(J) / A((BetaP*J)+1)
    End Barrier
    tempmin = min(J+Beta,N)
    Presched DO 310 I = J+1, tempmin
        RHS(I) = RHS(I) - A((BetaP*J)+I-J+1)*Y(J)
    End Presched DO
    Barrier
The backward solve (using col-sweep)
DO 400 J = N, 1, -1
    Barrier
    X(J) = Y(J) / A((BetaP*J)+1)
    End Barrier
    tempmax = max(J-Beta,1)
    Presched DO 410 I = J-1, tempmax, -1
        Y(I) = Y(I) - A((BetaP*J)+I-J+1)*X(J)
    End Presched DO
CONTINUE

C Print out the solution vector
Barrier
DO 500 J = 1, N
WRITE(8,680) J, X(J)
CONTINUE
FORMAT(* X('4,14,') = ',6E13.6)
End Barrier
Join
END

Appendix 3: Concurrent FORTRAN - local memory version

PROGRAM MAIN
C PRCNUM holds the physical proc number corresponding the
c the logical proc number
C NP is the number of processors
C NUMDONE
C Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
C PIVCOL
C TempA(30000)
C A contains the matrix
C ASSIGN(500)
C Assign contains the list of columns that each processor owns
C numcols(30000)
C NUMCOLS
C numcols is the number of columns that a processor owns
C CurL(2000)
C the current pivot column
C RHS(2000)
C RHS is the right-hand side vector
C Y(2000)
The Y vector in the forward solve
C X(2000)
The X vector in the forward solve
C Allocate a lock
C NUMLCK
C NUMLCK
C PID(20)
C PID
C ICFret
C tempmax, tempmin
C Allocate a lock
CALL CFgetl(&ICFret,'NUMLCK')
open(unit=2,cpu=1,file='/usr/u/l/mtj/concur/choleski/param.dat')
C Read in the number of processors
READ(2,525) NP
PRINT *, ' Using ',NP,' processors'
DO 15 I = 1, NP
   PRCNUM(I) = I + 2
15 CONTINUE

C Decide whether to read in or build the matrix
READ(2,525) I
IF (I.eq.0) THEN
   CALL INITMAT(TempA,RHS,N,Beta,BetaP)
ELSE
   CALL CRMAT(TempA,RHS,N,Beta,BetaP)
ENDIF
WRITE(6,600) N, Beta
FORMAT(' Order',I4,' matrix with a semi-bandwidth',I4,'.')
FORMAT(I4)
Load up the private copies of TempA
PRINT *, ' Making private copies'
COBLOCK
DO 155 I = 1, NP
   PROCESS(PID(i),LOADC(),PRCNUM(I))
155 CONTINUE
END COBLOCK
PIVCOL = 0
Start the factorization processes on each processor
NUMDONE = NP
COBLOCK
DO 150 I = 1, NP
   PROCESS(PID(i),ELCOL(),PRCNUM(I))
150 CONTINUE
END COBLOCK
PIVCOL = N + 1
Start the back solve processes on each processor
NUMDONE = NP
COBLOCK
DO 170 I = 1, NP
   PROCESS(PID(i),BACK(),PRCNUM(I))
170 CONTINUE
END COBLOCK
C print the solution vector
DO 500 J = 1, N
      WRITE(8,680) J, X(J)
500 500 CONTINUE
680 FORMAT(' X(',I4,') = ',6E13.6)

CALL CFkill(ICFret,0)
END

C private copies task
SUBROUTINE LOADCO
  Shared REAL /label6/ TempA(30000)
  REAL A(30000)
  common /pblkl/ A(30000)
  INTEGER ASSIGN(500)
  common /pblk2/ ASSIGN(500)
  INTEGER NUMCOLS
  common /pblk3/ NUMCOLS
  INTEGER plself
  INTEGER MYNUM
  INTEGER I, J
  Shared INTEGER /label2/ NP
  Shared INTEGER /label4/ Beta,BetaP,N

  MYNUM = plself()
  NUMCOLS = 0
  DO 10 I = MYNUM, N, NP
    NUMCOLS = NUMCOLS + 1
    ASSIGN(NUMCOLS) = I
    DO 20 J = 1, BetaP
      A((NUMCOLS*BetaP)+J) = TempA((I*BetaP)+J)
    CONTINUE
20 20 CONTINUE
10 10 CONTINUE

RETURN
END

C factorization task
SUBROUTINE ELCOLO
  INTEGER K,I,J,S
  Shared INTEGER /label2/ NP
  Shared INTEGER /label3/ NUMDONE
  Shared INTEGER /label4/ Beta,BetaP,N
  Shared INTEGER /label5/ PIVCOL
  Shared CHARACTER /label1/ NUMLCK
  INTEGER ICFret
  INTEGER MYPIV, MYPIV2
  INTEGER tempmin
  REAL A(30000)
  common /pblkl/ A(30000)
  INTEGER ASSIGN(500)
  common /pblk2/ ASSIGN(500)
  INTEGER NUMCOLS
  common /pblk3/ NUMCOLS
  REAL /label2/ CurL(2000)

