FORTRAN for clusters of IBM ES/3090 multiprocessors

by R. J. Sahulka
E. C. Plachy
L. J. Scarborough
R. G. Scarborough
S. W. White

IBM Clustered FORTRAN is a combination of software and hardware that allows two IBM Enterprise System/3090™ (ES/3090™) multiprocessors to be physically connected as a cluster and allows FORTRAN jobs to execute in parallel across all of the processors of the cluster. The FORTRAN compiler and library provided as part of Clustered FORTRAN are used for writing and executing the parallel programs in this hybrid environment of distributed and shared-memory systems. The compiler provides language extensions for explicit programming in parallel, as well as the ability to automatically generate both parallel and vector code. The Clustered FORTRAN language allows users to write parallel applications that are independent of the machine configuration and operating system. This paper describes the execution environment, compiler, and library, gives some variations of programming matrix multiplication, and shows that performance of one GigaFLOPS can be achieved using Clustered FORTRAN.

As applications are constructed for parallel processing, there is a desire to apply more processors to the application. Clustered FORTRAN is a combination of hardware and software that allows the connection of two IBM Enterprise System/3090™ (ES/3090™) multiprocessors to form a cluster of processors for execution of FORTRAN programs. IBM ES/3090 multiprocessors are tightly-coupled, shared-memory multiprocessor systems that support up to six processors and share a global memory; each of these processors may be equipped with the Vector Facility feature.[1] VS FORTRAN's Multitasking Facility (MTF),[2] IBM Parallel FORTRAN,[3,4] and the parallel extensions in VS FORTRAN Version 2, Release 5[5] allow a single FORTRAN job to use all of the processors of a single IBM 3090 multiprocessor. Clustered FORTRAN extends the parallel processing capabilities to allow a single FORTRAN job to use all of the processors of two IBM 3090 multiprocessors. Clustered FORTRAN provides a high-speed connection between two IBM 3090 multiprocessors, control program extensions for job control and interprocessor communications, and a FORTRAN compiler and library that allow FORTRAN jobs to use all of the processors of the cluster.[6] These two IBM 3090 multiprocessors are distributed, since there is no shared memory between the two multiprocessors.

Clustered FORTRAN has been superseded by IBM Enhanced Clustered FORTRAN, announced in November 1990, which is an extension of Clustered FORTRAN. Enhanced Clustered FORTRAN supports clusters of up to four IBM ES/9000™ multiprocessors with an additional global storage ac-

cessible by each multiprocessor in the cluster. This paper describes the execution environment and compiler for clusters of two IBM ES/3090s using Clustered FORTRAN. However, many of the concepts apply to clusters of four IBM ES/9000 multiprocessors using Enhanced Clustered FORTRAN.

Clustered FORTRAN

Clustered FORTRAN builds on the function in Parallel FORTRAN and continues the evolution by providing additional enhancements, allowing programmers to exploit all of the processors in a distributed cluster of two IBM 3090 multiprocessors. Announced in May 1989, Clustered FORTRAN has been available on a limited basis since the second quarter of 1990. It was developed for the Virtual Machine/Extended Architecture™ Systems Product (VM/XA™ SP) operating system by IBM at the Data Systems Division in Kingston, New York, and the Palo Alto Scientific Center and Programming Systems Santa Teresa laboratory, in California.

The major reason for using Clustered FORTRAN is to reduce the time required to execute a FORTRAN program. The time reduction is achieved when multiple processors of one or more computers of a Clustered FORTRAN Complex simultaneously execute portions of a single application program. Parallel execution does not reduce the total number of CPU (central processing unit) cycles required to execute a program and, in fact, an increase in CPU cycles is normally required. Clustered FORTRAN allows a program to be split into multiple independent instruction streams. When these instruction streams are executed simultaneously by different processors on the same or different computers, the program utilizes cycles from each of the assigned processors. Thus the program executes more CPU cycles in a given span of real time, and it can complete its computation more quickly.

Different forms of parallelism can occur in a FORTRAN program. An application may have subroutines that can execute concurrently on different data. Loops may have iterations that can execute at the same time. Independent sequences of statements may be eligible for concurrent execution. Parallel work may occur nested within other parallel work. To accommodate these different forms of parallelism, Parallel FORTRAN has two shared-memory programming models: (1) parallelism where memory is shared by default, for fine-grained tasks that need to work on the same data, and (2) parallelism where memory is not shared by default, for coarse-grained tasks that are relatively independent and share no memory, or only specified parts of memory.

An important feature of Parallel FORTRAN is the separation of the specific execution environment from both the user's source and the object code

Clustered FORTRAN allows a program to be split into multiple independent instruction streams.

generated by the compiler. At execution time, a user specifies the number of processors for parallel execution. The user's execution environment can be viewed by the user as a virtual multiprocessor.

Clustered FORTRAN builds on the Parallel FORTRAN functions by extending the execution environment from a single, virtual, multiprocessor computer to multiple, virtual, multiprocessor computers. In addition to the two Parallel FORTRAN programming models for shared-memory mentioned above, Clustered FORTRAN supports distributed, nonshared memory parallelism where tasks communicate by sending and receiving data. The extensions over Parallel FORTRAN include:

- A high-speed connection that allows two IBM ES/3090 multiprocessors to be connected as a cluster
- Control program extensions for communication on the high-speed connection
- FORTRAN run-time options for specifying a virtual run-time configuration for executing the program
- System extensions to build and control the virtual configurations required by Clustered FORTRAN programs
- Additional language for parallel subroutine execution to allow the scheduling of parallel work across multiple, virtual computers
- Library routines for synchronizing parallel

pieces of work executing in the same or different virtual computers
- Several enhancements to the parallel language constructs introduced by Parallel FORTRAN

Clustered FORTRAN maintains the execution environment and functions of Parallel FORTRAN within a single, virtual, multiprocessor computer. These functions include:

- Automatic parallel execution for eligible DO loops
- Automatic integration of parallel and vector processing
- Language for parallel loop iterations
- Language for parallel statement sequences
- Language for parallel subroutine execution
- Library routines for synchronizing parallel pieces of work

This paper discusses the execution environment, compiler, and library provided by Clustered FORTRAN to support parallelism, both for a single, virtual multiprocessor and for multiple, virtual multiprocessors. The paper concludes with several examples of a matrix multiplication program for both single and multiple, virtual computer environments, and includes performance results for matrix multiplication using Clustered FORTRAN with two IBM ES/3090 Model 600J computers.

A user of Clustered FORTRAN specifies a virtual configuration that suits the application.

