Clustering IBM Enterprise System/3090 computers for parallel execution of FORTRAN programs

by L. J. Scarborough
R. G. Scarborough
S. W. White

Two IBM Enterprise System/3090™ Model 600J computer systems, each with six processors capable of executing vector and scalar instructions, have been connected into a cluster for parallel execution of single FORTRAN programs. The clustering is achieved through a combination of software and hardware. When enabled for parallel execution and allowed to use all twelve processors in the cluster, FORTRAN programs have run as much as 11.7 times faster than when run on a single processor. The combined hardware and software technology is called IBM Clustered FORTRAN. It was achieved by modifying existing technology quickly to provide new capabilities. This paper discusses the modifications and the motivations behind them. It summarizes the performance of several applications executed with Clustered FORTRAN. Finally, it describes how clustering has been used to improve performance in novel ways.

Introduction

The use of computer systems for numerically intensive applications is increasing. New applications are being developed, and new and existing applications are employing ever-expanding quantities of computer resources in their execution. Users continue to want the results of their applications quickly.

In an effort to meet this challenge, many vendors of computer systems offer parallel-processing systems in which multiple processors or computers work together on behalf of one application. The systems differ in their methods, but it is common to divide the systems into two classes: shared-memory systems and distributed-memory

systems. In a shared-memory system, the multiple processors share a common memory and communicate through it. In a distributed-memory system, the multiple processors communicate via an interconnection network. Hybrid systems, which use a combination of these mechanisms, also exist. In a hybrid system, a small number of processors (four to sixteen) might access a shared memory, and several such groups of processors might be linked together by other means.

Proponents of shared-memory parallel machines point to the high-bandwidth, low-delay communication possible on these systems and also note that a commonly addressable memory often removes the need for copying data between processors. Supporters of distributed-memory parallel machines claim that a lack of scalability is inherent in shared-memory designs, since memory bandwidth cannot increase without limit and since it is difficult to keep the contents of all of the memories in agreement. Builders have used various mechanisms to connect the separate processors (or separate groups of processors) that form their distributed-memory machines. In some designs, such as that of the Cedar project [1–3], communication among groups of processors occurs through a separate "global" memory with longer access times than the "local" memory used within a group of processors. In others, such as the work reported here, communication between groups of processors occurs through a memory-to-memory message-passing connection.

This paper discusses a method for linking high-end multiprocessor computer systems into a single complex, which we call a cluster, so that all of the processors in the cluster can compute in parallel on a single FORTRAN program. Our work was done for a cluster composed of two IBM Enterprise System/3090™ (ES/3090™) Model 600J computer systems [4]. These are shared-memory multiprocessing computers. Each computer has six processors, and each processor in our cluster had an optional vector-execution element that enabled it to execute vector instructions. Our cluster therefore contained twelve vector processors executing in parallel. More recently, Extended Clustered FORTRAN, an extension of this work, was announced to support larger clusters [5]. These clusters may contain up to four Enterprise System/9000™ (ES/9000™) systems, each having up to six processors with integrated vector elements. This permits twenty-four vector processors to execute in parallel on a single FORTRAN program.

Our goals in this work were pragmatic. We wanted to provide clustering for our high-end systems because these systems are widely available, to us and our customers, and clustering would increase their capabilities. We wanted to use existing hardware and software to the maximum extent possible because we wanted to develop something quickly and inexpensively. We wanted to do this without disrupting the product-development activities underway in our company. So, in contrast to other approaches to clustering, we ruled out extensive modifications to the hardware or software. We ruled out plans that would require years of research work or years of product-development activity. We did not use the clever mechanisms that do result from extensive research and development. We did not, for example, consider adopting the global-memory approach used in the Cedar project, since this would have required a substantial redesign of the memory subsystem, or synchronization registers such as those present on some Cray systems [6]. Instead, we constrained ourselves to use simpler mechanisms. The cluster of two ES/3090 computers we used was created by providing a high-speed hardware link between the memory subsystems of the two computers and driving it with message-passing software. In the recently announced cluster that connects four ES/9000 computers, the high-speed hardware links from the computers are connected through a switch.

Although the physical cluster available to us contained only the two real ES/3090 computers, the software does not explicitly limit the number of real computers that may be linked into a cluster. Further, regardless of the real computers in a cluster, the software allows users to create arbitrary virtual clusters for running their programs. Just as a real cluster is composed of one or more real computers, each having one or more real processors, a virtual cluster is composed of virtual computers having virtual processors. Users can experiment with algorithms designed for two six-processor computers, twenty uniprocessor computers, ten 16-processor computers, or any other virtual configuration, and map this experimental configuration onto whatever real configuration is available at run time. (The software imposes an arbitrary limit of 64 virtual computers per virtual cluster.)

The investigation of clustering was done as a joint study with the Cornell National Supercomputer Facility (CNSF). CNSF was founded with a special charter to explore parallel computation and to provide parallel computing in a production environment. The Parallel FORTRAN language [7], designed for shared-memory multiprocessing, was defined during an earlier joint study with CNSF. It was implemented by IBM, placed into production at CNSF, and used successfully for parallel programming of the six-processor ES/3090 computer at CNSF. In 1989, CNSF added a second six-processor ES/3090 computer to its configuration and sought mechanisms that would allow all twelve processors of the two ES/3090 computers to be brought to bear concurrently on single application programs.

FORTRAN was selected as the language for the clustered system. FORTRAN is the language typically used by programmers of numerically intensive applications

on parallel systems, and it has changed rapidly in the past half decade to support a wide variety of parallel-computer systems. Indeed, a survey article published four years ago found twelve dialects of parallel FORTRAN for commercially available machines and noted that other machines had come into existence after the article was submitted for publication [8].

IBM and CNSF jointly defined a set of extensions to FORTRAN for parallel and clustered computing. The goal was to leave the meaning and flavor of FORTRAN unchanged, so that the established audience of scientists and engineers already experienced in FORTRAN could easily benefit from parallel computation. Completely automatic compilation for parallel processors is in theory the best way to achieve this goal, but compilers are not yet able in practice to do a complete enough job for production computing. They are able to analyze nests of loops, and this was done in Parallel and Clustered FORTRAN, but more is required. Programmers still must modify their FORTRAN code for parallel execution, and additions to FORTRAN are needed.