RETURN
END
C Start the choleski factorization loop
MYPIV = 1
DO 100 K = 1, N
  tempmin = min(K+Beta,N)
  C If I own column K then compute the pivot col
  IF (K.eq.ASSIGN(MYPIV)) THEN
    A((MYPIV*BetaP)+1) = sqrt(A((MYPIV*BetaP)+1))
    DO 110 S = K+1, tempmin
      A((MYPIV*BetaP)+S-K+1) = A((MYPIV*BetaP)+S-K+1) / A((MYPIV*BetaP)+1)
    CONTINUE
    WHEN (NUMDONE.eq.NP) CONTINUE
    DO 115 S = K, tempmin
      CurL(S-K+1) = A((MYPIV*BetaP)+S-K+1)
    CONTINUE
    MYPIV = MYPIV + 1
    NUMDONE = 0
    PIVCOL = PIVCOL + 1
  ELSE
    ENDIF
  C Update the rest of the columns
  DO 120 MYPIV2 = 1, NUMCOLS
    J = ASSIGN(MYPIV2)
    IF ((J.ge.K+1).and.(J.le.tempmin)) THEN
      DO 130 I = J, tempmin
        A((MYPIV2*BetaP)+I-J+1) = A((MYPIV2*BetaP)+I-J+1)
      C - CurL(I-K+1)*CurL(J-K+1)
      CONTINUE
    ENDIF
  CONTINUE
END

SUBROUTINE FORW()
INTEGER IJ
REAL A(30000)
common /pblk1/ A(30000)
INTEGER ASSIGN(500)
common /pblk2/ ASSIGN(500)
INTEGER NUMCOLS
common /pblk3/ NUMCOLS
Shared INTEGER label2/ NP
Shared INTEGER /label3/ NUMDONE
Shared INTEGER /label4/ Beta,BetaP,N
Shared INTEGER /label5/ PIVCOL
Shared REAL /label6/ RHS(2000)
Shared REAL /label7/ Y(2000)
Shared CHARACTER /label8/ NUMLCK
INTEGER ICFret
INTEGER MYPIV, MYPIV2
INTEGER tempmax

MYPIV = 1
DO 100 I = 1, N
  tempmax = max(I-Beta,l)
  Compute the amount to subtract from the RHS
  PriSUM = 0
  DO 110 MYPIV2 = 1, NUMCOLS
    J = ASSIGN(MYPIV2)
    IF ((J.ge.tempmax).and.(J.lt.I)) THEN
      END IF
      PriSUM = PriSUM + A((BetaP*MYPIV2)+I-J+1)*Y(J)
    CONTINUE
  Update the RHS
  CALL CFlock(ICFret,l ,"UMLCK")
  RHS(1) = RHS(1) - PriSUM
  NUMDONE = NUMDONE + 1
  CALL CFulck(ICFret,l ,"UMLCK")
  IF I own column I then compute Y(l)
  IF (I.eq.ASSIGN(MYPIV)) THEN
    WHEN (NUMDONE.eq.NP) CONTINUE
    Y(l) = RHS(l) / A((BetaP*MYPIV)+1)
    MYPIV = MYPIV + 1
    NUMDONE = 0
    PIVCOL = PIVCOL + 1
    ELSE
      WHEN (PIVCOL.eq.1) CONTINUE
    END IF
  CONTINUE
  RETURN
END

the bacward solve task using col-sweep
SUBROUTINE BACK()
INTEGER IJ
REAL A(30000)
common /pblkl/ A(30000)
INTEGER ASSIGN(500)
common /pblk2/ ASSIGN(500)
INTEGER NUMCOLS
common /pblk3/ NUMCOLS
Shared INTEGER /label2/ NP
Shared INTEGER /label4/ Beta,BetaP,N
Shared INTEGER /label5/ PIVCOL
Shared REAL /label8/ Y(2000)
Shared REAL /label9/ X(2000)
Shared CHARACTER /label11/ NUMLCK
INTEGER MYPIV, MYPIV2
INTEGER tempmax

MYPIV = NUMCOLS
DO 100 J = N, 1, -1
  If this proc owns column J then compute X(J)
IF (J.eq.ASSIGN(MYPIV)) THEN
  X(J) = Y(J) / A((BetaP*MYPIV)+1)
  MYPIV = MYPIV - 1
  IF (MYPIV.eq.0) MYPIV = 1
  PIVCOL = PIVCOL - 1
END IF
WHEN (PIVCOL.le.J) CONTINUE
tempmax = max(J-Beta,1)
DO 110 MYPIV2 = NUMCOLS, 1, -1
  1 = ASSIGN(MYPIV2)
  IF ((I.le.J-1).and.(I.ge.tempmax)) THEN
    Y(I) = Y(I) - A((BetaP*MYPIV2)+J-I+1)*X(J)
  END IF
  CONTINUE
100 CONTINUE
RETURN
END