Clustered FORTRAN execution environment

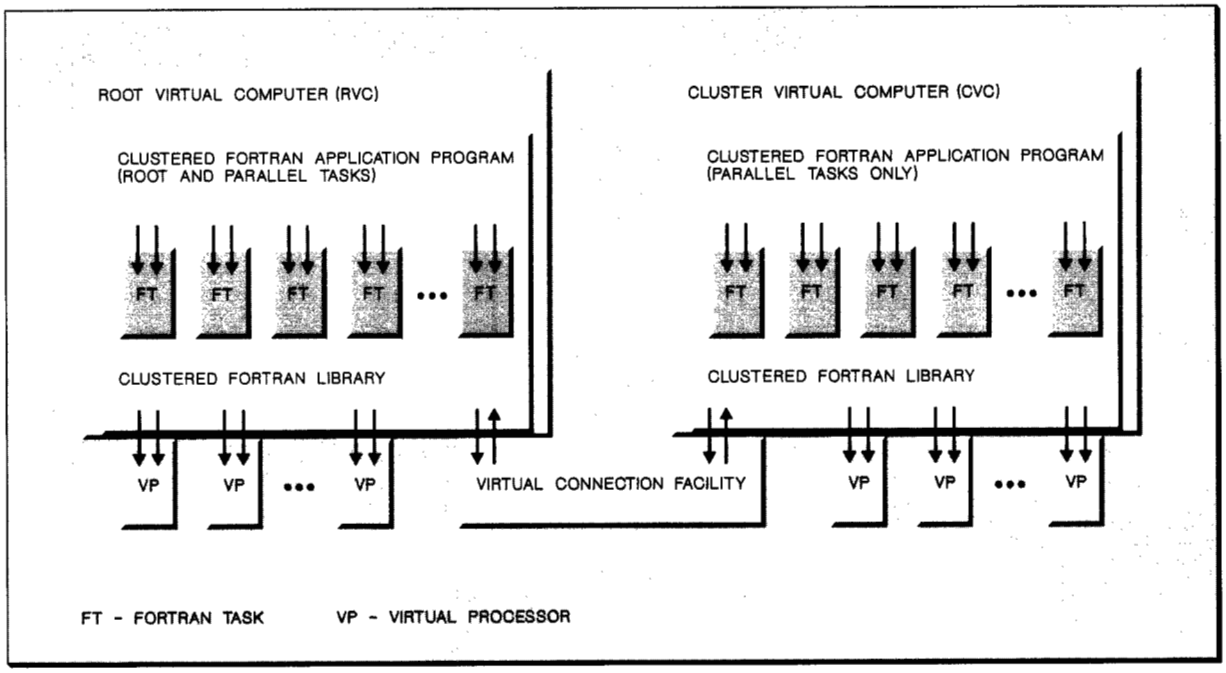
A Clustered FORTRAN Complex consists of two real computers grouped into a cluster by a high-speed connection facility. Each computer must be an IBM ES/3090 multiprocessor system with four, five, or six processors. Both of the computers of the complex must operate under VM/XA, and each computer must have the Clustered FORTRAN hardware and software installed.

A Clustered FORTRAN application sees a virtual configuration for its execution environment. The virtual configuration consists of one or more virtual computers that are connected together by a virtual connection facility. A virtual computer is mapped to a tightly-coupled multiprocessor. The virtual computers share no memory; this mimics the architecture of the collection of real computers in the complex. Communication between virtual computers is done by copying data. Each of the virtual computers may be a shared-memory multiprocessor; this mimics the architecture of any one of the real computers in the complex. A virtual computer is implemented as an IBM VM/XA virtual machine and the virtual processors in that computer are implemented as virtual CPUs defined in that virtual machine. A virtual processor is within a virtual computer and is mapped to one of the tightly-coupled multiprocessors by the operating system. A virtual configuration with two virtual multiprocessors for a Clustered FORTRAN application is shown in Figure 1.

A user of Clustered FORTRAN specifies a virtual configuration that suits the application at hand. A real cluster configuration consisting of two six-way multiprocessor systems might be thought of as 12 uniprocessor computers, as four computers each with three processors, as two computers each with six processors, or many other configurations. The number of virtual computers and processors can be greater than the number of real computers and processors. The virtual configuration is specified at execution time by a file containing COMPUTER statements, with one statement for each virtual computer desired. When a Clustered FORTRAN application is submitted for execution, the system extensions create the virtual configuration and map it onto the real configuration for the duration of the job. An important feature of Clustered FORTRAN is that the application source and the object code generated by the compiler are independent of the configuration used for execution. This means that an application does not have to be recompiled when the user changes the configuration file.

The FORTRAN main program obtains control after the virtual configuration has been built and initialized. This program runs as a task in the first virtual computer that is requested. This is known as the *root task* and this virtual computer is known as the *root virtual computer* (RVC). The other virtual computers, known as *cluster virtual*

Figure 1 A virtual configuration for a Clustered FORTRAN application



computers (CVCs), are fully initialized and waiting for work.

To achieve parallel execution among virtual computers (RVC and CVCs), a FORTRAN program must assign work to the CVCs. This is done by using Clustered FORTRAN's language for parallel tasks, which allows tasks running in one virtual computer to originate tasks in another virtual computer and then assign work to these remote tasks. Additional parallelism can be gained within a virtual computer by using Clustered FORTRAN's inline parallel constructs (automatic parallel DO loops, parallel loops, parallel cases) or by originating tasks in the same virtual computer and assigning work to them. Parallelism within a virtual computer allows a virtual computer, which was defined to be a multiprocessor, to use its multiple processors. The section on Clustered FORTRAN parallel functions explains the parallel functions provided by the Clustered FORTRAN compiler and library.

