NASA CR-178,16

NASA Contractor Report 178161

ICASE REPORT NO. 86-54

NASA-CR-178161
19860020908

ICASE

THE FORCE ON THE FLEX: GLOBAL PARALLELISM AND PORTABILITY

Harry F. Jordan

Contract No. NAS1-17070
August 1986

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23665

THE FORCE ON THE FLEX: GLOBAL PARALLELISM AND PORTABILITY.

HARRY F. JORDAN[†]

ABSTRACT

A parallel programming methodology, called the force, supports the construction of programs to be executed in parallel by an unspecified, but potentially large, number of processes. The methodology was originally developed on a pipelined, shared memory multiprocessor, the Denelcor HEP, and embodies the primitive operations of the force in a set of macros which expand into multiprocessor Fortran code. A small set of primitives is sufficient to write large parallel programs, and the system has been used to produce 10,000 line programs in computational fluid dynamics. The level of complexity of the force primitives is intermediate. It is high enough to mask detailed architectural differences between multiprocessors but low enough to give the user control over performance.

The system is being ported to a medium scale multiprocessor, the Flex/32, which is a 20 processor system with a mixture of shared and local memory. Memory organization and the type of processor synchronization supported by the hardware on the two machines lead to some differences in efficient implementations of the force primitives, but the user interface remains the same. An initial implementation was done by retargeting the macros to Flexible Computer Corporation's ConCurrent C language. Subsequently, the macros were caused to directly produce the system calls which form the basis for ConCurrent C. The implementation of the Fortran based system is in step with Flexible Computer Corporations's implementation of a Fortran system in the parallel environment.

[†]University of Colorado, Boulder, CO 80309-0425.

Research was supported in part by NASA Contract No. NAS1-17070 and by the Air Force Office of Scientific Research under Grant No. AFOSR 85-0189 while the author was in residence at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665.

N86-30380 #

The Global Parallelism Concept

The unifying idea behind the programming environment discussed in this paper is that of "global" parallelism. In contrast to the dataflow point of view we retain the idea of multiple instruction streams but insulate the user from the detailed management of the streams on an individual basis. One view of this unifying idea is as a way of incorporating parallelism into the structural hierarchy of a program. It is in contrast to the encapsulation of parallelism into one or more program modules and can be viewed as parallelism with the largest possible "grain" size.

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 into the description of an algorithm is an important issue. The leaf level, where SIMD parallelism is appropriate can be denoted as fine grained parallelism. As MIMD parallelism is applied at higher levels, we can speak of algorithms with coarser grained parallelism. With 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 (as in SIMD) for fine grained parallelism to make sense. 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 [1] 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. It is well known that even a small amount of sequential code in an otherwise parallel program can decrease efficiency significantly on a system with a large degree of parallelism. The encapsulation idea forces all code above a certain level of structure to be sequential. 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 parallelism at the highest level of program structure in the MIMD environment. Experience shows that it is quite feasible to write applications programs with "global" parallelism. In this environment 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 explicitly appearing code to deal with parallelism is all related to process synchronization and data sharing. The idea of global parallelism applies to the decomposition of algorithms on the basis of data rather than function. With a high degree of parallelism some data decomposition of an algorithm is surely necessary since the number of independent functions is limited. Thus this idea is probably most appropriate to systems supporting many processes.

The above concept of global parallelism has been incorporated into a programming methodology called the "force". The force [2] methodology for parallel programming arose in trying to produce high performance parallel programs in a shared-memory multiprocessor running up to 200 processes on the same user program [3]. Multiprogramming was not an issue, and all emphasis was on single problem solution speed. Partly for performance measurement purposes and partly for program manageability, a programming style emerged in which a single piece of code was written which could be executed by a force of processes in parallel. The number of processes constituting the force is constant during execution but is bound as late as the beginning of execution, and may be one. Similar techniques have been developed for programming some more recent multiprocessors, notably the Bolt Beranek and Newman Butterfly [4] and the IBM research processor RP3[5].

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.

Statements written in a force program are implicitly executed by all processes in parallel. Variables appearing in statements are divided into local variables, having separate instances for each process, and global variables, shared among all processes of the force. An assignment statement, for example, may combine the values of global and local variables to produce a local or global result. If the result is local, no assignment conflict is possible. If it is global, then assignment conflict must be prevented, either by allocation of disjoint sections of a global 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.

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 and of supporting synchronized access to global 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.

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.

Macros of the second class deal with variable declaration. This class currently includes only *Global* and *Local* macros. Global variables are associated with Fortran common while local variables are ordinary Fortran variables local to a separately compiled program module. Sharing of local variables among several program modules, but local to one process, can only be accomplished by parameter passing. The static allocation flavor of Fortran makes it difficult to build a structure of common variables with one instance for each process when the number of processes is not known until execution time.

