IBM Parallel FORTRAN

by L. J. Toomey
E. C. Plachy
R. G. Scarborough
R. J. Sahulka
J. F. Shaw
A. W. Shannon

IBM Parallel FORTRAN is a compiler and library for writing and executing parallel programs. It provides language extensions for explicitly programming in parallel, and it also provides compiler enhancements for automatically generating both parallel and vector code. Parallel FORTRAN offers a language for parallel programming that is independent of the machine configuration and the operating system. The combination of Parallel FORTRAN and IBM 3090 multiprocessors can provide a significant reduction in turnaround time for applications.

Parallel processing is a widely accepted technique for reducing the turnaround time for engineering and scientific computation-bound applications.[1] Recently there has been a renewed interest in parallel processing, and today a number of computer manufacturers are offering parallel processors.[2] The IBM 3090 multiprocessor is one such system. The 3090 supports up to six scalar and vector processors, all sharing a global memory.[3] IBM Parallel FORTRAN enhances the parallel processing capabilities of the 3090.[4]

Parallel FORTRAN is a compiler and library that allows FORTRAN programmers to exploit parallel processing on IBM 3090 multiprocessors. It operates under the MVS/XA and the VM/XA System Product (SP) operating systems. Parallel FORTRAN was developed in IBM jointly by the Programming Systems Santa Teresa Laboratory, the Palo Alto Scientific Center, and the Data Systems Division in Kingston, and it has been available on a limited basis since March 1988.

FORTRAN was introduced by IBM in 1954 and is the predominant language for scientific and engineering applications.[5] IBM FORTRAN compilers and libraries have been modified to support the evolving IBM hardware. VS FORTRAN Version 1 introduced a basic form of parallelism with the Multitasking Facility (MTF).[6] Automatic vectorization was added to VS FORTRAN Version 2 to support the IBM 3090 Vector Facility.[7,8] Parallel FORTRAN continues the evolution by providing language extensions for explicitly programming in parallel and by providing compiler enhancements for automatically generating both parallel and vector code. The extensions for parallel execution allow programmers to exploit the full hardware capability of the IBM 3090 multiprocessor.

Parallelism can occur in different forms 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. Sequences of statements may be eligible for concurrent execution. Parallel work may occur nested within other parallel work. Extensions in Parallel FORTRAN allow the specification of these various forms of parallelism, wherever they may occur.

This paper discusses the extensions in Parallel FORTRAN to support parallelism as follows:

- 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 subroutines
- Library routines for synchronizing parallel pieces of work

Also covered are execution environments and performance considerations when using Parallel FORTRAN. The paper concludes with examples that show both parallel and vector processing of a matrix-multiplication program.

Background

Parallel FORTRAN evolved from a prototype that had been developed as part of a joint study with Cornell University. The Cornell Theory Center is one of five National Science Foundation supercomputer centers that were initiated in 1985 to provide supercomputing resources for scientists nationwide. The Theory Center's primary supercomputing resource is the Cornell National Supercomputer Facility (CNSF).[9] The CNSF configuration is based on an IBM 3090 Model 600E with six Vector Facilities.

An objective of the Cornell Theory Center has been to explore parallelism and to provide parallel computing in a production environment. A parallel FORTRAN compiler was needed to achieve this objective; it was defined and specified jointly by IBM and Cornell. The resulting compiler, called the Parallel FORTRAN Prototype, was developed by IBM and delivered to Cornell in January 1987. Cornell has been using the prototype since then to explore parallel computing on its six-way IBM 3090.[10]

In addition to discussions with Cornell, the Parallel FORTRAN Prototype was built on experience gained with three previously existing parallel program packages developed at IBM. These three packages provide FORTRAN programmers the following ways to access the multiprocessing capabilities of an IBM mainframe:

- *The VS FORTRAN Multitasking Facility (MTF).* MTF is a set of subroutines incorporated into the VS FORTRAN Library,[6] that allow FORTRAN subroutines to be executed asynchronously from the mainline FORTRAN program. MTF operates only under the MVS operating system.
- *Environment for Parallel Execution (EPEX).* EPEX was developed at the IBM T. J. Watson Research Center to support the experimental, highly parallel RP3 computer,[11] which executes multiple instances of the same program, each running on its own processor. Each processor in the system, besides having its own local memory, is linked to the other processors by a common shared memory. EPEX simulates this environment under VM by using multiple virtual machines and writable shared-memory segments. The shared memory is used for sharing data between the instances of the program and for special coordination and synchronization variables defined by EPEX.
- *Loosely Coupled Array of Processors (ICAP).* ICAP was developed in IBM Kingston to support an experimental parallel system that consisted of a set of array processors attached via channels to an IBM mainframe system.[12] ICAP allowed FORTRAN subroutines to be executed asynchronously on the attached array processors. It now provides the ability to execute its programs totally on an IBM mainframe. ICAP operated under both the VM and the MVS operating systems.

Programming of parallel applications

If a program is to be executed in parallel, it is necessary to assign different pieces of the program to different processors during execution. This requires that the pieces eligible for execution in parallel be identified. In some cases, it is possible for the parallel pieces to be identified automatically by the compiler. In other cases, the programmer may have to specify the parallel pieces explicitly. Parallel FORTRAN supports both methods of achieving parallel execution.

Specifically, Parallel FORTRAN provides the following:

- Extensions to the compiler for automatically generating parallel code
- Extensions to the language for explicitly programming in parallel
- Extensions to the library for synchronizing parallel execution through locks and events

The FORTRAN program identifies the pieces of work that are eligible to run in parallel. The FORTRAN library then maps the parallel pieces of work onto virtual processors that are known as FORTRAN processors. The operating system maps the FORTRAN processors onto the real machine processors.

The number of FORTRAN processors is specified as a run-time option. By varying this number, the user can control the maximum degree of parallel execution that FORTRAN attempts to achieve during a given execution of an application. The number of real processors available depends on the machine configuration, on other work being executed in the system, and on the relative priorities assigned to the work. If more real processors are assigned to the parallel program by the operating system, more parallel execution may be achieved.

Automatic parallelism is the simplest way to introduce parallelism into an application.

The hardware configuration and differences between operating systems are not exposed in the FORTRAN program. Instead, the program simply specifies sections within the program that can be executed concurrently. The compiler and library accept this specification and execute the parallel sections, using the processors available to the program.

Automatic parallel

Automatic parallelism is the simplest way to introduce parallelism into an application. A new compiler option requests that the compiler analyze nests of DO loops to determine whether they are eligible for parallel execution. Parallel code is generated only if the results will be not be changed by parallel 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 dependences that prevent parallel execution, the compiler determines whether 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 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 analyzes nests of DO loops for both parallel and vector execution. Individual loops may be selected for vector or parallel execution, or for both vector and parallel execution. A loop selected for parallel may contain inner loops in any mode; a loop selected for vector may contain only inner loops which are scalar and serial. If it is found to be cost-effective, loops may be broken apart and different pieces executed in different modes.