Appendix 4: ConCurrent FORTRAN - shared memory version

PROGRAM MAIN
  Shared INTEGER /label1/ PRCNUM(20)
  C PRCNUM holds the physical proc number corresponding the
  C the logical proc number
  Shared INTEGER /label2/ NP
  C NP is the number of processors
  Shared INTEGER /label3/ NUMDONE
  Shared INTEGER /label4/ Beta,BetaP,N
  C Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
  Shared INTEGER /label5/ PIVCOL
  Shared REAL /label6/ A(30000)
  A contains the matrix
  Shared REAL /label7/ RHS(2000)
  C RHS is the right-hand side vector
  Shared REAL /label8/ Y(2000)
  C The Y vector in the forward solve
  Shared REAL /label9/ X(2000)
  X is the solution vector
  Shared CHARACTER /label11/ NUMLCK
  EXTERNAL ELCOL
  EXTERNAL FORW
  EXTERNAL BACK
  INTEGER PID(20)
  INTEGER ICFret
  INTEGER 1
  INTEGER I
  INTEGER ICFlret
  INTEGER tempmax, tempmin
  Allocate a lock
  CALL CFgetl(ICFret,'NUMLCK')
  open(unit=2,cpu=1,file='/usr/u1/mj/concur/choleski/param.dat')
  Read in the number of processors
  READ(2,525) NP
  PRINT *, 'Using ',NP,' processors'
  DO 15 I = 1, NP
    CALL CFgetl(ICFret,'NUMLCK')
    open(unit=2,cpu=I,file='/usr/u1/mj/concur/choleski/param.dat')
  END
35 PRCNUM(I) = I + 2
36 15 CONTINUE
37
38 C Decide whether to read in or build the matrix
39 READ(2,525) I
40 IF (I.eq.0) THEN
41 CALL INITMAT(A,RHS,N,Beta,BetaP)
42 ELSE
43 CALL CRMAT(A,RHS,N,Beta,BetaP)
44 END IF
45 WRITE(6,600) N, Beta
46 600 FORMAT(I4)
47
48 PIVCOL = 0
49 C Start the factorization processes on each processor
50 NUMDONE = NP
51 COBLOCK
52 DO 150 I = 1, NP
53 PROCESS(PID(i),ELCOL(),PRCNUM(I))
54 150 CONTINUE
55 END COBLOCK
56
57 PIVCOL = 0
58 C Start the forward solve processes on each processor
59 NUMDONE = NP
60 COBLOCK
61 DO 160 I = 1, NP
62 PROCESS(PID(i),FORW(),PRCNUM(I))
63 160 CONTINUE
64 END COBLOCK
65
66 PIVCOL = N + 1
67 C Start the back solve processes on each processor
68 NUMDONE = NP
69 COBLOCK
70 DO 170 I = 1, NP
71 PROCESS(PID(i),BACK(),PRCNUM(I))
72 170 CONTINUE
73 END COBLOCK
74
75 C Print out the solution vector
76 DO 500 J = 1, N
77 WRITE(8,680) J, X(J)
78 500 CONTINUE
79 680 FORMAT(' X('J4,') = ',6E13.6)
80 CALL CFkill(ICFret,0)
81 END
82
83 C The factorization task
84 SUBROUTINE ELCOL()
85 INTEGER MYNUM
86
INTEGER K, IJ

Shared INTEGER /label2/ NP
Shared INTEGER /label3/ NUMDONE
Shared INTEGER /label4/ Beta, BetaP, N
Shared INTEGER /label5/ PIVCOL
Shared REAL /label6/ A(20000)
Shared CHARACTER /label7/ NUMLCK
INTEGER ICFret
INTEGER MYPIV
INTEGER plself
INTEGER tempmin

Start the choleski factorization loop
Find out what processor I am
MYNUM = plself()
MYPIV = MYNUM

DO 100 K = 1, N
  tempmin = min(K+Beta, N)
  IF (K.eq. MYPIV) THEN
    WHEN (NUMDONE.eq.NP) CONTINUE
    A((K*BetaP)+1) = sqrt(A((K*BetaP)+1))
    DO 110 S = K+1, tempmin
    A((K*BetaP)+S-K+1) = A((K*BetaP)+S-K+1) / A((K*BetaP)+1)
  CONTINUE
  MYPIV = MYPIV + 1
  WHEN (PIVCOL.eq.K) CONTINUE
  ELSE
    WHEN (PIVCOL.eq.K) CONTINUE
ENDIF

DO 120 J = K+MYNUM, tempmin, NP
  DO 130 I = J, tempmin
    A((J*BetaP)+I-J+1) = A((J*BetaP)+I-J+1)
    - A((K*BetaP)+I-K+1)*A((K*BetaP)+J-K+1)
  CONTINUE
CONTINUE

