Overview of the Force Scientific Parallel Language

GITA ALAGHBAND¹ AND HARRY F. JORDAN²
¹University of Colorado at Denver, ²University of Colorado at Boulder

ABSTRACT
The Force parallel programming language designed for large-scale shared-memory multiprocessors is presented. The language provides a number of parallel constructs as extensions to the ordinary Fortran language and is implemented as a two-level macro preprocessor to support portability across shared memory multiprocessors. The global parallelism model on which the Force is based provides a powerful parallel language. The parallel constructs, generic synchronization, and freedom from process management supported by the Force has resulted in structured parallel programs that are ported to the many multiprocessors on which the Force is implemented. Two new parallel constructs for looping and functional decomposition are discussed. Several programming examples to illustrate some parallel programming approaches using the Force are also presented. © 1994 John Wiley & Sons, Inc.

1 INTRODUCTION
Programming languages have taken different approaches over the years. Parallel language philosophies range from the implicit parallelism approach as in data flow and functional programming languages [1] where the programmer is not concerned with the underlying parallelism to the explicit parallelism and control flow languages where the user is in control of what must execute in parallel. In the first approach the idea of instruction stream is eliminated, thus a completely automatic solution to the problem of operation scheduling is assumed. Furthermore, the programmer’s world is entirely separated from issues of load balancing, variable access conflict, synchronization overhead, and other performance-related problems.

The Force parallel programming language is a control flow language that gives the programmer some control and responsibility for the above-mentioned performance issues. The Force is a parallel extension of the Fortran language designed for large-scale shared memory multiprocessors and is implemented with a macro processor. The language efficiently implements a number of parallel constructs while hiding machine dependencies from the user. The parallel constructs provide means for writing parallel scientific programs in such a way that the programmer need not be concerned about the correct implementation of parallel loops, synchronizations, process management, and the number of processes executing the program. Being an extension of Fortran, the user programs in the familiar Fortran language with a few efficiently implemented parallel extensions. The Force language has been well received in the scientific programmer com-
munity and many large application programs written in the Force are in use today [2–7]. The Force language has been easily ported to several shared memory multiprocessors including the HEP, Flex/32, Encore Multimax, Sequent Balance and Symmetry, Alliant Fx, Convex, IBM 3090, and Cray multiprocessors [8]. Recently, the Force has been ported to the KSR multiprocessor. The global address space in the KSR, which is based on the distributed shared-memory approach, is very different from all of the above-mentioned architectures. However, porting the Force language to this new architecture has been straightforward. Hiding machine dependencies from the user leaves the programmer with the task of programming in one language. Programs are then ported to any machine that Force has been implemented on with no change. Some tuning of the code for architectural-specific performance may be required.

The unique feature of the Force is the idea of global parallelism [9] later called single program multiple data (SPMD) by IBM [4]. All processes execute a single program completely and in parallel. Programs are written for an arbitrary number of identical processes, but this number is fixed at run time. As all processes are identical, work is not assigned to specific processes. Parallel constructs distribute the work over the entire force of processes, and therefore the programmer is relieved from details of process management. Variables are either uniformly shared by all processes or are private to each process. The Force language provides a number of synchronization constructs. These constructs are not process specific and are hence called generic. The Force language has been specifically designed for shared memory multiprocessors. Several other programming languages that have been developed as extensions to sequential languages for shared memory multiprocessors are also available. Extensions to Fortran have been done in EPEX/Fortran [10], Cedar Fortran [11], and IBM Fortran [12]. Blaze [13] based on Pascal, Astor [14] based on Modula, and Concurrent C [15] and PCP [16] are extensions to the C language. The process scheduling problem has been addressed by the Uniform System Library on the BBN Butterfly [17] for Parafame [18], and in Schedule [19]. A portable set of subroutines for parallel programming is provided with the VMMP [20] and the Large Grain Data flow [21] libraries. The EPEX/Fortran and PCP language constructs have many similarities with the Force. The PCF parallel Fortran extensions [22], which are being studied by the ANSI X3H5 Standardization Committee, include process creation and termination during program execution. The Argonne Macro Library [23, 24] uses nested macro definitions to hide machine dependencies. However, the Argonne macros need the parallel environment variables as arguments in every operation, whereas these variables are hidden in the Force. This paper intends to give an overview of the Force language including a report on two new constructs for looping and functional parallelism. We do not attempt to make a comparative study of existing parallel languages. Good comparative studies have been presented elsewhere [25–27].

In the next section we will expand on the key Force ideas. In Section 3 we will present the Force constructs and their formats. Several programming examples to demonstrate the use of key Force constructs will be given in Section 4. The Force is implemented as a two-level macro processor and has been ported to several shared memory multiprocessors. Section 5 will cover more details on the implementation and portability of the Force. The Force language supports both fine and coarse grain parallelism. It also supports both data and functional parallelism. In Section 6 we will present a detailed description of a new construct "Resolve" for functional decomposition in the Force and explain its major differences with the notion of teams in PCP. We will conclude the paper with a summary in Section 7.