The directives provided by VS FORTRAN to influence automatic vectorization have been extended to support the automatic parallelization. A user may now indicate a preference for parallel or serial code for a given loop in the same manner as a preference for scalar or vector code might have been indicated previously. The user may also express a preference for the number of processors to assign to the loop at run time and for the number of iterations to be grouped together as a unit of work at run time.

Parallel language extensions

Although automatic detection of parallelism may be an easy way to introduce parallelism, it does have some limitations. Primary among these is the requirement that the answers not change during execution in parallel. Some algorithms are able to run effectively in parallel even though they contain data dependences that can cause their results to change from parallel run to parallel run. In chaotic relaxation, for example, the algorithms are designed to converge, and they are deemed to be successful when they converge to a value with some small tolerance. Any value with this tolerance is as acceptable as any other. Automatic parallelization insists on producing the same value as is produced by a serial execution of the program, and so it does not make parallel a program with such data dependences. Therefore, although automatic detection of parallelism is an easy way to obtain parallel execution, it does not provide the complete answer for a parallel programmer.

For this reason, Parallel FORTRAN provides language extensions with which the programmer may specify parallel execution. The language extensions can be categorized into two types: in-line extensions and out-of-line extensions. The in-line extensions, which define parallelism within a routine, identify loops or blocks of statements that can be executed concurrently. The out-of-line extensions, which define par-

allelism across routines, identify subroutines that are eligible for parallel execution. Both types of extensions can be nested. Thus programs that have been written to exploit parallel execution can be encapsulated as library routines and can be invoked from other programs, either serially or in parallel, in the usual FORTRAN manner.

The in-line extensions permit the code within a subroutine to be dynamically parceled out to more than one processor for execution. The parallel code sequences operate on the arrays and scalars known to the subroutine. In this, they are similar to DO loops, which are automatically parallelized. They are unlike the automatically parallelized loops, however, in that the programmer can specify operations that contain data dependences.

The out-of-line extensions permit a user to create new, disjoint, asynchronous execution environments for one or more FORTRAN subroutines. Each disjoint collection of subroutines, called a *task*, is independent of all other tasks; the execution of one task cannot affect another unless the programmer explicitly shares data between them. Thus, when algorithms have been programmed and debugged, they can be easily encapsulated within their own task environments. When users explicitly share data between tasks, they are also explicitly identifying data areas that should be reviewed if errors occur during parallel execution.

Parallel loops. A parallel loop is one in which each iteration of the loop may be executed concurrently. Some unspecified number of processors, possibly one per iteration, may be used to execute the loop. The number of processors is determined at run time, and that number can vary from one to the number of FORTRAN processors specified at run time. The order in which iterations are executed is therefore not guaranteed. All iterations are completed, however, before execution continues beyond the end of the loop. 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 that is similar to a DO loop, a simple form of which is shown in Figure 1.

This simple form of a parallel loop permits iterations to execute in parallel. Suppose, however, that a programmer needs to compute a temporary result, such as XTEMP, within an iteration. A statement like XTEMP=A(I)*B(I) cannot work in parallel. When several processors execute the statement simultaneously, each computing with a different value of I, only one value of XTEMP is saved, i.e., the one that by chance was stored last. Each processor therefore 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 2.

Figure 1 Simple form of PARALLEL LOOP

```
      PARALLEL LOOP label index=i1,i2,i3
       statements
label CONTINUE
```

Figure 2 Example of PRIVATE statement

```
      PARALLEL LOOP 1 I=I1,I2
       PRIVATE (XTEMP)
       XTEMP=A(I)*B(I)
       C(I)=XTEMP*D(I)
     1 CONTINUE
```

Further, 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. DOFIRST and DOFINAL statements are provided for this purpose. These statements delimit, respectively, a prolog and epilog 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 prolog or epilog. DOEVERY delimits the remaining body of the loop that is executed on each iteration.

Figure 3 Example of PARALLEL LOOP extensions

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

Figure 4 Simple form of PARALLEL CASES

```
      PARALLEL CASES
       PRIVATE (var, ...)
      CASE
       statements
      CASE
       statements
      CASE
       statements
      END CASES
```

The example in Figure 3 shows the use of these statements to implement a sum reduction. The problem is to compute a global sum GSUM of a vector AVAL. A private variable PSUM is initialized to zero for each processor and 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. When all elements have been summed, each processor adds its private partial sum PSUM into the global total sum GSUM. This final addition is done under control of a lock, so that GSUM is updated by only one processor at a time. (This example, which is used to illustrate the parallel loop, may not contain enough processing for profitable parallel execution.)

Parallel cases. It is often possible to execute blocks of statements in parallel. The blocks may contain straight-line code or loops, and the loops may be either parallel or vector loops. What is significant is that the blocks may be processed concurrently. At the limit, each block can be executed by a different processor, the number of which is not known but may range from one up to the number of blocks. The exact number is 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. That is to say, each block is computationally independent of the others, or the data interactions that arise between blocks are either controlled or intentional.

The PARALLEL CASES structure is provided to simplify the programming of such parallel blocks of statements. A simple form of parallel cases is shown in Figure 4. The three illustrated cases may execute concurrently, but there is no guarantee of the order of their execution. As with parallel loops, the cases may employ private variables as needed. All cases are completed before execution continues beyond the END CASES statement.

It is often helpful to make some cases wait until preceding cases complete. An early case could then, for example, compute data to be used by more than one subsequent case. To facilitate this, cases may be numbered, and any case may wait for any specified preceding case. A sample is shown in Figure 5. In this manner, a program containing an acyclic graph of dependences may be translated into a series of parallel cases.

Both parallel loops and parallel cases may contain nested parallel loops and parallel cases. Input and output statements may be used within parallel loops and parallel cases.

Parallel tasks. Parallel loops and cases are statements that yield in-line parallelism, spreading the work of a given subroutine across multiple processors. Out-of-line parallelism, in contrast, creates new and distinct FORTRAN execution environments and permits each of these distinct environments to execute concurrently. A shorthand name for these environments is *tasks*.

A Parallel FORTRAN program begins execution in a task referred to as the *root task*. The ORIGINATE statement may be used to create more tasks. Each task that is originated in this way has an identifier and its own storage. The identifier is returned to the programmer by the ORIGINATE statement and is used in other statements to manipulate the task. The storage associated with a task is private to that task and persists until the task is terminated. When a task is no longer needed, it can be deleted with a TERMINATE statement. The ORIGINATE and TERMINATE statements are shown in Figure 6.