In Parallel FORTRAN, the programmer writes code as if it were running on a single processor. Language extensions are used to indicate where separate code sequences might be assigned to separate processors. The sequences, though, are still written like normal FORTRAN for uniprocessors. The language extensions were designed to permit large segments of code and data to be isolated for execution by independent processors, as well as to allow processors to work cooperatively on code or data that could not be isolated. Both kinds of parallel possibilities arise naturally in programs.

The question then arose: What could be done to allow Parallel FORTRAN programs to be executed on a cluster of two ES/3090 computers? Clustered FORTRAN was developed as an answer to this question [9, 10]. Clustered FORTRAN allows a set of applications to achieve speedups when they are executed on a cluster. How many applications ultimately will benefit depends on how well their data can be distributed across the cluster, how often they need to communicate between the ES/3090 computers, and how much overhead is associated with that communication.

Clustered FORTRAN overview

IBM Clustered FORTRAN consists of three components. The first of these is the Clustered FORTRAN compiler and library software, which includes all of Parallel FORTRAN, implements the new language extensions designed for clustering, and extends the synchronizing routines of Parallel FORTRAN to operate across the entire cluster. A programmer uses the language extensions to identify the parts of a program that are eligible for parallel execution. The compiler then converts the program into pieces that the run-time library can manipulate to cause the application to execute in parallel on the available hardware configuration.

The second component of Clustered FORTRAN is the Node Manager, which is responsible for managing the real computers in the real cluster and for creating the virtual clusters of virtual computers requested by application programs at run time. A copy of the Node Manager runs on each of the real computers contained in the cluster; it acts essentially as an extension of the VM/XA™ operating system [11] of the real computer. The Node Managers in a cluster coordinate among themselves, allocating resources for the Clustered FORTRAN application programs executing on the cluster. They also permit the installation to tune, monitor, and control the execution of clustered applications.

The third component of Clustered FORTRAN is the high-speed connection facility, a hardware and software combination that facilitates the transmission of large amounts of data between the computers in a cluster. The hardware is attached to the memory subsystems of the real computers. It is managed by supervisor-state code in the operating system and is shared by the application programs executing in the cluster. The interface for communicating between virtual computers resembles that of the existing VM/XA Inter-User Communication Vehicle (IUCV) [12, 13]. IUCV provides communication between virtual computers on the same real computer. Our adaptation allows the virtual computers to lie on different real computers, and it calls on the high-speed hardware connection to transport data between real computers.

Figures 1 and **2** illustrate the relationships between these components of Clustered FORTRAN and the real computer systems on which they execute. Figure 1 is an overview, showing a cluster of two real computers connected by the high-speed connection facility. Each real computer has a copy of the Node Manager. Running on the real cluster is a single application program. The application programmer has specified a virtual cluster containing four virtual computers, two virtual computers on each of the two real computers. Figure 2 shows the structure of one of the virtual computers.

FORTRAN compiler and library

As indicated above, we wanted extensions to Parallel FORTRAN that would allow it to operate across a cluster of computers. The extensions were intended to conform in style and spirit to the language of Parallel FORTRAN and to enable the programmer to experiment with various methods of partitioning a program across the cluster.

The major difference between Parallel and Clustered FORTRAN follows from the nature of the memory on the machines for which they were written. Parallel FORTRAN was directed at programs operating on one computer;

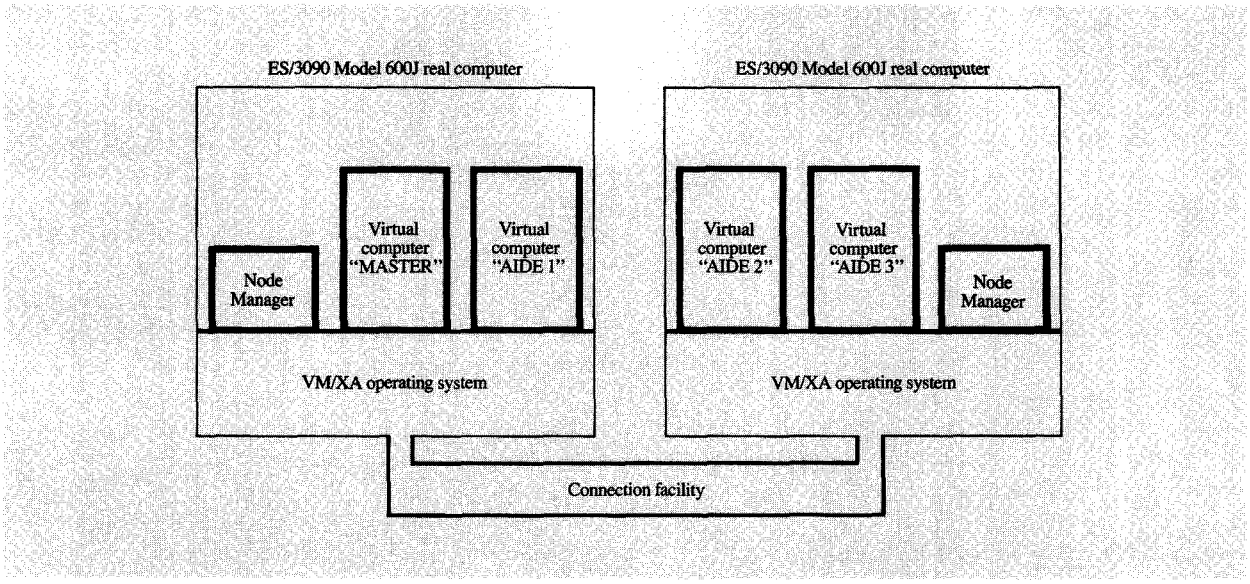

Figure 1

Clustered FORTRAN system components. An application program using four virtual computers is running on a cluster of two real computers.

Clustered FORTRAN was developed for programs operating across multiple computers that do not share memory. We were not able to devise techniques for automatically partitioning a program for computers that do not share memory; this is very much a current topic of computer science research [14, 15]. We implemented, instead, a small extension to the Parallel FORTRAN language that would allow a user to distribute the work of an application across more than one computer system. Compatibility with the existing language was important. But, since we could not ignore the difference between nonshared and shared memory, we felt that programmers had to be aware of the difference, and we sought simple extensions that acknowledged the difference.