2 KEY CONCEPTS OF THE FORCE

The Force parallel programming language is based on a global parallelism model in which programs are written under the assumption that an arbitrary number of processes, NP, will execute the entire program from the beginning to the end in parallel. A single program is executed by multiple processes each of which has a separate program counter. The user is not concerned with the creation, termination, or management of these processes. The fork/join mechanism or some variation of it to manage parallel processes is used in most other parallel languages. In this model, a single instruction stream must fork into multiple streams at the entry to a parallel section and then join into a single stream at the exit from the parallel section [28]. The overhead associated with managing processes and execution environment every time the programmer invokes fork and join
for parallel execution results in performance penalties. Furthermore, the encapsulation of parallel code sections within fork/join boundaries forces all code above the specified level to be sequential whereas experience shows that some of this code may be parallelizable. Efficiency is a sensitive function of the amount of sequential code in a parallel program. The efficiency reduction in the fork/join model will be more significant in systems supporting many active processes. This argument suggests that one must attempt to encapsulate as much code as possible in a parallel section to maximize the efficiency of a parallel program. In other words, the encapsulation overheads make larger grain parallelism more efficient. In a multiple instruction multiple data (MIMD) environment, performance is maximized by putting the parallelism at the highest possible level of the program structure. For these reasons, in the Force, we have adapted the global parallelism model where parallel processes created at the beginning of the program cooperate to execute the entire program in parallel and are joined only at the end. These processes are all identical and are not explicitly managed by the user.

Variables are either private to a process or are uniformly shared by all processes. This is the property of parallel execution and is independent of and orthogonal to the variable types supported by the underlying Fortran language. Thus, a private variable implies that there is a copy of the variable of a specific type for each process. Similarly, a shared variable could take on any allowable Fortran variable type, but only a single copy of it exists and is shared by all processes of the Force. Care must be taken in parallel update of shared variables. Synchronization operations are provided in the Force to ensure correct operations and results when accessing shared variables in parallel.

Programs written under the global parallelism model may be composed of any of the four categories of strictly sequential sections, replicated sections, multiple sequential sections, and multiple parallel sections of code, as illustrated in Figure 1. For strictly sequential code, that is, one at a time
 execution, the Force language supports both data-oriented synchronization through producer/consumer synchronization on specific variables and control-oriented synchronization through barriers and critical sections. Critical sections may be named or unnamed and they provide one process at a time access to the protected section of the code. Barriers introduce delays at specific points in the code. They can be used to ensure that all processes have reached a certain point in the code before any of them can proceed. The barrier can also have an associated section of sequential code. The semantics are that all processes pause when they reach the barrier. After all have arrived, one and only one process will execute the sequential code associated with the barrier, at its completion all processes exit the barrier. This is the only mechanism in the Force to indicate that a section of code must be executed sequentially. All other code segments are executed by the Force of processes in parallel. All Force synchronizations are “generic”; which means that processes are not explicitly named in the synchronization operations. Generic synchronization relieves the programmer from managing processes and simplifies the parallel programming task.

The multiple sequential sections of code are concurrently executable sections, each of which is executed by a single process, as presented in Figure 1b. Parallel loops or DoAlls are examples of this kind [29]. Several parallel loops for distribution of work to parallel processes are supported by the Force. Static work distribution is mostly desired when the amount of work to be carried out by each process is identical. Static loop scheduling incurs virtually no overhead, requires no synchronization in its implementation, and hence is the most efficient one to use when every process is assigned equal work. Dynamic loop scheduling on the other hand is most appropriate when processes are assigned different amounts of work. In this case the amount of work to be done at each iteration varies and thus it is advantageous to have processes to obtain a new loop index when they complete their previous loop iteration. The self-scheduling requires synchronized access and update of a loop index by processes and is in general more expensive than the first prescheduling alternative. However, it provides better load balancing and in many applications, as we will see later, proves superior to the first approach.

A third alternative and the most general concept for concurrent code segments provides a means of work distribution in cases where the degree of concurrency is not known at compile time. Rather, the program dynamically requests a new instance of the code segment to be executed. All applications involving tree expansion of work units such as the adaptive quadrature example presented in Section 4 would benefit from this approach.

In functional decomposition, concurrently executable code sections have different code bodies. Two mechanisms for functional decomposition of programs are available in the Force. The first and the simpler of the two assumes that each code segment must be executed by only one process, but segments are executed simultaneously. Here again processes may be prescheduled at compile time or self-scheduled at run time. The most general concept for functional decomposition in the Force, however, resolves the force of processes into several components executing different parallel code sections concurrently as seen in Figure 1c. These sections may in turn consist of any of the types of codes presented in Figure 1 and may or may not be mutually independent.

