The Force†

Harry F. Jordan
University of Colorado
Boulder, Colorado

Introduction

The Force is a parallel programming language and methodology which evolved in the course of implementing numerous algorithms, primarily of a numerical nature, on large scale shared memory multiprocessors. The number of processes supported was from 20 to 200. The Force was realized as a macro preprocessor extending the Fortran language and includes primitive operations supporting both fine and coarse grained parallelism. There is a unifying philosophy which determines the structure of algorithms implemented using the Force. In this programming paradigm, multiple processes execute a single program. The number of processes is arbitrary but fixed at run time and may be one. Work is not assigned to specific processes but distributed over the entire force of processes by parallel constructs. The variables on which work is performed are either uniformly shared among all of the processes or strictly private to a single process. Process synchronization is non–specific or generic. Generic synchronization primitives do not individually identify processes to be synchronized. The parallelism in a Force program is present from the beginning of the program; it is not encapsulated inside of some modules of the program hierarchy. The philosophy could thus be characterized as that of universal or global parallelism which suppresses explicit process management.

The Force is based on the shared memory multiprocessor model of computation. There are multiple instruction streams which could be thought to execute separate programs, but since all code resides in a common memory, a better formulation is that of a single program being executed by multiple processes, each of which has a separate program counter. The shared memory model implies that references to a variable need specify no associated process identity. The user does not explicitly write commands such as *send* and *receive* to communicate values between processes. Transfer of a datum between processes is based on coincidence in the shared name space of the write and read references to a variable. The Force as a language is at a high enough level to suppress machine dependence but is not intended to be a "very high level" language. No attempt is made to suppress the differences between multiprocessors (MIMD) and vector (SIMD) computers, nor is the shared memory versus explicit communication

† This work was supported in part by ONR under Grant N00014-86-k-0204, by AFOSR under Grant AFOSR 85-1089 and by NASA Langley Research Center under Grants NAG-1-640 and NAS1-17070.

distinction hidden. A language similar to the Force might be used as intermediate code output from a very high level language processor which did automatic dependency analysis, parallelization and/or scheduling.

The rationale behind the structure of the Force methodology is grounded in the desire to address large scale multiprocessing. With many processes, it is not feasible to write separate code for each process. Furthermore, independence from the number of processes is the only manageable structure for an algorithm to be executed by many of them. Any required specification of process identifiers significantly complicates the coding for many processes. If synchronization, variable naming, work partitioning or process control required process identification, the programmer might have to keep track of as many separate interactions as there are pairs of processes. As it is, there are no topological aspects of the parallel computer environment to be considered in algorithm design, and no processes are individually managed. It is interesting that this style of parallel programming seems to lead to programs in which a relatively local analysis is sufficient to determine correctness of synchronization. The inclusion of variables private to a process supports the need for temporaries and is particularly useful in coarse grained parallelism, where there are long sequences of serial code. The primitives which assign work can support load balancing by virtue of the fact that they are not process specific. Finally, the design of the Force as an extension to an existing sequential language provides all of the types, operations and control structures of that language and allows the user, or designer, to focus on the parallel aspects of programming.

Efficiency on existing multiprocessors corresponding to the abstract shared memory multiprocessor model of computation was also an important consideration in the design of the Force. It must be emphasized that the concept of efficiency used is based on minimum execution time for a single parallel program with a given number of processors. Throughput of a multiprogrammed multiprocessor system is simply not considered. Although multiprogramming throughput is an important topic in operating systems design, it can probably not be addressed reasonably at a programming language level. The purpose of parallel processing, after all, is to speed up the execution of individual programs. Multiprocessing throughput is probably optimized by running many multiprogrammed sequential machines simultaneously. To address a balance between one program completion speed and system utilization would require a specification of the relative importance of these aspects which is certainly not available at the language level. Efficiency is addressed in the Force both through its philosophy and through the selection, specification and design of individual parallel primitives. It was previously noted that the Force philosophy eliminates encapsulation of parallelism. Encapsulation would imply that all code above the level of encapsulation in the program hierarchy would be sequential, and it is well known that the efficiency penalty paid for a small amount of sequential code is high in a many processor system. As far as possible, the parallel primitives are kept conceptually small, each embodying only one concept. Complex parallelism is obtained by combining these elementary operations.

Implementations of the Force on various machines have certain features in common, many of which contribute to parallel performance. A parallel environment consisting of some shared and some private variables is maintained at run time. It contains implementation structures from which the user is insulated and its structure may help to specify and clarify the Force computational model. A shared variable always contained in the environment is the number of processes for this execution and a private variable always included is the process identifier. The user usually need not be concerned with these variables, but they are made available for algorithm performance determination and debugging. Other implementation details used, for example, in implementing barrier synchronization can be completely hidden from the user. As a result of the Force philosophy, even the implementation can avoid detailed process management; mutual exclusion and process counting are sufficient to implement most parallel primitives. The fact that processes are not created or destroyed in the course of a Force program implies that the implementation need not necessarily save and restore any process states.

The current implementations of the Force are constructed as macro preprocessors producing output consisting of Fortran augmented by whatever parallel extensions the manufacturer has supplied with the multiprocessor. The preprocessing requires a pattern matching macro processor. Since the macro processing has been done in a Unix environment which lacks a combined pattern matching macro processor, the pattern matching has been done with the stream editor (*sed*) and the macro expansion with the *m4* macro processor. As well as allowing a more syntactically consistent language, macro processing has some efficiency advantage over a subroutine library based implementation. Some macros can be expanded in line using information from the parallel environment, thus saving the performance cost of subroutine linkage. More or fewer macros can be handled this way depending on the suitability of the parallel extensions supported by the target multiprocessor. Disadvantages of the macro based implementation include the well known problem that compiler error messages refer to the macro expanded code, of which the user would rather not be aware, and the problem of name collision between user names and both macro keywords and internal names generated by the expansion.

To give the reader a flavor of programs written with the current implementation of the Force, Table 1 shows a complete Force program. This overly simple example generates and multiplies two real matrices. The constructs used will be described in detail later, but the program is presented early as a concrete example to which to tie the conceptual discussion.

The Force philosophy grew during work by the author [1] with the Denelcor HEP [2] computer starting in 1980. Work on the Force as a programming language extension [3] was begun while the author was on sabbatical leave at the Ballistics Research Laboratory of the U. S. Army during the 1983-84 academic year. It was used for fluid dynamics computations at BRL by Nisheeth Patel [4] [5] during the spring of 1984. The system was subsequently implemented on the Flexible Computer Corporation's Flex/32 multiprocessor at the NASA Langley Research Center with support from the Institute for Computer Applications in Science and Engineering (ICASE) [6] [7]. Support from Encore Computer Corporation enabled the completion and polishing of a Force implementation on the Encore Multimax at the University of Colorado, begun in the spring of 1986 by Muhammad Benten [8]. Mr. Benten is in the process of porting this implementation to the Sequent Balance 8000 multiprocessor at the time of writing.

```
          Force MATMP of NP ident ME
          Shared REAL A(200,200), B(200,200), C(200,200)
          Shared INTEGER M, LDIM
          Private INTEGER I, J, IRES
          Private REAL SUM
          End declarations
          Barrier
C            The order of the matrices - M
             READ (5, 50) M
50           FORMAT(I4)
C            Echo the input with number of processes.
             WRITE (6, 75) M, NP
75           FORMAT('  Order',I4,'  matrices using',I4,'  processes.')
          End Barrier
C         Set up the matrices to be multiplied.
          Presched DO  10 I = 1, M
             DO  5 J = 1, M
                A(I, J) = 1./3.
                B(I, J) = 3.
5            CONTINUE
10        End Presched DO
          Barrier
          End Barrier
          Selfsched DO  300 I = 1, M
C            Produce all of row I of the C matrix.
             DO  200 J = 1, M
                SUM = 0.0
                DO  100 K = 1, M
                   SUM = SUM + A(I,K)*B(K,J)
100             CONTINUE
                C(I,J) = SUM
200          CONTINUE
300       End Selfsched DO
          Barrier
C            Write the results.
             DO  20 I = 1, M
                WRITE (8, 500) (C(I, J1), J1=1,M)
20           CONTINUE
500       FORMAT(6E13.6)
          End Barrier
          Join
          END
```

Table 1: A Matrix Multiply Force Program.

Other macro based language extensions were produced for the Denelcor HEP and later ported to other multiprocessors. Lusk and Overbeek at Argonne National Laboratory produced a system [9] based on the monitor concept of C. A. R. Hoare [10], and Babb at the Oregon Graduate Center implemented a method known as Large-Grain Data Flow [11]. The idea of having a single body of code executed by many processes has also been employed by IBM in the RP3 project through the EPEX Fortran preprocessor [12] and by the BBN Corporation through the Uniform System [13] subroutine library for the Butterfly multiprocessor. The philosophy of the latter two systems is probably most closely related to that of the Force in that they tend to suppress process management while retaining the concept of instruction streams. Babb suppresses process management but also, at the top level, exchanges instruction streams for data flow scheduling of operations.

The Force Computational Model

It is important to characterize the model of parallel computation which underlies Force programs. It is this model which is loosely referred to as the Force philosophy. The computational model is based on, and strongly influenced by, the parallel machines used to execute Force programs, but for the purpose of machine independence, one should extract major architectural features of different machines to form more abstract models of parallel computation, each representing a number of machines. The partitioning of the class of all parallel machines into SIMD and MIMD, first proposed by Flynn [14], seems to have distinct impact on algorithm structure. The parallelism in SIMD or "vector" machines appears within single hardware instructions and is thus reflected at the lowest or leaf level of an hierarchically structured algorithm. MIMD parallelism is at the instruction stream (or process) level and could appear at any level of algorithm structure where code execution takes place. In practice, process management overhead favors algorithms in which parallelism appears at a high level.

The Force computational model retains the concept of instruction streams. The data flow [15] or functional programming models discard instruction streams entirely. The cost of eliminating the idea of instruction streams is that it assumes a completely automatic solution to the problem of scheduling operations. Programs written in a functional language contain no operation scheduling information, thus having the programmer specify only functional dependencies. This removes the issues of load balancing, variable access conflict, synchronization overhead and other performance related problems from the programmer’s world entirely. Since these are not closed topics in the research community, there is probably room for a computational model which leaves the programmer some control of, and responsibility for, these performance issues. Furthermore, the program counter is a well understood and successful scheduling mechanism. The step from one to many instruction streams seems sufficient to introduce a number of interesting extensions without demanding a complete solution to scheduling.

Within the class of MIMD machines, those based on shared memory and those with explicit interprocessor communication form distinct subclasses. The shared memory designation is appropriate when hardware mechanisms such as local cache, high connectivity switching networks and memory request pipelining are used to make at least a portion of the memory address space equally accessible to all processors. Explicit interprocess communication is the correct designation when a subset of the total memory is associated with each instruction stream, either because it is only accessible to that stream or because the access time for that stream is very low compared to other streams referencing that part of the address space. Uniform accessibility can be supported at various levels: software library, microcode firmware or memory interface hardware. For the purpose of specifying a computational model, it is appropriate to base the distinction between shared memory and explicit communication on whether the programmer can use one process to directly reference a value produced by another process or whether he must write explicit send and receive operations to communicate values between processes.