Work is assigned to a task with the DISPATCH and SCHEDULE statements, which name a subroutine to be executed asynchronously in the subtask and list the arguments to be passed to the called subroutine. The user may specify a particular task to be called or may request that the library choose any available task. Dispatched tasks complete their work automatically. Scheduled tasks require that the user subsequently issue a corresponding wait. Sample statements are shown in Figure 7.

Tasks, like subroutines in traditional FORTRAN, may communicate through arguments and common blocks. Figure 8 shows the optional clauses on the SCHEDULE and DISPATCH statements to control the use of common blocks. A SHARING clause may be used to name 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 subtask uses

Figure 5 Extended form of PARALLEL CASES

```
PARALLEL CASES
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 (1,2,3)
 statements
END CASES
```

Figure 6 ORIGINATE and TERMINATE statements

```
ORIGINATE ANY TASK itask
TERMINATE TASK itask
```

Figure 7 Sample SCHEDULE and DISPATCH statements

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

Figure 8 Options for SCHEDULE or DISPATCH statement

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

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 a task when work is assigned and copied out of the task when work is completed. Both the superior and the subordinate tasks have a private copy of these common blocks. COPYINGI and COPYINGO name commons that are to be copied respectively only into or only out of the subtask.

Tasks can be assigned a variety of pieces of work. A TAGGING clause is provided to allow the programmer to name or tag a particular piece of work. The values of the tags are saved when the task is scheduled. Subsequently, when the programmer 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 the specific piece of work that had been assigned to the task that 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, and wait for all tasks. Figure 9 shows the variations of the WAIT FOR statement, including its optional TAGGING clause.

Parallel library. The Parallel 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 the management of locks and events. Locks may be used to ensure that only one processor at a time gains access to a resource, such as a variable using a global counter. Events may be used to make a task wait until another task has reached some point in execution. Synchronization techniques of many kinds can be implemented using locks and events.

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 separate trace file can be produced for each task, or a single trace file can be produced covering all tasks. 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 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 9 Example of WAIT FOR statements

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

Execution of parallel applications

Three conditions are required for a program to execute in parallel: (1) multiple pieces of parallel work ready for execution; (2) multiple FORTRAN processors associated with the program; and (3) multiple real processors available to it. The programmer controls the first two conditions directly; the Parallel FORTRAN language and compiler are used to identify the parallel pieces of work, and a run-time option is used to specify the number of FORTRAN processors. The third condition, the number of real processors, is controlled by the operating system and depends on the amount of other work being done by the system and on the relative priorities assigned to that work. As real processors become available, the operating system can allocate them to the parallel program, and more of the parallel pieces of work can be executed simultaneously. The Parallel FORTRAN execution environment is shown in Figure 10.

Parallel execution and performance are improved when the queue does not empty.

Parallel FORTRAN allows the programmer to identify more pieces of parallel work than there are FORTRAN processors. The additional pieces may be thought of as sitting in a queue. Thus, when a FORTRAN processor finishes one piece of work, it selects the next piece from the queue and executes it. The FORTRAN processors can continue to run without operating system interactions as long as the queue contains work. When the queue empties and refills, however, the FORTRAN processors must use the less efficient mechanisms provided by the operating system to suspend and restart themselves.

The degree to which the queue of pending parallel work can be kept from emptying varies from application to application. Parallel execution and performance are improved when the queue does not empty, because operating system overhead is then normally not required to schedule and synchronize parallel work. The language extensions in Parallel FORTRAN and its support for multiple levels of parallel execution both facilitate the identification and execution of a continuing stream of eligible parallel pieces of work.

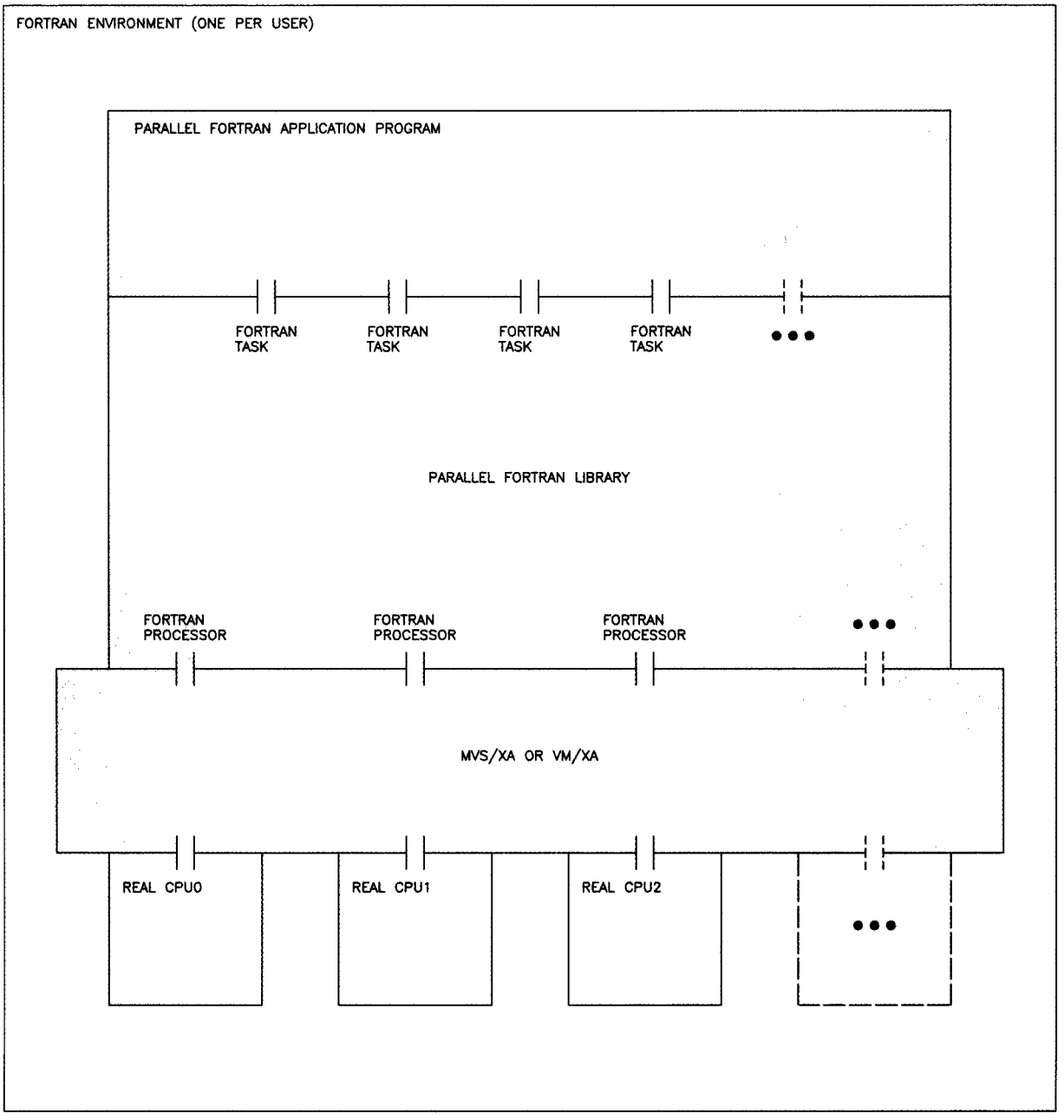
The method used for representing FORTRAN processors depends on the operating system. Under MVS/XA, each FORTRAN processor is implemented as an MVS task. Under VM/XA, where programs are run in virtual machines, the virtual machine is made into a virtual multiprocessing machine and each FORTRAN processor is executed by one of the virtual CPUs within the virtual multiprocessing machine. This is described in more detail subsequently in this paper. The programmer of a parallel program does not need to know about this difference between operating systems. Regardless of the operating system, the programmer refers only to the FORTRAN processors. It is the job of the FORTRAN compiler and library to provide these FORTRAN processors in a portable manner.