Parallel execution of statements is implicit in the Force. Therefore, statements modifying data private to each process are executed in parallel and constitute replicated sequential sections of code presented in Figure 1d. Programs are written for arbitrary number of processes and explicit process management is avoided by creating and terminating them at the top of the program hierarchy. The information that is needed by the system to provide process management for the user, maintain independence from the number of processes, and to implement the parallel and synchronization constructs is referred to as the Force parallel environment. This information is produced by the macro processor partly at compile time and partly at run time and, with the exception of two variables, number of processes NP and process identifier ME, the rest of the parallel environment information is not accessible to the user.

Figure 2 illustrates a simple Force program to find the maximum element of the array X of 1,000 elements. The program is written for a number NP of processes. NP is a shared variable and ME is a private process identification variable. These variables are made available to the users for testing purposes only. The processes will first compute a private maximum, PMAX, in a parallel loop, Pre-sched Do. Next, in a critical section of code they find a shared global maximum, GMAX. Processes are joined only at the end of the program. Figure 2 also serves to show several of the code segments
Establish the number of processes requested by the executes command, and set up the parallel environment.

Force PROC of NP ident ME

Declare private and shared variables.

Shared real X(l:1000), GMAX
Private real PMAX
Private integer I
End Declarations

Select one process to read and initialize the global maximum. Delay the rest until this is complete.

Barrier read(*, *)(X(!), I=1,1000)
GMAX=0.0
End barrier

Each process sets its private maximum.

PMAX=0.0
End parallel

Each process finds maximum of a disjoint subset of X.

Presched Do 101 = i,1000
If ( X(i) .gt. PMAX) PMAX = X(i)
End presched do

Each process uninterruptibly updates the global maximum.

Critical MAX if (PMAX .gt. GMAX) GMAX = PMAX
End critical

Wait until all processes arrive, then print maximum.

Barrier write(*,*) GMAX
End barrier

Wait for completion of all processes and return to the Operating system.

Join

End

FIGURE 2 Sample Force program to find the maximum array element.

The Force language provides the multiprocessing features of parallel programs and does not hinder the exploitation of vector parallelism on multiprocessors with vector capability. On machines such as the Cray's and the Alliant Fx, the optimizing compiler may be called to do the vectorization as an additional layer of parallelism to the Force programs.

3 THE FORCE LANGUAGE

The Force is a language extension to Fortran and is implemented with a macro processor. It is a scientific programming language specially designed for shared memory multiprocessors. The compilation of the Force programs in UNIX environment is done in four steps:

1. The SED pattern matcher translates the Force statements into parameterized function macros.
2. The M4 macroprocessor replaces the function macros with Fortran code and the specific parallel programming extensions of the target Fortran language. The underlying Fortran language must, of course, include mechanisms for process creation, memory sharing among processes, and synchronized access to shared variables.
3. The Fortran compiler translates the expanded code and links it with the Force driver and multiprocessing libraries.
4. The Force driver is responsible for creating processes and setting up the Force parallel environment and shared memory.

The Force macros can be divided into several categories: program structure, variable declaration, work distribution, and synchronization macros.

3.1 Program Structure Macros

The Force and Forcesub macros declare the start of a parallel main program and subprogram, respectively, as seen in Figure 3. These macros set up the parallel environment variables. As discussed before, only two of these variables, the shared number of processes NP and the private process id. ME, are made available to the user. The Externf statements are needed when the separate compilation of the Force modules is desired. The End Declarations macro terminates the
declaration section of a Force module and marks the beginning of the executable code. This mark is used to insert the declarations and any start up code that may be needed by the macros and memory sharing mechanism. The Join statement terminates execution of the parallel main program. A Force subroutine is terminated by a Fortran return statement provided that all processes execute it. To make a parallel subroutine call, all processes must execute the Forcecall statement corresponding to the desired subroutine. No entry or exit synchronization for subroutine calls is enforced and barriers may be used if such synchronizations are needed. A Force subroutine may be called by only one process, but the programmer must ensure that parallel constructs such as Barriers and parallel Doalls that depend on execution by all processes are not used in the subroutine.

### 3.2 Variable Declarations Macros

The Fortran local/common variable classification specifies textual scope of the variables’ program modules. The shared/private classification of the Force specifies scope of variables among processes and is presented in Figure 4. Both Fortran local and common variables may be either private or shared across processes. Standard Fortran declarations and all implicit declarations are assumed to be private.

Data synchronization is provided by primitive Produce/Consume operations on asynchronous variables. These operations will be described shortly. The asynchronous variables have a “full/empty” state associated with them and are shared among processes.