The algorithmic implications of classifying computers on the basis of SIMD versus MIMD and shared memory versus explicit communications seem large enough to give meaning to a shared-memory, MIMD model of parallel computation. The discussion of algorithms and program structure to follow assumes this computational model and the performance trade-offs discussed would be significantly impacted by a change in the model. This model is of interest because many-process MIMD machines seem to offer the best speedup potential for unrestricted algorithms and because notable progress has been made in hardware mechanisms to support the shared memory concept. We will thus assume a class of machines consisting of multiprocessors with a substantial portion of their address space devoted to uniformly accessible, shared memory. Separate memory hardware, address mapping or software may also support memory private to each processor.

Implicit information on the shared memory, MIMD model of computation is inherent in the various programming language extensions supported by the manufacturers of different shared memory multiprocessors. In the Fortran for the Denelcor HEP [16], common variables were automatically shared while Fortran local variables became process private, thus confusing a parallelism attribute of variables with a textual name scope issue. New processes could be created, binding the process to a Fortran subroutine which it began executing in parallel with its creator. Return from that subroutine meant self termination of the created process. Processes could not terminate others. The only synchronization mechanism supported by the language was producer/consumer access to variables, directly implemented in hardware for every word of memory. The Flexible Computer Corporation’s Flex/32 [17] multiprocessor has separate private and shared memory hardware, the distinction being made in extended Fortran at compile and link time. Thus either Fortran local or common variables may be of either parallelism class, private or shared. Processes can create and terminate other processes, and the code unit associated with a new process is again the Fortran subroutine. A variety of synchronization and communication primitives is implemented in the operating system, including process control, locks, events and messages. Initial support for parallel Fortran on the Encore Multimax [18] and Sequent Balance 8000 [19] multiprocessors consisted of standard Unix [20] fork and process control, dynamic sharing of memory with child processes, and spinlocks as the basic synchronization mechanism. A user library of parallel primitives built on this basic software support is available to Fortran programmers.

The Force computational model suppresses all explicit control of processes. An unspecified but fixed number of processes is assumed to be available at the start of execution and these processes execute the Force program to completion before terminating. The primary reason for this aspect of the computational model is that the explicit management of processes is not feasible in the case of a very large number of them, which is the case toward which the Force is directed. A secondary benefit is that many differences between the various machines' parallelism support lie in the process control area. No parent-child process relationship is required to support a Force implementation nor any control of one process by another. The Force makes few demands on the process model supported by a specific multiprocessor and has been successfully implemented on the four, rather different, systems mentioned above. The fact that no synchronization or communication primitives include an identifier for another process implies that there is no topological structure associated with process interaction.

In keeping with the lack of any topological structure to the set of processes executing a Force program, only two types of variables are distinguished: strictly private to one process and uniformly shared by all processes. This distinction is made on the parallel usage of variables by multiple processes and is independent of, and orthogonal to, any variable class specification inherited from the underlying sequential language, such as the Fortran local or common distinction. A private variable is a single name for a set of different variables, one per process. The name of a private variable appearing in a statement refers to the specific variable associated with the process executing the statement. Thus an assignment to a private variable which is executed by all processes represents as many separate assignments as there are processes. Private variables are normally used for process specific temporary storage by a programmer and will usually constitute a larger fraction of the variables in a Force program with large parallelism granularity. At the machine level, private variables are cacheable and might be implemented by cells in local memories associated with each processor if the machine had such storage in addition to its shared memory. A shared variable name refers to one storage item in the shared memory of the underlying machine. The variable is equally accessible to all processes of the Force and may be read or written by any of them. Most major problem data in a Force program is of this type. Of course, parallel writes to such variables must be synchronized and shared variables are only cacheable over program fragments in which use of them is read-only.

We have tried above to characterize the Force computational model from the bottom up, that is, to relate it to multiprocessor hardware. A top-down view of the principles of the Force comes from consideration of the structure of parallel algorithms and their representation as multiprocessor programs. The first issue in parallel algorithm structure is how the set of all operations to be performed is divided into sets of operations to be done in parallel. In a vector or SIMD machine there is only one way to decompose an algorithm into parallel parts. That is to decompose the problem data into items which can be processed in parallel by a single control structure which will become the single instruction stream. Decomposition of an algorithm on the basis of data is also possible in the MIMD environment, but in addition, one can partition the operations into classes or functions which can be applied in parallel. A simple division in the taxonomy of MIMD parallel algorithms may thus be made on the basis of data decomposition versus functional decomposition. Functionally parallel algorithms can perhaps be further classified by the way in which the parallel functions share data. For example, the macro-pipelining structure is well known from the use of coroutines [21] even on uniprocessors. A more important issue in the structure of functionally parallel algorithms, however, is the level at which the parallelism appears in an hierarchically structured computation.

The view of a computation as an hierarchically structured set of functions is well established and maps into the subroutine calling hierarchy in most programming languages. The level of the (usually tree structured) functional hierarchy at which parallelism enters the description of an algorithm is important. As noted, the leaf level is the only place where SIMD parallelism can be applied. We can denote this as fine grained parallelism. As MIMD parallelism is applied at higher levels, we speak of algorithms with coarser grained parallelism. The question of parallelism in programs is closely tied to programming language structure, but the current discussion is meant to be language independent. If parallelism is expressed at a high level in a program for an SIMD machine, and a clever compiler transforms it into fine grained parallelism to fit the execution environment, then the resulting program has fine grained parallelism, whether written by a person or produced by transformation from another form.

The usage of synchronization operations by a parallel program is related to its parallelism granularity. If a program has fine grained parallelism, the major issue in expressing the computation is to specify exactly what is to be done in parallel in each of the small grains. Very tight synchronization must be the rule for fine grained parallelism to make sense. In the SIMD case, this synchronization is part of the execution environment and does not appear in the code. In a program with coarse grained parallelism the amount of code devoted to expressing the parallelism may be very small and localized in a high level module. In exchange, the specification of synchronization becomes the major issue and may appear explicitly at any level of structure, all the way down to the leaf.

One possible way to fit MIMD parallelism into the calling hierarchy is to try to encapsulate parallelism below a certain level, or grain size. This has the advantage that the upper levels of the program can be written without knowing anything about parallel computation. Using the Fork/Join mechanism [22] to manage parallel processes, a single instruction stream would fork within some subroutine into multiple streams which would perform a parallel computation and then join into a single stream before returning from the subroutine. The drawbacks in this scheme lie in the area of performance. Efficiency is a sensitive function of the amount of sequential code in an otherwise parallel program on a highly parallel system. The encapsulation idea forces all code above a certain level of structure to be sequential, possibly reducing efficiency significantly if many processes are active. Furthermore, there is overhead associated with managing processes and execution environments in fork and join which is invoked whenever the program passes into or out of the parallel level of structure. Since encapsulation overheads tend to make larger grained parallelism more efficient regardless of the grain size, there is a good reason to locate MIMD parallelism at the highest level of program structure. Experience shows that it is feasible to write applications programs in which parallel processes are established at the beginning of execution and the process environment remains constant over the entire program hierarchy. We use the term "global" parallelism to refer to this method of managing processes. It is the most characteristic feature of the Force computational model.

In the global parallelism model one begins a program under the assumption that it may be executed by an arbitrary number P of processes. There is no explicit code for process management. The processes are managed by entry level, system dependent code which chooses the number of processes on the basis of hardware structure and available knowledge of algorithm needs. The code appearing explicitly to deal with parallelism is all related to process synchronization and distribution of work across processes. Work may be distributed on the basis of a data decomposition of the algorithm or on the basis of a functional decomposition. In algorithms with a strong Euclidean space connection, such as partial differential equation solution, data decomposition is often called domain decomposition. With many processes, some data decomposition of an algorithm is surely necessary since the number of independent functions is limited. Thus data decomposition is more fundamental to the structure of the Force than functional decomposition. Typically, a high level decomposition into a few parallel functions would involve a large amount of data based parallel decomposition within each function. Statements written in a Force program are implicitly executed by all processes in parallel, unless limited by an explicit parallel primitive. An assignment statement, for example, may combine the values of shared and private variables to produce a private or shared result. If the result is private, no assignment conflict is possible. If it is shared, then assignment conflict must be prevented, either by allocation of disjoint sections of a shared data structure to multiple processes or by synchronizing the assignment across processes, say by enclosing it in a critical section or by using producer/consumer synchronization on the variable assigned. Library or user subroutines which are either free of side effects or carefully synchronized can be invoked in parallel, one copy for each process.

The effect of removing process management from the user's area of responsibility by placing it above the top of the program hierarchy is in some cases quite similar to the effect of encapsulation. An example is the parallel loop, or DOALL [23] construct, in which data decomposition of work associated with a loop index is used to distribute parallel operations over processes. Using the encapsulation idea, processes would be forked on entry to the DOALL, perform the body computations for disjoint sets of values of the index and join at the end of the DOALL. With global parallelism, the DOALL construct may be expressed identically. The interpretation is that the processes, already established and with their own private environments, are to execute the body for disjoint sets of values of the index. The processes may either use synchronized access to a shared copy of the loop index to self schedule the work, or the process number, contained in a parallel environment managed implicitly by the Force, can be used to preschedule a fixed set of index values for the process with a given number.

Since parallel execution of statements is implicit in the Force, a construct is needed to satisfy data dependencies by forcing sequential completion of distinct parallel sections of code and to introduce strictly sequential operations, such as input or output on a sequential device. The "barrier" construct is used to supply both of these needs in the Force and has a strong influence on the structure of many Force programs. The barrier is a control oriented synchronization primitive and has an associated section of sequential code, which may be null. The semantics of this construct are that all processes pause when they reach the barrier. After all have arrived, one process executes the associated sequential code, after which all processes exit the barrier. Barrier synchronizations introduce process delay and some sequentialization, even if the body of the barrier is null. The semantics are inherently simpler, however, that those of a Join followed by a sequential code section followed by a Fork, which is another way to accomplish the sequentialization required. The difference is that the barrier does not require process environments to be allocated or released and private variables retain their values across a barrier.

Several advantages arise out of independence from the number of processes. It is not necessary to design algorithms with a detailed dependence on the, potentially very large, number of processes executing them. The choice of the optimal number of processes can be made at run time on the basis of system hardware configuration and load. Since complete independence from the number of processes implies correct execution with only one process, the issues of arithmetic correctness and multi-process synchronization can be separated in the testing of a program. Experience indicates that programs written to be independent of the number of processes executing them are not significantly less efficient than programs tailored to a specific number of processes, at least for the shared memory class of machines. In fact, this programming style evolved, to a large extent, from attempts to produce maximally efficient programs for this class.

The information required by the system to manage processes for the user, maintain independence from the number of them and to implement cooperative synchronization constructs is referred to as the Force's parallel environment. Part of the parallel environment is fixed, being the same for any Force program, and part is variable, depending on the specific Force primitives included in a program. The values of two items in the fixed portion of the environment are made available to the user through the Force program or subroutine header. These are the shared value of the number P of processes and the private value of a unique index between one and P assigned to each process. The process index is primarily useful for debugging, while the number of processes is useful in performance measurements and optimizations. The number of processes could, for example, be used to select one of two different algorithms to be applied to some subcomputation on the basis of the (possibly data dependent) size of the computation and the number of processes available to perform it. Other parts of the fixed portion of the environment are the shared variable required to initially assign process indices and the synchronization mechanism associated with implementing the barrier, which is the most complex and cooperative of the synchronizations provided. The variable part of the parallel environment is automatically generated by constructs appearing in a specific Force program. The items generated are implementation dependent but include such things as shared index variables for self scheduled DOALLs and full/empty locking variables for producer/consumer synchronized variables.