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. 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 these conditions may be true simultaneously. No order of evaluation of the conditions is specified, and each will be evaluated by one arbitrarily selected process. Thus conditions depending only on global 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

```
Macros associated with program structure:
    Force <name> of <# procs> ident <proc #>
        <declarations>
    End declarations
        <force program>
    Join

    Forcesub <name> of <#procs> ident <proc #>
        <declarations>
    End header
        <subroutine body>
    RETURN

    Forcecall <name>(<parameters>)

Declaration macros:
    Global <variable names>
    Local <Fortran declaration>

Macros specifying parallel execution:
    Pcase on <variable>
        <code block>
    Usect
        <code block>
    ...
    End pcase

    [Pre|Self]sched DO <n> <var>= <i1>, <i2>, <i3>
        <loop body>
<n>  End [pre|self]sched DO

Synchronizing macros:
    Barrier
        <code block>
    End barrier

    Critical $<variable>
        <code block>
    End critical

    Produce <variable> = <expression>         (producer)
    ... = ... Use(<variable>) ...             (consumer)
```

Figure 1: Specific Macros for a Force Program

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 all local 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 based 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 global variables have a binary state, full or empty, as well as a value. Execution by some process of the macro, *Produce <variable> = <expression>,* waits for the variable to be in the empty state, sets its value to that of the expression and makes it full, all in a manner which is atomic with respect to the progress of any other process. Similarly, the macro, *Use(<variable>),* appearing in an expression returns the value of the variable when it 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 full/empty variables are *Purge <variable>,* which sets a variable empty regardless of its previous state, and *Copy(<variable>),* which waits for the variable to be full and returns its value but does not empty it.

A major weakness in the current set of force macros 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. 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. If the number of components is larger than the number of processes in the force, 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 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 contain a mechanism for efficiently handling such algorithms without making the user responsible for explicit 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.

Status and Applications

The force macros described above represent a parallel programming environment in which process management is suppressed, and programs are independent of the number of processes executing them, except for performance. The system makes parallel execution the normal mode; sequential operation must be explicitly invoked. Two features combine to ensure that there is no topological structure to the parallel environment. First, processes are identical in capability, and, second, all variables are either strictly local to one process or uniformly shared among all of them. This eliminates much of the complexity of the "mapping problem" encountered in constructing parallel versions of algorithms for machines with visible processor topology.

Primitive operations of the force are available to support both fine-grained and coarse-grained parallelism. Many of the primitives, especially those supporting fine grained interaction, require only local analysis to determine correctness of the synchronization. This locality strengthens the case for being able to automate this analysis. The ability to recursively subdivide the force, coupled with the support for parallelization on the basis of data partitioning, orients the system towards "massive" parallelism in that the activity of large numbers of processes can be compactly specified.

The system is currently tied fairly tightly to shared memory with undifferentiated processes and, for that reason, does not support message passing. One could view the *Produce* and *Consume* primitives as a weak form of send and receive operations with the associated variable playing the role of an unbuffered, one word, message channel.

The force system has been used to produce a parallel Gaussian elimination subroutine[2] identical in interface and operation to the SGEFA routine of LINPACK[6]. 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[7], [8] and MacCormack's method for a shock tube model[9]. It has also been used to implement a new parallel pivoting algorithm for solving sparse systems of linear equations[10].

The Machines

The issues which arise in implementing the force on a shared memory multiprocessor will be addressed by considering implementations on two, fairly different, such machines: the Denelcor HEP[3] and the Flexible Computer Systems Flex/32[11]. Not only are the two systems fairly different in architecture, the HEP being a pipelined multiprocessor while the Flex/32 is built from multiple microprocessors, but the primitive operations for establishing and controlling parallel processes which are supported by the systems are quite different. These parallel primitive operations are a combined result of hardware, compiler support, operating system and run-time libraries. A summary of the hardware, parallelism model and primitive operations for each of the machines follows.