The Parallel FORTRAN language is summarized in **Figure 3**. It supports two primary styles of programming to exploit parallelism. The first allows multiprocessing of the statements within a given subroutine. The prime example of this style of programming is the parallel loop, in which different processors execute independent iterations of the loop. All of the local and common data known in the subroutine are available for access by all of the processors (although selected scalar temporary variables may be declared to be private). In a similar manner, the construct for parallel cases allows different processors to execute different blocks of statements concurrently.

In the second style of programming for parallelism, the user originates and schedules subprograms that execute independently of the program calling them and of each other. These independently executing subprograms are called tasks. When originated, each task has its own local data area, its own set of common blocks, and its own set of paths (called unit numbers in FORTRAN) for communicating with files. Work is assigned to a task by a SCHEDULE statement, which names the subroutine at which the task is to begin its execution. The SCHEDULE statement also specifies what data (if any) belonging to the routine executing the SCHEDULE statement are to be shared with the task being scheduled. Only the specified data are shared, and the sharing lasts only for the single call to the task.

The strong separation of tasks and the strong control over data sharing were intended to isolate the operation of tasks, so that programming errors could be more easily located. Parallel programming errors are often intermittent and are notoriously difficult to find. Sharing only the data explicitly listed on a SCHEDULE statement accomplishes two things. It reduces enormously the amount of data subject to inadvertent update, and it also delimits exactly the data that are exposed to asynchronous manipulation. (If a user fails to share data that must be shared for the program to operate correctly, the error will be repeatable and therefore more easily found.) This strong separation of tasks also makes it easy to incorporate existing routines, either serial or parallel, into larger parallel programs. Provided they are given the data they expect, tasks can

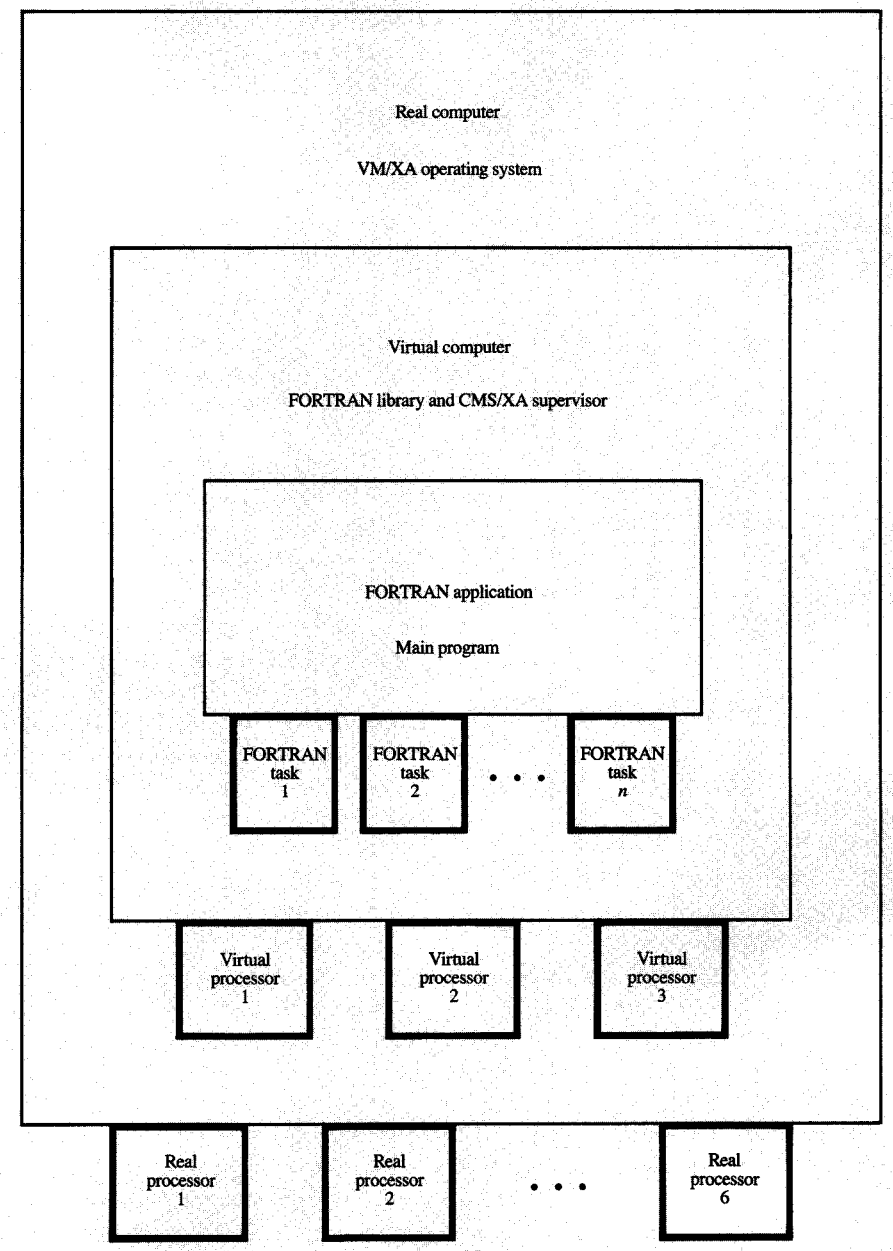

Figure 2

Structure of a virtual computer. A virtual computer with three virtual processors is running on a real computer with six real processors. The application program has originated a number of tasks and bound them to the virtual computer.

run securely in isolation, unaffected by other parts of the larger program.