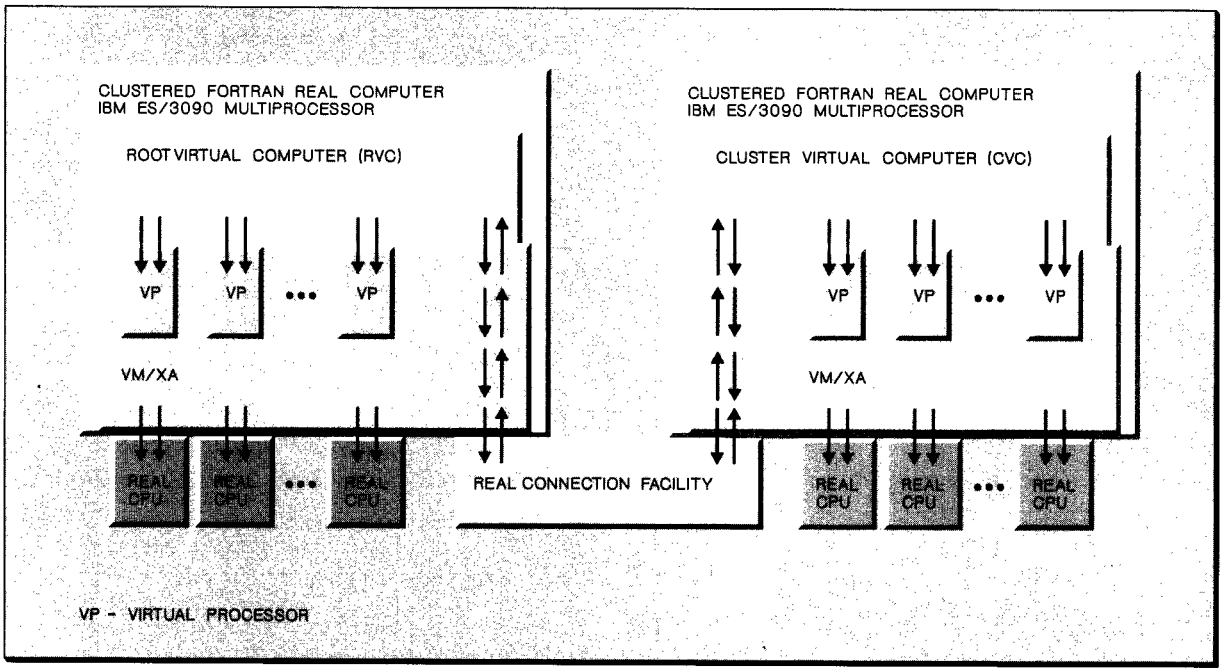
Figure 2 shows a Clustered FORTRAN Complex with two real computers. In this figure a Clustered FORTRAN job has the RVC of its virtual configuration executing in one of the real computers and a CVC executing in the other.

As can be seen from the preceding description, the Clustered FORTRAN language, compiler, and library are used to build a virtual configuration for a run-time execution environment, identify the parallel pieces of work, and map the parallel pieces of work onto the logical resources provided by that configuration. The underlying Clustered FORTRAN software support and the VM/XA operating system map the logical resources of the virtual configuration to the real resources of the Clustered FORTRAN Complex.

Four conditions are required for a Clustered FORTRAN program to be able to fully exploit the processing capabilities of a Clustered FORTRAN Complex. These conditions follow:

1. The virtual configuration for executing the program should define one or more virtual computers in each real computer of the complex.
2. The virtual computers in each real computer should be defined with sufficient virtual proc-

Figure 2 A Clustered FORTRAN Complex



essors (in total) to use all of the processors of the real computer.

3. The virtual computers in each real computer should have sufficient pieces of parallel work ready for execution to keep all of the processors of the real computer busy.
4. In each real computer of the complex, the virtual computers should be assigned a scheduling priority high enough to allow them to acquire real processors when needed.

The first two conditions can normally be satisfied by specifying an appropriate virtual configuration when the program is submitted for execution. Satisfying the third condition is very dependent on the nature of the program. Some programs may be able to satisfy this condition for almost the full duration of their execution; others may only be able to partially satisfy this condition; and yet others may not be suitable for use with Clustered FORTRAN.

The fourth condition depends on the scheduling policies set up for the Clustered FORTRAN Complex in which the program is run. In general, if a Clustered FORTRAN program is to see performance benefits over those it could see on a single IBM 3090 multiprocessor, it must be allowed the simultaneous use of more computational resources than it could receive in any one of the IBM 3090 multiprocessors of the cluster. (Without the simultaneous use of the resource, an application may perform better on a single IBM 3090 multiprocessor.)

Clustered FORTRAN parallel functions

Clustered FORTRAN supports a variety of programming styles. To express concurrent execution, a user can choose to specify a parallel SUBROUTINE, a parallel DO loop, or sections of parallel code. Synchronization can be accomplished by waiting for a task or by using locks, events, and synchronization counters. Communication is achieved by using shared variables or by copying COMMON blocks and SUBROUTINE arguments.

Parallel tasks. Clustered FORTRAN allows new and distinct FORTRAN execution environments to be created and permits the concurrent execution of these distinct environments. A short-hand name

for these environments is *tasks* and their use is referred to as out-of-line parallelism. Use of these out-of-line tasks allows a programmer to encapsulate a subroutine for parallel execution with varying and well-controlled degrees of sharing, and to treat the tasks as objects that maintain a set of data that is local and persistent to a task environment. A task can be created that communicates with the task that created it by using shared memory. Alternatively, a task can be created that shares no memory with the task that created it, and that communicates with it by sending data.

A Clustered FORTRAN program begins execution in the root task, which executes in a root virtual computer (RVC). The ORIGINATE statement may be used to create more tasks and to specify in which virtual computer they run. The BINDING clause in an ORIGINATE statement is used to specify the virtual computer that will contain a task. If the BINDING clause is omitted, then the task is originated in the same virtual computer as the originating task. Any task may originate a task in any virtual computer. The originated task has its storage allocated in a particular computer and it runs only in that computer. Each task has a global identifier returned to the program by the ORIGINATE statement. The global identifier can be used in subsequent statements to manipulate the task. When a task is no longer needed it can be deleted with a TERMINATE statement. The ORIGINATE and TERMINATE statements are illustrated in Figure 3. The first ORIGINATE statement in the figure causes a new task to be originated. The new task may share memory with and will reside in the same virtual computer as the originating task. The second and third ORIGINATE statements use the BINDING clause to specify that the task will not share memory with the originating task and to indicate the target virtual computer for the originated task. The virtual computer may or may not be the same virtual computer of the originating task. If the originated task is on a different virtual computer than its originating task, then these two tasks cannot share memory, and they communicate by sending data. These ORIGINATE statements show tasks being originated in specific virtual computers, the second relative computer in the virtual cluster and a virtual computer that was assigned the name "cvc01" in its COMPUTER statement. The use of a name provides flexibility in defining the virtual cluster for the program since virtual computers can be renamed in the COMPUTER statements.

Figure 3 ORIGINATE and TERMINATE statements

```
ORIGINATE ANY TASK itask
ORIGINATE ANY TASK itask, BINDING(2)
ORIGINATE ANY TASK itask, BINDING('cvc01')
TERMINATE TASK itask
```

Figure 4 Sample DISPATCH and SCHEDULE statements

```
 DISPATCH ANY TASK itask,
*  CALLING subnam(arg1,arg2, ...)
 SCHEDULE TASK itask,
*  CALLING subnam(=arg1=,=arg2,arg3=, ...)
```

Work is assigned to a task with the SCHEDULE and DISPATCH statements. These statements name a subroutine to be executed, asynchronously, in the invoked task and list the arguments to be passed to the called subroutine. The user may request that a specific task be used or may request that the library choose any available task. SCHEDULE allows user control over the detection of task completion and the assignment of new work. DISPATCH allows the run-time library to detect task completion and assign new work to tasks.

