ICASE

A LANGUAGE COMPARISON FOR SCIENTIFIC COMPUTING ON MIMD ARCHITECTURES

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

Contract Nos. NAS1-18107, NAS1-18605
January 1989

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

Operated by the Universities Space Research Association

NASA
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665
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 share/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 | Version B
---|---
sum = sum + 1 | sum = sum + 1
LOCK(sumlock) | 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 $L^Tx = y$ 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 $\min(k + \beta, N)$

\[ l_{ss} = a_{ss}/l_{kk} \]

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.

\[
\text{The Inner Product (ij) Algorithm for } Ly = b
\]

for $i = 1$ to $n$

for $j = \max(i - \beta, 1)$ to $i - 1$

\[ b_i = b_i - l_{ij}y_j \]

\[ y_i = b_i / l_{ii} \]

\[
\text{Column Sweep (ji) Algorithm for } LTx = y
\]

for $j = n$ to $1$

\[ x_j = y_j / l_{jj} \]

for $i = j - 1$ to $\max(j - \beta, 1)$

\[ y_i = y_i - l_{ij}x_j \]

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 $LTx = y$ 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 expressibility of functional parallelism and data partitioning, support for communication and synchronization, ease of learning the language, ease of converting existing programs, readability 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(1) will be executed by processor number PROCNUM(1) 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 communicated 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 “handlers” 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 ConCurrent 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-CFunlock 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 it's 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 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 in 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',I4,'.')
41  525 FORMAT(I4)
42
43  End Barrier
44  C Transfer the matrix from Shared memory to local memory
45  LCol = 1
46  Presched DO 700 I = 1, N
47     DO 710 J = 1, BetaP
48        A((LCol*BetaP)+J) = TempA((I*BetaP)+J)
49  710 Continue
50  Assign(LCol) = I
51  LCol = LCol + 1
52  End Presched DO
53  NumCols = LCol - 1
54  700
Start the choleski factorization loop

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

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

DO 110 S = K + 1, tempmin

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

C

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)

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)

130 Continue

END IF

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

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

END IF

310 Continue

Update the RHS

Critical UPDTRH

RHS(I) = RHS(I) - PriSUM

End Critical

IF (I.le.Assign(LCol)) THEN

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

LCol = LCol + 1

END IF

Barrier

Y(I) = RHS(I) / CurDiv
END Barrier

Backward Solve (using col-sweep)

DO 400 J = N, 1, -1

IF (J.eq.Assign(LCol)) THEN
  X(J) = Y(J) / A((BetaP*LCol)+l)
  LCol = LCol - 1
ELSE IF (LCol.eq.0) LCol = 1
END IF

END Barrier

DO 410 L2Col = NumCols, 1, -1

I = Assign(L2Col)

IF ((I.le.J-1).and.(I.ge.tempmax)) THEN
  END IF
ELSE
  CONTINUE
  Y(I) = Y(I) - A((BetaP*L2Col)+J-I+l)*X(J)
  CONTINUE
END IF

Print the solution vector

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

END

Appendix 2: Force - shared 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 IJ,K,S
5 Shared INTEGER tempmin
6 Private INTEGER tempmax
7 Shared REAL A(100000)
8 C A contains the matrix
9 Shared REAL RHS(2000)
10 C RHS is the right-hand side vector
11 Shared REAL Y(2000)
12 C The Y vector in the forward solve
13 Shared REAL X(2000)
14 C X is the solution vector
15 End declarations
16
17 Barrier
18 C Decide whether to read in or build the matrix
19 READ(9,525) I
20 IF (I.eq.0) THEN
21 CALL INITMAT(A,RHS,N,Beta,BetaP)
22 ELSE
23 CALL CRMAT(A,RHS,N,Beta,BetaP)
24 END IF
25 WRITE(6,600) N, Beta
600 FORMAT(' Order',I4,' matrix with a semi-bandwidth',I4,'.')
525 FORMAT(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
 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)+J-K+1)*A((K*BetaP)+J-K+1)
 CONTINUE
 End Presched DO
100 CONTINUE