The HEP

The HEP computer is a multiple instruction stream computer categorized as MIMD by Flynn[12]. 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. 2. Even within a single PEM, however, HEP is still an MIMD computer. 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

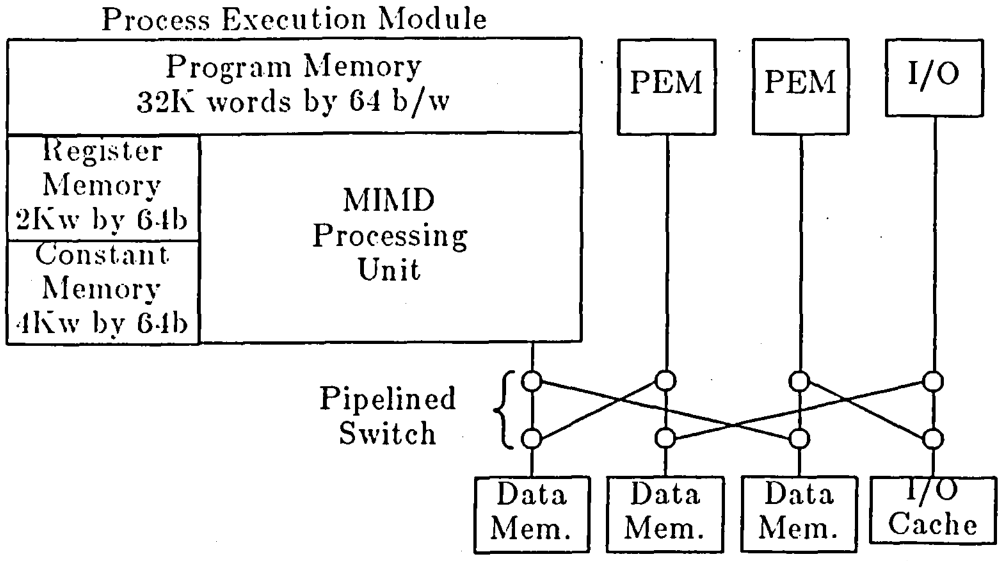

Figure 2: Architecture of the HEP Computer

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 major flavor of the architecture can be given by considering only one of them, the main execution pipeline. Heavy use has been made of pipelines in vector processors (SIMD computers). In such machines the operating units are broken into small stages with data storage in each stage. Complete processing of a pair of operands involves the data passing sequentially through all stages of the "pipeline." Parallelism is achieved by having different pairs of operands occupying different stages of the pipeline simultaneously. The main execution pipeline of HEP can be viewed as a unified structure which processes most instructions using a pipeline with eight steps. Independent instructions (along with their operands) flow through the pipeline with an instruction being completely executed in eight steps. Independence of the activities in successive stages of the pipeline is achieved not by processing independent components of vectors but by alternately issuing instructions from independent instruction streams. Multiple copies of process state, including program counter, are kept for a variable number of processes. 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.

The previous paragraph describes the register to register instructions. Those dealing with main memory (data memory) behave differently. Data memory is shared between PEMs and words are moved between register and data memories by means of a class of Storage Function Unit (SFU) instructions. The relationship between the main execution pipeline and the SFU is shown in Fig. 3. A process is characterized by a Process Status Word (PSW) containing a program counter and index offsets into both register memory and constant memory to support the writing of reentrant code. Under the assumption that multiple processes will cooperate on a given job or task and thus share memory, memory is allocated and protected on the basis of a structure called a task. There are a maximum of 16 tasks, eight supervisor tasks and eight user tasks. The 128 possible processes are divided into a maximum of 64 users and 64 supervisor processes which must belong to tasks of corresponding types. Aside from this restriction a task may have any number of processes, from zero to 64.

An active process is represented in the hardware by a Process Tag (PT) which points to one of the 128 possible PSWs. The instruction issuing operation maintains a fair allocation of resources between tasks first and between processes within a task second by means of 16 task queues, each containing up to 64 PTs and a secondary queue called the snapshot queue. PTs coming one at a time from the snapshot queue cause the issuing of an instruction from the corresponding process into the execution pipeline.

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. When a PT comes to the head of the SFU snapshot queue a