Arguments to the called subroutine may be passed by address or by copy. When the address is used, the same copy of the argument is shared between the scheduled task and the scheduling task. Addresses can be used to identify arguments only when the scheduled and scheduling tasks reside in the same virtual computer. When tasks reside in different virtual computers, a copy of the argument must be passed to the subroutine. When an argument is passed by copy, a copy is made of the value of the argument in the called task. Each task is executed with its own copy of the argument. Sample statements are shown in Figure 4. The CALLING clause in each of the three statements specifies the name of the subroutine to be executed asynchronously. The DISPATCH statement in the figure shows arguments being passed by address. The SCHEDULE statement shows the arguments being passed by copy. The "=arg=" means that the value of the argument is to be

Figure 5 SCHEDULE statement—Options allowed for tasks in the same or different virtual computers

```
SCHEDULE TASK itask,
*  COPYING (common, common, ...),
*  COPYINGI(common, common, ...),
*  COPYINGO(common, common, ...),
*  TAGGING (tagone, tagtwo, ...),
*  CALLING subnam(=arg1=,
*                 =arg2,arg3= ...)
```

Figure 6 SCHEDULE statement—Additional option allowed for tasks originated without BINDING

```
SCHEDULE TASK itask,
*  SHARING (common, common, ...),
*  CALLING subnam(arg1,arg2,
*                 arg3 ...)
```

copied into the scheduled task when work is assigned and copied out of the scheduled task when the work is completed. The "=arg" means the value is only copied into the scheduled task when work is assigned, and the "arg=" means the value is only copied out of the scheduled task when the work is completed. A variable, array element, full array, or array stripe may be passed by copy in either or both directions; an expression or constant may only be passed into the called subroutine. An array stripe is a section of an array that is contiguous in storage. It is specified by giving a lower bound and upper bound in one dimension of an array. For example, assuming an array specified by DIMENSION A(10,10,10), the notation A(:,2:6,3) specifies an array stripe with A(1,2,3) as its first element and A(10,6,3) as its last element.

Tasks, like subroutines in traditional FORTRAN, may communicate through common blocks of storage as well as arguments. Figures 5 and 6 show the optional clauses for a SCHEDULE statement. These clauses also apply to the DISPATCH statement. Figure 5 shows the optional clauses allowed in the SCHEDULE statement when the task being invoked is in the same or different virtual computers as the caller. Figure 6 shows the additional clause allowed in the SCHEDULE statement when the task being invoked is in the same virtual computer as the caller. The difference is simply that SHARING of common blocks may be specified only when the tasks are in the same virtual computer. A SHARING clause may be used to name the common blocks to be shared with the task selected to execute the subroutine. Shared common blocks are accessed in the same location by both tasks. The scheduled or dispatched task uses the same copy of the common block as the task that invoked it. A COPYING clause may be used to name common blocks that are to be copied into the invoked task when work is assigned and copied out of the task when work is completed and waiting. Both the invoked and the invoking tasks have a private copy of these common blocks. COPYINGI and COPYINGO name common blocks that are to be copied respectively only into, or only out of, the invoked task.

Figure 7 Example of WAIT FOR statements

```
WAIT FOR TASK itask,
*  TAGGING(varone,vartwo)
WAIT FOR ANY TASK itask,
*  TAGGING(varone,vartwo)
WAIT FOR ALL TASKS
```

Tasks can be assigned a variety of pieces of work. A TAGGING clause is provided to allow the program to name, or tag, a particular piece of work. The values of the tags are saved when the task is scheduled. When, subsequently, the program issues a wait for a task, the values of tags for the completing task may be retrieved. This makes it easy for the program to determine what specific piece of work was assigned to the task that was just completed.

The WAIT FOR statement is used to detect when a task has completed its assigned work. Three types of WAIT FOR statements are available: wait for a specific task, wait for any task, or wait for all tasks. Figure 7 shows the variations of the WAIT FOR statement, including its optional TAGGING clause.

In-line parallel function. In addition to the out-of-line extensions for parallel tasks, Clustered FORTRAN also provides in-line parallelism. In-line

parallelism permits the code within a subroutine to be dynamically parceled to more than one processor for execution. Each of the processors executes in its own logical environment. All the array and scalar data known to the subroutine are automatically shared. The sharing of data between these environments means that in-line parallelism is confined to the processors of the virtual computer in which the owning subroutine was executing. Processors may also name their own private data.

Clustered FORTRAN provides three forms of in-line parallel function: automatic parallelism, parallel loops, and parallel cases.

Automatic parallelism. Automatic parallelism is the simplest way to introduce in-line parallelism into an application. A compiler option requests that the compiler analyze nests of DO loops to determine if they are eligible for parallel execution. Parallel code is generated only if parallel execution would produce the same results as serial execution. An extension of the data-dependence algorithms used for vectorization determines whether loops, or selected statements within loops, may be executed in parallel. If there are no data dependences which prevent parallel execution, the compiler determines if it is cost-effective to execute the loop in parallel. If so, parallel code is generated for the loop; otherwise serial code is generated. Besides being a simple way to introduce in-line parallelism, automatic parallelism also allows a program to remain portable to other FORTRAN compilers.

The vector and parallel compiler options may both be specified. In this case, the compiler will analyze nests of DO loops for both parallel and vector execution. Individual loops may be selected for vector, for parallel, or for both vector and parallel execution. A loop selected for parallel execution may contain inner parallel or vector loops. A loop selected for vector execution may only contain inner scalar loops. If it is found to be cost-effective, a loop may be broken into two or more separate loops, each of which has the same induction parameters as the original. The new loops may be executed in different modes. Directives are provided for users to indicate a preference for parallel or serial code for a given loop as well as a preference for scalar or vector code. Thus the user can override the compiler's economic decisions for the type of code generated for a loop.

Although automatic detection of parallelism is an easy means to obtain in-line parallel execution, it does not provide a complete answer for in-line parallel programming. The primary reason for this is the requirement that parallel execution must produce the same results as serial execution. Some algorithms are able to run effectively in parallel even though they contain data dependences which may cause their results to change from run to run. An example is chaotic relaxation. The algorithms are designed to converge, and are deemed to be successful when they converge within some small tolerance. Any value that meets this criterion is acceptable. When using automatic parallelization, the same results must be produced as would be produced by the serial execution of the program.

Clustered FORTRAN provides three forms of in-line parallel function: automatic parallelism, parallel loops, and parallel cases.

Clustered FORTRAN provides language extensions with which the programmer may specify in-line parallel execution. These extensions, which define parallelism within a routine, identify loops or blocks of statements that can be executed concurrently.