To summarize the characteristics of the Force computational model, it suppresses explicit process management by creating and terminating them at the top of the program hierarchy. Parallel computation is thus the normal mode of execution on entry to a Force program, and any sequential operation must be explicitly invoked using a control primitive. Programs are independent of the number of processes executing them, except of course, for performance, which should increase with the number of processes. Since the Force is devoted to optimizing single program completion time, the number of processes can remain fixed during execution, and operating system throughput issues are not addressed. All processes are identical in capability, and all execute the same program. Variables are either strictly private to one process or uniformly shared among all of them. The combination of process and variable structure implies that there is no topology associated with data access by processes of the Force. There is never any need for one process to identify another one explicitly. Because programs following the Force computational philosophy can be executed with one process, correctness of program execution can be tested independently of effects due to improper synchronization. Finally, the Force allows parallel constructs at any level of the program hierarchy and thus supports both coarse and fine grained parallelism.

Realization of the Concept

The programming language associated with the Force consists of some simple extensions to the Fortran language, which are currently implemented as macros expanded by a language independent preprocessor. The target Fortran system must, of course, include ways of creating multiple processes, sharing memory between processes, and of supporting synchronized access to shared variables. The macros interact through the variables of a parallel environment, which contains some general information such as the number of processes and some machine dependent items. A macro preprocessor has advantages over the use of a subroutine library to provide language extensions in that it allows a more readable syntax for the parallel operations and does not require subroutine linkage overhead in the case of simple, and potentially very efficient, macros. It is at a disadvantage, however, with respect to a compiler for the extended language. A key dividing point in capability is that the compiler builds a symbol table while the macro preprocessor does not. This not only makes it impossible for the preprocessor to check correct variable usage, but also requires the user to explicitly supply variable type information in some parallel constructs which would be available implicitly if the program were compiled.

The macros currently constituting the Force can be divided into several classes, as shown in Fig. 1. The first class deals with parallel program structure. The macros *Force* and *Forcesub* respectively begin parallel main programs and parallel subroutines. They make the parallel environment variables available to the macros within that program module as well as making the number of processes and a unique identifier for the current process available to the user at run time. An *End Declarations* macro marks the beginning of executable code and provides target locations for declarations and start up code which may be generated by the macros. A *Join* macro terminates the parallel main program. It is the last statement executed by all processes of the Force. The Fortran RETURN statement is sufficient to terminate a parallel subroutine, provided that all processes eventually execute it. The *Forcecall* macro is executable rather than

Program Structure:

```
Force <name> of <# of procs> ident <proc id>
        < declaration of variables >
        [ Externf < Force module name > ]
End declarations
        <Force program>
Join

Forcesub <name>([parameters]) of <# of procs> ident <proc id>
        < declarations >
        [ Externf < Force module name > ]
End declarations
        < subroutine body >
RETURN

Forcecall <name>([parameters])
```

Declaration of Variables:

```
Private <FORTRAN type> <variable list>
Private Common /<label>/ <FORTRAN type> <variable list>

Shared  <FORTRAN type> <variable list>
Shared  Common /<label>/ <FORTRAN type> <variable list>

Async <FORTRAN type> <variable list>
Async Common /<label>/  <FORTRAN type> <variable list>
```

Figure 1a: Parallel Constructs of the Force - Structure and Declaration

declarative but must correspond to the semantics of the *Forcesub.* For correct operation of a *Forcesub* all processes must execute a *Forcecall* so that *Barrier*'s, for example, contained in the *Forcesub* will operate correctly. No implicit synchronization is forced on entry to, or exit from, a *Forcesub*. Synchronization constructs can be included explicitly, either inside or outside the subroutine, if it is required.

Macros of the second class deal with variable declaration. The primary new variable characteristic which appears with the Force computational model is the shared versus private distinction. This is a specification of name scope across multiple processes, just as the Fortran local/common distinction specifies name scope across program modules. Both Fortran local and common variables may be either private or shared across processes. The memory sharing mechanism in some machine implementations requires the size of shared variables to be known to the Force preprocessor, so the type information is required in the *Shared* declaration. Standard Fortran declarations and any implicitly declared variables will be taken as *Private* in order to maintain compatibility between sequential Fortran code blocks and Force programs which might

Parallel Execution:

```
        Pcase on <variable>
        [Pcond (<Fortran IF condition>)]
                <code block>
        [Usect]
        [Pcond (<Fortran IF condition>)]
          . . .
        End Pcase

        Presched Do <n> <var> = <i1>,<i2>[,<i3>]
                <loop body>
<n>     End Presched Do

        Selfsched Do <n> <var> = <i1>,<i2>[,<i3>]
                <loop body>
<n>     End Selfsched Do

        Pre2do <n> <var1>=<i1>,<i2>[,<i3>]; <var2>=<j1>,<j2>[,<j3>]
                <doubly indexed loop body>
<n>     End Presched Do

        Self2do <n> <var1>=<i1>,<i2>[,<i3>]; <var2>=<j1>,<j2>[,<j3>]
                <doubly indexed loop body>
<n>     End Selfsched Do
```

Synchronization:

```
        Barrier
                < code block >
        End barrier

        Critical <lock-var>
                < code block >
        End critical

        Void  <async variable>
        Produce  <async variable>  =  <expression>
        Consume <async variable> into <variable>
        Copy <async variable>  into <variable>
        ...  Isfull( <async variable> ) ...
```

Figure 1b: Parallel Constructs of the Force - Execution and Synchronization

incorporate them. Most implementations of the Force require that extra synchronization items be included with variables which are accessed using the data synchronizations, *Produce* and *Consume*, to be described shortly. Thus, a distinct variable type declaration, *Async*, for asynchronous variable, is introduced. Appropriate additional declarations are generated by *Async* to implement the full/empty variable state required by *Produce* and *Consume* operations.

Macros of another class distribute work across processes. The most familiar construct is the DOALL, which is employed when instances of a loop body for different index values are independent and can thus be executed in any order. Two versions are provided. The *Presched Do* divides index values among processes in a fixed manner which depends only on the index range and the number of processes. The *Selfsched Do* allows processes to schedule themselves over index values by obtaining the next available value of a shared index as they become free to do work. For situations in which it is desirable to parallelize over both indices of a doubly nested loop, both prescheduled, *Pre2do,* and self scheduled, *Self2do,* macros are available. Independence of the loop body instances over both indices is, of course, required for correct operation. Note that only one process will execute the body of a DOALL for any one specific value of an index, or index pair. Thus the body of a DOALL should be considered a sequential code block. A similar construct is the parallel case, *Pcase,* which distributes different single stream code blocks over the processes of the Force. Execution conditions can be associated with each block, and any number of conditions may be true simultaneously. No order of evaluation of the conditions is specified, and each is evaluated by an arbitrarily selected process, so conditions depending only on shared variables are most meaningful.

At the heart of the Force methodology are the synchronization macros. They characterize the approach to parallel programming and provide the means for controlling the Force so that coherent and deterministic computation can be performed. Two subclasses of synchronization are control flow oriented synchronizations and data oriented synchronizations. The key control oriented synchronization is the barrier since it provides control of the entire force. Its semantics are that all processes must execute a *Barrier* macro before one arbitrarily chosen process executes the code block between *Barrier* and *End Barrier.* When the code block is complete, the entire force begins execution at the statement following the *End Barrier.* Although all but one process are temporarily suspended by a barrier, no process termination or creation takes place, and private process states are preserved across the barrier. Operations which depend on the past computation, or determine the future progress, of the entire force are typically enclosed in a barrier. Another control synchronization is the critical section, familiar from the operating systems literature. Statements between *Critical* <*variable*> and *End Critical* may only be executed by one process of the Force at a time. This mutual exclusion extends to any other critical section with the same associated variable.

Data oriented synchronization is provided by the elementary producer-consumer mechanism, in which shared variables have a binary state, full or empty, as well as a value. Such variables are called asynchronous to indicate that the access mechanisms associated with them allow them to be updated asynchronously by different processes. Execution by some process of the macro, *Produce* <*async variable*> = <*expression*>, waits for the variable to be empty, sets its value to the expression and makes it full, all in a manner which is atomic with respect to the progress of any other process. The macro, *Consume* <*async variable*> *into* <*variable*>, sets the second variable, which should be private, to the value of the asynchronous variable when the latter becomes full and sets it empty. Variables in the wrong state may cause these macros to block the progress of a process. Auxiliary macros for asynchronous variables are *Void <async variable>,* which sets a variable empty regardless of its previous state, and *Copy,* which waits for the asynchronous variable to be full and copies its value to a private variable, but does not empty it. The primitive, *Isfull(<async variable>),* is a logical function which can be used in conditions to test the state of an asynchronous variable without blocking on either full or empty. The test, however, will not be synchronized with any change in the state.

A weakness in the set of Force macros in Fig. 1, which is the set supported in current implementations, is that it does not smoothly support decomposition of a program into parallel components on the basis of functionality. The *Pcase* macro offers the rudiments of this, but only allows one process to execute each of the parallel functions. In discussing the computational model, we argued that data decomposition is more fundamental when the Force involves many processes, but that some functional decomposition was appropriate at the upper levels of the program hierarchy. What is desired is a macro, *Resolve*, which will resolve the force into components executing different parallel code sections. The section of code for each component would start with *Component <name> strength <number>,* which would name the component and specify the fraction of the force to be devoted to this component. The component strengths would be estimated by the programmer on the basis of any knowledge available about the computational complexity of each component. A macro, *Unify,* would reunite the components into a single force. The implementation of *Resolve* is complicated by the conflicting demands of generality and efficiency. It is quite simple if the number of processes of the Force is larger than the number of components of the *Resolve*; but complete independence from the number of processes implies that the number of components may be larger than the number of processes. Then inter-component synchronization may deadlock unless the components are co-scheduled over the available processes. An implementation which produces process rescheduling at every possible deadlock point, and is still efficient when the number of processes exceeds the number of components, is under development. Incorporation of a *Resolve* macro will make it useful to extend the barrier idea. A barrier should be able to specify whether only the processes in the current component are to be blocked or whether all processes in the parent force are to participate. In the case of recursively nested *Resolve* constructs, the barrier might specify a nesting level relative to the one in which it appears.

The *Resolve* idea promises a mechanism for functional decomposition of programs into parallel components, but there is one more capability of parallel programming environments with explicit process management which is not addressed by the Force. This is the ability to create new processes to handle new parallel tasks as they arise, which translates in the Force environment into the ability to give away work to "available" processes in a dynamic manner during execution. This ability is most called for by tree algorithms and dynamic divide-and-conquer methods. It would be desirable for the Force to have a mechanism for handling such algorithms without making the user responsible for process management or losing the benefits of independence of the number of processes. A mechanism related to resolve might be applied at each tree node but could lead to much process management overhead in cases where the correct thing to do is merely to traverse a subtree with the one remaining process.