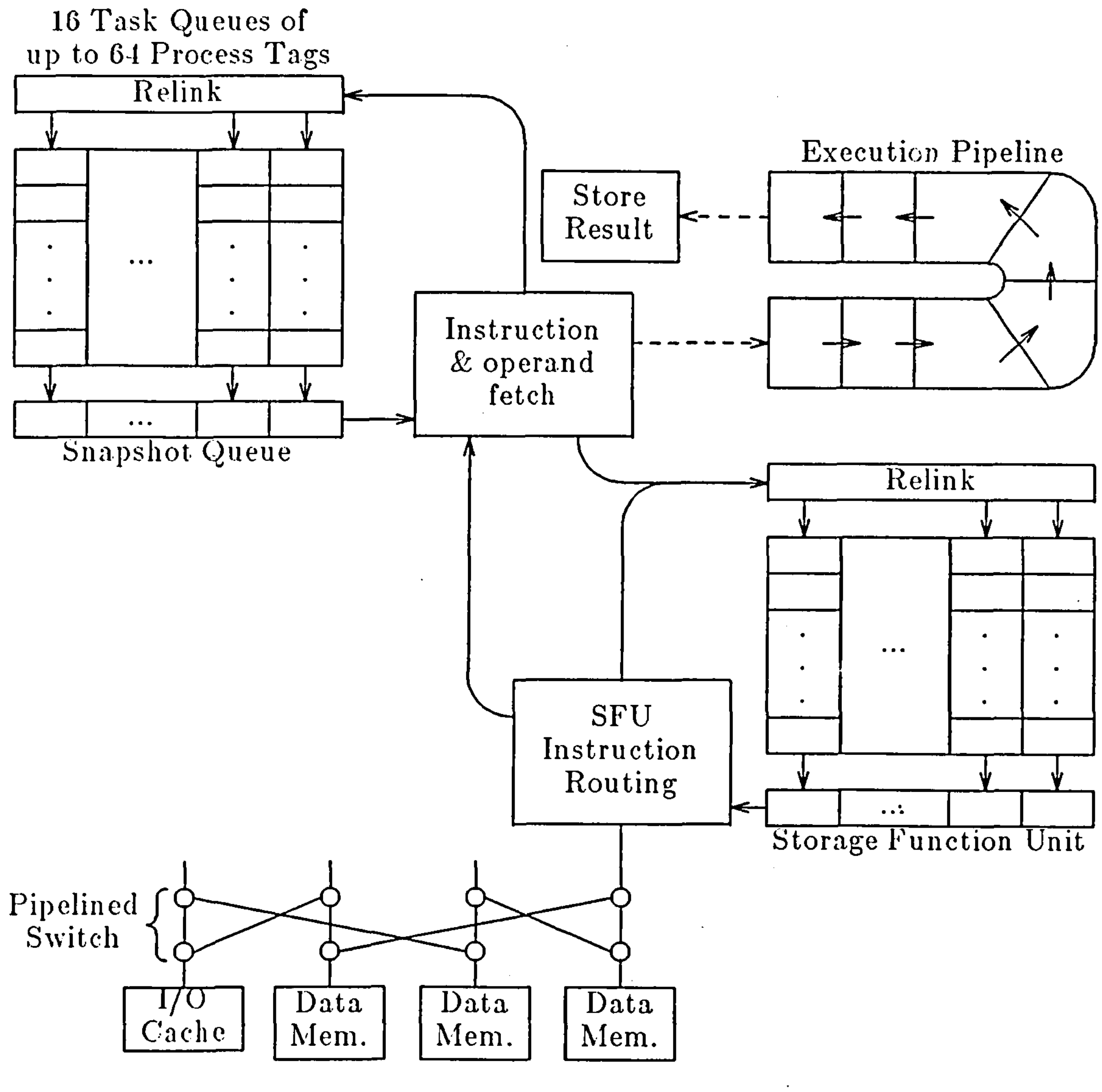

Figure 3: HEP Pipeline Architecture

memory transaction is built and sent, along with the PT, into the attached node of a pipelined, message-switched switching network. The transaction propagates through the switch to the appropriate memory bank and returns to the SFU with status and perhaps data. An SFU instruction behaves as if it were issued into a pipeline longer than the eight step execution pipeline but with the same step rate.

Hardware support for process synchronization is based on producer/consumer synchronization. Each cell in register and data memories has a full/empty state and synchronization is performed by having an instruction wait for its operands to be full and its result empty before proceeding. The synchronizing conditions are 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.

Compiler level support consists of minimal language extensions to give the user access to the parallelism of the hardware. The extensions can be represented as subroutine calls or incorporated into the language definition. Since the force is based on Fortran, the extensions to that language are described. To allow for the fact that an independent process usually requires some local variables, the process concept is tied to the Fortran subroutine. 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 states of asynchronous variables to empty regardless of their previous states.

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. The parallel computation model supported by the Fortran compiler and run time system can thus be viewed as shown in Fig. 4. 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 1. 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 giving rise to the condition.