### 3.3 Work Distribution Macros

Several Doall constructs are provided in the Force language and are categorized according to the work scheduling mechanism they employ (Fig. 5). In a Presched Do the total range of indices is statically divided among the processes. Therefore,

![FIGURE 3 Force program structure.](image)

![FIGURE 4 Parallel constructs of the Force declarations.](image)

![FIGURE 5 The Force work distribution constructs.](image)
process \( j \) of \( NP \) processes gets the \((k \times NP + j)\)th index value for all \( k \) yielding an index in range. For shared memory multiprocessors with vector capability, this division of work interferes with vectorization. A block-prescheduled loop is implemented via \texttt{VPresched Do} for the Alliant Fx and the Cray supercomputers, which divides the total range of indices by the number of processes. Each process is then responsible for handling its block. Vectorization can now be applied to each of the blocks. When processes must move information in between regions, they must know about their region boundaries. The boundary values are made available to the programmer through optional private variables \( b_1 \) and \( b_2 \).

In the \texttt{Selfsched Do} each process obtains the next value of the index by incrementing a shared variable at run time. This loop can adapt to a varying work load, but the implementation requires synchronized access to the shared index.

Doubly nested loops, when the loop body instances are completely independent, can be parallelized over both indices using prescheduled, \texttt{Pre2 Do}, and self-scheduled, \texttt{Self2 Do}, macros.

The most general and complex form of \texttt{Doall} is the \texttt{Askfor Do} loop and is used in cases where the number of units of parallel work may increase over time. The situation mostly arises in recursive algorithms. Initially, there are \( i \) units of work to be executed by parallel processes. A typical loop body will contain a section of code in which processes will obtain the next index and update it atomically. The loop body may also contain a \texttt{More Work} statement that will increase the number of work units of the loop by \( \langle j \rangle \). The loop execution must be completed before any process can exit the loop.

The \texttt{Case} statement distributes processes over different single stream code segments. This is a similar concept to the \texttt{Doalls} where code segments correspond to different instances of the loop body. The different code segments are processed concurrently but each is executed by one process only. Similar to the \texttt{Doalls} there is a prescheduled, \texttt{Pcase}, and a self-scheduled, \texttt{Scase}, statement. The code sections may be conditionally, \texttt{Csect}, or unconditionally, \texttt{Usect}, executed.

The last construct is \texttt{Resolve}, which resolves the force into components executing parallel code sections. Each component has an associated user specified \texttt{strength} variable as the fraction of the force to be devoted to this component. The \texttt{Unify} macro reunites the components into a single force. This is the newest and the most complex \texttt{Force} construct and will be described in more detail in Section 6.

### 3.4 Synchronization Macros

Synchronization operations are essential in the design and implementation of explicitly parallel programs. The \texttt{Force} provides both data and control-oriented synchronizations. Data-oriented synchronization is used to synchronize updating of shared variables by processes. The producer/consumer synchronization is implemented through \texttt{Produce/Consume/Copy} primitive operations as shown in Figure 6. These atomic operations are applied to the class of \texttt{Async} variables with full/empty status and operate as follows:

1. A \texttt{Consume} waits for the state of the variable to be full, reads the value into a private variable, and sets it to empty.
2. A \texttt{Produce} waits for the state of the variable to empty, sets its value to the expression, and sets it to full.
3. A \texttt{Copy} waits for the asynchronous variable to become full, copies its value into a private variable, but leaves its state as full.
4. A \texttt{Void} operation sets the state of its asynchronous variable argument to empty regardless of its previous state.

The primitive \texttt{Isfull} is a logical function that indicates whether the state of an asynchronous variable is full or empty.

Data-oriented synchronization can be used to provide for a powerful \texttt{MIMD} form of data parallelism which is very different from single instruction multiple data (SIMD) data parallelism. Using the \texttt{Produce/Consume/Copy} synchronization operations, one can implement a data flow approach to parallelism for different data structures such as irregular grid structures that would be extremely difficult in \texttt{SIMD} computers.

**FIGURE 6** The \texttt{Force} synchronization constructs.

- Synchronization:
  - \texttt{Consume <async var> into <variable>}
  - \texttt{Produce <async var> = <expression>}
  - \texttt{Copy <async var> into <variable>}
  - \texttt{Void <async var>}
  - \texttt{Isfull <async var>}
  - \texttt{Critical <lock var>}
  - \texttt{Barrier <code block>}
  - \texttt{End critical}
  - \texttt{End barrier}
Control-oriented synchronization, as was discussed before, uses the two concepts of critical sections for mutual exclusion and barriers. At a Barrier, all processes wait. Upon arrival of all processes one process will execute the code body of the Barrier, which may be empty, sequentially. All other processes wait at the end of the Barrier until the single process completes the sequential code.

In the next section we will present several examples to demonstrate some parallel programming approaches using the Force.