A degree of structural consistency must be maintained in putting together Fortran code and parallel primitives from Fig. 1 to make a Force program. A Force program begins with a header which allows the user to declare variables to be private or shared. Fortran assignment and conditional statements are executed by all processes of the Force simultaneously unless limited by a process synchronization construct. Normally, parallel assignments are only made to private variables while both private and shared variables appear in expressions. If only shared variables appear in a conditional, the flow of control will be the same for all processes of the Force. Private variables appearing in a conditional can cause different processes to take different paths through the program. If processes are split among different execution paths, the paths must rejoin again before executing any primitive requiring the entire force. Such primitives include *Barrier, Pcase*, DOALL and *Forcecall*.

The semantics of the parallelism constructs in the Force imply that certain interrelationships among the primitives be observed when they are used together in a program. Several of the constructs restrict execution to a single stream within some code block. *Barrier* and *Pcase* limit execution of enclosed blocks to a single process while critical section code is eventually executed by all processes, but only one at a time. Thus constructs which depend on multiple, simultaneous execution, such as DOALL, *Pcase* or *Barrier* should not appear within such blocks. A critical section within a *Barrier* is meaningless, but critical sections have definite use within two or more code blocks of a *Pcase* construct. Nested critical sections have meaning when the associated locking variables are different. Data oriented synchronization primitives may occur within singly executed code without restriction, other than the natural possibility of deadlock. In fact, initialization of asynchronous variables is usually done within a singly executed block.

Parallel loops do not restrict the execution of their bodies to a single process, but they do limit execution of the body for each index value to one process. Thus constructs which depend on full parallel execution cannot appear within DOALLs. These include *Barrier, Pcase* and other DOALLs. The inconsistency in the parallelism requirements of nested DOALLs is the reason for supplying multiple index DOALLs for parallel execution of loop bodies which are independent over the Cartesian product of two or more index sets. Critical sections, *Produce* and *Consume* are quite useful within DOALLs, and often lead to programs in which the distributed nature of synchronization reduces its effect on program performance.

Subroutine invocation within a Force program can be done either with a *Forcecall* or an ordinary Fortran CALL. Only the *Forcecall* makes the parallel environment available to the subroutine called. Since a Force subroutine invoked by *Forcecall* assumes that all processes of the Force will enter it, a *Forcecall* must not appear within a code body in which parallel execution has been restricted. Thus, Forcecalls are not meaningful within *Barrier, Pcase, Critical* or DOALL constructs. An ordinary CALL implies execution of a subroutine in single stream on behalf of one or more processes.

Since any Fortran based parallel system must support multiple independent execution of subroutines, such as those in the mathematical library, subroutines must have separate private variable states for all processes executing them. An ordinary Fortran subroutine or function call may thus appear within any code section of a Force program. The subroutines or functions so invoked contain no parallel constructs and access by them to any shared variables must be controlled externally if it is desired.

The *Resolve* construct is intended to produce a new parallel execution environment within each of its components, differing from the original only in the number of processes and their identifying indices. Thus all of the parallelism primitives have meaning within a Force component. The implementations of the primitives must, of course, refer to the parallel environment of the component rather than of the original force. The meaning of the barrier, as has been noted, can be extended to refer to higher levels of a nested component structure, but it retains its original meaning with respect to the immediate component with no modification of its semantics. *Barrier, Pcase* and the DOALLs have an action limited to the component in which they appear. Critical sections and data oriented synchronizations can synchronize operations within the current component with operations in any other components which share the corresponding variables.

It can thus be seen that a fairly small number of parallel primitive operations, superimposed on a sequential programming language, and used to produce programs which manage processes as a single entity is the essence of the current implementations of the Force computational model. The level of the parallel extensions is similar to the level of the Fortran language, making it intermediate across the range of parallel languages which have been considered in the literature. Implementation as a macro preprocessor to Fortran provides easy and efficient realizations on different machines, but has drawbacks in the areas of consistency checking, error messages and use of implicit information from the base language. Further work is under way to make the Force a coherent parallel programming language for shared memory multiprocessors, complete with compiler and run time environment.

Applications and Performance Issues

The Force system has been used to produce a parallel Gaussian elimination subroutine[3] identical in interface and operation to the SGEFA routine of LINPACK[23]. As well as being effective in this library subroutine type of application, it has been used to write large parallel fluid dynamics programs, including SOR algorithms for incompressible flow [4,5]. It has also been used to implement a new parallel pivoting algorithm for solving sparse systems of linear equations[24]. The structure of the Gaussian elimination and SOR algorithms is simple enough to serve to illustrate the use of global parallelism and the specific macros in writing applications code. The first example is an LU decomposition routine from the LINPACK package of subroutines for the solution of simultaneous linear equations. It has been tested extensively on a wide range of computer architectures [25]. The algorithm is Gaussian elimination with partial pivoting and leads to a Force program which uses control oriented mechanisms for most synchronizations. The second example is a fluid dynamics calculation relying on a Successive Over Relaxation (SOR) kernel to compute the stream functions in a driven cavity vortex. In contrast to the Gauss elimination, the SOR algorithm relies almost entirely on producer/consumer synchronization.

The SGEFA routine is implemented as a Force subroutine. As in a sequential Gauss elimination, the outer structure is a loop over pivots. This sequential DO loop is executed in parallel by all processes of the Force, each process having a private loop index variable. Within the loop body there are three computational phases, separated by barrier synchronizations. They involve finding the pivot element, swapping the pivot row with the top row of the sub-matrix, and reducing all non-pivot rows. In each phase a process does 1/P of the work, where P is the number of processes. The overall skeleton of the algorithm is shown in Fig. 2. It can be seen that all synchronization is done by the three barriers separating the computation phases along with one critical section in which each process updates the shared maximum pivot candidate on the basis of its own private maximum.

The first and second phases of the outer loop body contain order K DOALLs while the third phase has order K^2 computation, where K is the size of the square submatrix below the diagonal. None of the DOALLs has any explicit variability of execution time over processes, but the order K^2 phase may involve enough computation that asymmetric processor-memory interconnection could cause process speed to vary. Thus the first two DOALLs are implemented as pre-scheduled DO loops and the outermost one of the nested loops in phase three as a self scheduled DO. (Detailed measurements show that the synchronization required by self scheduling reduces performance for a large number of processes but improves it for fewer processes compared to that obtained by prescheduling the row reduction.)

The fluid dynamics application involves a complete user program of about 350 Fortran statements as opposed to a single subroutine and test driver. Since the Force main program is already parallel, overhead work such as array initialization and boundary condition calculation, which one might be tempted to do before forking parallel processes, is done naturally in parallel. The overall program structure is a sequential

```
DO loop over pivots.
  DOALL - search part of pivot column for private maximum.
  Critical section - update global maximum.
  Barrier - record pivot when all have finished.
  DOALL - swap part of pivot row into position.
  Barrier - take reciprocal of pivot.
  DOALL containing sequential loop over elements of a row:
    reduce part of non-pivot rows.
  Barrier - reset global maximum.
End of loop over pivots.
```

Figure 2. Structure of the Parallel Gauss Elimination

loop over simulated time with two sets of linear equations solved by SOR iteration at each time step. One set is for stream function and one for vorticity at the points of a rectangular grid. The method used for the parallel SOR is a simplified version of a more general algorithm [26] which is guaranteed to have the same convergence properties as a rowwise sequential sweep of the grid. The simplified algorithm involves processes sweeping different rows of grid points with interprocess synchronization used to guarantee that updated information (and old information) is used to compute a new value at a grid point exactly as in a single process rowwise sweep of the grid.

Consider only the Force subroutine to perform a single relaxation iteration for the stream function on the grid. It returns to the calling program the maximum change in stream function value over the grid. The overall structure of the parallel algorithm is shown in Fig. 3. One barrier separates the initialization of the boundary elements to full from the relaxation sweep. In addition, the code to zero the maximum change is contained in this barrier. Within the loop over rows, producer/consumer synchronization prevents a process from overtaking the one sweeping the previous row and thus ensures that the same values will be calculated as for a sequential rowwise sweep. Producer/consumer synchronization is also used to update the shared maximum, in contrast to the critical section used for a similar purpose in SGEFA.

Although fairly different in structure, both applications presented result in clear programs with parallelism constructs appearing in natural ways. The importance of DOALL type constructs opens the way to application of the research already done in parallelizing Fortran loops automatically [27]. The barrier makes possible the separation of a computation into sequential phases without invoking the process environment management overhead of Fork/Join. Producer/consumer synchronization and critical sections make it easy to deal with mutual exclusion type restrictions on access to shared variables. Each of the parallel constructs used seems closely related to the type of data dependencies considered by a programmer or an automatic parallelizer in producing parallel programs.

Various features of the Force methodology are related to the performance of a parallel computer system. An overall principle used in selection of primitive operations

```
Entry.
Set the initial boundary row elements full.
Barrier - zero the maximum change.
Self scheduled DO loop over rows of the grid.
  Sweep row sequentially, waiting for corresponding elements of
    previous row to be full before updating and filling a value.
  Update private maximum change.
End of self scheduled DO over rows.
Return.
```

Figure 3: Force Subroutine for a Single SOR Sweep

for inclusion in the Force was that the semantics of each primitive should be simple enough to admit of an efficient implementation across the range of shared memory multiprocessors. The simple process model, consisting of program counter, private variables and unique identifying index, also contributes to low overhead implementation on most shared memory machines. Process priorities and parent-child relationships, for example, can significantly complicate the implementation of a parallel programming system on some multiprocessors which do not directly support such features. The concept of efficient implementation of a parallel primitive through macro expansion and an implicit parallel environment is well illustrated by the *Presched DO* macro. Two elements always contained in the parallel environment are the shared variable containing the number of processes, here referred to as <np>, and the private variable, <me>, which contains the unique index, 1 ≤ <me> ≤ <np>, for each process. In all implementations of the Force, the parallel construct,

Presched DO 10 I = I1, I2, I3

expands into the ordinary Fortran DO statement,

DO 10 I = I1+<me>-1, I2, I3*<np>.

The primitive operations of the Force define a virtual machine, and the generality of this machine yields independence from the details of the underlying hardware. This benefit of machine independence and portability need not, however, suppress all machine performance issues at the level of Force programming. Pratt [28] points out that a virtual machine for parallel execution should make "visible," as programming alternatives, distinctions which may reflect major hardware performance differences. The clearest example of such alternatives within the Force is the existence of both a prescheduled and a self scheduled DOALL.

At the level of the abstract machine, the process interactions implied by pre- and self scheduling are different. Prescheduling, since it allocates index values to processes in a fixed way as soon as the number of processes is determined, will split the workload evenly across processes only if processors run at similar speeds and the amount of computation specified by the DOALL body is independent of index value. On the other hand, no process interaction is required to allocate the index values; each process can determine its own portion of the work independently. In contrast, the self scheduling technique allows processes to load balance at execution time by obtaining further index values whenever they complete the work connected with previous values. This is done at the expense of a short critical section to obtain, increment and store a shared index variable, unless the host machine supports some cooperative synchronization mechanism such as fetch-and-add [29] by means of a combining network. In this case the price paid is the cost of using the combining network.

For a given underlying hardware, these distinctions at the abstract machine level can be translated into performance differences by using a few general characteristics of the hardware system. The most important parameters for the pre- versus self scheduling comparison are the size, in execution time, of a minimal critical section to access and update a shared index and the number of processes competing for this access. When combined with the program dependent parameters of the mean and standard deviation of the DOALL body size over the set of index values, they allow a determination of which type of scheduling will lead to better performance.