CALL CFlock(ICFret, 1, 'NUMLCK')
NUMDONE = NUMDONE + 1
CALL CFulck(ICFret, 1, 'NUMLCK')

CONTINUE
RETURN
END

The forward solve task (using col-sweep)

SUBROUTINE FORW()
INTEGER MYNUM
INTEGER IJ
Shared INTEGER /label2/ NP
Shared INTEGER /label3/ NUMDONE
Shared INTEGER /label4/ Beta, BetaP, N
Shared INTEGER /label5/ PIVCOL
Shared REAL /label6/ A(20000)
Shared REAL /label7/ RHS(2000)
Find out which processor I am
MYNUM = plself()  
MYPIV = MYNUM  
DO 100 J = 1, N  
tempmin = min(J+Beta,N)  
IF (J.eq.MYPIV) THEN  
WHEN (NUMDONE.eq.NP) CONTINUE  
Y(J) = RHS(J) / A((BetaP*J)+1)  
MYPIV = MYPIV + NP  
NUMDONE = 0  
PIVCOL = PIVCOL + 1  
ELSE  
WHEN (PIVCOL.eq.J) CONTINUE  
ENDIF  
DO 310 I = J+MYNUM, tempmin, NP  
RHS(I) = RHS(I) - A((BetaP*J)+I-J+1)*Y(J)  
CONTINUE  
CALL CFlock(ICFret,1,"NUMLCK")  
NUMDONE = NUMDONE + 1  
CALL CFulck(ICFret,1,"NUMLCK")  
CONTINUE  
RETURN  
END

The back solve task using col-sweep
SUBROUTINE BACK()  
MYNUM  
IJ  
NP  
NUMDONE  
Beta,BetaP,N  
PIVCOL  
A(20000)  
Y(2000)  
X(2000)  
ICFret  
MYPIV  
plself  
tempmax  

find out which processor I am  
MYNUM = plself()  
MYPIV = N + 1 - MYNUM  
DO 100 J = N, 1, -1  
tempmax = max(J-Beta,1)  
IF I am responsible for col J then compute X(J)
200 IF (J.eq.MYPIV) THEN
201 WHEN (NUMDONE.eq.NP) CONTINUE
202 X(J) = Y(J) / A((BetaP*I)+1)
203 MYPIV = MYPIV - NP
204 NUMDONE = 0
205 PIVCOL = PIVCOL - 1
206 ELSE
207 WHEN (PIVCOL.eq.J) CONTINUE
208 ENDIF
209 DO 410 I = J-MYNUM, tempmax, -NP
210 Y(I) = Y(I) - A((BetaP*I)+J-I+1)*X(J)
211 410 CONTINUE
212 CALL CFlock(ICFret,1,'NUMLCK')
213 NUMDONE = NUMDONE + 1
214 CALL CFUnlock(ICFret,1,'NUMLOCK')
215 100 CONTINUE
216 RETURN
217 END

Appendix 5: PISCES message passing version

1 tasktype chol
2 integer N, M, P, Beta, BetaP
3 C N is the matrix size, M is the max number of columns per proc
4 C P is the max number or processors, Beta is the semi-bandwidth,
5 C BetaP is Beta + 1
6 integer MaxN, MaxBetaP
7 C MaxN is the max matrix size, MaxBetaP is the max semi-bandwidth
8 parameter (MaxN=305)
9 parameter (MaxBetaP=80)
10 parameter (P=25)
11 * M should be at least as great as N/P
12 parameter (M=305)
13 integer colasgn(P,M)
14 C colasgn is the array of which columns a processor owns
15 integer numcols(P)
16 C numcols is the number of columns owned by each processor
17 taskid tasknum(P)
18 C the array of task id’s
19 common /blk/tasknum(P)
20 handler getid
21 real X(MaxN)
22 handler mnewx
23 integer xok
24 common /blk1/xok
25 real Curx
26 common /blk2/Curx
27 real A(MaxN,MaxBetaP)
28 C A is the matrix
29 common /result/A(MaxN,MaxBetaP)
30 real B(MaxN)
31 C B is the right hand side
32 common /rhs/B(MaxN)
33 integer owner(MaxN)
the array of who owns each column

Generate test matrices

CALL SETCPU(1)
open(unit=2,file='/usr/u1/mchol3/param.dat')
READ(2,500) N
print *, ' N = ',N
READ(2,500) Beta
print *, ' Beta = ',Beta
500 FORMAT(I4)
BetaP = Beta + 1
do 10 i = 1,N
   A(i,1) = Beta * 4.0
   do 20 j = 2,Beta+1
      A(i,j) = -1.0
20  continue
10  continue