4 PROGRAMMING EXAMPLES

We start with a Gaussian elimination program to solve a system of linear equations $Ax=b$. The main program gausselim performs the matrix generation and forward elimination to triangularize the A matrix. Back substitution is performed by the Force subroutine bsolve. The program presented in Figure 7 starts with a Force header, which sets up the parallel environment for nproc processes. Following the header are the variable declarations. The matrix and the right-hand side vector $b$ are declared as Shared Common so that they can be accessed in the back solve subroutine as well as the main program. The program starts by reading the order of the matrix, $n$, inside a barrier. Input and output are done sequentially within a barrier. The A matrix and vector $b$ are generated in parallel within a prescheduled do loop. Therefore, rows of A and $b$ are produced in parallel. Note that initializations of shared variables are done by one process within a barrier. All processes execute the do loop for forward elimination. For each pivot $a_{kk}$ parallel processes will reduce different rows, $i$, of the matrix in parallel.

The back substitution is performed through the bsolve Force subroutine and is presented in Figure 9. This subroutine is called after a barrier synchronization to ensure that all processes have completed their matrix triangularization task. The

```
C Gaussian elimination program
C------------------------------------------------------------------------
  Force gausselim of nprocs ident me
  Shared common /matrix/real a(500,500),b(500)
  Async common /sol real x(500)
  Shared real pivot
  Private integer i,j,k
End declarations

Read in matrix dimension
Barrier
  read(*,IO) n
End Barrier
C Generate the matrix
C Presched DO 100 i=1,n
  if(i.eq.j) then
    a(i,j) = i*i/2.
  else
    a(i,j) = (i+j)/2.
  b(i) = i
End if
100 continue
End presched do

Initialize shared variables
Barrier
  read(*)(10) n
  format(i4)
End Barrier

C Compute the pivot
Barrier
  pivot = -1.0/a(k,k)
End Barrier

C Perform row reductions
Presched DO400 i=k+1,n
  temp = pivot*a(i,k)
  DO 410 j=k,n
    a(i,j) = a(i,j) + temp*a(k,j)
  410 continue
  b(i) = b(i) + temp*b(k)
400 End presched do

Perform back solve
Forcecall bsolve(n)

Output section
Barrier
  if(msing .eq. 1) then
    write(*,*)' The matrix is singular'
  else
    format(2x,I6.2)
    write(*,*)' The solution vector is as below:'
    DO 600 j=1,n,10
      jo = min(j+9,n)
      write(*,30) (x(k),k=jo,j)
    600 continue
  endif
End Barrier

C Join
if (msing .eq. 1) goto 9000
end
```

FIGURE 7 Gaussian elimination program in the Force language.
A parallel back substitution procedure is illustrated in the back solve procedure. The Force call statement is executed by all processes. The use of asynchronous variables and self-scheduled do loop is illustrated in the back solve procedure.

Parallelization of this procedure may seem difficult at first, because the values of \(x_{i+1}, x_{i+2}, \ldots, x_n\) are needed before we can solve for \(x_i\). However, as Figure 8 shows, once the value of \(x_n\) is produced, it can be used in all other equations by parallel processes, \(p_i\), towards partial solution of their corresponding \(x_i\). Initially all \(x\) entries are set to the empty state, Void. The \(x_n\) is produced by one process first. In the Selfsched loop, all processes can now succeed to make a copy of \(x_n\) into a private temp variable and calculate their corresponding partial sum in the private psnum variable, as seen in Figure 9. After each iteration, \(i\), a new value of \(x_i\) is produced that can be used by other processes attempting a Copy on \(x_i\). As can be seen from Figure 8 each process has a different amount of work to perform due to the upper triangular matrix structure. This is also clear from the range of the sequential nested do loop in the self-scheduled loop. The self-scheduled loop is used so that when a process has completed producing an \(x\) value, it can obtain the next row index, \(i\), for the calculation of the \(x_i\) value. This way the work is better balanced among processes as it is possible for some processes to execute more loop iterations of shorter length and some to execute fewer iterations of longer length. A prescheduled loop would have preassigned the same number of iterations to all processes and would not be the best choice for this application.

The next example demonstrates the use of Askfor Do loop. This loop is primarily used with recursive algorithms. An interesting simple example is the adaptive quadrature integration method, for which two basic operations are needed. First, the integral is approximated on the interval \((A, B)\) [Fig. 10]. Then the error is estimated in the approximation on \((A, B)\). The sequential procedure for this integration can be described with the following steps:

1. Apply the approximation to the interval.
2. Apply the error estimate.
3. (a) If the error is small enough add contribution to the integral.
   (b) If not, split the interval in two and recursively do each half.
4. Return from this recursion level.

![FIGURE 8 Parallel back substitution.](image)

![FIGURE 9 Back substitution force subroutine.](image)

![FIGURE 10 Adaptive quadrature integration.](image)
To parallelize this procedure, Step 3 can be revised as follows:

3. (a) If the error is small enough cooperatively add to the integral and quit.
   (b) If not, split interval into two, create a process to do one half, and do the other half.