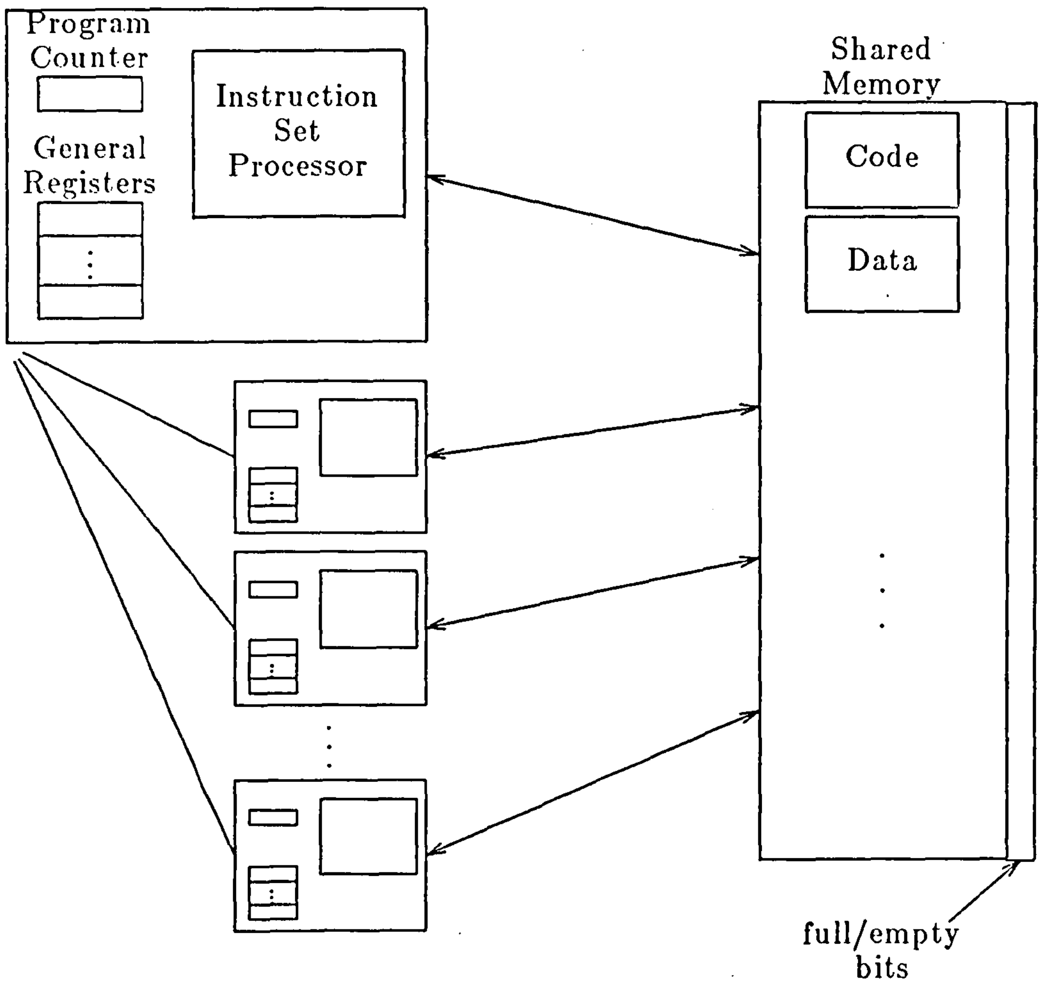

Figure 4: HEP Run Time System Model

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

Table 1: HEP Parallel Primitives

The Flex/32®

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. 5, there are a set of local buses, ten of them, each of which can connect two boards, which 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 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 process model in the Flex/32 is somewhat different from that of the HEP and is shown pictorially in Fig. 6. Since not all of the address space is