Implementations and Portability

As stated in the introduction, the Force was designed for implementation on the class of machines known as shared memory multiprocessors. Before giving examples of specific members of this class on which the Force has been implemented, it is useful to try to characterize a generic shared memory multiprocessor. As seen at the level of a user programming language, the system consists of two distinct layers: the hardware and the software. There are two distinct contributions to the software layer which are difficult, and usually unimportant, for the user to distinguish: the operating system calls and run time language support library functions built onto the system calls.

The hardware of any shared memory multiprocessor must support multiple instruction streams and will thus have multiple program counters. The different instruction streams control multiple execution units which yield parallelism in the processing of data. Multiple memory accesses must be able to progress simultaneously to provide instructions and data to the control and execution units. The feature described by the modifier "shared memory" means that at least a substantial portion of the data memory is accessible to all instruction streams. Program memory might be shared or private to a subset of the instruction streams, and some part of the data memory might be private. All of these memory variations have appeared in the machines described below on which the Force has been implemented.

In addition, the hardware of a shared memory multiprocessor must support synchronization between the instruction streams. Although synchronization is theoretically possible using only atomic read and write on a shared memory, no efficient multiprocessor program can be written without a hardware primitive which at least does a test and conditional change of value indivisibly on a shared memory item. Variations in support provided here lie in whether the indivisible test and update is performed by mutual exclusion or whether the effect of indivisibility is obtained through a combining network; in whether or not any instruction stream control features are included in the synchronization to support low overhead waiting; and in the details of the data test and update: test-and-set, compare-and-swap, fetch-and-add, etc.

The operating/run-time system software primitives available to the user of a generic multiprocessor must allow the user to do process management, sharing of data among instruction streams, and synchronization. Minimal process management support must allow the user to start parallel activity and terminate it when complete. Data is communicated among instruction streams using the shared memory, but the notion of an independent instruction stream implies that much data must be unique to that stream. Thus there must be user level support for the private versus shared data distinction. Software support for synchronization can either be very simple, supplying a single primitive on which the user can build more complex synchronizations, or very complex, supporting a wide range of expensive but powerful operations of the style familiar to the designers of operating systems.

We now describe the hardware of four computers on which the Force has been implemented; the HEP, the Flex/32, the Multimax and the Balance 8000. The HEP computer is a pipelined multiprocessor in which several processing units, called Process Execution Modules (PEMs), may be connected to a shared memory consisting of one or more memory modules as shown in Fig. 4. Even within a single PEM, however, HEP is still a multiprocessor. Only the number of instructions actually executing simultaneously, about 12 per PEM, changes when more PEMs are added to a system. Separate memories store program and data with smaller memories devoted to registers and frequently used constants. Only data memory is shared between PEMs. We will concentrate on the architecture of a single PEM which implements multiprocessing by using the technique of pipelining.

There are several separate, interacting pipelines in a PEM but the overall effect is that, on the average, instructions from about 12 different instruction streams are being operated on simultaneously. Copies of process state, including program counter, are simultaneously available for all processes in the pipeline. A PEM is an MIMD processor in exactly the same sense in which a pipelined vector processor is an SIMD machine. In both, independent data items are processed simultaneously in different stages of the pipeline while in the HEP, independent instructions occupy pipeline stages along with their data.

It is important that data memory instructions occupy a separate, noninterfering pipeline consisting of the Storage Function Unit, pipelined switch and memory module. The relationship between the main execution pipeline and the SFU is shown in Fig. 5. An active process is represented in the hardware by a Process Tag (PT) which points to one of the 128 possible process states. When an SFU instruction (data memory access) is issued, the PT leaves the queues of the main scheduler and enters a second set of identical queues in the SFU. The PT then remains within the SFU-switch-memory pipeline until the memory reference is completed, and does not consume processor execution time.

HEP hardware support for process synchronization is based on producer/consumer synchronization. Each cell in data memory has a full/empty state and synchronization is performed by having an instruction wait in the SFU-memory pipeline for a read cell to be full or a write cell empty before proceeding. The synchronizing conditions are

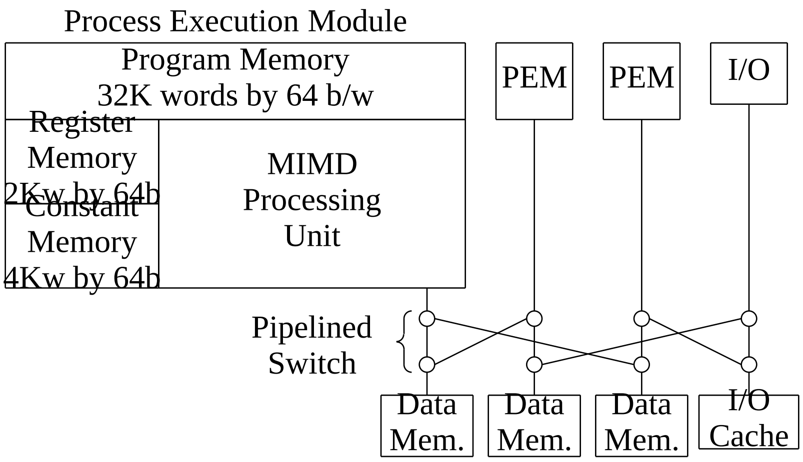


Figure 4: Architecture of the HEP Computer

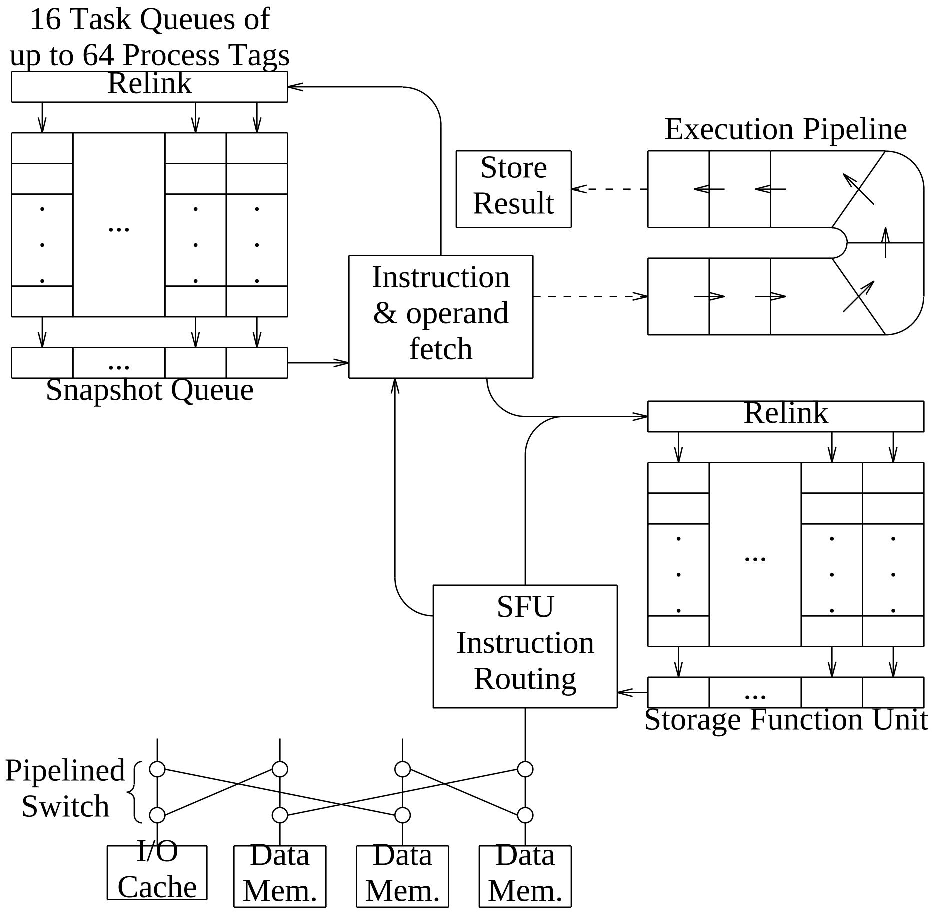
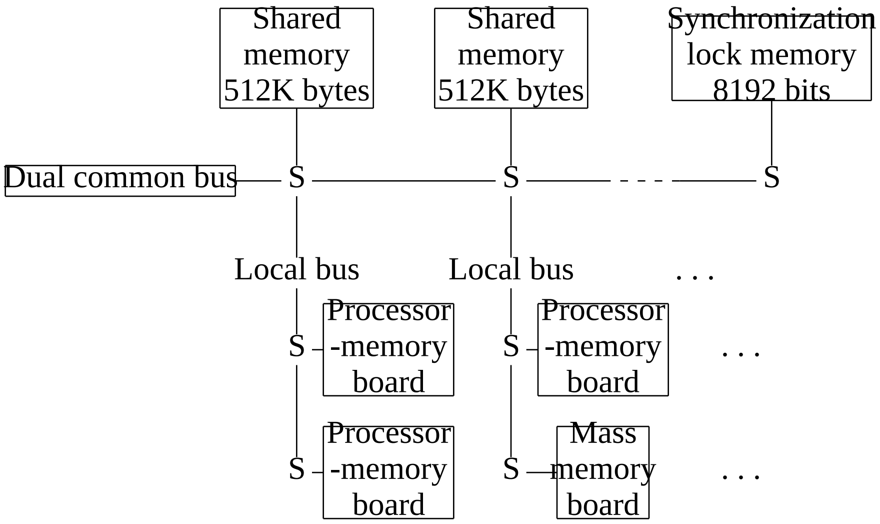


Figure 5: HEP Pipeline Architecture

optionally checked by the instruction issuing mechanism and, if not fulfilled, cause the PT to be immediately relinked into its task queue with the program counter of the PSW unaltered.

The architecture of the Flex/32 is conceptually simpler than that of the HEP, but the system support for parallelism is more complex. The machine consists of a set of single board microcomputers connected by several buses to each other and to some common memory and synchronization hardware. As shown in Fig. 6, there is a set of ten local buses, each of which can connect two boards. These are either single board computers, consisting of processor and memory, or mass memory boards. Two common buses connect the local buses together and to the common memory and synchronization hardware. The memory on the common bus is faster for a processor to access than that on the mass memory boards, but both are shared by all processors. The memory on a processor board is accessible only to that processor.

Hardware support for synchronization is supplied by an 8192 bit lock memory. This structure is meant to remove the requirement for repeated tests by a processor



S -- bus switching

Figure 6: Flex/32 Architecture

trying to obtain a lock. There is an interrupt system connected with each processor, which provides underlying hardware support for an event signaling mechanism between processors as well as for exception handling within a single processor. The processor/memory boards are based on the National Semiconductor 32032 microprocessor chip. There may be one or four megabytes of memory on a board and a VME bus interface is provided to connect an individual processor to I/O devices. A self-test system, connected to all processors, provides a mechanism for testing, bootstrapping and initializing the multiprocessor.

The Encore Multimax and Sequent Balance 8000 are related to the Flex/32 in that they all use the same microprocessor chip. The Multimax and the Balance systems, however, are both organized around a single, high speed bus instead of the local and common bus hierarchy exhibited by the Flex/32. The high level diagram is simpler for the Multimax, as shown in Fig. 7. The heart of the system is comprised of processor boards, each containing two complete processors and cache memory, and memory boards containing a dual bank memory per board. These processor and memory boards are connected with each other by means of a 96 bit pipelined bus which is arbitrated by a Bus Arbiter and System Control board. Secondary memory and peripherals are connected through Ethernet/Mass Storage boards, which also communicate with the system through the single bus.