Other mechanisms have been used by other parallel processing packages for the IBM 3090 multiprocessors. The VS FORTRAN Multitasking Facility (MTF), for example, uses the technique of suspending and restarting tasks for each independent piece of work. This can result in two interactions with the operating system for each dispatch of a piece of parallel work. A different approach, microtasking,[13] keeps the processors in a *busy-wait spin loop* when they have nothing to do. This avoids the overhead of suspending and restarting work by the operating system for the user of microtasking, but the cycles used for spinning are lost to other users in the system.

Parallel FORTRAN programs may be run on dedicated or undedicated systems. On a dedicated system, the program runs most quickly. On an undedicated system, the speed of the parallel program is affected, because the real processors are used to support execution of other concurrent programs. Relative priorities may be used to bias the operating system toward or against the parallel program.

The programmer should have in mind whether the computer system is dedicated or undedicated. On a dedicated system, the programmer might assign each FORTRAN processor a fixed and equal amount of work to do concurrently. This is called *static mapping*, and it works well when each FORTRAN processor is assured of being given a real processor. Because

Figure 10 Parallel FORTRAN execution environment



equal amounts of work are assigned to each processor, they should all complete work at the same time. Therefore, they all remain busy as long as there is parallel work assigned to them. However, in an environment where there is contention for real processors among multiple users, this static mapping may not perform well.

Suppose an application has been statically mapped and all processors are working on it. Further suppose

that one processor is taken away to do a small piece of work for the operating system, while the other processors continue to execute on their statically mapped partitions. When these other processors finish their partitions, roughly all at the same time, the interrupted processor still has some work to do on its partition, because of the time it lost working for the operating system. Also, because the program has divided its work statically, there is nothing for the other processors to do until the final processor finishes its partition. All processors are affected by a temporary loss of one processor.

On undedicated systems, an application that partitions work dynamically rather than statically probably performs better. Dynamic partitioning allows the program to make use of real processors as they become available during execution. Parallel loops and cases and automatically parallelized loops are executed with dynamic scheduling. Certain out-of-line language statements, such as WAIT FOR ANY TASK, also allow dynamic load balancing on the available real processors. The matrix-multiply example described later in the paper explores this issue further.

When a Parallel FORTRAN program is executed with a single FORTRAN processor, the order of execution of the program is repeatable. The statements always execute in the same order for the same data. This allows for the development and debugging of a parallel program in an environment where bugs are reproducible. When the program is executed with multiple FORTRAN processors, the order of execution is not repeatable. Such bugs may be thought of as nondeterministic and not easily reproducible. It is beneficial to remove the deterministic class of bugs prior to parallel execution.

Parallel processing within a virtual machine on VM/XA

Under VM/XA, as indicated earlier in this paper, programs are run within virtual machines. When a Parallel FORTRAN program is executed, the virtual machine is made into a virtual multiprocessor. Each FORTRAN processor is executed by a virtual CPU within that virtual multiprocessor. (See Figure 11 for an illustration of this point.)

Programs under VM/XA are normally executed under control of a simple operating system, CMS, which runs within the virtual machine. CMS, however, is not a multiprocessing operating system. It assumes that it is running on a uniprocessor, and it cannot be executed concurrently by multiple virtual CPUs within a virtual machine. The virtual CPUs that are executing as FORTRAN processors, therefore, cannot be allowed to execute the internal routines of CMS concurrently and asynchronously. Nonetheless, the FORTRAN processors may from time to time require a service from CMS, such as input, output, or storage allocation.

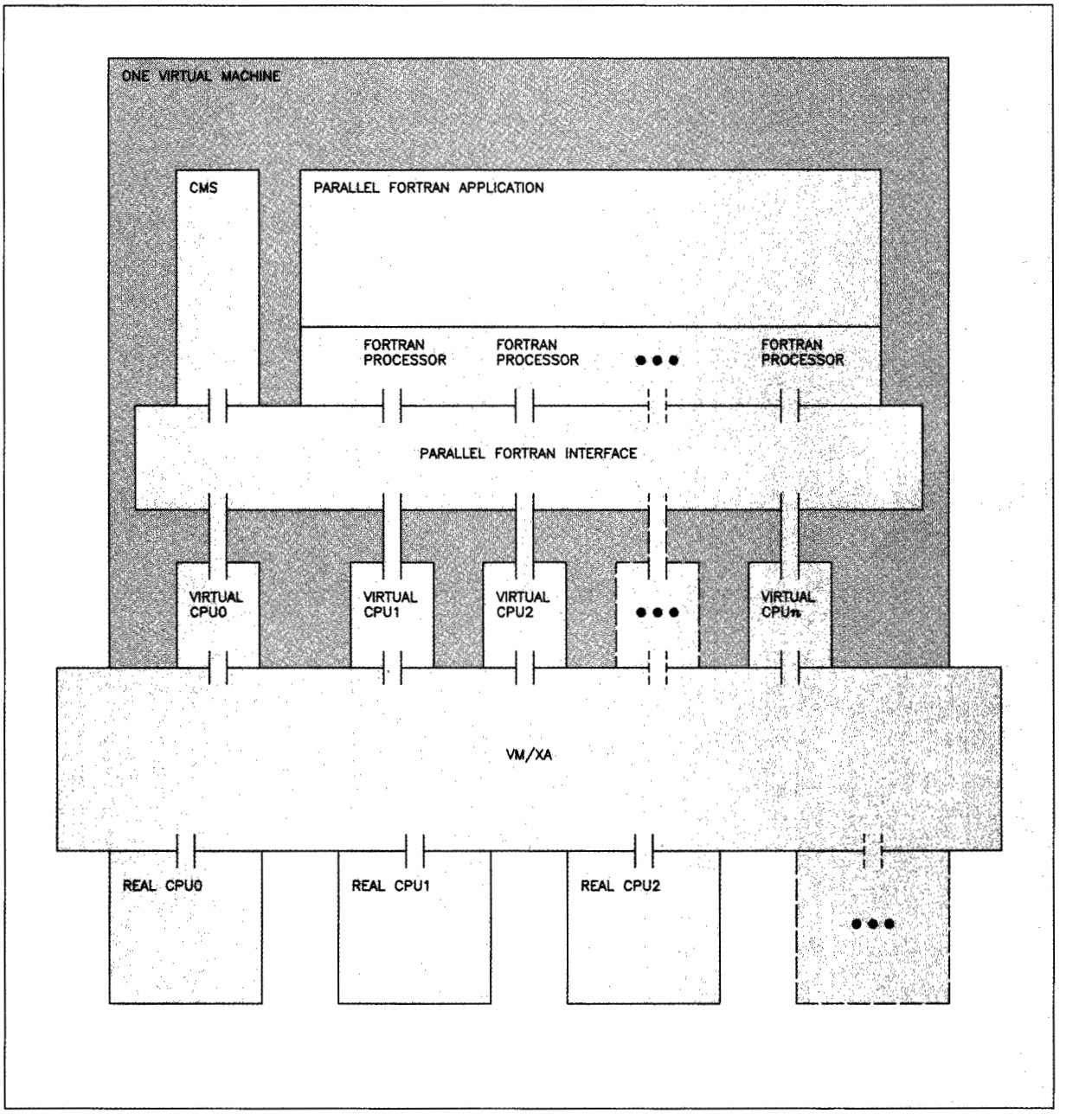
The solution to this problem is to define one virtual CPU—in addition to the virtual CPUs used as FORTRAN processors—for use as a CMS processor. When a FORTRAN processor requests a CMS service, the request is intercepted and queued for execution by the CMS processor, and the requesting FORTRAN processor is suspended until the CMS processor completes the request. The CMS processor executes these requests one at a time, finishing each completely before beginning the next. Each FORTRAN processor, since it is suspended while its request is processed, sees its requests handled in a fully synchronous manner, just as though it were on a uniprocessor. The CMS processor executes requests, one at a time and from start to finish in a fully synchronous manner, just as though it too were on a uniprocessor. This maintains the integrity of the CMS internal implementation.