Parallel loops. A parallel loop is one in which each iteration of the loop may be executed concurrently. Some number of processors, possibly one per iteration, may be used to execute the loop. The number of processors is not specified. It will be determined at run time and it can vary from one to the number of virtual processors associated with the virtual computer. The order in which iterations are executed is not guaranteed. However, all iterations are completed before execution continues beyond the end of the loop.

Figure 8 Simple form of PARALLEL LOOP

```
      PARALLEL LOOP 10 I = 1, 100, 4
         statement
            .
            .
            .
10    CONTINUE
```

Figure 9 Example of PRIVATE statement

```
      PARALLEL LOOP 10 I = 1, 100
         PRIVATE (XTEMP)
         XTEMP=A(I)*B(I)
         C(I)=XTEMP*D(I)
10    CONTINUE
```

The programmer is responsible for ensuring that the loop is valid for parallel execution. Normally, each iteration should be computationally independent of other iterations. Alternatively, the user can ensure that the proper synchronization is used between iterations or that the results are meaningful in the absence of such synchronization.

A PARALLEL LOOP has a syntax which is similar to a DO loop. A simple form is shown in Figure 8.

This simple form of a parallel loop permits iterations to execute in parallel. But suppose a programmer needs to compute a temporary result, such as XTEMP, within an iteration. A statement like XTEMP=A(I)*B(I) will not work in parallel. If several processors execute the statement simultaneously, each computing with a different value of I, because there is only one copy of XTEMP, the processors will try to update a single copy and incorrect behavior may result. Therefore, each processor needs its own private copy of XTEMP for the program to operate correctly. Such private variables may be declared with a PRIVATE statement, as shown in Figure 9.

Furthermore, given such private variables, it is sometimes desirable to initialize them before executing iterations of the loop, or to reference their final value after all loop iterations are complete. For this purpose, DOFIRST and DOFINAL statements are provided. These delimit, respectively, a prologue and epilogue block for the loop; they may specify, by a LOCK operand, that only one processor at a time is to be permitted to execute the prologue or epilogue. DOEVERY delimits the body of the loop that is executed on each iteration.

Figure 10 Extended form of PARALLEL LOOP

```
      GSUM=0
      PARALLEL LOOP 10 I = 1, 100
      PRIVATE (PSUM)
      DOFIRST
        PSUM=0
      DOEVERY
        PSUM=PSUM+AVAL(I)
      DOFINAL LOCK
        GSUM=GSUM+PSUM
10    CONTINUE
```

The example shown in Figure 10 shows how these statements might be used to implement a sum reduction. This loop calculates a global sum, GSUM, of a vector AVAL. A private variable, PSUM, initialized to zero for each processor, will be used in each processor to accumulate a sum of the elements of the vector AVAL assigned to that processor. The number of elements accumulated in each local PSUM is determined dynamically at run time. After each processor has executed its last iteration, it adds its private partial sum, PSUM, into the global total sum, GSUM. This final addition will be done under control of a lock so that GSUM is updated by only one processor at a time. (It should be noted that this example, written to illustrate the parallel loop, may not contain enough processing for profitable parallel execution.)

Parallel cases. Often it is possible to execute blocks of statements in parallel. The blocks may contain straight-line code, branches to other statements in the same block, intrinsic functions, calls to FORTRAN library subprograms, or loops; the loops themselves may be parallel or vector. What is significant is that the blocks may be processed concurrently. At the limit, each block could be executed by a different processor. The number of processors is not known; it may range from one through the number of blocks; the exact number will be determined at the time the blocks are executed. As with parallel loops, the programmer is

responsible for ensuring that such blocks are valid for parallel execution—either that each block is computationally independent of the others or that the data interactions that arise between blocks are controlled or intentional.

The PARALLEL CASES structure is provided to simplify the programming of such parallel blocks of statements. The example in Figure 11 shows statements that may execute concurrently in the first lexical CASE block, CASE 1 and CASE 2; there is no guarantee of the order of their execution. A WAITING clause is provided for situations where an early case may compute data that would be used by more than one subsequent case. In this manner, a program containing an acyclic graph of dependences may be translated into a series of parallel cases. As with parallel loops, the cases may employ private variables as needed. All cases are completed before execution continues beyond the END CASES statement.

Both parallel loops and parallel cases may contain nested parallel loops and parallel cases. The nested loops may be scalar or vector. Input and output statements may be used within parallel loops and parallel cases.

Clustered FORTRAN library. The Clustered FORTRAN library also has extensions for parallelism. Some of these extensions are internal, supporting the parallel language and the automatic parallel capabilities of the compiler. Other extensions are external and may be used directly by the programmer.

Routines are provided for determining the configuration of the virtual cluster. The program can determine the number of virtual computers in the virtual cluster and the location, system identification, associated names, and the number of virtual processors for each virtual computer.

Intrinsic functions are provided for protected updates of shared variables by multiple tasks that reside in the same virtual computer. Arithmetic functions are provided to add, subtract, update if smaller and update if larger to INTEGER*4, REAL*4 and REAL*8 variables or array elements. Logical functions are provided to do AND, OR, and exclusive-OR updates of INTEGER*4 variables or array elements.

Figure 11 Example of PARALLEL CASES

```
PARALLEL CASES
CASE
   statements
CASE 1
   statements
CASE 2
   statements
CASE 3, WAITING FOR CASE 1
   statements
CASE 4, WAITING FOR CASES (1,2)
   statements
CASE 5, WAITING FOR CASES (2,3)
   statements
END CASES
```

Routines are provided for the management of locks, events, and synchronization counters. Locks may be used to ensure that only one task at a time gains access to a resource, such as a shared table modified by multiple tasks. Events may be used to make a task wait until another task has reached some point in execution. Synchronization counters may be used to implement various types of intertask signaling and waiting schemes. Locks, events, and synchronization counters may be used between tasks executing in different virtual computers, as well as the same virtual computer.

An optional trace of the parallel execution may be requested via a run-time option. The trace may be an aid in tuning or debugging a program that executes in parallel. A trace file can be produced for each virtual computer in the virtual cluster. Each trace record identifies the executing task, subroutine, and statement. The system provides trace records for such events as start and end of program execution, origination and termination of tasks, assignment and completion of work to tasks, sharing and copying of common blocks, start and end of parallel loops and parallel cases, and uses of locks and events. Programmers may also enter trace records into these files by calling a library subroutine. The level of detail generated in the trace file is controlled by the run-time option or by a library call.