Make the assignment of columns to tasks

clust=pppcmin0
numclust = 0
do 50 i = 1,100000
   numcols(clust) = 0
   clust = pppcnxt(clust)
   numclust = numclust + 1
50 if (clust.eq.pppcmin()) goto 55
      continue
55 myclust = pppgclu (pppself)
clust = pppcmin0
   do 60 i = 1,N
      clust = pppcnxt(clust)
      numcols(clust) = numcols(clust) + 1
      colasgn(clust,numcols(clust)) = i
      owner(i) = clust
56 60 continue
57 * Make the assignment of tasks to clusters
58 *
59 clust=pppcmin0
60   numclust = 0
61   do 70 i = 1,100000
62      numcols(clust) = 0
63      clust = pppcnxt(clust)
64      numclust = numclust + 1
65      if (myclust.eq.clust) clust = pppcnxt (clust)
66      owner(i) = clust
67      clust = pppcnxt(clust)
68 70 continue
70 *
* Make the assignment of tasks to clusters
71 clust=pppcmin0
72   do 70 i = 1,100000
73      if (myclust.eq.clust) clust = pppcnxt (clust)
74      if (myclust.eq.clust) clust = pppcnxt (clust)
75      on cluster(clust) initiate colsrv (N,Beta,numclust, pppv1 (numcols, clust, clust),
76 & pppm1 (colasgn, P, M, clust, clust, 1, M),
77 & pppv1 (owner,1,N))
89  clust = pppcnxt(clust)
90  if (clust.eq.pppcmin()) goto 75
91 70  continue
92 75  continue
93 *
94 * Get the taskid of every task
95 *
96  accept numclust-1 of getid
97  endaccept
98 *
99 * Send the collection of taskid's to every task
100  to all send allids(pppv1(tasknum,1,P))
101 *
102 * Send the columns that are assigned to each processor to that processor
103 *
104  clust=pppcmin()
105  do 80 i = 1, 100000
106  if (myclust .eq. clust) clust = pppcnxt (clust)
107  do 90 j = 1, numcols(clust)
108  to tasknum(clust) send incol
109  & (j,BetaP,pppm1(A,MaxN,MaxBetaP,colasgn(clust,j),
110  & colasgn(clust,j),1,BetaP))
111 90  continue
112  clust = pppcnxt(clust)
113  if (clust.eq.pppcmin()) goto 85
114 80  continue
115 85  continue
116 *
117 * Wait for results to come back
118 *
119  accept N of newcol
120  endaccept
121 *
122 * Initialize the RHS to all 1's
123 *
124  do 120 i=1,N
125  B(i) = 1.0
126 120  continue
127 *
128 * Start the forward solve
129 *
130  do 130 i=1,N
131  to tasknum(owner(i)) send bval(B(i))
132 130  continue
133 *
134 * Start back solve
135 *
136  accept n of mnewx
137  endaccept
138 *
144 * print the solution vector
145 * 
146     do 450 i=1,N
147     WRITE (8,650) i,X(i)
148 450  continue
149 650   FORMAT(' X(',14,') = ','E13.6)
150 terminate
151 end
152
153 * HANDLER: Store the taskid in the array
154 *
155     handler getid (index, tasknum(index))
156     integer index
157     integer P
158     parameter (P=25)
159     taskid tasknum(P)
160     common /tblk/tasknum(P)
161     enddeclarations
162     return
163     end
164
165 * HANDLER: Store the incoming column in the array
166 *
167     handler newcol (col, BetaP,
168       & integer MaxN, MaxBetaF', BetaP)
169     integer MaxN, MaxBetaP, BetaP
170     parameter (MaxN=305)
171     parameter (MaxBetaP=80)
172     real A(MaxN,MaxBetaF')
173     common /result/A(MaxN,MaxBetaP)
174     integer col
175     enddeclarations
176     return
177     end
178 *
179 * factorization, back solve and forward solve task
180 *
181     tasktype colsrv (N, Beta, numclust, numcols,
182       & pppv1 (mycols, 1, M), pppv1(owner, 1, N))
183 * These parameter must match that in the chol tasktype definition
184     integer M, BetaP
185     integer MaxBetaP
186     parameter (M=305)
187     parameter (MaxBetaP=80)
188     integer P
189     parameter (P=25)
190     integer MaxN
191     parameter (MaxN=305)
192     integer owner(MaxN)
193     integer numclust
194     integer N, Beta, numcols
195     common /mblk1/ numcols
196     integer mycols(M)
197     common /mblk2/ mycols(M)
198     handler incol
handler pivot
real Amine(M,MaxBetaP)
common /blk1/Amine(M,MaxBetaP)
real piv(MaxBetaP)
common /blk2/piv(MaxBetaP)
integer pivnum
common /blk3/pivnum
integer curk
integer curin
common /blk5/curin
real Ymine(M)
common /blk4/Ymine(M)

real sum
integer sent
integer k,s,i
handler bval
handler bup
handler newx
integer xok
common /bblk1/xok
real Curx
common /bblk2/Curx
real B(M)
common /blk2/ B(M)
integer bcount(M)
common /blk1/ bcount(M)
taskid tasknum(P)
common /blk/tasknum(P)
handler allids
enddeclarations