FORTRAN processors not executing CMS requests are able to run without impediment from this serialization. Parallel FORTRAN is intended for use with large computationally intensive applications, and requests for CMS services should therefore be occasional rather than frequent. If repeated use were made of CMS services, alternative processing mechanisms would have to be pursued.

Performance of parallel applications

The major reason for using Parallel FORTRAN is to reduce the real time required to execute a FORTRAN program. The time reduction is achieved when multiple processors simultaneously execute portions of a single application program. Parallel FORTRAN does not reduce the total number of CPU cycles required to execute a program; in fact, a modest increase in CPU cycles is normally required. Instead, it allows a program to be split into multiple independent instruction streams. When these are executed simultaneously by different CPUs of a 3090 multiprocessor system, the program receives cycles from each of the assigned CPUs. Thus the program receives more CPU cycles in a given span of real time, and it completes its computation more quickly.

Figure 11 Parallel processing within a virtual machine on VM/XA



For programs that are vectorizable, the IBM 3090 Vector Facility also reduces the real time required to execute a FORTRAN program. The Vector Facility improves performance because it uses fewer CPU cycles than the scalar processor for many numerically intensive computations. The Vector Facility can be viewed as providing faster CPUs to the program. Vector and parallel execution complement each other. Their combined use results in more and faster CPUs executing on a program and can lead to

a larger reduction in the real time required to execute a program than either vector or parallel alone.

The improvement in turnaround time for an application converted to parallel is limited by the amount of serial processing that remains in the converted application. Not everything can be done in parallel. For example, the reading of initial data and the printing of final results are often done by a single processor. The following equation, often referred to as Amdahl's law, can be used to estimate an upper limit for the speedup expected for an application. Given the fraction of the original serial execution time that can be converted to parallel, p, and the fraction that must remain in serial, $1 - p$, the equation computes the maximum speedup for a given number of processors, n, as follows:

$$\text{Speedup} = \frac{n}{n(1 - p) + p}.$$

The improvement in turnaround time for an application converted to parallel is limited by the amount of serial processing that remains in the converted application.

Measurements of performance. Applications vary in the degree to which they can be parallelized. An indication of this can be seen in results of measurements on four applications representing different areas of scientific research. Measurements were taken using from one through six processors. Figure 12 shows the speedups achieved. The speedups shown are relative to the serial vectorized versions of the applications. The primary factors affecting the speedup of an application are the number of processors allocated to an application and the percentage of the application's processing that can execute in parallel.