It is obvious that isolated tasks can map naturally onto a clustered system. In contrast, we did not see an efficient way to distribute across computers the execution of statements within a subroutine, such as the statements within a parallel loop. The data referenced by the statements within a subroutine must be partitioned if the

Statements for Parallel Loops

```
PARALLEL LOOP stmtno [,] indvar=exp1,exp2[,exp3]
PRIVATE (var [,var ...])
DOFIRST [LOCK]
DOEVERY
DOFINAL [LOCK]
STOP LOOP stmtno
```

Statements for Parallel Cases

```
PARALLEL CASES
PRIVATE (var [,var ...])
CASE [m [,WAITING FOR {CASE n | CASES (n1 [,n2 ...])}]]
END CASES
```

Statements for Parallel Tasks

```
ORIGINATE ANY TASK taskid
TERMINATE TASK taskid
SCHEDULE [ANY] TASK taskid,
     [TAGGING (tagval1 [,tagval2 ...]),]
     [SHARING (shrcom1 [,shrcom2 ...]),]
     [COPYING (cpycom1 [,cpycom2 ...]),]
     [COPYINGI(cpicom1 [,cpicom2 ...]),]
     [COPYINGO(cpocom1 [,cpocom2 ...]),]
     CALLING subnam [([arg1 [,arg2 ...]])]
WAIT FOR [ANY] TASK taskid [, TAGGING (tag1 [,tag2 ...])]
WAIT FOR ALL TASKS
```

Figure 3

Parallel FORTRAN statements for parallel processing.

data are to be distributed, but in general we do not know how to partition such data. An alternative would have been to simulate a shared-memory system on the distributed-memory cluster, sending pages back and forth as they were accessed by the different computers in the cluster. The cost of this simulation was expected to be very high. It would have required extensive programming in privileged operating-system code and would have led to more frequent communication of smaller units of data. Instead, in order to optimize the performance of the link between the computers, we wanted an interface that allowed large amounts of data to be communicated at each transmission. For these reasons we focused on extending the Parallel FORTRAN tasking statements to work across the cluster, something that could be done primarily in the FORTRAN compiler and library with minimal changes to the privileged operating system code, and we provided programming constructs that send large amounts of data between virtual computers in one transmission.

The SCHEDULE statement in Parallel FORTRAN allows a programmer to provide data to a called task in several ways. Named common blocks may be copied into the task before it executes, out of the task after it executes, or both, with different choices possible for each common block. Named common blocks may also be shared between the tasks, which means that the called task accesses directly the caller's copy of the common block. Finally, the caller may specify arguments for the subroutine at which the called task begins its execution. These are shared with the subroutine in the normal manner of FORTRAN (accessed by address and not by value). For example, the statement

```
SCHEDULE TASK N,
     SHARING (COMON1, COMON2),
     COPYING (COMON3, COMON4),
     CALLING SUBX (ARGA, ARGB, ARGC)
```

directs the sharing of common blocks COMON1 and COMON2, the copying of common blocks COMON3 and COMON4 (into the task before execution of the subroutine and out of the task afterward), and the sharing of arguments ARGA, ARGB, and ARGC. COPYINGI or COPYINGO clauses can be used to specify copying in or copying out only.

In a clustered environment, common blocks and arguments must be copied, rather than shared, when they are used to communicate between tasks on different computers in the cluster. Language was already available in Parallel FORTRAN for copying common blocks, but not arguments, so a language extension was defined to provide for the copying of arguments. A user may specify the phrase "ARG(*)" in the COPYING, COPYINGI, or COPYINGO clauses of a SCHEDULE statement to specify how arguments should be copied. Individual arguments may also be tagged to indicate different forms of copying: "=A" (modeled on the assembly-language literal) means to copy the value of the variable A into the subroutine as its argument (i.e., copy on input); "A=" (modeled on the FORTRAN assignment statement) means to copy the value returned for the argument by the subroutine into the variable A (i.e., copy on output); and "=A=" causes copying in both directions. (The Clustered FORTRAN language is summarized in **Figure 4**.)

One additional extension for arguments was thought important. If an argument is an array, the syntax above permits copying the entire array. But sometimes a loop may be used to partition an array computation across tasks, so language was needed for passing array partitions to the tasks. The Fortran 90 colon notation for array sections [16] was selected as the method for specifying array partitions.

With these language extensions, programmers can copy common blocks and arguments between tasks running on different computers. The data to be copied are specified at the point where the subroutine in the task is scheduled for execution, and transmission of the data takes place synchronously, from the perspective of the caller, when the caller schedules a task or waits for its completion. This

is sufficient for many applications. However, message-passing constructs that allow tasks to communicate at any point during execution are known to be an additional requirement.

Since tasks are expected to be long-lived and relatively large, it seems reasonable to place them on specific computers in the cluster when they are originated and to keep them there. But what computers are available, and which computer shall be used for a given task? We left this open to the programmer.

As mentioned earlier, the programmer may specify an arbitrary virtual configuration for executing an application—an arbitrary virtual cluster of virtual computers, each virtual computer having its own storage, disks, and processors. The programmer may also specify a mapping between the virtual computers in the virtual cluster and the real computers in the real cluster. The virtual configuration is specified at run time and may be changed, along with the mapping from virtual to real configuration, from run to run. It is the job of the system software running on the real computers to construct and operate the virtual cluster specified by the programmer.

The user specifies the virtual configuration in a file that is read when the program begins execution. The configuration file contains one record for each virtual computer in the virtual cluster. As an example, the file

```
COMPUTER PROCS(6), STORAGE(128M), BINDING(MASTER)
COMPUTER PROCS(3), STORAGE(128M), BINDING(HELPER1)
COMPUTER PROCS(3), STORAGE(128M), BINDING(HELPER2)
COMPUTER PROCS(3), STORAGE(128M), BINDING(HELPER3)
```

requests a virtual cluster containing four computers, each with 128 megabytes of storage. One computer has six processors; the other three have three processors apiece. The programmer may refer to specific virtual computers by number (1 through 4) or by arbitrary symbolic names (MASTER, HELPER1, HELPER2, and HELPER3 in the example). Additional options (see [9]) may be used to place the virtual computers onto specific real computers.

The virtual configuration file is consulted when a task is originated. An ORIGINATE statement creates a task and returns an identifier to the user. When used in its basic form,

```
ORIGINATE ANY TASK itask ,
```

the ORIGINATE places the new task on the same virtual computer as the originating task. The two tasks can therefore share arguments and common blocks. But the user may also specify the virtual computer on which the task should be placed by using an extended form,

```
ORIGINATE ANY TASK itask, BINDING(ibind) ,
```

and giving, in the variable or constant ibind, the number or symbolic name of the virtual computer for the new task.

Statements for Parallel Tasks

```
ORIGINATE ANY TASK taskid, BINDING (ibind | cbind)
SCHEDULE [ANY] TASK taskid,
     [COPYING ([ARG(*) [, cpycom1 [,cpycom2 ...]]]),]
     [COPYINGI([ARG(*) [, cpicom1 [,cpicom2 ...]]]),]
     [COPYINGO([ARG(*) [, cpocom1 [,cpocom2 ...]]]),]
     CALLING subnam[([[=]arg1[=] [,[=]arg2[=] ...]])]
```

Figure 4

Extensions to Parallel FORTRAN for clustered computing. The new constructs are the BINDING clause, the ARG(*) operand, the "=" notation, and the Fortran 90 syntax for partitioning array arguments (not shown).