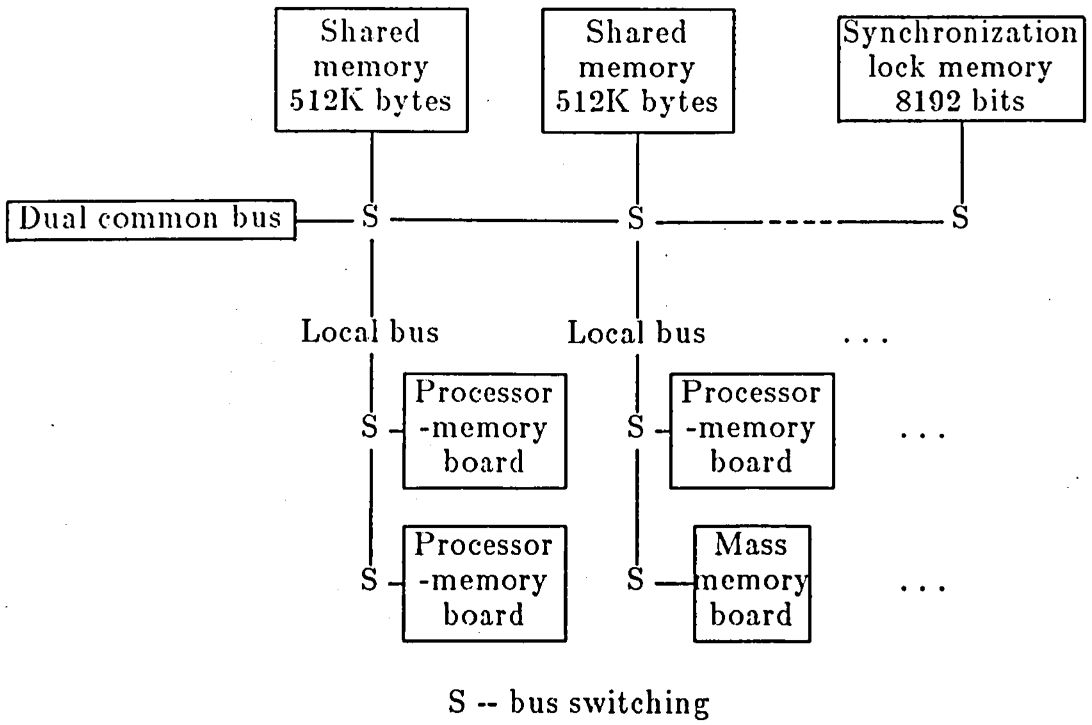

Figure 5: Flex/32 Architecture

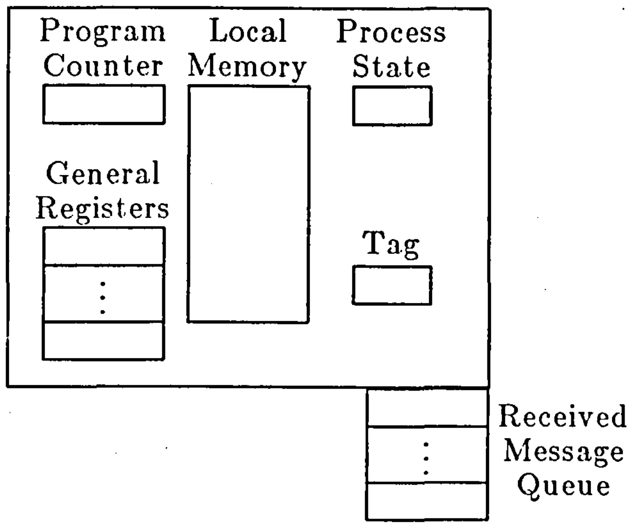

Figure 6: Flex/32 Run Time System - Process Model

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 a more complex model of synchronization facilities linking processes. The total systems model is shown in Fig. 7. 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 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 thus not shown separately in the system model.

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 2. The primitives are implemented through system calls. Since most of them interact with the multiprogramming of single processors, operating system intervention is usually required. Only a small part of this fairly extensive parallel programming model is needed to support the implementation of the force constructs.