BetaP = Beta + 1
myclust = pppgclu (pppself)
Send my taskid to the parent
to parent send getid(myclust,pppself)
Accept the vector of taskids
accept 1 of
allids
endaccept
receive the columns that we are assigned
accept numcols of
incol
endaccept
Begin the factorization
myk=1
do 10 k=1,N
if I own column k then compute and broadcast the pivot
if (mycols(myk).eq.k) THEN
    Amine(myk,1)=sqrt(Amine(myk,1))
    do 20 s=2,(min(k+Beta,N)-k+1)
    Amine(myk,s)=Amine(myk,s)/Amine(myk,1)
    continue
  to all send pivot(mycols(myk), BetaP,
  pppm1(Amine,M,MaxBetaP,myk,myk,1,BetaP))
  to parent send newcol(mycols(myk), BetaP,
  pppm1(Amine,M,MaxBetaP,myk,myk,1,BetaP))
  do 30 s=1,BetaP
  p(s)=Amine(myk,s)
  continue
  myk = myk + 1
ELSE
  accept 1 of pivot
  endaccept
  if a pivot column is received out of order then
    send it back to myself and get another
    if (pivnum.ne.k) THEN
      to self send pivot(pivnum, BetaP,
      pppv1(piv,1,BetaP))
      goto 40
    ENDIF
    ENDIF
  update the rest of the columns that I own
  do 50 s = myk,numcols
  if ((mycols(s).gt.k).and.(mycols(s).le.min(k+Beta,N)))
  THEN
    do 60 i=1,min(Beta+k,N)-mycols(s)+1
    Amine(s,i)=Amine(s,i)-piv(i+mycols(s)-k)*
    piv(mycols(s)-k+1)
    continue
    ENDIF
  continue
  continue
* start forward solve (using inner product)
  curin = 1
  receive the right hand side values that I own
  accept numcols of
  bval
  endaccept
  do 90 i = 1, numcols
  bcount(i) = 0
  sum = 0
  sent = 0
  do 110 s = 1, numcols
  if ((mycols(s).ge.max(1,i-Beta)).and.(mycols(s).le.i))
  THEN
    sum = sum + Amine(s,i-mycols(s)+1)*Ymine(s)
ELSE if (mycols(s).eq.i) THEN
if I own this column then wait for everyone
to send me the updates and then compute Y(l)
b(s) = b(s) - sum
sent = 1
if (bcount(s).eq.numclust-2) goto 160
accept all of
bup
endaccept
goto 150
continue
Ymine(curin) = B(curin) / Amine(curin,1)
curin = curin + 1
ENDIF
continue
if (sent.eq.0) then
endif
to parent send fordon
to tasknum(owner(i)) send bup(i,sum)
continue
if (bcount(s).eq.numclust-2) goto 160
bUP
ENDIF
continue
start backsolve (using col-sweep)
curk=numcols
do 200 j=N,1,-1
if (curk.eq.0) goto 300
if (mycols(curk).eq.j) THEN
Curx=Ymine(curk)/Amine(curk,1)
to all send newx(j,Curx)
to parent send mnewx(j,Curx)
curk = curk - 1
ELSE
xok = j
accept 1 of
newx
endaccept
if (xok.ne.0) goto 210
ENDIF
do 220 s = curk, 1, -1
if (((mycols(s).lt.j).and.(mycols(s).ge.max(lj-Beta)))
THEN
Ymine(s) = Ymine(s) - Amine(sj-mycols(s)+l)*Curx
ENDIF
do 220
continue
terminate
end
HANDLER: receive a column and place it in Amine
handler incol(col, BetaP,
pppm1(Amine,M,MaxBetaP,col,col,1,BetaP))
integer M, BetaP
parameter (M=305)
364   integer MaxBetaP
365   parameter (MaxBetaP=80)
366   real Amine(M,MaxBetaP)
367   common /blk1/Amine(M,MaxBetaP)
368   integer col
369   enddeclarations
370
371   return
372
373   HANDLER: receive a pivot column and place it in piv
374   handler pivot(pivnum, BetaP, pppv1(piv,1,BetaP))
375   integer BetaP
376   integer MaxBetaP
377   parameter (MaxBetaP=80)
378   real piv(MaxBetaP)
379   common /blk2/piv(MaxBetaP)
380   integer pivnum
381   common /blk3/pivnum
382   enddeclarations
383
384   return
385
386   HANDLER: take in the updated bval
387   handler bval (B(curin))
388   integer M
389   parameter (M=305)
390   real B(M)
391   common /blk1/B(M)
392   integer curin
393   common /blk5/curin
394   enddeclarations
395
396   curin = curin + 1
397
398   return
399
400   HANDLER: take in the new x and place in x(row) for main task
401   handler mnewx (row,X(row))
402   integer row
403   parameter (MaxN=460)
404   real X(MaxN)
405   common /blk1/X(MaxN)
406   enddeclarations
407
408   return
409
410   end
411
412   HANDLER: take in the new x
413   handler newx (row,Curx)
414   integer row
415   integer xok
416   common /bblk1/xok
417   real Curx
if (row.eq.xok) THEN
  xok = 0
ELSE
  to self send newx(row, Curx)
ENDIF