A task created with the BINDING keyword can communicate with the originating task only through copied data.

The BINDING clause of the ORIGINATE statement and the syntax for copying arguments and array partitions in the SCHEDULE statement are the only new language constructs needed in Parallel FORTRAN to enable it to operate in a cluster. The programmer must still analyze the application to ascertain how data can be partitioned and work can be distributed. For some applications, the partitioning may correspond naturally to the existing algorithm and its serial implementation. For others, the partitioning may require a new algorithmic approach.

Programs that are partitioned to run on multiple computers can still be run with a single computer (assuming they still fit within storage). Regardless of the configuration, Clustered FORTRAN maintains the semantics for copied data specified by the user. Thus, a program written for clustering can execute correctly with or without a clustered system. Executing on a single-computer system, especially on a uniprocessor, can simplify debugging. On a uniprocessor, the program executes in a repeatable manner (the statements execute in the same order for the same data). Deterministic bugs can be found and removed. Thereafter, the program may be run on a cluster, where parallel components of the program can execute concurrently and the order of statement execution may or may not be repeated.

The FORTRAN library was extended in several ways for clustered computing. Inquiry functions were added to allow the user to ask about the number of virtual computers or, indeed, to retrieve the entire virtual configuration along with its mapping to the real configuration. The synchronizing functions that Parallel FORTRAN provides to users—its lock routines and event routines—were extended to operate across the entire cluster. New synchronization functions were added, most notably some that perform interlocked arithmetic and

logical operations on variables. These functions, which follow suggestions from the Los Alamos National Laboratory [17], are more natural for some applications than are events and locks. They provide the ability for tasks to modify a shared counter or to wait until it reaches a specified value. Tasks running across the cluster may use such counters in many ways, such as dynamically partitioning the pieces of a computation among themselves.

Clustered FORTRAN also incorporates some changes that are independent of clustering. Subroutines, for example, may be called from within parallel loops. Routines to do fast atomic updates of variables in memory are provided to coordinate tasks running in the same virtual computer. These improvements were suggested by user experience with Parallel FORTRAN.

In summary, with a very few syntax additions, Parallel FORTRAN has been extended for clustered computing. Now, with Clustered FORTRAN, an application programmer can use a single tool to exploit parallel computing over a wide range of machine capabilities. The user can partition a program at the highest level to take advantage of clustered (distributed) parallel execution. Within each partition, the user can identify places for shared-memory multiprocessor execution. And within each of these pieces, the user can take advantage of the vector execution capabilities of the compiler and hardware. When the program is executed, the Clustered FORTRAN library continues the approach of the Parallel FORTRAN library [18], mapping the user-specified parallelism onto the resources available. The user prepares one source program, augmented to reveal parallelism but independent of a run-time configuration, and it remains the job of FORTRAN to execute that program on the hardware available at run time.

Node Manager

Our software was intended to operate initially on two ES/3090 Model 600J computers. Rather than design it for "two six-ways" or "twelve one-ways" (either of which would have imposed a particular view of how to program a cluster of shared-memory multiprocessor computers), we wanted the software to be open-ended and flexible. We therefore allow users to write programs for arbitrary clusters of computers. The chosen configuration—a virtual cluster of virtual computers—is mapped at run time to the real cluster of real computers. This flexibility supports a variety of programming styles and run-time configurations. Indeed, it has always been a goal of Parallel and Clustered FORTRAN to make FORTRAN programs independent of the real hardware system, so that they can run on whatever hardware is available.

It is the job of the Node Manager software to construct and operate the virtual cluster specified by the programmer. A copy of the Node Manager runs on each of the real computers in the cluster, each controlling the allocation of resources on its own computer. But the Node Managers in the cluster also communicate as peers, jointly coordinating the allocation of resources across the complex.

When a Clustered FORTRAN application begins execution, the Node Manager running on the local real computer is asked to allocate the virtual configuration specified by the programmer. It communicates with the other Node Managers; together they allocate the specified virtual computers and connect them into a virtual cluster. Direct virtual communication links are initialized between each pair of virtual computers. (The direct links are used during execution of the application; the Node Managers are not involved in communications between the virtual computers.) Once the virtual cluster has been created and initialized, the application program is free to execute on the cluster. When it completes, it notifies the Node Manager on its local computer. The Node Manager communicates with its peers, and collectively they delete the virtual cluster from the system.

The Node Managers customize themselves when they first begin execution, as directed by a control file. The control file can be modified to suit local conditions. The file specifies such parameters as the real computers in the Clustered FORTRAN complex, the limits on resources that any application is permitted, and the names of the authorized system operators. The authorized operators may issue commands to control the operation of the complex. Most elements of the control file can be changed by operator commands while the system is running. Commands are also provided for such operations as querying status, canceling jobs, and coupling and uncoupling real computers from the real cluster.

The Node Managers also have the ability, at specific points in their processing, to call subroutines provided by the local installation. These subroutines can be used to modify or refine the operation of the system for the local environment. For example, the CNSF at Cornell defines for itself how accounting information is accumulated and processed for a clustered application. It screens jobs and prioritizes them. It controls dynamically the total number of jobs and the total number of virtual computers that can be active at once. It also defines and implements its own policies for balancing resources among users. CNSF strives to provide parallel-computing resources to a set of high-priority users while still supporting other users on a fully loaded system. The high-priority users expect quick turnaround time because they have adapted their programs for parallel execution. Priority for parallel users, though, means that other users may be delayed. CNSF is continually refining its methods and policies to balance the needs of both kinds of users, and it has developed facilities beyond those provided by the Node Manager to balance

the system dynamically between short-running and long-running jobs, between serial jobs and parallel jobs, and between high-priority users and other users [19].

Connection facility

As mentioned above in the Clustered FORTRAN overview, the real computers in a Clustered FORTRAN complex are joined by means of the Clustered FORTRAN connection facility, a combination of hardware and software that facilitates transmission of large amounts of data between the computers. The software interface to the connection facility is based on the Inter-User Communication Vehicle (IUCV) interface provided by VM/XA. IUCV permits communication between virtual computers on a single real computer; minor extensions were made to permit communication between virtual computers on different real computers.