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
300 CONTINUE

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*I)+J-I+1)*X(J)
 End Presched DO
410 CONTINUE
CONTINUE
Print out the solution vector
CONTINUE
END

Appendix 3: Concurrent FORTRAN - local memory version

PROGRAM MAIN
Shared INTEGER /labell/ PRCNUM(20)
PRCNUM holds the physical proc number corresponding to
the logical proc number
Shared INTEGER /label2/ NP
NP is the number of processors
Shared INTEGER /label3/ NUMDONE
Shared INTEGER /label4/ Beta,BetaP,N
Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
Shared INTEGER /label5/ NUMCOLS
Shared REAL /label6/ TempA(30000)
REAL A(30000)
A contains the matrix
common /pblk1/ A(30000)
INTEGER ASSIGN(S00)
Assign contains the list of columns that each processor owns
common /pblk2/ ASSIGNED(500)
INTEGER NUMCOLS
numcols is the number of columns that a processor owns
common /pblk3/ NUMCOLS
Shared REAL /label12/ CurL(2000)
the current pivot column
Shared REAL /label17/ RHS(2000)
RHS is the right-hand side vector
Shared REAL /label18/ Y(2000)
The Y vector in the forward solve
Shared REAL /label19/ X(2000)
X is the solution vector
Shared CHARACTER /label11/ NUMLCK
EXTERNAL LOADC
EXTERNAL ELCOL
EXTERNAL FORW
EXTERNAL BACK
INTEGER PID(20)
INTEGER I
INTEGER ICFret
Allocate a lock
CALL CFgetl(ICFret,'NUMLCK')

open(unit=2,cpu=1,file='/usr/u1/mj/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) = 1 + 2
15 CONTINUE

C Decide whether to read in or build the matrix
READ(2,525) I
IF (I.le.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)

C 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
C 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
C Start the forward solve processes on each processor
NUMDONE = 0
COBLOCK
   DO 160 I = 1, NP
      PROCESS(PID(i),FORWO(),PRCNUM(I))
160 CONTINUE
END COBLOCK

PIVCOL = N + 1
C 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
print the solution vector
DO 500 J = 1, N
WRITE(8,680) J, X(J)
CONTINUE
FORMAT(' X(',I4,') = ',6E13.6)
CALL CFkill(ICFret,0)
END

private copies task
SUBROUTINE LOADCO
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
  CONTINUE
10 CONTINUE
RETURN
END

factorization task
SUBROUTINE ELCOLQ
INTEGER K,I,J,S
INTEGER /label21/ NP
INTEGER /label31/ NUMDONE
INTEGER /label4/ Beta,BetaP,N
INTEGER /label51/ PIVCOL
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)
INTEGER MYP1V, MYP1V2
INTEGER tempmin
REAL A(30000)
common /pblkl/ A(30000)
INTEGER ASSIGN(500)
common /pblk2/ ASSIGN(500)
INTEGER NUMCOLS
common /pblk3/ NUMCOLS
Shared REAL /label12/ CurL(2000)
DO 100 K = 1, N
   tempmin = min(K+Beta,N)
   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
      WHEN (PIVCOL.eq.K) CONTINUE
      ENDIF
   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) - CurL(I-K+1)*CurL(J-K+1)
         CONTINUE
      ENDIF
   CONTINUE
   CALL CFlock(ICFret,1,'UMLCK')
   NUMDONE = NUMDONE + 1
   CALL CFulck(ICFret,1,'UMLCK')
100 CONTINUE
RETURN
END

SUBROUTINE FORW()
INTEGER 1J
REAL A(30000)
common /blk1/ A(30000)
INTEGER ASSIGN(500)
common /blk2/ ASSIGN(500)
INTEGER NUMCOLS
common /blk3/ NUMCOLS
Shared INTEGER /label2/ NP
Shared INTEGER /label3/ NUMDONE
Shared INTEGER /label4/ Beta,BetaP,N
Shared INTEGER /label5/ PIVCOL
Shared REAL /label7/ RHS(2000)
Shared REAL /label8/ Y(2000)
Shared CHARACTER /label11/ NUMLCK
INTEGER ICFret
INTEGER MYPIV, MYPIV2
INTEGER tempmax