end

* HANDLER: Update the rhs

handler bup (row, bval)
integer MaxN,M
parameter (M=305)
parameter (MaxN=305)
integer row
real bval
real B(M)
common /fblkl/ BO
integer bcount0
common /fbW bcount(M)
integer numcols
common /mblkl/ numcols
integer mycols(M)
common /mblk2/ mycols(M)
enddeclarations

do 10 i= 1, numcols
  if (row.eq.mycols(i)) goto 15
  continue
10 continue
B(i) = B(i) - bval
bcount(i) = bcount(i) + 1
return
end

* HANDLER: accepts vector of taskids

handler allids (pppvl(tasknum,1,P))
integer P
parameter (P=25)
taskid tasknum(P)
common /tblk/tasknum(P)
enddeclarations
return
end

Appendix 6: PISCES FORCE version

tasktype chol
INTEGER I,J,K,S
INTEGER tempmin
INTEGER tempmax
shared
INTEGER Beta,BetaP,N
C Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
REAL A(20000)
C A contains the matrix
REAL RHS(1000)
C RHS is the right-hand side vector
REAL Y(1000)
C The Y vector in the forward solve
REAL X(1000)
C X is the solution vector
INTEGER STARTTIME,ENDTIME

common /blk1/Beta,BetaP,N,A
common /bLk2/RHS,Y,X,STARlTIME,ENDTIME
end shared
end declarations
forcesplit

DO 100 K= 1,N
C Compute the first element of the pivot column
barrier

A((K*BetaP)+1) = sqrt(A((K*BetaP)+1))
end barrier

tempmin = min(K+Beta,N)
C Compute the rest of the pivot column
presched do 110 S = K+1, tempmin
A((K*BetaP)+S-K+1) = A((K*BetaP)+S-K+1) / A((K*BetaP)+1)
CONTINUE

C Update the rest of the columns
presched do 120  J = K+1, tempmin
Do 130  I = J, tempmin
A((J*BetaP)+I-J+1) = A((J*BetaP)+I-J+1)
CONTINUE

CONTINUE

CONTINUE

CONTINUE

CONTINUE
The forward solve (using col-sweep)

```fortran
DO 300 J = 1, N  
  barrier  
  Y(J) = RHS(J) / A((BetaP*J)+1)  
  end barrier  
  tempmin = min(J+Beta,N)  
  presched do 310 I = J+1, tempmin  
  RHS(I) = RHS(I) - A((BetaP*J)+1-J+1)*Y(J)  
  310 CONTINUE  
  300 CONTINUE
```

The backward solve (using col-sweep)

```fortran
DO 400 J = N, 1, -1  
  barrier  
  X(J) = Y(J) / A((BetaP*J)+1)  
  end barrier  
  tempmax = max(J-Beta,1)  
  presched do 410 I = J-1, tempmax, -1  
  Y(I) = Y(I) - A((BetaP*I)+J-I+1)*X(J)  
  410 CONTINUE  
  400 CONTINUE
```

Print out the solution vector

```fortran
DO 500 J = 1, N  
  barrier  
  WRITE(8,680) J, X(J)  
  500 CONTINUE
```

```fortran
WRITE(8,680)  
680 FORMAT(' X('J4,)=',6E13.6)
```