Hardware support for synchronization in the Multimax is supplied by providing system support for the indivisible test-and-set instruction which is present in the instruction set of the microprocessor chip used in the system. A busy wait can then be implemented on any addressable unit of memory (byte) as the lowest level of synchronization. The busy wait consumes minimal resources since the byte being tested is

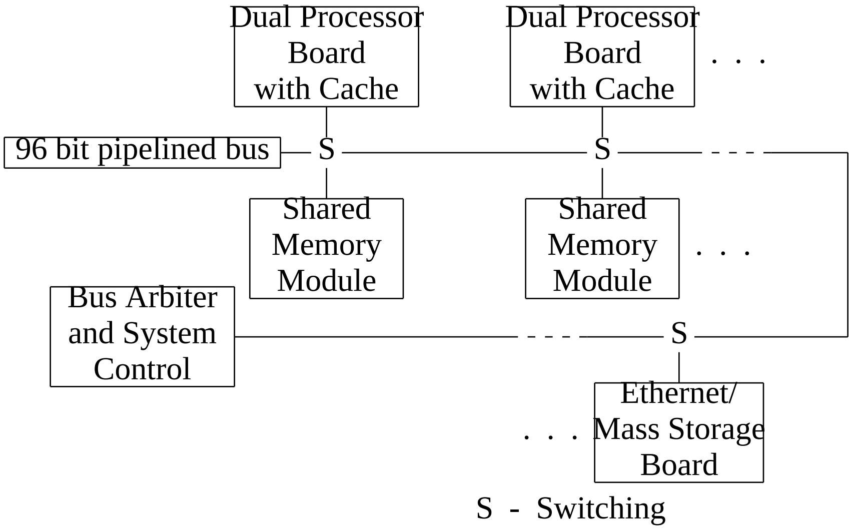


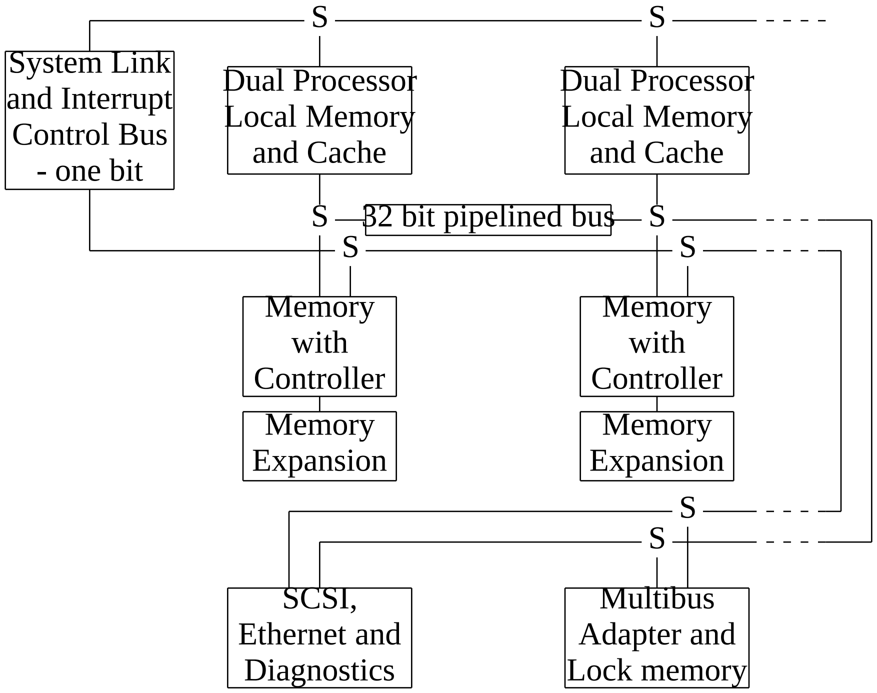
Figure 7: Encore Multimax Architecture

moved to the cache of the waiting processor, and no bus traffic is generated until the byte is changed, giving the wait loop a chance to exit.

The central pipelined bus in the Sequent Balance 8000 is 32 bits wide and links processors, memories and I/O interfaces, as in the previous system. See Fig. 8. Processors are again packaged two to a circuit board, but there are two different kinds of memory boards. A memory controller board contains the controller along with 2-Mbytes of memory and is capable of controlling up to 8-Mbytes. Additional memory, up to 6-Mbytes more, is contained on a memory expansion board which is connected to a specific controller board. A second, single bit, bus, called the System Link and Interrupt Controller (SLIC) bus, is used for interprocessor interrupts and other system control. Secondary memory and I/O is connected to the system through one of two different types of interfaces installed on the system buses. One is an adapter for the widely used MULTIBUS and the other a Small Computer System Interface.

There are two distinct hardware mechanisms used to support interprocessor synchronization in the system. The operating system kernel uses a set of 64 single bit "gates" contained in the SLIC subsystem to support mutual exclusion in access to shared kernel data structures. Since the 64 SLIC gates constitute a scarce resource, parallel user program synchronization is supported by a memory structure known as atomic lock memory. A 16K bit Atomic Lock Memory is contained on each MULTIBUS adapter board, and the indivisible test-and-set instruction of the processor is supported only on parts of the address space mapped into these memories.

Having seen examples of the hardware of four different shared memory multiprocessors on which the Force has been implemented, we now consider the range of parallel programming primitives offered to the user of these systems. Use of these parallel primitives from Fortran can be considered as a language extension. In two of the systems actual syntactic and semantic extensions of the Fortran language are implemented



S - Switching SCSI - Small Computer Systems Interface

Figure 8: Sequent Balance 8000 Architecture

through preprocessing, compiler modification or a mixture of the two. In the other two, strictly semantic extensions are implemented as Fortran function or subroutine calls, perhaps supported by linker options. The degree to which the operating system is involved in carrying out a parallel primitive operation is important to performance. A system call usually requires about three orders of magnitude longer than an operation implemented directly by the hardware.

The HEP system was one of those which actually modified the Fortran language by adding parallel constructs, although this was retracted to some extent in a second software release. To allow for the fact that an independent process usually requires some local variables, the process concept is tied to the Fortran subroutine. Fortran local variables are automatically private to one process and all COMMON variables are shared by all processes. The Fortran extension is merely a second version of the CALL statment, CREATE. Control returns immediately from a CREATE statement, but the created subroutine, with a unique copy of its local variables, is also executing simultaneously. The RETURN in a created subroutine has the effect of terminating the process executing the subroutine. Parameters are passed by address in both CALL and CREATE.

The only other major conceptual modification to Fortran allows access to the synchronizing properties of the full/empty state of memory cells. Any Fortran variable may be declared to be an "asynchronous" variable. Asynchronous variables are distinguished by names beginning with a $ symbol and may have any Fortran type. They may appear in Fortran declarative statements and adhere to implicit typing rules based on the initial letter. If such a variable appears on the right side of an assignment, wait for full, read and set empty semantics apply. When one appears on the left of an assignment, the semantics are wait for empty, write and set full. To initialize the state (not the value) of asynchronous variables, a new statement, PURGE, sets the state of an asynchronous variable empty regardless of previous state. The wait for full, read and leave full semantics of the *Copy* operation in the Force are also supported by the HEP.

The HEP Fortran extensions of CREATE and asynchronous variables are the simplest way to incorporate the parallel features of the hardware into the Fortran language. Since process creation is directly supported by the HEP instruction set and any memory reference may test and set the full/empty state that is associated with each memory cell, the Fortran extensions are direct representations of hardware mechanisms requiring no operating system intervention. The parallel computation model supported by the Fortran compiler and run time system can thus be viewed as shown in Fig. 9. A process with its own program counter and registers may spawn others like it using CREATE, and the processes interact by way of full/empty shared memory cells.

The parallel programming primitive operations can be characterized as in Table 2. Note that all the parallel primitives are user level operations requiring no operating system intervention. Interrupts are not present in the HEP. Conditions which would normally lead to an interrupt, including supervisor calls, result in the creation of a supervisor process to handle the condition and may or may not suspend the process causing the

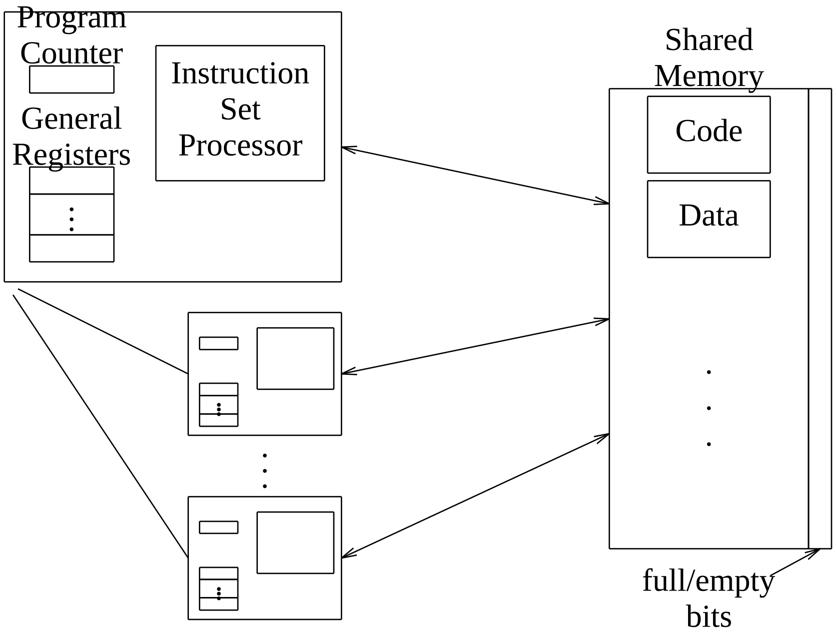


Figure 9: HEP Run Time System Model

Process		
Create		Quit and save state
Synchronization		
Produce	- Wait for empty, write and fill	Set location empty
Consume	- Wait for full, read and empty	
Copy	- Wait for full, read	

Table 2: HEP Parallel Primitives

condition.

The Flex/32 offers a fairly high level Fortran extension called ConCurrent Fortran, which is implemented by a preprocessor. On this machine the Force only makes use of the preprocessor's facility for allocating variables in shared memory. Otherwise the Force constructs are implemented by Fortran calls which give the user access to operating system functions supporting process management and interaction.

The process model in the Flex/32 is somewhat different from that of the HEP and is shown pictorially in Fig. 10. Since not all of the address space is shared, a process has a certain amount of strictly local memory. The system also manages a unique identifying tag for each process and maintains a process state which may be one of: running, non-existent, dormant, ready or suspended. There is also a received message queue for each process, which is managed by the system.