Therefore, one process starts the integration and every time the interval is split a new process is created. The unlimited recursive creation of processes will produce a breath first expansion of an exponentially large problem, and is unnecessary. In spite of virtual processes, no parallel system will execute this approach efficiently.

The method to implement the adaptive quadrature integration efficiently is to allow a single process to be responsible for integrating a subinterval. Define two intervals for a process:

1. Interval \((A, B)\) for which the process is responsible for computing an answer.
2. Interval \((Au, Bu)\), which is the currently active subinterval.

A high level description of the algorithm can now be presented.

0. Initialize \((Au, Bu)\) to \((A, B)\).
1. Apply approximation to \((Au, Bu)\).
2. Estimate the error on \((Au, Bu)\).
3. (a) If the result is accurate add it to the total integral, report process free, and quit.
   (b) If not accurate enough, split \((Au, Bu)\) and make the left half the new active interval.
4. Assign a free process, if there is one, to integrate a remaining inactive interval.
5. Go to Step 1.

The Force subroutine ADAPT, which implements the adaptive quadrature integration, is presented next. Following this parallel programming strategy, we now have a fixed number of processes and as they become available we assign them to the

```fortran
REAL *8 FUNCTION funct(x)
real*8 x,dexp
funct = x*dexp(x**2)*2.0
RETURN
END
```

FIGURE 11 The adaptive quadrature integration in the Force.
newly created regions for integration. This is a perfect situation to use the `Ask for` Do construct. Initially, the `Region` array is set to hold the original boundary points of the integration area. A single process picks up the private active boundary values in the `Ask for` Do loop. Notice that this is done inside a critical section of code to ensure correct update of the array index IMC. Next, the integration and a test for accuracy is performed. This section of code is done in parallel in subsequent iterations. During the first iteration, however, there is only one active process due to the value of `Init` variable. If the integration approximation is accurate enough, the process adds its private contribution, `sump`, to the global variable, `SUM`, cooperatively in a critical section. Otherwise, the process splits the interval in half. It then puts the right half in the `Region` array to be assigned to a free process at some future point and adds one unit of work for the `Ask for` construct all within a critical section. The process then repeats the procedure for the new left half interval.

The `Ask for` construct allows an efficient implementation of this otherwise complicated situation and should in general be used in recursive applications.

5 IMPLEMENTATION AND PORTABILITY

Implementation of the Force relies on the constructs provided by a specific multiprocessor manufacturer for process creation, synchronization, and shared memory description. To hide these machine dependencies and to ease portability across shared memory multiprocessors, the Force language is implemented as a two-level macro structure [8]. The lower level consists of a small set of generic machine-dependent macros. The upper level macros are machine independent and are built on top of the lower level macros. They implement all of the Force language constructs such as loops, critical sections, etc. There are only a total of 10 machine-dependent macros for lock synchronization operations, memory allocation operations, and the parallel environment. In porting the language from one machine to another, only these few low level macros need to be implemented for the new machine. A detailed description of the portability of the Force and the two-level macro implementation is presented in [8]. As a second step higher level macros might be rewritten to improve performance. The Force has been relatively easily implemented on the HEP, Flex/32, Encore Multimax, Sequent Balance and Symmetry, Alliant Fx, Convex. IBM 3090, and the Cray XMP and YMP/UNICOS multiprocessors.

Implementation of the Force on systems involving rather different process models has not been difficult. In addition to portability, the two-level macro structure has the advantage that any new constructs can be implemented and added to the language using only the machine-independent macros. An example of this is the `Resolve` construct described in the next section.

As reported by Alaghband et al. [8], performance of the Force language constructs was measured by implementing the Linpack routines `sgefa` and `sgesl` using the Force, sequential Fortran, and microtasking directives on the Sequent Balance and Symmetry and Cray YMP, respectively. For the two versions of these algorithms the cost of the Force language constructs is comparable to the cost of microtasking directives. In fact Force performs better than microtasking directives on the Sequent as it minimizes the start up cost of parallel loops by creating all of the processes before the start of the program execution. The Force efficiently implements the machine-independent high level operations for most architectures and little tuning of the high level macros has been needed. Programs written in the Force can be ported to any machine on which the Force has been implemented. Thus, programs can be developed and tested on smaller, less expensive, and more available machines. A correctly tested program can then be run on supercomputers for performance with only some fine tuning that may be needed for the target architecture.