terminate
<table>
<thead>
<tr>
<th>Language</th>
<th>PE’s</th>
<th>Fact</th>
<th>Forw</th>
<th>Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force (Local)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>1</td>
<td>201.80</td>
<td>6.00</td>
<td>7.38</td>
</tr>
<tr>
<td>&quot;</td>
<td>2</td>
<td>105.00</td>
<td>3.06</td>
<td>3.76</td>
</tr>
<tr>
<td>&quot;</td>
<td>4</td>
<td>56.62</td>
<td>1.66</td>
<td>2.00</td>
</tr>
<tr>
<td>&quot;</td>
<td>8</td>
<td>33.38</td>
<td>1.06</td>
<td>1.22</td>
</tr>
<tr>
<td>&quot;</td>
<td>16</td>
<td>21.04</td>
<td>0.94</td>
<td>1.04</td>
</tr>
<tr>
<td>Force (Shared)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>1</td>
<td>239.94</td>
<td>3.04</td>
<td>3.14</td>
</tr>
<tr>
<td>&quot;</td>
<td>2</td>
<td>121.82</td>
<td>1.62</td>
<td>1.68</td>
</tr>
<tr>
<td>&quot;</td>
<td>4</td>
<td>62.22</td>
<td>0.94</td>
<td>0.98</td>
</tr>
<tr>
<td>&quot;</td>
<td>8</td>
<td>32.56</td>
<td>0.68</td>
<td>0.72</td>
</tr>
<tr>
<td>&quot;</td>
<td>16</td>
<td>18.46</td>
<td>0.92</td>
<td>0.94</td>
</tr>
<tr>
<td>Concur (Local)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>1</td>
<td>180.52</td>
<td>8.46</td>
<td>7.28</td>
</tr>
<tr>
<td>&quot;</td>
<td>2</td>
<td>96.22</td>
<td>7.44</td>
<td>5.86</td>
</tr>
<tr>
<td>&quot;</td>
<td>4</td>
<td>55.66</td>
<td>10.36</td>
<td>7.56</td>
</tr>
<tr>
<td>&quot;</td>
<td>8</td>
<td>41.44</td>
<td>17.62</td>
<td>14.14</td>
</tr>
<tr>
<td>&quot;</td>
<td>16</td>
<td>44.28</td>
<td>30.88</td>
<td>28.08</td>
</tr>
<tr>
<td>Concur (Shared)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>1</td>
<td>217.70</td>
<td>5.76</td>
<td>5.98</td>
</tr>
<tr>
<td>&quot;</td>
<td>2</td>
<td>116.76</td>
<td>6.04</td>
<td>6.10</td>
</tr>
<tr>
<td>&quot;</td>
<td>4</td>
<td>68.04</td>
<td>7.38</td>
<td>7.50</td>
</tr>
<tr>
<td>&quot;</td>
<td>8</td>
<td>49.26</td>
<td>17.48</td>
<td>17.50</td>
</tr>
<tr>
<td>&quot;</td>
<td>16</td>
<td>51.20</td>
<td>30.96</td>
<td>30.98</td>
</tr>
<tr>
<td>Pisces (Message-passing)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>1</td>
<td>287.22</td>
<td>14.38</td>
<td>9.44</td>
</tr>
<tr>
<td>&quot;</td>
<td>2</td>
<td>161.04</td>
<td>12.74</td>
<td>7.34</td>
</tr>
<tr>
<td>&quot;</td>
<td>4</td>
<td>97.78</td>
<td>7.80</td>
<td>5.10</td>
</tr>
<tr>
<td>&quot;</td>
<td>8</td>
<td>67.50</td>
<td>5.36</td>
<td>4.70</td>
</tr>
<tr>
<td>&quot;</td>
<td>16</td>
<td>55.14</td>
<td>4.18</td>
<td>5.20</td>
</tr>
<tr>
<td>Pisces (Force)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>1</td>
<td>225.50</td>
<td>3.04</td>
<td>3.08</td>
</tr>
<tr>
<td>&quot;</td>
<td>2</td>
<td>114.24</td>
<td>1.68</td>
<td>1.70</td>
</tr>
<tr>
<td>&quot;</td>
<td>4</td>
<td>58.72</td>
<td>1.12</td>
<td>1.16</td>
</tr>
<tr>
<td>&quot;</td>
<td>8</td>
<td>30.80</td>
<td>0.86</td>
<td>0.86</td>
</tr>
<tr>
<td>&quot;</td>
<td>16</td>
<td>17.56</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td>Lines</td>
<td>Words</td>
<td>Chars</td>
<td>Program</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>--------------------------</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>305</td>
<td>2846</td>
<td>Force (Local)</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>170</td>
<td>1525</td>
<td>Force (Shared)</td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>649</td>
<td>6460</td>
<td>ConCurrent (Local)</td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>501</td>
<td>4784</td>
<td>ConCurrent (Shared)</td>
<td></td>
</tr>
<tr>
<td>344</td>
<td>859</td>
<td>9706</td>
<td>Pisces (Message-passing)</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>157</td>
<td>1572</td>
<td>Pisces (Force)</td>
<td></td>
</tr>
</tbody>
</table>
FLEX/32 20-PROCESSOR CONFIGURATION

Arbitration and locks

Shared memory 4.5 MB

Common bus

1 2 3 4 5 6 7 8 9 10

UNIX

11 12 13 14 15 16 17 18 19 20

Local memory 4 MB / processor

Local busses
Choleski's method for solving banded symmetric, positive definite systems is implemented on a multiprocessor computer using three FORTRAN based parallel programming languages, the Force, PISCES and Concurrent FORTRAN. The capabilities of the language for expressing parallelism and their user friendliness are discussed, including readability of the code, debugging assistance offered, and expressiveness of the languages. The performance of the different implementations is compared. It is argued that PISCES, using the Force for medium-grained parallelism, is the appropriate choice for programming Choleski's method on the multiprocessor computer, Flex/32.