In addition to a slightly more complicated process model, the Flex/32 system supports more complex synchronization facilities. The total systems model is shown in Fig. 11. At the outset, processes are bound to individual processors. The processors may be multiprogrammed, so more than one process may be bound to a processor. The processes share communication and synchronization support supplied by the operating

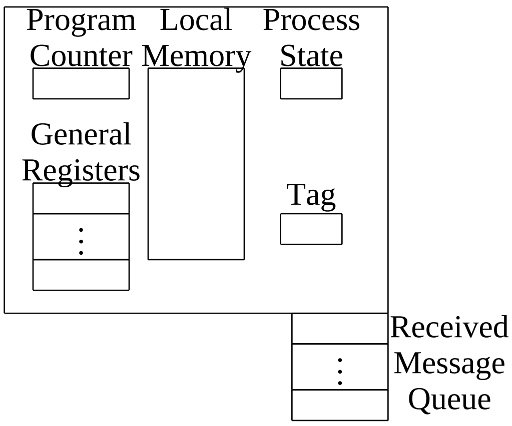


Figure 10: Flex/32 Run Time System - Process Model

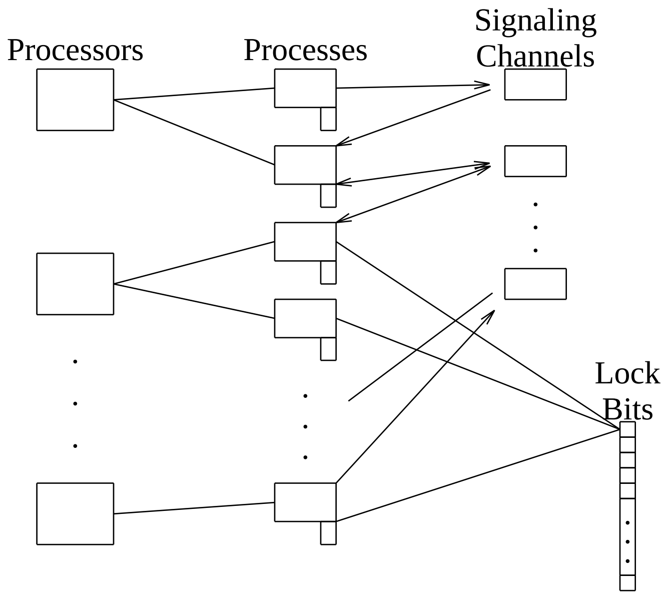


Figure 11: Flex/32 Run Time System - Overall Structure

system. The Signaling Channels implement the Event mechanism and may be attached to a process as a receiver of the event, an originator, or both. Lock bits may also be connected to several processors for mutual exclusion enforcement. The message passing facility is represented by the received message queue in each process and is not shown separately.

The Flex/32 system provides numerous parallel processing primitives. They may be divided into classes dealing with four different parts of the system model: Processes, Messages, Events and Locks. The structures associated with each of these parts and the primitives which act on the structures are summarized in Table 3. The primitives are implemented through system calls since most interact with the multiprogramming of single processors. Only a small part of this extensive parallel programming model is needed to support the implementation of the Force constructs.

Both the Encore Multimax and the Sequent Balance 8000 build parallel programming support on the Unix [19] process model. Both process models are identical except that the Sequent machine has a portion of the address space mapped to the Atomic Lock Memory unit of that machine. Since the Encore machine performs the function of this unit using the standard shared memory, it need not appear separately in the hardware model. Otherwise the Sequent process model of Fig. 12 represents that for both machines. Unix process management primitives are accessed through Fortran compatible calls to the operating system. Memory is shared among Unix processes differently in the two systems. In the Multimax, a Fortran function takes a common name and size as parameters and causes all subsequently forked child processes to inherit access to that same common area of memory. In the Balance 8000, a compiler/linker option accepts the common block names for the shared common blocks and generates the

Process				
		Structure		
State:	•running	•dormant	Tag:	unique,
	•suspended	•nonexistent		system-wide
	•ready			identifier
		Primitives:		
-create	-startup	-kill	-wait for termination	
		-get tag	-give up processor	
Messages				
Structure:	•type		•source id	
	•length		•destination id	
	•pointer			
Primitives:	-send		-receive/wait	
			-receive/fail	
Events				
Structure:	list of sources and destinations			
Primitives:	-configure	-activate	-on event call	
	-remove	-wait	-set timer	
		-passive test		
Locks				
Structure:		8192 single bits		
Operating mode:		polling or interrupt		
Primitives:		-allocate	-lock	
			-unlock	

Table 3: Flex/32 Parallel Primitives

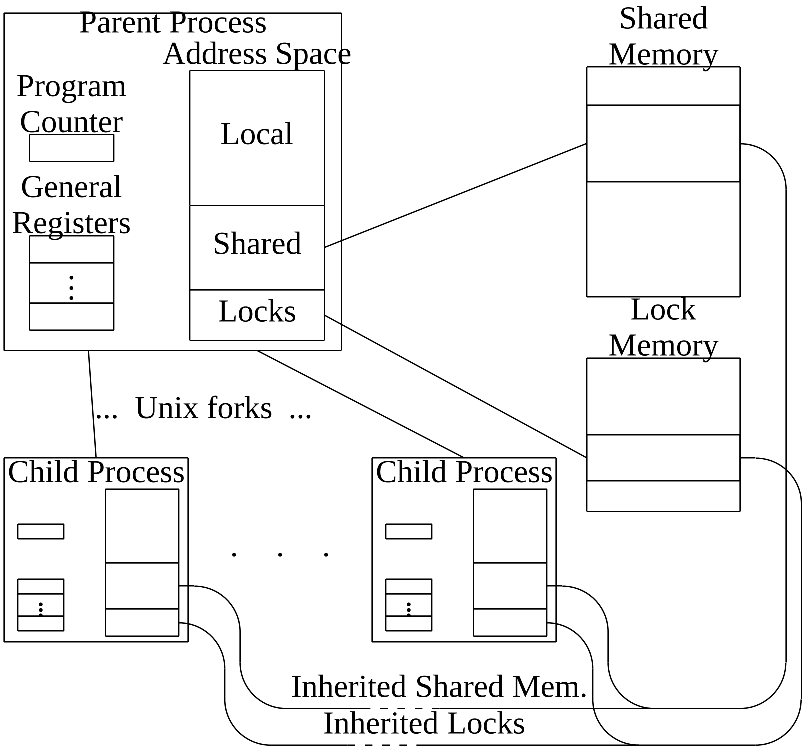


Figure 12: Sequent Process Model

appropriate system calls to make them available to all processes. Synchronization in both machines is based on the so-called spinlock [30] which causes a process to busy wait until the lock is clear. Spinlocks can use any byte of memory in the Multimax, while in the Balance 8000 they must be done on a bit of atomic lock memory. The parallel programming primitives for the two machines are summarized in Table 4.

The Force implementations are built on top of the parallel process models supplied by the several machines. On each machine there is a parallel environment consisting of synchronization and process management variables which are suppressed and hidden

Process Management

•Unix fork •Unix wait •Unix exit

Memory Sharing

Multimax	Balance 8000
Function:	Compiler/linker option:
SHARE(<common name>, <size>)	-F /<common block>/

Synchronization

lock	-wait for zero and set	unlock	-set to zero, no wait

Table 4: Parallel Primitives Used by the Force on Unix Based Systems

from the user by the Force macros. The parallel environment is established by a driver program which also manages the processes by forking them at the beginning, and synchronizing their join at the end, of the parallel program. The driver is the "main" program from the point of view of the Fortran system, and the Force "main" program is executed as a subroutine by the processes created by the driver.

The main difference in the overall structure of the driver is between the Unix based and non-Unix based systems. In the Multimax and Balance 8000 machines, which use the Unix process model, a *fork* operation produces an identical replica of the executing process. The entire core image of the parent process is copied, with the exception of those portions of the address space which have been specified as shared prior to the *fork*. The latter are made to refer to a shared memory area by means of the address map of the child process. Copying the core image has the effect of treating any private variables used by the child process as call-by-value parameters and shared variables as call-by-reference parameters. The non-Unix systems, HEP and Flex/32, handle *fork* by creating a process executing a specified subroutine, rather than by replicating the entire core image. In this case one must use care in passing parameters to a created subroutine.

The usual implementation of Fortran supports call-by-reference, and this is the case for both the HEP and the Flex/32. In the HEP, all physical memory is shared so a reference to either a local or common variable of the Fortran program can be passed to a created subroutine. Potential problems arise from our tendency to think that call-by-reference and call-by-value are equivalent for parameters which are read-only for the created subroutine. This is not correct for a parallel program, since the parameters would have to be read-only, i.e. constant, for the future execution of both the creating program and the subroutine for the two parameter passing methods to be equivalent. On the Flex/32, physical memory is divided between shared memory and that private to a single processor, and only variables in shared memory can be used as call-by-reference parameters. Since the host Fortran system only allows Fortran common variables to be shared, the utility of passing parameters to a created subroutine is limited. The main use of parameter passing on subroutine creation by Force implementations is to supply a unique process index to each process forked by the driver. This can also be accomplished, however, by initialization code at the beginning of the Force main program which uses a shared numbering variable in the parallel environment to cooperatively compute unique indices.

A second difference in driver structure is imposed by the way the underlying Fortran supports the specification of variables which are shared. If shared variables are specified as part of the extended parallel Fortran language, as is the case on the HEP and Flex/32, or if shared common areas are specified as part of the linker command, as in the Balance 8000, the structure of the driver is not influenced. If, however, sharing of variables must be specified through an operating system call at execution time, as in the Multimax, the driver must invoke this operation for all of the shared variables of the Force program before forking the parallel processes.

When the process executing the driver has finished establishing other processes executing the parallel main program, it calls that subroutine itself and becomes one of the force of processes. Once all processes are executing the Force main program, the parallelism constructs of importance are executable primitives implemented by macros which expand into executable code of the underlying Fortran system, including the parallel extensions supplied by the manufacturer. The macros may introduce auxiliary implementation variables or use variables in the parallel environment. Apart from the private unique index and the shared number of processes, which are always present, the environment may contain shared variables associated with cooperative computation of the process index, the *join* at the end of parallel execution, with barriers and with other constructs, such as self scheduled DOALLs, which use shared variables for process cooperation.

A few selected examples of the implementation of executable macros will serve to illustrate some of the differences encountered using the various hardware and software primitives supplied by the different systems. A major hardware difference between the HEP and the rest of the systems influences even the lowest level synchronizations. This is that the basic synchronization mechanism on the HEP is the producer/consumer operation applied to any memory cell while the basic synchronization on the other three machines is an atomic test-and-set operation, which is used within a loop to implement the low level lock/unlock mechanism. Although the lock/unlock primitives are not supported directly by the Force they are closely related to the named critical section, which is. It is therefore interesting to look at the implementations of critical sections and asynchronous variables on the two different types of hardware. Table 5 compares the implementation of critical sections on the HEP and on the other machines. The table shows that the only real difference is that the full power of the HEP hardware primitive is not used, since the value of the asynchronous variable is ignored; only its state is used to perform locking. Of course, some of the machines implement the lock/unlock operations with spinlocks, so that processes do a busy wait for the lock to clear, while others include a process suspension, possibly after a timeout. The effect of different waiting mechanisms can be seen in the performance of critical sections in tightly and loosely synchronized code sections but does not appear in the implementation. The implementation of *Produce*, *Consume* and *Copy* on the HEP are directly in terms of single user instructions, so only their implementation on other machines is of interest. Table 6 shows a method of implementing asynchronous variables using two locks to encode the

HEP	**Others**
System state and initialization:	System state and initialization:
Single full/empty variable - full	Single bit lock - clear
Critical section code:	Critical section code:
Consume critical section variable	Set critical section lock
Execute code body	Execute code body
Produce critical section variable	Clear critical section lock

Table 5: Implementation of Critical Sections

full/empty state.