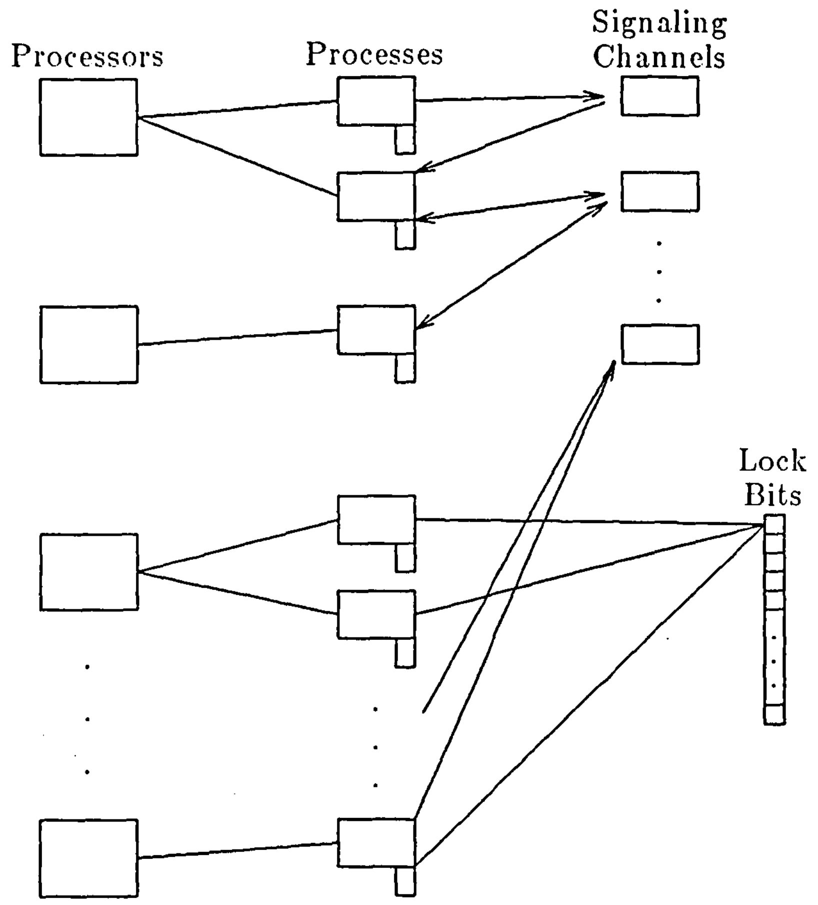

Figure 7: Flex/32 Run Time System - Overall Structure

Process				
Structure	State:	•running	Tag:	unique,
		•suspended		system-wide
		•ready		identifier
		•dormant		
		•nonexistent		
Primitives:	get tag		create	
	start up		wait for termination	
	kill		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 2: Flex/32 Parallel Primitives

Implementation of Force Primitives

Basic hardware support for synchronization on the HEP is through the produce and consume operations on full/empty memory cells. The basic hardware support for synchronization on the Flex/32 is supplied by the common lock memory and the interrupt hardware. Table 3 compares the implementation of critical sections on the two machines. The implementations are very similar, but a detailed look at the differences will introduce the issues to arise in more disjoint implementations of other primitives to follow.

The basic HEP synchronization is somewhat more powerful than is needed for critical sections. A single full/empty variable suffices to control entry to the section, but only its state is significant; the value of the variable

	HEP	
Ststem state and initialization:		
	Single full/empty variable	- full
Critical section code:		
	Consume critical section variable	
	Execute code body	
	Produce critical section variable	
Performance:		
	Consume and produce are single user-mode instructions, but may result in some resource usage by waiting processes.	
	Flex/32	
System state and initialization:		
	Single bit lock	- clear
Critical section code:		
	Set critical section lock	
	Execute code body	
	Clear critical section lock	
Performance:		
	Set and clear locks are done by system calls. Processor rescheduling is possible, and wakeup of a delayed process may be by interrupt or polling.	

Table 3: Implementation of Critical Sections

is unused. The Flex/32 locks are well suited in complexity to what is needed for critical section control. The process delay which may be required on critical section entry is supported by the hardware of the HEP, making critical section entry a user level operation with no operating system intervention. On the other hand, a small amount of system resources is consumed by waiting processes, which may cause congestion if many processes wait simultaneously. The Flex/32 implements locking and unlocking through system calls. This is costly in terms of performance but allows processor rescheduling. Wakeup of blocked processes may either be by polling or by interrupt.

There is considerably more structure to the implementation of the *Barrier* macro on both machines.. Table 4 summarizes the implementations, including two implementations for the HEP having quite different performance characteristics. The two HEP implementations emphasize the difference between suspended and partially active waiting, which was mentioned in connection with the critical section code. This issue was not important in connection with critical sections because the control is very simple

HEP - Active Waiting

```
System State          Initialization
   Entry lock         -     clear
   Exit lock          -     set
   Counter            -     zero
Barrier Code
   Wait for entry lock clear
   Count arriving process
   If last process then
     execute code body
     set entry lock
     clear exit lock
   Wait for exit lock clear
   Count exiting process
   If last process then
     set exit lock
     clear entry lock
```

HEP - Process Suspending

```
System state            Initialization
   Process state        -     empty
     save area
   Counter              -     zero
Barrier Code
   Count arriving process
   If not last one then
     save state and quit
   else
     recreate other processes
     clear counter
```

Flex/32

```
System State       Initialization
   Barrier event   -  connected to all processes
                        as source/destination
   Counter         -  zero
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
```

Table 4: Implementation of Barriers

and because the probability that many processes will simultaneously wait on entry to critical sections with the same lock is low. In the *Barrier*, it is guaranteed that all processes simultaneously access the same blocking condition. There is only one implementation for the Flex/32 since all synchronization support is through operating system calls and involves process suspension rather than active waiting.

The critical section and the Barrier implementations serve to give an idea of the range of differences in the implementation of Force primitives on the two architectures. Many of the primitives, such as prescheduled DOALL, did not change at all between the machines, while others, such as self-scheduled DOALL, build on the same techniques used in the critical section and Barrier. One other implementation issue which deserves mention is the implementation of a data oriented synchronization on a machine which has hardware support only for control oriented synchronization.

The Force includes primitive operations for the simplest data oriented synchronization, produce and consume. The HEP hardware supports these operations directly, using the full/empty state bit for each memory cell. In the Flex/32, locks are separate items, not associated with data. To implement producer/consumer synchronization, a boolean data item must be allocated to the full/empty state and a lock must be allocated to bind the data transfer to the state change as an atomic unit. The lock itself cannot be used to model the full/empty state because there is no way to bind it to the data transmission. Furthermore, since the full/empty state is a data item, the system supported process waiting mechanism cannot be used to wait for its change. Critical section code must be repeatedly executed to monitor a change in the state variable. In contrast, it is very easy to model the lock/unlock synchronization using produce/consume. The full/empty state of a memory cell is used for the lock and the value of the cell is simply ignored.