Figure 12 Serial matrix multiplication

```
      COMMON /AC/ A(1000,1000), B(1000,1000),
     *            C(1000,1000)
      REAL*8 A, B, C
        .
        .
        .
      DO 20 I = 1, 1000
       DO 20 K = 1, 1000
        C(I,K) = 0.0
        DO 30 J = 1, 1000
30       C(I,K) = C(I,K)+B(J,K)*A(I,J)
20     CONTINUE
        .
        .
        .
```

Figure 13 Vector report for matrix multiplication

```
VECT +----   DO 20 I = 1, 1000
SCAL |+---    DO 20 K = 1, 1000
     ||         C(I,K) = 0.0
SCAL ||+--     DO 30 J = 1, 1000
     |||__30     C(I,K) = C(I,K)+B(J,K)*A(I,J)
     ||___
     |____
```

Figure 14 Matrix multiplication with automatically parallelized DO loop

```
      COMMON /AC/ A(1000,1000), B(1000,1000),
     *            C(1000,1000)
      REAL*8 A, B, C
        .
        .
        .
      DO 20 K = 1, 1000
       DO 20 I = 1, 1000
        C(I,K) = 0.0
        DO 30 J = 1, 1000
30       C(I,K) = C(I,K)+B(J,K)*A(I,J)
20     CONTINUE
        .
        .
        .
```

Parallel programming example: matrix multiplication

In the following examples, a matrix multiplication problem is programmed using different methods to illustrate the features of Clustered FORTRAN and to explore issues in parallel and vector programming. Most of these examples are for illustrative purposes only and do not necessarily provide the best parallel performance for the matrix multiplication problem. The best parallel performance for matrix multiplication on a single IBM 3090 multiprocessor can probably be obtained by using the DGEMLP subroutine from the IBM Engineering and Scientific Subroutine Library (ESSL).[7] The examples begin with those for a single virtual computer and end with two for multiple virtual computers. Performance results are given for matrix multiplication on two ES/3090 Model 600J computers.

The first matrix multiplication program, shown in Figure 12, is a serial program that has been optimized for use on the IBM 3090 Vector Facility. This is the code that will be parallelized in the remaining examples.

The objective now is to parallelize the matrix multiplication without degrading the vector performance. The compiler report in Figure 13, which was generated by the compiler when compiling Figure 12 using the option for automatic vectorization, shows that the matrix multiplication is vectorized over the I loop and that C(I,K) needs to be stored only after the J loop has completed. This leaves the K loop as the prime candidate for parallelization.

Distribution of parallel pieces of work (iterations of the K loop) among tasks can be either static or dynamic. Dynamic balancing of work is likely to be preferable when systems cannot be dedicated. The next example shows how dynamic load balancing can be accomplished. The example begins with the observation that the ordering of the I and K loops may be reversed. When this is done, the K loop becomes the outermost loop, where it is suitable for parallelization. Note that to have both a parallel loop and a vector loop in a nested loop, the vector loop must be at a nested level contained within the parallel loop. Also, generally performance will be better when the parallel loop is the outermost loop. The I and J loops, meanwhile, maintain their relationship to each other, allowing efficient vector code.

Figure 14 shows how dynamic load balancing can be achieved using automatically generated parallel DO loops (using the AUTO option). Figure 15

illustrates how the compiler parallelized and vectorized the program. This is a very small reformulation of the original matrix multiplication. A user could also have used the PARALLEL LOOP statement in place of the DO statement for the outermost loop.

When parallelizing code with many fine-grain loops, efficiency can often be improved by moving to coarser grain parallelism, which is the subroutine level. Although the efficiency is obviously not improved by moving a single loop into a subroutine, Figure 16 illustrates subroutine-level parallelism using the SCHEDULE and WAIT FOR statements. The core of the matrix multiplication code is placed in a subroutine named MLT and modified so that it computes one Nth of the matrix, where N is the number of tasks. Its arguments tell it how many tasks there are and which block of columns of the result matrix it is to compute. It is scheduled for parallel execution with multiple executions of the SCHEDULE statement. Note that the compiler will automatically create multiple copies of the argument K, so that each virtual process has its own value of K.

This example also illustrates a static mapping of work to tasks; each task receives a predefined amount of work. It can work well on a dedicated system where each scheduled task should have a real processor immediately available. However, if the program is executing in an environment where there is competition for the real processors, the parallel performance actually achieved will be determined by the task that receives the lowest level of service.

In all of the preceding examples, the matrix multiplication was being performed in only one virtual computer. In all cases, parallelism was achieved by having logically concurrent tasks generate independent columns of the result matrix. (A column of the result matrix corresponds to one iteration of the K loop.) The example in Figure 14 used dynamic load balancing with compiler-generated chunks of iterations being distributed among the virtual processors for the K loop. In Figure 16, the iterations were explicitly chunked such that each task received exactly one chunk using static load balancing.

The following examples show how the matrix multiplication may be extended across multiple virtual computers. Figure 17 shows dynamic load

Figure 15 Parallel report for matrix multiplication with automatically parallelized DO loop

```
PARA +----   DO 20 K = 1, 1000
VECT |+---    DO 20 I = 1, 1000
     ||          C(I,K) = 0.0
SCAL ||+--      DO 30 J = 1, 1000
     |||__30     C(I,K) = C(I,K)+B(J,K)*A(I,J)
     ||___
     |____
```

Figure 16 Matrix multiplication with SCHEDULE

```
      COMMON /AC/ A(1000,1000), B(1000,1000),
     *            C(1000,1000)
      REAL*8 A, B, C
        .
        .
        .
      DO 20 K = 1, NTASK
20      SCHEDULE ANY TASK ITASK,
     *    SHARING (AC),
     *    CALLING MLT (K,NTASK)
      WAIT FOR ALL TASKS
        .
        .
        .
      END

      SUBROUTINE MLT (KN,KT)
      COMMON /AC/ A(1000,1000), B(1000,1000),
     *            C(1000,1000)
      REAL*8 A, B, C
      KUB = 1000*KN/KT
      KLB = 1+1000*(KN-1)/KT
      DO 20 I = 1, 1000
        DO 20 K = KLB, KUB
          C(I,K) = 0.0
          DO 30 J = 1, 1000
30          C(I,K) = C(I,K)+B(J,K)*A(I,J)
20        CONTINUE
      RETURN
      END
```

balancing across (up to) four virtual computers where the chunk size is defined by the parameter KCHNK. Figure 18 uses static load balancing and defines only two chunks, the left half and the right half of the result matrix. In both of these last two examples, we chose to use DGEMLP to illustrate the simplicity of using the parallel ESSL routines to provide parallelism within the virtual computer.