One of the most cooperative synchronizations supported by the Force is the *Barrier*, and its implementations illustrate several issues which result from the underlying process models and parallel primitives. One machine independent feature which must be taken into account in implementations of the barrier is that it must be possible to execute several barriers in a row which share implementation variables. This would be necessary even if separate barriers generated unique implementation variables because a single barrier might be enclosed in a sequential loop. Thus it is not enough to ensure that processes in the barrier do not conflict with those entering it, but also that processes entering the barrier for a second time do not conflict with those exiting from a prior execution of the barrier.

The HEP implementation is based on the use of asynchronous variables to implement the process delay at the barrier. If processes wait for an asynchronous variable to be full using *Copy* instead of *Consume*, then a single *Produce* can release many processes. This is the action needed by a barrier implementation, which always involves counting processes as they arrive and causing all but the last one to wait. The last arriving process then executes the code body of the barrier and releases the waiting processes. The lock/unlock synchronization is not as well suited to this because, if many processes are waiting to lock a single lock, only one at a time can succeed. Thus the straightforward implementation is for each process which successfully locks a lock unlock it for the next process. Another synchronization which embodies the idea of many waiting processes released by a single action is the event mechanism supported by the Flex/32. Here all but the last arriving process wait on an event which is activated (posted) by the last one after executing the body of the barrier. Proper implementation of the event mechanism by the underlying system ensures that each process "sees" a single activation of an event exactly once. Table 7 shows implementations of barriers on the machines discussed. Two implementations are shown for the HEP. The

Shared variable - S   Private variable - P   Locks - L1, L2

L1	L2	Meaning
0	1	Empty
1	0	Full
1	1	Operation in progress
0	0	unused

Produce	**Consume**	**Copy**
lock L1	lock L2	lock L2
S = expression	P = S	P = S
clear L2	clear L1	clear L2

Table 6: Implementation of Asynchronous Variables with Locks.

HEP - Active Waiting			**HEP - Process Suspending**		
System State	Initialization		System state	Initialization	
Entry lock	-	full	Process state save area	-	empty
Exit lock	-	empty	Counter	-	zero
Counter	-	zero			

HEP - Active Waiting — Barrier Code

```
Copy entry lock
Count arriving process
If last process then
  Execute code body
  Consume entry lock
  Produce exit lock
Copy exit lock
Count exiting process
If last process then
  Consume exit lock
  Produce entry lock
```

HEP - Process Suspending — Barrier Code

```
Count arriving process
If not last one then
  Save state and quit
Else
  Recreate other processes
  Clear counter
```

Flex/32		**Multimax & Balance 8000**	
System State	Initialization	System State	Initialization
Barrier event	- connected to all processes as source/destination	Locks	- IN, OUT
Counter	- zero	Counter	- zero

Flex/32 — Barrier Code

```
Lock counter
  Count arriving process
  Clear counter if last
Unlock counter
If last process then
  Execute code body
  Activate barrier event
else
  Wait for barrier event
```

Multimax & Balance 8000 — Barrier Code

```
Lock IN
If not last in then
    Count process in
    Clear IN
    Lock OUT
  Else
    Execute code body
Endif
If not last out then
    Count process out
    Clear OUT
  Else
    Clear IN
Endif
```

Table 7: Implementations of Barriers

second illustrates what is done if resource utilization by processes doing a busy wait at the barrier is large enough to slow processes which have not yet reached it.

The only class of macros not yet mentioned in this implementation discussion is that labeled Parallel Execution in Fig. 1b. Both the *Pcase* ant the DOALLs have the function of allocating different code sections to different processors for parallel execution. In *Pcase*, the code sections are distinct bodies of text, while in the DOALLs the different sections are distinguished by the value of an index. Thus the only real implementation issue for this set of macros lies in the difference between pre- and self scheduling. The extreme simplicity of the prescheduled DOALL implementation has been shown above, so a few comments on the *Selfsched DO* are appropriate. In this construct, a shared index must be asynchronously initialized, updated and tested for exhaustion. As with the *Barrier*, it must be possible to execute a *Selfsched DO* repeatedly within an enclosing sequential loop. Thus no process may enter a *Selfsched DO* a second time until all processes have finished with the shared index mechanism on the previous execution. It is necessary that the index be initialized before any process attempts to access its value, but it is not strictly necessary that all of the processes enter the DOALL "together". A sufficient, but overly strong, synchronization mechanism is to enclose the loop index initialization in a barrier at the start of the DOALL. Weakening the synchronization so that processes can get started on the body before all have arrived is an interesting exercise, but it has little performance benefit in a tightly synchronized program, where processes make well coordinated progress through the code.

Conclusions

The Force is primarily a programming methodology for managing many processes on a shared memory multiprocessor. Secondarily, it is a parallel extension to Fortran which allows identical parallel programs to be run on several existing multiprocessors. Some key ideas embodied in the Force are independence of the number of processes executing a parallel program, high performance of a tightly coupled program on a dedicated machine, suppression of process management, and reliance on "generic" synchronizations. The Force is a true multiprocessor programming language in that it has no specifically vector constructs and does not avoid the idea of instruction streams, as is done in functional or dataflow languages. It is oriented toward shared memory multiprocessors, and there are basic problems in its structure which make implementation on a distributed memory machine not only difficult but somewhat inconsistent. It is difficult to reconcile send and receive operations in a distributed memory system with the need to suppress process identity. One might require all shared variables to be asynchronous, since *send* and *receive* are closely related to *Produce* and *Consume*, and variable identity might replace process identity. But the use of such small messages would entail severe performance penalties on currently available distributed memory systems.

The Force is an evolving system, as indicated by the discussion of the *Resolve* macro, which is not yet included in implementations. Pushing this minimal support for functional partitioning to accommodate support for divide and conquer type programs, without reverting to explicit process management is one direct line of evolution. Another is to address the requirement that the number of processes be constant during execution or that all processes need to be coscheduled on real processors for reasonable performance. Removing these requirements will be particularly important when the Force is used on a multiprogrammed multiprocessor. Of course there is some incompatibility between the use of parallelism in a single program, which is done to reduce that program's completion time, and multiprogramming, which is intended to maximize the throughput of a system running multiple jobs at some expense to the completion time of each job.

Implementation of the Force on systems involving three, rather different, process models has not been difficult, and portability between machines with similar system supported primitives is almost trivial. Given the fairly strong differences between the machines already hosting the Force, there should be no major difficulty in porting the system to any shared memory multiprocessor with which the author is familiar.

References

[1] H. F. Jordan, "Experience with pipelined multiple instruction streams," *Proc. IEEE,* Vol. 72, No. 1, pp. 113-123 (Jan. 1984).

[2] M. C. Gilliland, B. J. Smith and W. Calvert, "HEP - A semaphore-synchronized multiprocessor with central control," *Proc. 1976 Summer Computer Simulation Conf.,* Wash., DC, pp 57-62 (July 1976).

[3] H. F. Jordan, "Structuring parallel algorithms in an MIMD, shared memory environment," *Parallel Computing,* Vol. 3, No. 2, pp. 93-110 (May 1986).

[4] Patel, N. R. and Jordan, H. F., "A parallelized point rowwise successive over-relaxation method on a multiprocessor," *Parallel Computing,* Vol. 1, No. 3&4 (Dec. 1984).

[5] N. Patel, W. B. Sturek and H. F. Jordan, "A parallelized solution for incompressible flow on a multiprocessor," *Proc. AIAA 7th Computational Fluid Dynamics Conf.,* Cincinnati, OH, pp. 203-213 (July 1985).

[6] H. F. Jordan, "Parallel computation with the Force," *ICASE Rept. No. 85-45,* NASA Langley Res. Ctr., Hampton, VA (Oct. 1985).

[7] H. F. Jordan, "The Force on the Flex: global parallelism and portability," *ICASE Rept. No. 86-54,* NASA Langley Res. Ctr., Hampton, VA (Aug. 1986).

[8] H. F. Jordan, M. S. Benten and N. S. Arenstorf, "Force User's Manual," *ECE Tech. Rept. 86-1-4,* Dept. of Electrical and Computer Engineering, University of Colorado, Boulder, CO (Oct. 1986).

[9] E. L. Lusk and R. A. Overbeek, "Implementation of monitors with macros: A programming aid for the HEP and other parallel processors," *Tech. Rept. ANL-83-97,* Argonne National Lab., Argonne, IL (Dec. 1983).

[10] C. A. R. Hoare, "Monitors: an operating system structuring concept," *Comm. ACM,* pp. 549-557 (Oct. 1974).

[11] R. G. Babb II, "Parallel processing with large-grain data flow techniques," *Computer,* Vol. 17, No. 7, pp. 55-61 (July 1984).

[12] F. Darema-Rogers, D. A. George, V. A. Norton and G. F. Pfister, "VM/EPEX - A VM environment for parallel execution," *IBM Research Rept. RC11225(#49161), IBM T. J. Watson Res. Ctr.* (Jan. 1985).

[13] "The Uniform System Approach to Programming the Butterfly Parallel Processor," BBN Laboratories Inc., draft (Nov. 1985).

[14] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.* Vol. C-21, No. 9, pp. 948-960 (1972).

[15] J. B. Dennis, "Data flow supercomputers," *IEEE Computer,* pp. 48-56 (Nov. 1980).

[16] J. S. Kowalik, Ed., *Parallel MIMD Computation: The HEP Supercomputer and its Applications,* MIT Press (1985).

[17] *The Flex/32® System Overview,* Flexible Computer Corp., Dallas, TX, (1986).

[18] *Multimax Technical Summary,* Encore Computer Corporation, Marlboro, MA, (1986).

[19] *Balance® 8000 System Technical Summary,* Sequent Computer Systems, Beaverton, OR (1984).

[20] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. ACM,* Vol. 7, No. 7, pp. 365-375 (July 1974).

[21] M. E. Conway, "Design of a separable transition-diagram compiler," *Comm. ACM,* Vol. 6, No. 7, 396-408 (1963).

[22] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Comm. ACM* Vol. 9, No. 3, pp. 143-155 (1966).

[23] J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK Users Guide,* SIAM Publications, Phil., PA (1979).

[24] G. Alaghband and H. F. Jordan, "Multiprocessor sparse L/U decomposition with controlled fill-in," *ICASE Rept. No. 85-48,* NASA Langley Res. Ctr., Hampton, VA, 1985.

[25] J. J. Dongarra, "Performance of various computers using standard linear equations software in a Fortran environment," *ANL Tech. Memo.* (Argonne National Lab., 1983).

[26] L. M. Adams and H. F. Jordan, "Is SOR color-blind?," *SIAM J. Sci. Stat. Comput.,* Vol. 7, No. 2, pp. 490-506, April 1986.

[27] U. Banerjee, S. C. Chen, D. J. Kuck and R. A. Towle, "Time and parallel processor bounds for FORTRAN-like loops," *IEEE Trans. Comput.,* **C-28** (9) (1979) 660-670.

[28] T. W. Pratt, "Pisces: An environment for parallel scientific computation," *ICASE Rept. No. 85-12,* NASA Langley Research Center, Hampton, VA (February 1985).

[29] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer - Designing an MIMD shared memory parallel computer," *IEEE Trans. Comput.,* Vol. C-32, No. 2 (Feb. 1983).

[30] W. A. Wulf, R. Levin and S. P. Harbison, *HYDRA/C.mmp: An Experimental Computer System,* McGraw-Hill, N. Y. (1981).