Elements of the communication protocols should be mentioned. First, the nature of the communication affects the amount of data to be sent. Much of the communication in Clustered FORTRAN can be supported with short transmissions that convey synchronization information or identify specific functions to be performed. However, the application program may also ask for transmission of hundreds of megabytes of data when common blocks and arguments are to be copied. Second, user data move out of or into user storage synchronously with the user's request to send or receive. If the amount of data is small, the data may be packaged into a buffer that can be sent later, and the sender is permitted to proceed. If the amount is large, though, the sender may be delayed until all of the data have been sent. The choice is made by the FORTRAN library. A third element of the protocol is that, from the perspective of the library, the transmissions usually are asynchronous. The library is designed so that commands generally do not require acknowledgment. For example, when a program requests that a task be originated on a different virtual computer, the local library creates an identifier for the remote task and returns it to the calling program immediately. The calling task is then allowed to proceed with its execution; the protocol requires only that the caller be given a unique identifier for the new task in order to continue beyond the ORIGINATE statement. Subsequently, and asynchronously, the local library sends a command to the other virtual computer, directing it to do the processing required to create the actual task. This processing may proceed asynchronously and need not be acknowledged.

The communication paths required to support a Clustered FORTRAN application run from the FORTRAN library to a Node Manager (to obtain resources), from Node Manager to Node Manager (to allocate and initialize resources), and from FORTRAN library to FORTRAN library (to distribute and synchronize FORTRAN work and data). The same software interface is used for all of these interactions. System-level software actually moves the data between virtual computers. When the virtual computers are on the same real computer, the data can be copied easily. When the virtual computers are on different real computers, the data are sent over the high-speed hardware connection.

Clustered FORTRAN performance

Although Clustered FORTRAN is still relatively new at Cornell, applications have been executed, and performance measurements have been made. Some of these are the traditional observations of speedups as a function of the number of processors. Others, though, reveal that new possibilities for improving performance exist with clustering. The examples presented in this section show

- Speedups due to additional real processors.
- Speedups due to additional real memory. (A cluster of two ES/3090 computers has twice the real memory of a single ES/3090.)
- Simplification of programs due to additional virtual memory. (Each virtual computer used in a computation gives the programmer over 900 megabytes of virtual memory in which to place data.)

The initial users of Clustered FORTRAN were interested in performance, and they chose applications that were expected to adapt naturally to the underlying physical architecture. These users partitioned their data to use nonshared memory; sometimes large data arrays were communicated in the process. We have these programs available for measurement. We wanted to ensure that the implementation of Clustered FORTRAN enabled such applications to perform well. Applications will be found, however, for which the costs of communication between the ES/3090 computers will be too high to allow reasonable speedups. Research in algorithms for distributed-memory systems may identify alternative ways for implementing applications to minimize the communication costs [20–23]. Since Clustered FORTRAN allows the virtual configuration for executing a program to be varied from run to run, programmers of such algorithms will be able to experiment easily with a variety of configurations, ranging from a single scalar uniprocessor computer to a cluster of vector multiprocessor computers, using a single source program.

Matrix multiplication, a well-known algorithm that often benefits from additional processors, is the first algorithm for which we have measurements. The nature of the computation allows uniform partitioning of the work among computers, simply by giving each an equal amount of the result matrix to calculate. The Clustered FORTRAN language was used to split the calculation into two equal

Table 1 Speedups due to clustering of additional processors for a human genome application.

Problem size (number of clones)	*One computer (6 processors)*		*Two computers (12 processors)*	
	Speedup of total job (start to end of whole job)	*Speedup of FORTRAN program (first to final user statement)*	*Speedup of total job (start to end of whole job)*	*Speedup of FORTRAN program (first to final user statement)*
2000	5.03	5.71	6.08	10.93
4000	5.60	5.77	9.45	11.44
8000	5.77	5.81	11.13	11.78

pieces and to assign them to two virtual computers. The virtual computers were defined to be six-processor multiprocessors; one virtual computer was placed on each of the two real ES/3090 Model 600J computers in the real cluster. Parallelism within the virtual computers was easily achieved by exploiting the parallel matrix multiplication routine (DGEMLP) available in Release 4 of the Engineering and Scientific Subroutine Library (ESSL) [24]. With the use of DGEMLP on the pair of ES/3090-600J computers, execution rates above 1200 million floating-point operations per second have been achieved. This is 10.8 times faster (using twelve processors) than the serial version of the DGEMLP routine (DGEMUL) on the same machine.

Table 1 shows the speedups experienced for a program involving human genome research from the Los Alamos National Laboratory [25, 26]. (Speedup is the time for execution on one processor divided by the time for execution on multiple processors.) Results are shown for execution of the same Clustered FORTRAN program on a one-computer cluster (six processors) and a two-computer cluster (twelve processors). The table reports the speedup achieved in the application program itself, as determined from the time interval between the first and the last user statement (in FORTRAN). It also reports the speedup achieved over the whole job, including the system overhead of initializing and terminating the virtual cluster, as determined from the interval between the time the job began and the time it ended. As the problem size increased, the roughly constant system overhead was amortized over longer job times, so the two measures of speedup converge. The problems of interest at Los Alamos are even larger than the ones we measured. Furthermore, the processing time for this program increases as the square of the problem size. We therefore anticipate that these large problems will display speedup factors in excess of 11.8 with twelve processors.

The above examples emphasize the one major factor that is most readily associated with parallel processing—the benefit from additional processors. Clustered FORTRAN offers additional possibilities because of the flexibility of its virtual-cluster configurations. Experiments with a variety of virtual configurations have demonstrated performance and programming benefits that stem from clustering but are not directly related to additional processors. Some applications have so much data that memory itself, not CPU power, is the prime bottleneck to performance. In these cases, clustering offers new ways to improve performance.

Since each virtual computer is a complete virtual machine with its own virtual memory, a user can obtain additional virtual memory simply by defining additional virtual computers. Some of the Cornell users have applications that require several gigabytes. Previously they have had to write code that would read and write their data a piece at a time, in order to stay within the 999-megabyte limit of a VM/XA virtual machine. Furthermore, some of these programs iterate over the data, which means that the data must be read and written repeatedly. For problems in which the data can be partitioned, placed in distinct virtual computers, and operated on by local tasks, additional virtual computers may allow the programs to contain all of the data they need and to reference the data directly instead of by reading and writing external storage.