Several considerations were used in developing the example shown in Figure 17. They included:

Figure 17 Dynamic load balancing with DGEMLP

```
@PROCESS DC(AA,BBCC) OPT(3)
      REAL*8 A,B,C
      PARAMETER (N=1000,KCHNK=25)
      COMMON /AA/ A(N,N)   /BBCC/B(N,N),C(N,N)
      INTEGER ID(4)
C     ...
      NCHNK = 1 + (N-1)/KCHNK
      NCOMP = MIN(NCOMPS(),NCHNK,4)
      M = MIN(KCHNK,N)
      ORIGINATE ANY TASK ID(1)
      SCHEDULE TASK ID(1), SHARING(AA),
     *  CALLING MLT(B(:,1:M), C(:,1:M), -M)

      DO 20 I=2,NCHNK
        L = M + 1
        M = MIN(M+KCHNK, N)
        J = M+1-L
        IF(I.LE.NCOMP) THEN
          ORIGINATE ANY TASK ID(I),BINDING(I)
          SCHEDULE TASK ID(I), COPYINGI(AA),
     *    CALLING MLT(-B(:,L:M),C(:,L:M)-,-J)
        ELSE
          WAIT FOR ANY TASK ITKD
          IF(ITKD.EQ.ID(1)) THEN
            SCHEDULE TASK ITKD, SHARING(AA)
     *      CALLING MLT(B(:,L:M),C(:,L:M),-J)
          ELSE
            SCHEDULE TASK ITKD, CALLING
     *      MLT(-B(:,L:M),C(:,L:M)-,-J)
          ENDIF
        ENDIF
 20   CONTINUE
      WAIT FOR ALL TASKS
C     ...

      DO 101 I=1,NCOMP
 101    TERMINATE TASK ID(I)
      STOP
      END
@PROCESS DC(AA) OPT(3)
      SUBROUTINE MLT(B,C,K)
      PARAMETER(N=1000)
      REAL*8 A,B(N,K),C(N,K)
      COMMON /AA/ A(N,N)
      CALL DGEMLP(A,N,'N',B,N,'N',C,N,N,N,K)
      RETURN
      END
```

- The program should be able to dynamically adapt to the number of virtual computers and virtual processors specified for the run-time configuration. It was assumed that a user would want to specify a virtual cluster that would match the physical configuration of the Clustered FORTRAN Complex being used.
- The program would normally be run on a complex where each node would be fully populated with vector facilities; vector performance should not be sacrificed to achieve parallelism.
- The program should be able to dynamically load balance between virtual computers as well as within a virtual computer.
- The synchronization and moving of data between virtual computers should be minimized. This consideration, to some extent, is in conflict with dynamic load balancing between virtual computers.

Figure 18 Static load balancing using DGEMLP

```
PARAMETER (J= 1000, JP=J+1, N=2*J)
REAL*8 A(N,N), B(N,N), C(N,N)
 .
 .
 .
ORIGINATE  ANY  TASK  K,  BINDING(2)
DISPATCH  TASK  K,  COPYINGI(ARG(*)),
*  CALLING DGEMLP (A,N,'N',B(:,JP:N),
*  N, 'N', C(:,JP:N)-, N, N, N, J)
CALL DGEMLP(A,N,'N',B,N,'N',C,N,N,N,J)
WAIT FOR TASK  K
 .
 .
 .
END
```

The preceding considerations led to the adoption of the approach shown in Figure 17. The K loop is broken into chunks with each chunk being distributed dynamically to the virtual computers. The chunk size (KCHNK) is parameterized to provide a control over the tradeoffs between load balancing among virtual computers, synchronization among virtual computers, and load balancing among the virtual processors within each virtual computer. The first ORIGINATE and SCHEDULE pair gives a chunk of columns (1 through M) of the B and C matrix to a task on the RVC, sharing all arrays to minimize communications. (The third argument in the CALLING clause specifies the number of columns in the chunk.) During the first (NCOMP-1) iterations of loop 20, if other virtual computers exist, chunks of columns are given to a task on each of the CVCs. Array stripes allow a compact specification of the minimal amount of data that must be exchanged. Once an initial chunk is given to each of the virtual computers, the WAIT FOR ANY TASK statement returns the task identifier, ITKD, of the first completing task. If it was the task in the root virtual computer, (ID(1) = ITKD), another chunk is given to it. Addressability to the A matrix is reestablished each time with the SHARING clause. If

the returning task was from a cluster virtual computer, then another chunk is given to it. Note that since the data of a task environment are persistent, the read-only data of the A matrix need not be copied after the first schedule of each of the tasks in the clustered virtual computers. After the last iteration of the loop, all chunks have been distributed and the work will be complete after the WAIT FOR ALL TASKS has been satisfied. The parallelism within each virtual computer is obtained by using the parallel ESSL routine for matrix multiply, DGEMLP. Dynamic load balancing within a virtual computer for DGEMLP is achieved by requesting additional virtual processors, which in turn will result in there being more pieces of work than there are real processors.

Figure 18 illustrates a simple way to obtain excellent performance from all 12 processors on a pair of dedicated systems. The work is statically partitioned between computers; the RVC calculates the left half of the result matrix while the right half is computed by the CVC. ESSL provides the parallel support within a computer. Performance measurements for this code are shown in Table 1. The code was run on the Cornell National Supercomputer Facility, which consisted of two ES/3090 Model 600J computers, each with 512 MB of main memory and with six Vector Facilities; VM/XA System Product Release 2; and the clustered FORTRAN hardware and software. The execution times include all necessary data transfers (input and result matrices) between the computers.

Performance of 1 GigaFLOPS can be achieved on matrix multiplication using Clustered FORTRAN with two ES/3090 Model 600J computers, as shown in Table 1.

Table 1 MATRIX MULTIPLY using DGEMLP in ESSL

N	MB	Computation and Transfer	
		Time (s)	GFLOPS
2000	96	20.03	0.798
3000	216	49.76	1.085
4000	384	108.79	1.176
4400	465	148.68	1.145
5000	600	221.48	1.128

Concluding remarks

IBM Clustered FORTRAN is a combination of software and hardware that allows two IBM ES/3090 multiprocessors (selected models), to be physically connected as a cluster and allows FORTRAN jobs to execute in parallel across all of the processors of the cluster. The FORTRAN compiler and library provided as part of Clustered FORTRAN are used for writing and executing the parallel programs. They provide a rich spectrum of function that supports a wide range of parallel application programming styles.

The parallelism in an application may be expressed in ways that are natural to the application. The SCHEDULE and DISPATCH statements may be used to execute independent subroutines in parallel, either in the same virtual computer or in a different virtual computer. The PARALLEL LOOP and PARALLEL CASES statements may be used to parallelize the statements within a routine. Automatic parallel and automatic vector may be used to gain faster execution of nests of eligible DO loops. Parallel execution is not restricted to a single level but may be specified wherever it occurs. Operating system and machine configuration differences are not exposed to the program.