MYPIV = 1
DO 100 I = 1, N
   tempmax = max(I-Beta,l)
   C 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.l)) 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(I)
   IF (I.eq.ASSIGN(MYPIV)) THEN
      WHEN (NUMDONE.eq.NP) CONTINUE
      Y(I) = RHS(I) / 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 /pblk1/ 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
   I = 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
110 CONTINUE
100 CONTINUE
RETURN
END

Appendix 4: ConCurrent FORTRAN - shared memory version

PROGRAM MAIN
   Shared INTEGER /label1/ PRCNUM(20)
   PRCNUM holds the physical proc number corresponding the
   Shared INTEGER /label2/ NP
   NP is the number of processors
   Shared INTEGER /label3/ NUMDONE
   Shared INTEGER /label4/ Beta,BetaP,N
   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)
   RHS is the right-hand side vector
   Shared REAL /label8/ Y(2000)
   The Y vector in the forward solve
   Shared REAL /label9/ X(2000)
   X is the solution vector
   Shared CHARACTER /label10/ NUMLCK
   EXTERNAL ELCOL
   EXTERNAL FORW
   EXTERNAL BACK
   INTEGER PID(20)
   INTEGER I
   INTEGER ICFret
   INTEGER tempmax, tempmin
   Allocate a lock
   CALL CFgetl(ICFret,"NUMLCK")
   open(unit=2,cpu=1, file='/usr/mlj/concur/choleski/param.dat')
   Read in the number of processors
   READ(2,525) NP
   PRINT *, 'Using ', NP, ' processors'
   DO 15 I = 1, NP
      CALL...
PIRCNUM(I) = I + 2
CONTINUE

C Decide whether to read in or build the matrix
READ(2,525) 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

PIVCOL = 0
C Start the factorization processes on each processor
NUMDONE = NP
COBLOCK
   DO 150 I = 1, NP
       PROCESS(PID(i),ELCOL(),PRCNUM(I))
   CONTINUE
END COBLOCK

PIVCOL = 0
C Start the forward solve processes on each processor
NUMDONE = NP
COBLOCK
   DO 160 I = 1, NP
       PROCESS(PID(i),FORW(),PRCNUM(I))
   CONTINUE
END COBLOCK

PIVCOL = N + 1
C Start the back solve processes on each processor
NUMDONE = NP
COBLOCK
   DO 170 I = 1, NP
       PROCESS(PID(i),BACK(),PRCNUM(I))
   CONTINUE
END COBLOCK

C Print out the solution vector
DO 500 J = 1, N
   WRITE(8,680) J, X(J)
CONTINUE
500 FORMAT(' X(',I4,') = ',6E13.6)

CALL CFkill(ICFret,0)
END
C The factorization task
SUBROUTINE ELCOL()
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 I own the pivot column then compute it
   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 + NUMDONE
      PIVCOL = PIVCOL + 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
   CONTINUE
   CONTINUE
   CONTINUE
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 I am responsible for col J then compute Y(J)
IF (J.eq.MYPIV) THEN
    WHEN (NUMDONE.eq.NP) CONTINUE
    Y(J) = RHS(J) / A((BETA*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((BETA*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()
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 /label8/ Y(2000)
Shared REAL /label9/ X(2000)
Shared CHARACTER /label11/ NUMLCK
INTEGER ICFret
INTEGER MYPIV
INTEGER plself
INTEGER 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)
IF (J.eq.MYPIV) THEN
  WHEN (NUMDONE.eq.NP) CONTINUE
  X(J) = Y(J) / A((BetaP*J)+1)
  MYPIV = MYPIV - NP
  NUMDONE = 0
  PIVCOL = PIVCOL - 1
ELSE
  WHEN (PIVCOL.eq.J) CONTINUE
ENDIF
DO 410 I = J-MYNUM, tempmax, -NP
  Y(I) = Y(I) - A((BetaP*I)+J-I+1)*X(J)
CALL CFlock(ICFret,1,'NUMLCK')
NUMDONE = NUMDONE + 1
CALL CFulck(ICFret,1,'NUMLCK')
CONTINUE