This can be a programming convenience, but it is also a way to improve performance. External storage is relatively slow compared to the electronic storage used by the processors of a computer. Clustering, even on a single real computer, can give a user access to additional real electronic storage, since the virtual memory of multiple virtual computers, each at the 999-megabyte limit, can be contained concurrently within the real electronic storage of a single real ES/3090 computer. This may eliminate the need not only for reading and writing from external storage but also for paging from external storage.

When a program is run on a pair of real ES/3090 computers, additional performance may be gained. These systems have two classes of electronic storage: a fast electronic storage (central storage) which is accessed directly by the processors as they execute instructions in the application program, and a larger, but slower, electronic storage (expanded storage), which cannot be used directly by the application program [27]. Both

categories of storage are realized with semiconductor memory devices. The operating systems of the real computers page data between these two categories of storage.

A Clustered FORTRAN program running on a pair of ES/3090 computers can have twice as much fast, directly accessible, central storage available for its execution as it can when running on a single computer. If a program pages on a single system but does not page when spread across the cluster, there may be a reduction in the total work performed for the program, and the program may execute faster with respect to both CPU time and elapsed time. In this case, system throughput may actually be improved by clustered execution.

Seismic modeling code [28] from the IBM Rome Scientific Center illustrates this effect. Using all six processors of an ES/3090-600J computer, the program runs only 1.95 times faster than it does with a single processor. A speedup factor closer to six could not be achieved because of paging overhead. However, a cluster of two ES/3090-600J computers has enough central storage to hold the entire problem and execute it without paging. When the problem was partitioned between the two real computers, the speedups achieved actually exceeded the number of processors allocated when two, four, or six processors were used. (The processors were divided equally between the two real computers.) A speedup of 10.96 was achieved when using all twelve processors in the cluster. The paging on the single-computer system was done between central storage and expanded storage. Had the program been run on a system that backs central storage with disk files instead of with expanded storage, one would expect even worse speedups on the single-computer system. **Table 2** summarizes the results of these experiments.

Most scientific problems are not inflexible with regard to storage requirements. Scientists can vary grid dimensions, storage structures, the number of variables retained rather than recalculated, and the degree of computational accuracy required. These parameters are often constrained artificially in order to make a problem fit into the limited amount of storage available on a particular computer. Because it makes additional storage accessible, clustering can ease these constraints.

One may ask, Why not simply expand the storage of one computer, instead of clustering several computers together? One reason is that the hardware designers may not be able to add more memory to a given computer. Memory systems are typically implemented with hierarchies of fast to slow storage. Factors such as cable lengths, power supplies, and cabinet dimensions may preclude adding more fast storage. Another reason is that the computer may be well balanced, with respect to memory and processors, for its normal run of jobs.

Table 2 Speedups due to clustering of additional real memory as well as processors for a seismic modeling application (adapted from [28]; reproduced with permission).

Number of vector processors	*Small problem*		*Large problem*	
	Speedup using one computer	*Speedup using two computers*	*Speedup using one computer*	*Speedup using two computers*
1	1.00	—	1.00	—
2	1.98	1.95	1.57	2.21
4	3.85	3.78	2.02	4.33
6	5.68	5.51	1.95	6.24
12	—	9.71	—	10.96

Additional, expensive memory may help only a few applications. Clustering, in contrast, lets an application use more memory without affecting the overall balance of the system. Given a group of existing computers and given a job that needs more memory than is available on any one of the computers, a system manager now has a choice. Buying a bigger machine is one way to run the job. Linking the computers into a cluster is another.

Concluding remarks

Clustered FORTRAN extends the supercomputing capabilities of the IBM ES/3090 and ES/9000 computer systems by allowing customers with appropriate applications to employ the resources of multiple high-end systems for the execution of single FORTRAN programs. Clustering can substantially reduce the time required to run such applications, since additional processors can be brought to bear on computations and since additional storage is available for data. Through clustering, a scientist can run problems today which otherwise would have to wait for future machines.

Clustered FORTRAN enables programmers to take advantage simultaneously of multiple computers with distributed memory, of multiple processors sharing memory, and of multiple vector-processing elements. Clustered applications can be, but need not be, executed with dedicated system resources. They can therefore be developed and executed in normal system environments. When, however, a customer such as the CNSF at Cornell desires it, all of the dedicated resources of two ES/3090-600J computers can be used for a single FORTRAN application.

The design of Clustered FORTRAN permits experimentation and leaves open paths for possible future exploration in parallel programming. For example, at all of the language interfaces where Clustered FORTRAN communicates data between virtual computers, the mode and type of the data involved are known. If it becomes necessary to translate number formats, perhaps to support clustering for machines of different architectures, the

library can be given the information needed for data conversion. This information can be accumulated by the compiler without requiring the user to specify modes and types explicitly. Similarly, virtual configurations need not map directly to the real configurations available, so virtual configurations can be used to simulate other clusters of computers. For example, a configuration of 32 virtual computers has been used to simulate the IBM Parallel Processing Compute Server installed at CERN in Geneva [29, 30]. Such experimentation may provide valuable information for the design of larger clustered systems.

Acknowledgment

In conclusion, we thank the Cornell National Supercomputer Facility. High-end customers such as CNSF are continually stretching the limits of their computing resources and continually demanding new capabilities of their computing systems. In the case of Clustered and Parallel FORTRAN, CNSF challenged us to provide advances in computing for FORTRAN programmers. CNSF then participated with us in devising solutions to the challenges and in testing the resulting offerings in a production environment. We thank CNSF for its challenge to us, for its involvement in our work, and for its provision of a facility where programmers may experiment with clustered scientific applications.