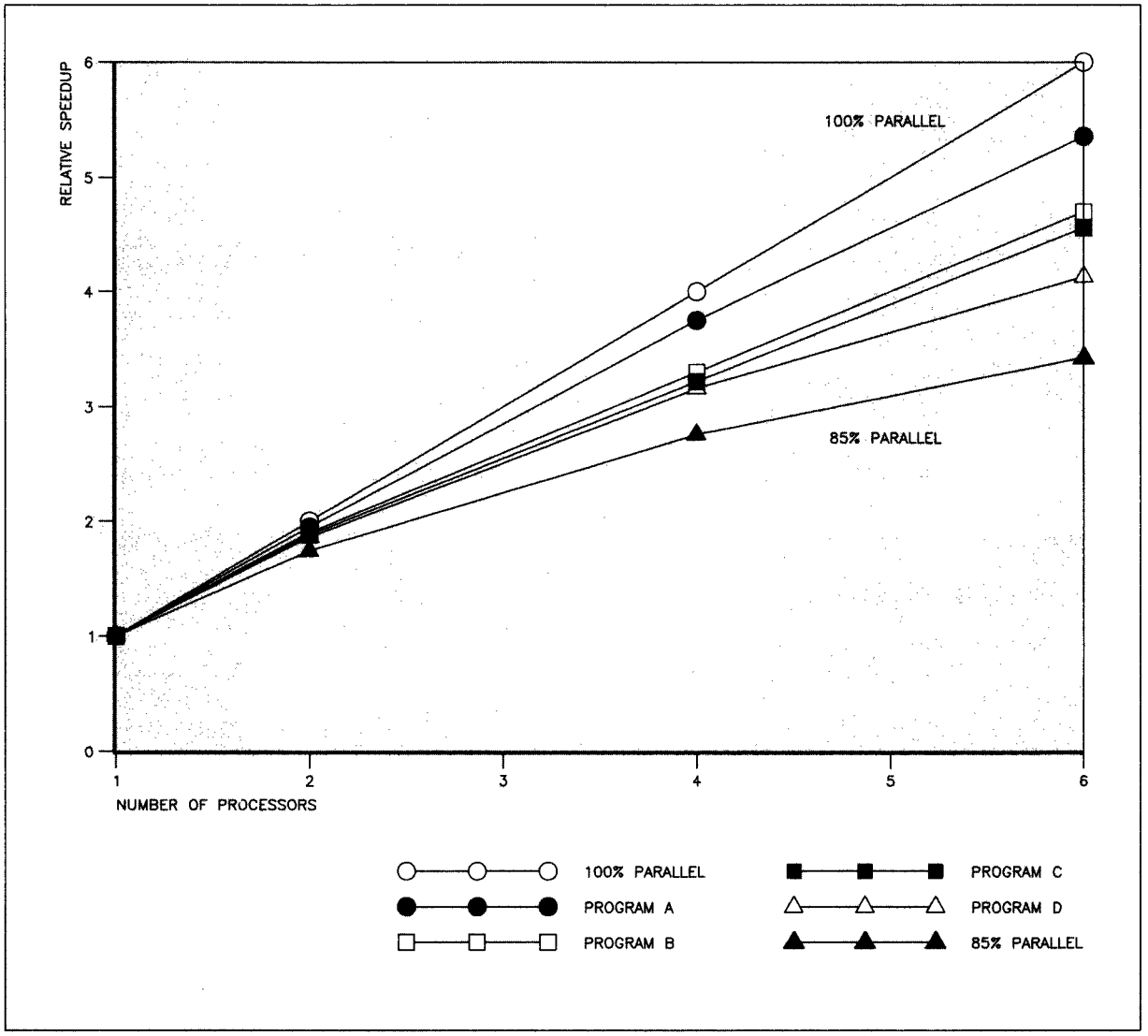
Programs A and B were measured at Cornell University on a VM/XA system using the Parallel FORTRAN prototype.[10] Program A, which studies protein folding and the 3D structure of polypeptides, uses a Monte Carlo technique to evaluate the free energy of the system. It was parallelized with the out-of-line language extensions. Program B, which involves research on statistical methods, performs multiplication, factorization, and inversion of very large matrices using the automatic parallel and automatic vector compiler options. Programs C and D were measured under MVS/XA, using the released version of Parallel FORTRAN. Program C is a high-energy physics application employing a Monte Carlo simulation for the simulation of quantum chromodynamics. It too is a vector program, but it uses parallel subroutines. Program D is a thin-layer fluid dynamics application using parallel tasks. All runs were made in a dedicated environment on a 3090 Model 600E equipped with six Vector Facilities.

The four programs ran from 4.1 to 5.3 times faster on the six-way machine; therefore they show effective parallelism of 90 to 97 percent, according to Amdahl's law. Effective percent parallelism is calculated by observing the actual speedup of an application and then using Amdahl's law.

Parallel programming guidelines. The results of the measurements of parallel applications show that the speedups can vary and that each application is different. It is difficult to predict in advance how a given application will perform. However, the following are guidelines for writing successful parallel programs on the IBM 3090:

- *Optimize code for serial processing.* Traditional optimization remains just as important in parallel codes as it does in serial codes.[14] Optimization reduces the absolute number of CPU cycles required to execute the program. Parallelization by itself merely spreads the remaining cycles across more than one processor. It is still important to optimize the code for the scalar and vector capabilities of the processor when writing a program with Parallel FORTRAN.
- *Maximize the parallel use of multiple Vector Facilities.* For best performance, a program should be structured to take maximum advantage of multiple Vector Facilities. In general, good vector operation should not be sacrificed to obtain parallel operation.
- *Minimize the work that must be done serially.* The time it takes to do this work is part of the minimum amount of time it will take a program to

Figure 12 Performance of selected application programs



execute. In other words, the more serial the work, the slower the program execution.

- *Minimize the overhead due to executing parallel constructs.* Overhead adds to the minimum amount of time it takes to execute a program. In Parallel FORTRAN, overhead can be minimized by several means. Originating tasks once and assigning work to them many times will save the repeated overhead of originating and terminating tasks. Distributing parallel work in larger rather than smaller pieces reduces the number of distributions and the processing required to do the distributions. However, judgment is required, because the goals of assigning work dynamically to balance the workload on the processors and that of assigning work in large chunks are to some extent in conflict.
- *Assign workloads that are dynamically self-balancing.* Unbalanced workloads have the same negative effect on performance as serial work. Parallel FORTRAN provides several facilities to help dynamically balance workloads. PARALLEL LOOPS, PAR-

Figure 13 Serial matrix multiplication

```
      COMMON /AC/ A(500,1500), B(1500,200), C(500,200)
      REAL*8 A, B, C, T
       .
       .
       .
      DO 20 I=1, 500
       DO 20 K=1, 200
        T=0.D0
        DO 30 J=1, 1500
30       T=T+(B(J,K)*A(I,J))
20     C(I,K)=T
```

ALLEL CASES, and automatically parallelized DO loops are all dynamically load-balanced by the library. The SCHEDULE statement, when used with the WAIT FOR TASK or WAIT FOR ANY TASK statements, and the DISPATCH statement both provide a way to assign additional work to tasks without waiting for all tasks to complete their assignments. Parallel locks and events allow the programming of many different types of application-specific load-balancing algorithms.

- *Minimize storage contention.* Try to make maximum use of cache and avoid storing adjacent words in memory from different processors. Methods for achieving these objectives include choosing the rightmost dimension of an array for parallelization and specifying large chunk sizes for a parallelized DO loop or PARALLEL LOOP construct.

Parallel programming example: Matrix multiplication

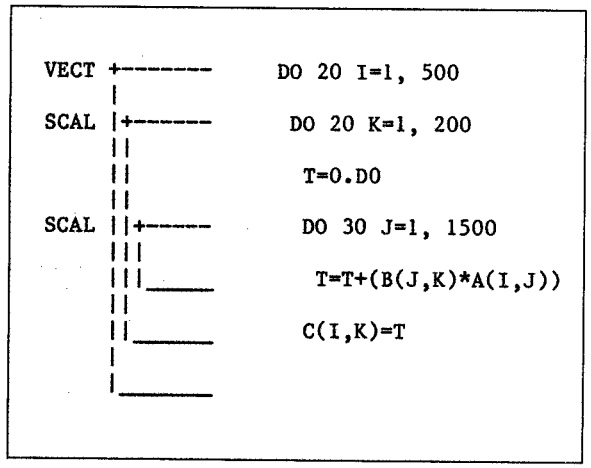
In the following examples, a matrix-multiplication problem is programmed repeatedly in different ways to illustrate the features of Parallel FORTRAN and to explore issues in parallel and vector programming.

The first matrix-multiplication program is shown in Figure 13, in which a serial program has been optimized for use on the IBM 3090 Vector Facility. This is the code that is to be parallelized in the remaining examples. In the example, the temporary variable T allows the compiler to use the vector MULTIPLY-AND-ADD instruction. As a result, the program in Figure 13 computes 128 different values of $T = T + (B(J,K)*A(I,J))$ with a single instruction, and it keeps these resultant values in a register and stores the 128 values of $C(I,K)$ only once.