6 FUNCTIONAL DECOMPOSITION IN THE FORCE

Functional decomposition is supported in the Force through the `Resolve` construct. Functional decomposition is especially desirable in cases where several parallel program segments, usually coarse grain, can be executed in parallel with respect to each other and the amount of parallelism in each segment is not large enough to use the complete force of processes. The `Resolve` statement resolves the Force into several subsets to execute different parallel code segments, called components, in parallel. It differs significantly from the other Force constructs, because it creates multiple parallel environments with independent implementation variables. A new parallel environ-
ment is created for each component of the Resolve. The Resolve components specify the fraction of the force of processes they need in a strength variable. This relative strength can either be specified statically or be computed dynamically before entering the construct. The strength is provided statically if the programmer can estimate the relative fraction of the total processes in advance. Otherwise, if it is dependent on a particular execution it can be computed at run time. A sequential component specifies 0 or seq for its strength. All of the subforces are reunited into a single force by the Unify macro. The Resolve macro may be nested to any level.

Figure 12 is an example of the Resolve. The example calculates vector $Z$ as:

$$Z(Y) = Y \times \int_{start}^{end} 2xe^{x^2} dx$$

where the vector $Y$ is obtained from the solution of the linear system:

$$AY = b.$$ 

The two terms in $Z$ can be computed in parallel and concurrently with respect to each other.

Therefore, they constitute the two components of Resolve with strengths quad and solve, respectively. This example is purposely selected to demonstrate the use of Resolve and not to require presentation of extensive coding. Hence, the code for adaptive quadrature, LU decomposition, and back substitution are directly used in this example. Once the two components are completely executed the Unify statement reunites the two subforces into one. The elements of the vector $Z$ are computed in a Presched Do loop using all of the processes.

The Resolve construct is most useful when there are enough processes that all of the components can acquire the number of processes they need to obtain parallel performance benefits of this partitioning. An interesting issue is the partitioning of the force of processes for assignment to different components of the Resolve. There are two cases to consider. The first is when the number of processes executing the program, NP, is greater than or equal to the number of components of the Resolve, ncomp. The second case is when NP<ncomp. The second case complicates the implementation in several ways. When there are not at least as many processes available as there are components, then some of the compo-

```fortran
Force RESOLVE of nprocs ident me
Shared common /order/integer n,prflg,msing
Shared common /matrix/real a(500,500),b(500)
Async common /sol/real y(500)
Shared common /piv/real pivot
Shared common /res/real z(500)
Shared common /bound/Double Precision EPS,START,ENDS,SUM
Shared common /frac/integer quad.solve
Shared integer max
Private integer i,j,k
End declarations

Read in matrix dimension and output flag
Read in strengths and integration boundaries
Barrier

read(*, I0)n,prflg
format(2i4)
read(*, *)quad.solve
read(*. *)start.ends
read(*. *)eps
if (n .gt. 500) goto 5
Initialize variables.
sum=0.0
count=0
eps=mp/2/(ends-start)
input the matrix a and the vector b.
call matrix
End Barrier
Resolve into 2
```

FIGURE 12 An example of the Resolve construct.
Components are not assigned processes at first, that is, components are not co-scheduled. In this case a complex scheduler is needed to reassign processes to other components when required. If the components of Resolve are independent of each other, then the scheduler will simply assign a process that has completed the execution of its component to another component to be executed. When components have data dependencies, the scheduler has to handle possible artificial deadlock situations that may arise every time a blocking synchronization is executed. Blocking synchronizations in this context are referred to Produce/Consume/Copy and the critical sections. When processes are not co-scheduled, a process that blocks on a Produce, Consume, or a Copy operation must wait for a variable to be consumed or produced by another process. If a second such process is not available, the system deadlocks. As a result, the process must call the scheduler upon execution of a blocking synchronization operation and give up its process. The scheduler must then assign the process to another component waiting to be executed.

The above must also be performed for critical sections. Figure 13 illustrates one possible deadlock situation if the scheduler is not called upon execution of a blocking critical section. Even though critical sections are not by themselves blocking, it is possible for them to have nested Produce/Consume synchronizations. Assume only one process is available and the execution order is component 1, 2, and 3. The process will block on Consume x statement of component 1 and calls the scheduler. The scheduler will assign this process to component 2. The process will block on the Critical max statement in component 2. If the scheduler is not called at this point the system is deadlocked. The implementation of the blocking synchronizations must hence be modified to ensure calls to the scheduler at their entry.

A second complication in the case of NP<ncomp is related to the scope of variables of the components. On swapping a process between components, the scheduler must be able to retain the latest values of the local variables associated with the partially executed component in order to continue its execution at a later point. The solution to this problem is to use co-routines, where the local variables environment is retained at each resume point. The scheduler therefore must implement a co-routine mechanism. In fact to avoid these problems in PCP [10], the teams, which to some degree correspond to the components of Resolve, must be independent of each other and the programmer must ensure the existence of all processes needed by the teams in order for the program to complete execution. These restrictions limit the usage of teams in the language.