References

1. D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "Cedar—A Large Scale Multiprocessor," *Proceedings of the 1983 International Conference on Parallel Processing*, Columbus, OH, August 1983, IEEE Computer Society Press, pp. 524–529.
2. D. Kuck, E. Davidson, D. Lawrie, and A. Sameh, "Parallel Supercomputing Today and the Cedar Approach," *Science* **231**, 967–974 (1986).
3. P. Emrath, D. Padua, and P. Yew, "Cedar Architecture and Its Software," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, Vol. 1, Kailua, Kona, Hawaii, January 1989, IEEE Computer Society Press, pp. 306–315.
4. S. G. Tucker, "The IBM 3090 System: An Overview," *IBM Syst. J.* **25**, 4–19 (1986).
5. *Supercomputing Systems Extensions*, Order No. G320-9944, March 1991; available through IBM branch offices.
6. *CRAY X-MP Computer Systems Functional Description Manual*, Order No. HR-3005, Cray Research Inc., Mendota Heights, MN, 1987.
7. *IBM Parallel FORTRAN Language and Library Reference*, Order No. SC23-0431, March 1988; available through IBM branch offices.
8. A. Karp and R. Babb, "A Comparison of 12 Parallel FORTRAN Dialects," *IEEE Software* **5**, 52–67 (1988).
9. *IBM Clustered FORTRAN Language and Library Reference*, Order No. SC23-0523, May 1990; available through IBM branch offices.
10. *IBM Clustered FORTRAN Installation and Operation for VM*, Order No. SC23-0524, May 1990; available through IBM branch offices.
11. *Virtual Machine/Extended Architecture System Product General Information (VM/XA SP Release 2)*, Order No. GC23-0362, July 1988; available through IBM branch offices.
12. *Virtual Machine/Extended Architecture System Product CP Programming Services (VM/XA SP Release 2)*, Order No. SC23-0370, November 1988; available through IBM branch offices.
13. *CMS Application Program Development Reference (VM/XA SP Release 1 and Release 2)*, Order No. SC23-0402, March 1988; available through IBM branch offices.
14. D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *J. Supercomputing* **2**, 151–169 (1988).
15. C. Koelbel, P. Mehrotra, and J. van Rosendale, "Supporting Shared Data Structures on Distributed Memory Architectures," *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 14–16, 1990 [*SIGPLAN Notices (USA)* **25**, No. 3 (March 1990)], pp. 177–186.
16. M. Metcalf and J. Reid, *Fortran 90 Explained*, Oxford University Press, Oxford, England, 1990, p. 117.
17. F. W. Bobrowicz, S. H. Dean, D. A. Mandell, and W. H. Spangenberg, *LANL Multitasking Overview*, Order No. LA-UR 87-759, Los Alamos National Laboratory, 1987.
18. L. J. Toomey, E. C. Plachy, R. G. Scarborough, R. J. Sahulka, J. F. Shaw, and A. W. Shannon, "IBM Parallel FORTRAN," *IBM Syst. J.* **27**, 416–435 (1988).
19. P. Bogdonoff, "Accommodating Parallel Processing in a Mixed Serial-Parallel Computing Environment," presentation to the SUPER Conference, Lexington, KY, April 1989.
20. S. Johnsson, "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures," *J. Parallel & Distributed Computing* **4**, 133–172 (1987).
21. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, Vol. 1, Prentice-Hall Publishing Co., Englewood Cliffs, NJ, 1988.
22. Y. Azmy, "On the Adequacy of Message-Passing Parallel Supercomputers for Solving Neutron Transport Problems," *Proceedings of Supercomputing '90*, New York, November 12–16, 1990, IEEE Computer Society Press, pp. 693–699.
23. G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel & Distributed Computing* **7**, 279–301 (1989).
24. *Engineering and Scientific Subroutine Library Guide and Reference (Release 4)*, Order No. SC23-0184, December 1990; available through IBM branch offices.
25. D. Torney, C. Whittaker, S. White, and K. Schenk, "Computational Methods for Physical Mapping of Chromosomes," *Proceedings of the First International Conference on Electrophoresis, Supercomputing, and the Human Genome*, Tallahassee, FL, April 10–13, 1990, World Scientific Publishing Company, pp. 268–278.
26. S. White, D. Torney, and C. Whittaker, "A Parallel Computational Approach Using a Cluster of IBM ES/3090 600Js for Physical Mapping of Chromosomes," *Proceedings of Supercomputing '90*, New York, November 12–16, 1990, IEEE Computer Society Press, pp. 112–121.
27. E. I. Cohen, G. M. King, and J. T. Brady, "Storage Hierarchies," *IBM Syst. J.* **28**, 62–76 (1989).
28. A. Kamel, P. Squazzero, and V. Zecca, "Large Scale Computing on Clustered Vector Multiprocessors," *Proceedings of Supercomputing '90*, New York,

November 12–16, 1990, IEEE Computer Society Press, pp. 418–427.
29. "Cross-Point Switch Based IBM/370 Parallel Processing Compute Server," IBM/CERN Joint Study Report, IBM Germany Research and Development Laboratory, Boeblingen, Germany, September 1990.
30. E. M. Ammann, R. R. Berbec, G. Bozman, M. Faix, G. A. Goldrian, J. A. Pershing, Jr., J. Ruvolo-Chong, and F. Scholz, "The Parallel Processing Compute Server," *IBM J. Res. Develop.* **35**, 653–666 (1991, this issue).
31. R. J. Sahulka, E. C. Plachy, L. J. Scarborough, R. G. Scarborough, and S. W. White, "FORTRAN for Clusters of IBM ES/3090 Multiprocessors," *IBM Syst. J.* **30**, 296–311 (1991).

Received October 1, 1990; accepted for publication April 1, 1991

Leslie J. Scarborough *IBM Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304.* Ms. Scarborough joined IBM in 1978 as an application programmer in East Fishkill, New York. Her initial assignments involved numeric computing with FORTRAN for graphics postprocessing. She began working on engineering and scientific compiler development in 1983, and contributed to Parallel and Clustered FORTRAN while a member of the IBM High-Performance/Supercomputing Systems Development Laboratory in Kingston, New York. She received an IBM Outstanding Innovation Award for her contributions to the Parallel FORTRAN project. In 1990 Ms. Scarborough joined the Palo Alto Scientific Center. She is the IBM representative to the ANSI X3H5 committee, a group that is defining a model for parallelism in 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 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, and worked 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 computer. 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 been working on 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 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 to work on scientific and engineering processor development, architecture, and system design. From 1986 to 1988, he worked with the Computational Physics Group at Lawrence Livermore National Laboratory. In 1989, he joined the Clustered FORTRAN/HIPPI development group in the High-Performance/Supercomputing Systems Development Laboratory in Kingston, New York. He is currently in the Advanced Workstations Division Processor Architecture and Performance group in Austin, Texas. His primary interests are parallel and distributed architectures and memory hierarchies.