The Clustered FORTRAN program identifies the pieces of work that are eligible to run in parallel and how the parallel work is to be scheduled on each of its virtual computers. When the program is compiled and executed, the library in each of the virtual computers puts the parallel work on a queue and distributes it to the virtual processors of the virtual computer. Real processors are allocated dynamically by the operating system. As additional real processors are allocated, additional virtual processors can execute concurrently. Programs that partition work dynamically, employ multiple levels of parallelism, or use other strategies to keep the queues of parallel work full can take advantage of these additional real processors as they become available during execution.

Clustered FORTRAN applications run under the VM/XA System Product operating system. The degree of parallel execution can be controlled at run time through the number of virtual computers and virtual processors requested. When parallel execution is requested in a cluster consisting of two ES/3090 Model 600 systems that are fully equipped with Vector Facilities, Clustered FORTRAN makes it possible for a single FORTRAN program to make simultaneous use of up to 12 processors and Vector

Facilities as a means of reducing its turnaround time, or increasing its performance. Performance of one GigaFLOPS has been achieved on matrix multiplication using Clustered FORTRAN with two ES/3090 Model 600J computers.

Acknowledgments

IBM Clustered FORTRAN was developed as part of a joint study with the Cornell National Supercomputer Facility (CNSF). In addition to providing input on the requirements for Clustered FORTRAN, the CNSF provided an environment where scientists could explore parallelism in their applications and give valuable feedback on Clustered FORTRAN.

Cited references

1. S. G. Tucker, "The IBM 3090 System: An Overview," *IBM Systems Journal* **25**, No. 1, 4–20 (1986).
2. *IBM VS FORTRAN Version 2, Release 3 Language and Library Reference*, SC26-4221-3, IBM Corporation; available through IBM branch offices.
3. *IBM Parallel FORTRAN Language and Library Reference*, SC23-0431-0, IBM Corporation; available through IBM branch offices.
4. L. J. Toomey, E. C. Plachy, R. G. Scarborough, R. J. Sahulka, J. F. Shaw, and A. W. Shannon, "IBM Parallel FORTRAN," *IBM Systems Journal* **27**, No. 4, 416–425 (1988).
5. *IBM VS FORTRAN Version 2, Release 5 Language and Library Reference*, SC26-4221-5, IBM Corporation; available through IBM branch offices.
6. *IBM Clustered FORTRAN Language and Library Reference*, SC23-0523-0, IBM Corporation; available through IBM branch offices.
7. *IBM Engineering and Scientific Subroutine Library Guide and Reference, Release 3*, SC23-0184-3, IBM Corporation; available through IBM branch offices.

Richard J. Sahulka *30 Bluestone Road, Woodstock, New York 12498*. Mr. Sahulka retired from IBM Data Systems Division in 1990 after nearly 33 years of service. Prior to his retirement, Mr. Sahulka was a member of the engineering and scientific systems development and technology organization in Kingston where he had overall design responsibility for the system software for Clustered FORTRAN. Mr. Sahulka led the team which designed and developed the VS FORTRAN Multitasking Facility and received an IBM Outstanding Technical Achievement Award for that effort. Mr. Sahulka has extensive experience in multiprocessing and multitasking, having worked on both the TSS and MVS operating systems. He received his Sc.B. degree in electrical engineering from Brown University, Rhode Island, in 1951. After service in the U.S. Navy during the Korean war he returned to Brown University to continue his studies. He joined IBM in Poughkeepsie in 1957.

Emily C. Plachy *IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598*. Dr. Plachy is a research staff member in the computer sciences department at the T. J. Watson Research Center. Her primary interests are parallel compilers and parallel programming environments. After working for Exxon Production Research Company in Houston, Texas, as a seismic applications programmer, she joined IBM in 1982 to work on engineering and scientific compiler development. Dr. Plachy provided the overall project management for the Parallel FORTRAN prototype. She managed the software development for both IBM Clustered FORTRAN and the IBM High-Performance Parallel Interface. She received a B.S. degree in applied mathematics and computer science from Washington University, St. Louis, Missouri, in 1970, an M.S. degree in computer science from the University of Waterloo, Waterloo, Ontario, in 1971, and a D.Sc. degree in computer science from Washington University in 1980.

Leslie J. Scarborough *IBM Palo Alto Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304*. Ms. Scarborough is a member of the IBM Palo Alto Scientific Center where she works on software for parallel computer systems. She joined IBM's East Fishkill facility in 1978 as an application programmer. Her assignment involved numeric computing with FORTRAN for graphics postprocessing. In 1983 Ms. Scarborough joined the IBM Kingston laboratory to work on engineering and scientific compiler development. She led the design and implementation of the Parallel FORTRAN Interface for VM/XA, and she received an Outstanding Innovation Award for her contributions to IBM's Parallel FORTRAN. Ms. Scarborough then worked on the definition of the FORTRAN language and library extensions for IBM's Clustered FORTRAN. She represents IBM on the ANSI X3H5 committee, a group formed to define a standard model for parallel programming in FORTRAN and other high-level languages. Ms. Scarborough received a B.S. degree in mathematics from the State University of New York at Albany in 1977 and an M.S. degree in computer science from Syracuse University in 1984.

Randolph G. Scarborough *IBM Palo Alto Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304*. Mr. Scarborough is an IBM Fellow at the Palo Alto Scientific Center. His primary interests are FORTRAN and new machine architectures, especially parallel and distributed systems. He joined IBM in 1969 as a systems engineer in Trenton, New Jersey, to work on large scientific and state government accounts. In 1973 he joined the Palo Alto Scientific Center to develop the APL microcode for the System/370™ Model 135. In 1978 he produced the FORTRAN H Extended Optimization Enhancement. In 1983 this work was augmented to include the new expanded-exponent extended-precision (XEXP) number format. Between 1982 and 1985 he produced the vectorizer incorporated into VS FORTRAN Version 2. Since then he has produced the compiler and library for Parallel and Clustered FORTRAN. Mr. Scarborough received a B.A. from Princeton University in 1968. He has received many IBM awards, including four Outstanding Innovation Awards (one for Clustered FORTRAN) and two Corporate Awards.

Steven W. White *IBM Advanced Workstations Division, 11400 Burnet Road, Austin, Texas 78758*. Dr. White received his Ph.D. from Texas A&M University where he also taught

in the electrical engineering department for three years. In 1982, he joined IBM in Poughkeepsie to work on large system, scientific and engineering processor development, architecture, and system design. In 1986, he started a two-year assignment with the Computational Physics Group at Livermore National Laboratory. In 1989, he joined the High-Performance/Supercomputing Systems Development Laboratory in Kingston, New York, to work on Clustered FORTRAN/HIPPI development. He currently works in the processor architecture group defining and evaluating potential members to the RISC System/6000™ product family. His primary interests are parallel and distributed architectures and memory hierarchies.

Reprint Order No. G321-5436.