RETURN
END

Appendix 5: PISCES message passing version

1    tasktype chol
2    integer N, M, P, Beta, BetaP
3     N is the matrix size, M is the max number of columns per proc
4     P is the max number or processors, Beta is the semi-bandwidth,
5     Beta is Beta + 1
6     integer MaxN, MaxBetaP
7     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     colasgn is the array of which columns a processor owns
15    integer numcols(P)
16     numcols is the number of columns owned by each processor
17    taskid tasknum(P)
18     the array of task id's
19    common /tblk/tasknum(P)
20    handler getid
21    real X(MaxN)
22    handler mnewx
23    integer xok
24    common /bblkl/xok
25    real Curx
26    common /bblk2/Curx
27    real A(MaxN,MaxBetaP)
28     A is the matrix
29    common /result/A(MaxN,MaxBetaP)
30    real B(MaxN)
31     B is the right hand side
32    common /rhs/B(MaxN)
33    integer owner(MaxN)
the array of who owns each column

signal fordon
integer numclust
integer clust
handler newcol
enddeclarations

* Generate test matrices

CALL SETCPU(1)
open(unit=2,file='/usr/u1/pisces/mchol3/param.dat')
READ(2,500) N
print *,' N = ',N
READ(2,500) Beta
print *,' Beta = ',Beta
500 FORMAT(I4)

A(i,j) = Beta * 4.0

do 20 j = 2,Beta+l
A(i,j) = -1.0
20 continue
10 continue

* Make the assignment of columns to tasks

clust=pppcmin()
numclust = 0
do 50 i = 1,100000
numcols(clust) = 0
clust = pppcnext(clust)
umclust = numclust + 1
if (clust.eq.pppcmin()) goto 55
50 continue
55 continue

myclust = pppgclu (pppsel)
clust = pppcmin()
do 60 i = 1,N
60 continue

myclust = pppgclu (pppsel)
clust = pppcmin()
do 60 i = 1,N
60 continue

* Make the assignment of tasks to clusters

clust=pppcmin()
do 70 i = 1,100000
70 continue

* Skip the cluster on which this task is running
if (myclust.eq.clust) clust = pppcnext (clust)
numcols(clust) = numcols(clust) + 1
colasgn(clust,numcols(clust)) = i
owner(i) = clust
clust = pppcnext(clust)
70 continue
clust = pppcnxt(clust)
if (clust.eq.pppcmin()) goto 75
continue
continue

* Get the taskid of every task
* accept numclust-1 of
gid
endaccept
* Send the collection of taskid's to every task
to all send allids(pppv1(tasknum,1,P))
* Send the columns that are assigned to each processor to that processor

clust=pppcmin()
do 80 i = 1, 100000
if (myclust .eq. clust) clust = pppcnxt (clust)
do 90 j = 1, numcols(clust)
to tasknum(clust) send incol
(j,BetaP,pppm1(A,MaxN,MaxBetaP,colasgn(clust,j),
colasgn(clust,j),1,BetaP))
continue
clust = pppcnxt(clust)
if (clust.eq.pppcmin()) goto 85
continue
continue

* Wait for results to come back
* accept N of
  newcol
  endaccept
* Initialize the RHS to all 1's
* do 120 i=1,N
  B(i) = 1.0
  continue
* Start the forward solve
* do 130 i=1,N
  to tasknum(owner(i)) send bval(B(i))
  continue
* Start back solve
* accept n of
  mnewx
  endaccept