The objective now is to parallelize the matrix multiplication without degrading the vector performance. Figure 14 shows that the matrix multiplication is vectorized over the I loop and that vector register reuse is obtained by storing $C(I,K)$ outside the J loop only. This leaves the K loop as the prime candidate for parallelization.

Figure 15 shows how this can be done by using the SCHEDULE and WAIT FOR statements. The matrix multiplication is placed into a subroutine named MLT and is modified so that it computes one Nth, where N is the number of processors of the matrix.

Figure 14 Vector report for matrix multiplication



The arguments of the matrix multiplication tell it how many processors there are and which *N*th it is to compute. The matrix multiplication is scheduled for parallel execution with multiple executions of the SCHEDULE statement.

The matrix multiplication is an example of a static mapping of work to processors that can work well on a dedicated system if each scheduled task is assured of having a real processor immediately available. However, if the program is executing in an environment where there is contention for the real processors, the parallel performance actually achieved is determined by the task that receives the lowest level of service.

Dynamic balancing of work to processors is likely to be preferable when systems cannot be dedicated. Parallel FORTRAN provides several methods for dynamic load balancing, as illustrated in the next three examples. All of these examples begin 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. The *I* and *J* loops, meanwhile, maintain their relationship to each other for generating efficient vector code.

Figure 16 shows the way in which dynamic load balancing can be done with the DISPATCH statement. For this technique, subroutine MLT is modified so that, rather than doing one *N*th of the matrix multiplication, it does one iteration of the new outermost *K* loop.

Figure 17 shows dynamic load balancing being done by using the PARALLEL LOOP statement. Note that this code is a small modification of the original matrix multiplication shown in Figure 13. Finally, Figure 18 shows dynamic load balancing being done using automatically generated parallel DO loops, and Figure 19 shows the way in which the compiler parallelized and vectorized the program.

Concluding remarks

Parallel FORTRAN provides a rich spectrum of function that supports a wide range of parallel application programming styles. It can easily be used to exploit the parallel and vector capability of IBM 3090 systems.

The parallelism in an application may be expressed in ways that are natural to the application. PARALLEL LOOPs and PARALLEL CASES may be used to parallelize the statements within a routine; SCHEDULE and DISPATCH may be used to execute independent subroutines in parallel. 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 Parallel FORTRAN program identifies the pieces of work eligible to run in parallel. When the program is compiled and executed, the library puts the parallel work in a queue and distributes it to Parallel FORTRAN processors. Real processors are allocated dynamically by the operating system. As additional real processors are allocated, additional FORTRAN processors can execute concurrently. Programs that partition work dynamically, employ multiple levels of parallelism, or use other strategies to keep the queue of parallel work full can best take advantage of these additional real processors as they become available during execution.

Parallel FORTRAN applications can run under the MVS/XA and the VM/XA SP operating systems. The degree of parallel execution can be controlled at run time through the number of FORTRAN processors. When parallel execution is requested and multiple real processors are available, Parallel FORTRAN can be a valuable aid in reducing the turnaround time of applications.

Figure 15 Matrix multiplication with SCHEDULE

```
      INTEGER KN(200)
      COMMON /AC/ A(500,1500), B(1500,200), C(500,200)
      REAL*8 A, B, C
       .
       .
       .
      DO 20 K=1, NTASK
       KN(K)=K
20     SCHEDULE ANY TASK ITASK,
     *   SHARING (AC),
     *   CALLING MLT (KN(K),NTASK)
      WAIT FOR ALL TASKS
       .
       .
       .
      END

      SUBROUTINE MLT (KN,KT)
      COMMON /AC/ A(500,1500), B(1500,200), C(500,200)
      REAL*8 A, B, C, T
      KUB=200*KN/KT
      KLB=1+200*(KN-1)/KT
      DO 20 I=1, 500
       DO 20 K=KLB,KUB
        T=0.D0
        DO 30 J=1, 1500
30       T=T+(B(J,K)*A(I,J))
20     C(I,K)=T
      RETURN
      END
```

Figure 16 Matrix multiplication with DISPATCH

```
      INTEGER KN(200)
      COMMON /AC/ A(500,1500), B(1500,200), C(500,200)
      REAL*8 A, B, C
       .
       .
       .
      DO 20 K=1, 200
       KN(K)=K
20     DISPATCH ANY TASK ITASK,
     *   SHARING (AC),
     *   CALLING MLT (KN(K))
      WAIT FOR ALL TASKS
       .
       .
       .
      END

      SUBROUTINE MLT(K)
      COMMON /AC/ A(500,1500), B(1500,200), C(500,200)
      REAL*8 A, B, C, T
      DO 20 I=1, 500
       T=0.D0
       DO 30 J=1, 1500
30     T=T+(B(J,K)*A(I,J))
20    C(I,K)=T
      RETURN
      END
```

Figure 17 Matrix multiplication with PARALLEL LOOP

```
      COMMON /AC/ A(500,1500), B(1500,200), C(500,200)
      REAL*8 A, B, C, T
       .
       .
       .
      PARALLEL LOOP 20 K=1, 200
       PRIVATE(T)
       DO 20 I=1, 500
        T=0.D0
        DO 30 J=1, 1500
30       T=T+(B(J,K)*A(I,J))
20      C(I,K)=T
       .
       .
       .
```


Acknowledgments

We wish to thank the Cornell National Supercomputer Facility (CNSF) staff, the Parallel FORTRAN prototype users, and the IBM on-site team at Cornell University for their contributions to Parallel FORTRAN. In addition to providing us with requirements for a Parallel FORTRAN compiler, the CNSF provided an environment where scientists could explore parallelism in their applications. This allowed us to receive valuable feedback on our parallel compiler. We also discussed parallel functions and received valuable input and support from the following IBM organizations: the Numerically Intensive Computing (NIC) Center in the Palo Alto Scientific Center, the Engineering/Scientific Systems NIC Center in IBM Kingston, NY, the VM Advanced Technology group in IBM Kingston, NY, the Scientific/Engineering Computations group in Kingston, NY, the Research Parallel Processing Prototype (RP3) project at the T. J. Watson Research Center, and the Parallel Translator (PTRAN) project at the T. J. Watson Research Center.

Cited references

1. R. W. Hockney and C. R. Jesshope, *Parallel Computers*, Adam Hilger Ltd., Bristol, Great Britain (1981).
2. Eric J. Lerner, "Parallel processing gets down to business," *High Technology* **5**, No. 7, 20–28 (July 1985).
3. S. G. Tucker, "The IBM 3090 system: An overview," *IBM Systems Journal* **25**, No. 1, 4–20 (1986).
4. *IBM Parallel FORTRAN Language and Library Reference*, SC23-0431-0, IBM Corporation; available through IBM branch offices.
5. J. Backus, "Programming in America in the 1950s—some personal impressions," *A History of Computing in the Twentieth Century*, N. Metropolis, J. Howlett, and Gian-Carlo Rota, Editors, Academic Press, Inc., New York (1980), pp. 125–135.