In weighing the advantages and disadvantages of including a Resolve construct in the Force, one must consider the following:

The Force at the present is a macro preprocessor and not a compiled language. Implementation of the scheduler as proposed about with co-routine mechanism for swapping processes at the Fortran level is not without difficulty. The main objectives of the Force has been efficiency of implementation and execution. The modifications required in the blocking synchronization constructs to allow for calls to the scheduler and the added overhead argue against the inclusion of a general Resolve construct.

The above problems disappear when one considers the case for NP>=ncomp. If there are enough processes to ensure assignment of at least one process per component, then components can be co-scheduled and there is no need for co-routines or modifications of any of the Force constructs. The above considerations led us to adapt the policy that the programmer must ensure the existence of at least as many processes as there are components of any Resolve constructs in the program. This is rather restrictive when one considers that a program must run with any number of processes, even one. The portability issue may also become a problem if some systems do not allow for creation of the desired number of processes. Furthermore, the only situation that would show performance benefits from using the Resolve is when there are enough processes to execute the components in parallel. For these reasons we have restricted the Resolve construct to the case where NP>=ncomp and a brief description of our implementation for this case will follow.
The partitioning of processes among components is done by allocating one process to each component first, regardless of additional processes in the force, to ensure co-scheduling. The rest of processes are then distributed to the components according to their relative strength. Figure 14 describes the algorithm for process allocation. In Figure 14, req(i) is the number of additional processes (after the first process allocation) requested by component i. This number is not necessarily an integer. The integer value of req(i), nalloc(i), is the actual number of processes allocated. The quantities sum and totalloc represent the running sum of requests and allocations, respectively. Multiple parallel environments are created by holding all fixed parallel environment variables in arrays of fixed size and indexing the variables by a unique component index. The component identity of the current component is held by each process in its own environment pointer (Fig. 15). The index of the parent component is saved in the parent pointer field of the new component’s parallel environment. A parallel environment index counter is set to 1 at the start of the program. The main program is given this index value. Components inside the main program and Force subroutines acquire a unique index from the shared parallel environment index counter at run time. The dynamic allocation of parallel environment indices allows separate compilation of the main Force program and Force subroutines. When processes hit a Unify they exit from the current environment and reenter the parent environment by restoring their old process identity and old environment pointer.

The globally shared or totally private nature of the variables in a Force program is preserved by the Resolve construct. Hence, all processes have access to the shared portion of all parallel environments. But, at any time processes access only their current parallel environment.

Resolve is a new construct and was added to the language after the machine-dependent macros were designed, implemented, and in use for a while. Its implementation did not require any new machine-dependent macros, although simple changes to the parallel environment were necessary, as was discussed. The fact that Resolve is implemented only with generic macros implies that it can run on any computer on which the Force machine-dependent macros are implemented.

7 CONCLUSION

The Force is a control flow parallel programming language designed for large-scale shared memory multiprocessors. The key Force ideas covered in this paper are global parallelism, independence of the number of processes executing a parallel program, high performance of tightly coupled programs, suppression of process management, and reliance on generic synchronizations. We have described the language parallel constructs. Several examples illustrate the use of the language in solving problems and implementing parallel algorithms. The implementation of the Force relies on the constructs provided by specific multiprocessors for process creation, synchronization, and shared memory allocation. Hiding these machine dependencies from the programmer through the two-level macro processor implementation has been one of the key achievements of the Force. Using this structure the Force has been successfully implemented and ported to a variety of shared memory systems. The addition of new parallel constructs to the Force can be done without the introduction of new machine-dependent macros. This was shown by the addition of a new and complex Resolve construct. Finally the discussion of the design decisions and the trade-offs between flexibility and performance of Resolve presented in Section 6 shows the importance and motivations behind the decisions made throughout the language design for high efficiency. Many scientific applications such as direct sparse matrix solver, nonlinear finite element computation, and fluid dynamics problems have been implemented.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Significance</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZZ</td>
<td>environment pointer</td>
<td>Private</td>
</tr>
<tr>
<td>ZZINDX</td>
<td>environment index counter</td>
<td>Shared</td>
</tr>
<tr>
<td>ZZNAME(XX)</td>
<td>name of component X</td>
<td>Shared</td>
</tr>
<tr>
<td>ZZPPIR(XX)</td>
<td>parent pointer to env. X</td>
<td>Shared</td>
</tr>
<tr>
<td>ZZMYID(XX)</td>
<td>process identity inside env. X</td>
<td>Private</td>
</tr>
<tr>
<td>ZZNPRS(XX)</td>
<td># of processes inside env. X</td>
<td>Shared</td>
</tr>
</tbody>
</table>
in the Force. As the language constructs are implemented as macros, there is no compiler to aid in debugging those aspects of the program other than those provided by the underlying Fortran compiler. However, by encapsulating the constructs in a simple macro, structured programs can be written without concern about implementing them every time the programmer is to use any of the constructs, which substantially reduces the debugging time of parallel programs written in the Force.

REFERENCES