*
print the solution vector

* handle getid (index, tasknum(index))
  integer index
  integer P
  parameter (P=25)
  taskid tasknum(P)
  common /tblk/tasknum(P)
enddeclarations

return
end

* HANDLER: Store the incoming column in the array
handler newcol (col, BetaP, MaxN, MaxBetaP, BetaP)
  integer MaxN, MaxBetaP, BetaP
  parameter (MaxN=305)
  parameter (MaxBetaP=80)
  real A(MaxN,MaxBetaP)
  common /result/A(MaxN,MaxBetaP)
  integer col
enddeclarations

return
end

* factorization, back solve and forward solve task

* tasktype colsrv (N, Beta, numclust, numcols, mycols, owner, M)
  These parameter must match that in the chol tasktype definition

integer M, BetaP
integer MaxBetaP
parameter (M=305)
parameter (MaxBetaP=80)
integer P
parameter (P=25)
integer MaxN
parameter (MaxN=305)
integer owner(MaxN)
integer numclust
integer N, Beta, numcols
common /mblkl/ numcols
integer mycols(M)
common /mblk2/ mycols(M)
handler incol

*
handler pivot
real Amine(M,MaxBetaP)
Amine contains the columns that this processor owns
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 Yminec(M)
Ymine contains the Y values that this processor owns
common /blk4/Ymine(M)
real sum
integer sent
integer k,s,i
handler bval
handler bup
handler newx
integer xok
common /blk1/xok
real Curx
common /blk2/Curx
real B(M)
B contains the right hand side values this processor owns
common /blk1/ B(M)
integer bcount(M)
bcount contains the number of updates to B(i) received
common /blk2/ 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
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)
  continue
  to all send pivot(mycols(myk), BetaP,
  pppml(Amine,M,MaxBetaP,myk,myk,1,BetaP))
  to parent send newcol(mycols(myk), BetaP,
  pppml(Amine,M,MaxBetaP,myk,myk,1,BetaP))
  do 30 s=1,BetaP
  piv(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))
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)))
  piv(s)=Amine(myk,s)
  continue
ENDIF
continue
continue
continue
continue
continue
continue
continue
continue
continue
continue
continue
continue
curin = 1
receive the right hand side values that I own
accept numcols of
bval
endaccept
do 90 i = 1, numcols
bcount(i) = 0
continue
curin = 1
do 100 i = 1, N
  sum = 0
  sent = 0
  do 110 s = 1, numcols
     if ((mycols(s).gt.1-Beta).and.(mycols(s).le.i))
     THEN
     sum = sum + Amine(s,i-mycols(s)+1) * Ymin(s)
    ENDIF
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
  continue
  to tasknum(owner(i)) send bup(i,sum)
to parent send fordon
* 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
  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
  continue
  continue
  continue
  terminate
  end
  HANDLER receive a column and place it in Amine
  handler incol(col, BetaP,
  & ppm1(Amine,M,MaxBetaP,col,col,1,BetaP))
  integer M, BetaP
  parameter (M=305)
integer MaxBetaP
parameter (MaxBetaP=80)
real Amine(M,MaxBetaP)
common /blk1/Amine(M,MaxBetaP)
integer col
enddeclarations

return
end

HANDLER receive a pivot column and place it in piv
handler pivot(pivnum, BetaP, pppv1(piv,l,Be@))
integer BetaP
integer MaxBetaP
parameter (MaxBetaP=80)
real piv(MaxBetaP)
common /blk2/piv(MaxBetaP)
integer pivnum
common /blk3/pivnum
enddeclarations

return
end

HANDLER take in the updated bval
handler bval (B(curin))
integer M
parameter (M=305)
real B(M)
common /blk1/ B(M)
integer curin
common /blk5/curin
enddeclarations

curin = curin + 1
return
end

HANDLER: take in the new x and place in x(row) for main task
handler mnewx (row,X(row))
integer row
parameter (MaxN=460)
real X(MaxN)
common /blk1/ X(MaxN)
enddeclarations

return
end

HANDLER: take in the new x
handler newx (row,Curx)
integer row
integer xok
common /bblik/xok
real Curx
if (row.eq.xok) THEN
  xok = 0
ELSE
  to self send newx(row,Curx)
ENDIF
return
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
  continue
  if (row.eq.mycols(i)) goto 15
10 continue
B(i) = B(i) - bval
bcount(i) = bcount(i) + 1
return
end