Figure 18 Matrix multiplication with parallel DO loop

```
      COMMON /AC/ A(500,1500), B(1500,200), C(500,200)
      REAL*8 A, B, C, T
       .
       .
       .
      DO 20 K=1, 200
       DO 20 I=1, 500
        T=0.DO
        DO 30 J=1, 1500
30      T=T+(B(J,K)*A(I,J))
20     C(I,K)=T
      .
      .
      .
```

Figure 19 Parallel report for matrix multiplication with parallel DO loop

```
PARA +--------    DO 20 K=1, 200
     |
VECT |+-------     DO 20 I=1, 500
     ||
     ||              T=0.DO
     ||
SCAL ||+------      DO 30 J=1, 1500
     |||
     |||_____        T=T+(B(J,K)*A(I,J))
     ||
     ||_______      C(I,K)=T
     |
     |________
```


6. *IBM VS FORTRAN Version 2, Language and Library Reference*, SC26-4221-1, IBM Corporation; available through IBM branch offices.
7. R. G. Scarborough and H. G. Kolsky, "A vectorizing FORTRAN compiler," *IBM Journal of Research and Development* **30,** No. 2, 163–171 (1986).
8. D. H. Gibson, D. W. Rain, and H. F. Walsh, "Engineering and scientific processing on the IBM 3090," *IBM Systems Journal* **25,** No. 1, 36–50 (1986).
9. *Forefronts* (whole issue) **3,** No. 2, Cornell Theory Center, Cornell University, Ithaca, NY (1987).
10. C. G. Hecht, "Parallel processing on the IBM 3090: Measurements of PFP," *Forefronts* **3,** No. 9, 2–4, Cornell Theory Center, Cornell University, Ithaca, NY (1988).
11. F. Darema, D. George, A. Norton, and G. Pfister, *A Single-Program-Multiple-Data Computational Model for EPEX-FORTRAN*, Research Report RC-11552, IBM T. J. Watson Research Center, Yorktown Heights, NY (1985).
12. E. Clementi, J. Detrich, S. Chin, G. Corongiu, D. Folsom, D. Logan, R. Caltabiano, A. Carnevali, J. Helin, M. Russo, A. Gnudi, and P. Palamidese, "Large-scale computations on

a scalar, vector and parallel 'supercomputer,'" *Parallel Computing* **5,** Nos. 1 and 2, 13–44 (July 1987).
13. P. Carnevali, P. Sguazzero, and V. Zecca, "Microtasking on IBM multiprocessors," *IBM Journal of Research and Development* **30,** No. 6, 574–582 (1986).
14. B. Liu and N. Strother, "Programming in VS FORTRAN on the IBM 3090 for maximum vector performance," *Computer* **21,** No. 6, 65–76 (1988).

Leslie J. Toomey *IBM Data Systems Division, Neighborhood Road, Kingston, New York 12401.* Ms. Toomey is Manager, Compiler Technology, in the Engineering/Scientific Systems Development and Technology function in IBM Kingston, NY. She joined IBM East Fishkill, NY, in 1978 as an application programmer. The assignment involved numeric computing with FORTRAN for graphics postprocessing. In 1983 Ms. Toomey joined IBM Kingston to work on engineering/scientific compiler development. She led the design and implementation of the Parallel FORTRAN Interface support on VM/XA, and received an Outstanding Innovation Award for her work on the Parallel FORTRAN project. Ms. Toomey 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.

Emily C. Plachy *IBM Data Systems Division, Neighborhood Road, Kingston, New York 12401.* Dr. Plachy is Manager, Software Technology, in the Engineering/Scientific Systems Development and Technology function. After working for Exxon Production Research Company in Houston, TX, as a seismic applications programmer, she joined IBM in 1982 to work on engineering/scientific compiler development. Dr. Plachy provided the overall project management for the Parallel FORTRAN Prototype and managed the development of the Parallel FORTRAN Interface for VM/XA. She received a B.S. degree in applied mathematics and computer science from Washington University, St. Louis, MO, in 1970, an M.S. degree in computer science from the University of Waterloo, Ontario, in 1971, and a D.Sc. in computer science from Washington University in 1980.

Randolph G. Scarborough *IBM Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304.* Mr. Scarborough is Manager of FORTRAN Technology at the Palo Alto Scientific Center. His primary focus is that of extending FORTRAN for new machine architectures. He joined IBM in 1969 as a systems engineer in Trenton, NJ, 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 been working on Parallel FORTRAN. Mr. Scarborough had overall project responsibility for the Parallel FORTRAN language and library extensions. He received a B.A. from Princeton University in 1968, and has received many IBM awards, including four Outstanding Innovation Awards (one for Parallel FORTRAN) and two Corporate Awards.

Richard J. Sahulka *IBM Data Systems Division, Neighborhood Road, Kingston, New York 12401.* Mr. Sahulka joined IBM in 1957. He is currently working on the Parallel FORTRAN project in the Engineering/Scientific Systems Development and Technology function in Kingston, NY. He led the team that developed the VS FORTRAN Multitasking Facility, receiving 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, Providence, RI, in 1951.

Jin F. Shaw *IBM General Products Division, Santa Teresa Laboratory, P.O. Box 50020, San Jose, California 95150.* Mr. Shaw is a member of the VS FORTRAN compiler group. He received an M.S. degree in computer science from the State University of New York at Stony Brook, and joined IBM Poughkeepsie in 1981 to work on vectorization algorithms for compiler development. In 1985, he moved to the Santa Teresa Laboratory and joined the VS FORTRAN compiler group. Since 1986, he has been working on automatic parallelization for the Parallel FORTRAN project. Mr. Shaw received an Outstanding Innovation Award for his work on the Parallel FORTRAN project.

Alfred W. Shannon *IBM General Products Division, Santa Teresa Laboratory, P.O. Box 50020, San Jose, California 95150.* Mr. Shannon received his M.S. degree in computer science from the University of California at Davis in 1976. He worked for the Lawrence Livermore National Laboratory for eight years in application programming and in the development of the vectorizing LRLTRAN compiler for the Cray-1. Mr. Shannon joined IBM in 1982, working in compiler performance and in environment development. He was involved with the implementation of the Parallel FORTRAN library, and is currently working on environment development in the Santa Teresa Laboratory. Mr. Shannon received an Outstanding Innovation Award for his work in Parallel FORTRAN. He is a member of ACM and SIGPLAN.

Reprint Order No. G321-5336.