Conclusions

The implementation of a parallel programming environment on two shared memory multiprocessors with quite different architectures has been described. The primitive operations of the system make fairly efficient implementations possible on both machines. One major difference has to do with whether parallelism is supported directly by hardware accessible to the user or is supported only through the operating system. In the latter case, the implementer must work in terms of the software run-time model presented by the system rather than in terms of a model related more directly to the hardware, which makes the prediction and optimization of performance somewhat more difficult. The mechanism by which processes wait at a synchronization is a key issue. If the waiting mechanism is tied to multiprogramming through the operating system call, throughput will be optimized, but a large overhead will be incurred for potentially short synchronization delays.

The use of interrupts in the system architecture leads to natural support for the Event concept. The implementation of Barrier type synchronizations can be tied to the event concept fairly naturally. On machines which do not support events, attention must be paid to minimizing the utilization of resources by waiting processes. The Barrier differs from the critical section in this regard because it is guaranteed that many processes will simultaneously wait at the Barrier while critical section conflict is probabilistic, and the liklihood of many processes waiting at the entry to a critical section is low in a normally constructed program.

REFERENCES

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

[2] H. F. Jordan, "Structuring parallel algorithms in an MIMD, shared memory environment," *Proc. 18th Hawaii Int'nl Conf. on Systems Sciences,* Vol. II, pp. 30-38 (1985); to appear in *Parallel Computing,* 1985.

[3] H. F. Jordan, "HEP architecture, programming and performance," in *Parallel MIMD Computation: The HEP Supercomputer and its Applications,* J. S. Kowalik, Ed., MIT Press (1985).

[4] "The Uniform System Approach to Programming the Butterfly Parallel Processor," Draft of Oct. 23, 1985, Copyright BBN Laboratories Inc. (R. H. Thomas, private communication).

[5] F. Darema-Rogers, D. A. George, V. A. Norton and G. F. Pfister, "A VM Parallel Environment," *Rept. RC11225 (#49161),* IBM T. J. Watson Res. Ctr. (Jan. 1985).

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

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

[8] 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, Ohio, pp. 203-213, July 1985.

[9] N. Patel, private communication.

[10] 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.

[11] *The Flex/32® System Overview,* Flexible Computer Corp., Dallas, Texas, 1986.

[12] Flynn, M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Trans. on Computers,* pp. 948-960 (1972).

Standard Bibliographic Page

1. Report No. NASA CR-178161 ICASE Report No. 86-54	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle THE FORCE ON THE FLEX: GLOBAL PARALLELISM AND PORTABILITY		5. Report Date August 1986
		6. Performing Organization Code
7. Author(s) Harry F. Jordan		8. Performing Organization Report No. 86-54
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No.
		11. Contract or Grant No. NAS1-17070
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code 505-31-83-01

15. Supplementary Notes
Langley Technical Monitor:
J. C. South

Final Report

Additional support provided by AFOSR Grant No. 85-0189.

16. Abstract
A parallel programming methodology, called the force, supports the construction of programs to be executed in parallel by an unspecified, but potentially large, number of processes. The methodology was originally developed on a pipelined, shared memory multiprocessor, the Denelcor HEP, and embodies the primitive operations of the force in a set of macros which expend into multiprocessor Fortran code. A small set of primitives is sufficient to write large parallel programs, and the system has been used to produce 10,000 line programs in computational fluid dynamics. The level of complexity of the force primitives is intermediate. It is high enough to mask detailed architectural differences between multiprocessors but low enough to give the user control over performance.

The system is being ported to a medium scale multiprocessor, the Flex/32, which is a 20 processor system with a mixture of shared and local memory. Memory organization and the type of processor synchronization supported by the hardware on the two machines lead to some differences in efficient implementations of the force primitives, but the user interface remains the same. An initial implementation was done by retargeting the macros to Flexible Computer Corporation's ConCurrent C language. Subsequently, the macros were caused to directly produce the system calls which form the basis for ConCurrent C. The implementation of the Fortran based system is in step with Flexible Computer Corporations's implementation of a Fortran system in the parallel environment.

17. Key Words (Suggested by Authors(s)) multiprocessors, shared-memory, parallel programming	18. Distribution Statement 61 - Computer Programming and Software 62 - Computer Systems Unclassified - Unlimited

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	21	A02

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA Langley Form 63 (June 1985)

End of Document