HANDLER: accepts vector of taskids
handler allids (pppv1(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 /blk/l/Beta,BetaP,N,A
common /bLk2/RHS,Y,X,STARTTIME,ENDTIME
end shared
end declarations
forcesplit
barrier
C Decide whether to read in or build the matrix
CALL SETCPU(1)
open(unit=9,file='usr/u1/mj/pisces/fchol/param.dat')
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
end barrier
C Start the choleski factorization loop
DO 100 K= 1,N
C Compute the first element of the pivot column
barrier
   tempmin = min(K+Beta,N)
   A((K*BetaP)+1) = sqrt(A((K*BetaP)+1))
end barrier
 presched do 110 S = K+1, tempmin
    A((K*BetaP)+S-K+1) = A((K*BetaP)+S-K+1) / A((K*BetaP)+1)
 CONTINUE
110
C Compute the rest of the pivot column
 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+1)
 CONTINUE
120
 CONTINUE
100 CONTINUE
59 C The foward solve (using col-sweep)
60 DO 300 J = 1, N
61     barrier
62     Y(J) = RHS(J) / A((BetaP*J)+1)
63     end barrier
64     tempmin = min(J+Beta,N)
65     presched do 310 I = J+1, tempmin
66     RHS(I) = RHS(I) - A((BetaP*J)+I-J+1)*Y(J)
67     CONTINUE
68     300 CONTINUE
69 C The backward solve (using col-sweep)
70 DO 400 J = N, 1, -1
71     barrier
72     X(J) = Y(J) / A((BetaP*J)+1)
73     end barrier
74     tempmax = max(J-Beta,1)
75     presched do 410 I = J-1, tempmax, -1
76     Y(I) = Y(I) - A((BetaP*I)+J-I+1)*X(J)
77     CONTINUE
78     400 CONTINUE
79 C Print out the solution vector
80 DO 500 J = 1, N
81     barrier
82     WRITE(8,680) J, X(J)
83     500 CONTINUE
84 FORMAT(' X(',I4,') = ',6E13.6)
85 terminate
86 end
### Appendix 7

<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></td>
<td>1</td>
<td>201.80</td>
<td>6.00</td>
<td>7.38</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>105.00</td>
<td>3.06</td>
<td>3.76</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>56.62</td>
<td>1.66</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>33.38</td>
<td>1.06</td>
<td>1.22</td>
</tr>
<tr>
<td></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></td>
<td>1</td>
<td>239.94</td>
<td>3.04</td>
<td>3.14</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>121.82</td>
<td>1.62</td>
<td>1.68</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>62.22</td>
<td>0.94</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>32.56</td>
<td>0.68</td>
<td>0.72</td>
</tr>
<tr>
<td></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></td>
<td>1</td>
<td>180.52</td>
<td>8.46</td>
<td>7.28</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>96.22</td>
<td>7.44</td>
<td>5.86</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>55.66</td>
<td>10.36</td>
<td>7.56</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>41.44</td>
<td>17.62</td>
<td>14.14</td>
</tr>
<tr>
<td></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></td>
<td>1</td>
<td>217.70</td>
<td>5.76</td>
<td>5.98</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>116.76</td>
<td>6.04</td>
<td>6.10</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>68.04</td>
<td>7.38</td>
<td>7.50</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>49.26</td>
<td>17.48</td>
<td>17.50</td>
</tr>
<tr>
<td></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></td>
<td>1</td>
<td>287.22</td>
<td>14.38</td>
<td>9.44</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>161.04</td>
<td>12.74</td>
<td>7.34</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>97.78</td>
<td>7.80</td>
<td>5.10</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>67.30</td>
<td>5.36</td>
<td>4.70</td>
</tr>
<tr>
<td></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></td>
<td>1</td>
<td>225.50</td>
<td>3.04</td>
<td>3.08</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>114.24</td>
<td>1.68</td>
<td>1.70</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>58.72</td>
<td>1.12</td>
<td>1.16</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>30.80</td>
<td>0.86</td>
<td>0.86</td>
</tr>
<tr>
<td></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>
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.