Parallel Language Constructs for Efficient Parallel Processing

Russell M. Clapp

Sequent Computer Systems
15450 SW Koll Parkway
Beaverton, Oregon 97006-6063
rmc@sequent.com

Trevor N. Mudge

Advanced Computer Architecture Laboratory
The University of Michigan
Ann Arbor, Michigan 48109-2122
tnm@eecs.umich.edu

Abstract

In this paper we propose some basic language extensions to incorporate a parallel procedure model into the C programming language. In order to improve on other proposals, we set the goals of our design to attain increased efficiency, flexibility, and expressiveness, and to improve parallel program structure. We begin by discussing the motivationfor these goals, and then present an overview of our proposed model for parallel procedures. The following section describes the design of the run-time system that supports the parallel procedure model. A novel scheme for nesting parallel procedure contexts in multiple stack frames is included in the run-time system, thus eliminating the need for costly process control blocks. After describing the details of the language and run-time system design, we then present detailed performance data for two parallel programs using this system.

1 Motivation and Goals

The recent emergence of commercial multiprocessors has provoked computer scientists and engineers to take a closer look at parallel programming. However, practical parallel programming systems are in a state of infancy, and their performance leaves considerable room for improvement. Many commercial systems incorporate ad hoc techniques for expressing parallelism, most involving machine dependencies [1, 2, 3, 4]. Programming languages for parallel systems have ranged from adapting sequential languages for parallel execution to the use of explicitly parallel concurrent languages such as Ada. Some success has been achieved in implementing *procedural* or *imperative* languages on shared-memory multiprocessors. However, in many cases, the difficulty of the problem has led to inefficient implementations.

Our approach is intended to offer a middle ground between parallelizing compilers and concurrent languages, while eliminating any ad hoc techniques for run-time support from the language. A major goal in our design, however, is an efficient implementation. An important consequence of this goal is that the language constructs for parallelism must be designed with an eye toward their run-time support requirements. It is critical that parallel language constructs requiring excessive run-time system overhead be omitted from our language proposal. Furthermore, we require the run-time support for these new constructs to be efficient enough to support loop-level parallelism similar to that described in [5, 6, 7]. This last requirement enables us to benefit from an existing and generally accepted language model of parallelism while incorporating a new model capable of greater generality.

The key problem in developing a high performance parallel processing system is the design of efficient run-time support code. This support includes both the operating system (OS) and the language specific run-time system. The fact that low-level and ad hoc techniques have surfaced at the language level is a symptom of the problem that current run-time support software is inadequate. Although gains are being made in OS design for shared-memory multiprocessors [8, 9, 10, 11, 12], there is still no widely accepted standard for OS level support of parallel programs. Furthermore, even with the support for parallelism provided directly by the operating system, there is still a need for language-specific run-time systems to support the execution of parallel programs. Because OS calls are relatively expensive, it is inefficient to invoke the operating system at every opportunity to exploit parallelism in a program (see the performance figures in [12]). Instead, a user-level language run-time system should be called to manage the spawning and merging of program parallelism. Because this system may be invoked with very low overhead, it allows parallelism of both fine and coarse grains to be profitably exploited.

Run-time efficiency is a key factor in the design of a parallel processing system because it is the amount of run-time overhead that dictates the amount of useful program parallelism. In a simplified model where there is a constant overhead involved in initiating a parallel execution thread and a program may be broken up into any number of parallel threads, it has been shown that there is a limit on the number of parallel threads that should be used in order to achieve maximum speedup [6]. This model implies that the *granularity*, the number of instructions executed for a parallel thread, decreases as the program is divided up into more and more threads. If the program is divided to the point where the overhead is equal to the granularity, the program's running time is the same as it is in the uniprocessor case, and any further reduction in the granularity will result in a speedup of less than 1.

This overhead/granularity tradeoff is analyzed by compilers that parallelize at the loop level. Because fine grain loop iterations must be combined in some cases to compensate for overhead, other sources of parallelism within a program are being investigated, so that more processors can be effectively used to increase speedup. Research involving interprocedural data dependency analysis in sequential programs demonstrates a desire to extract more parallelism of a larger granularity [13]. Our approach to parallel programming also addresses this need for larger grain parallelism.

With these points in mind, we can state our goals in developing a parallel language model and run-time system. Our approach centers around providing language-level mechanisms for expressing parallelism. We will use the following words to denote the design goals regarding our parallel programming system:

1. **Efficiency.** The overhead required to support parallel execution threads should be as low as possible. As a result, only a minor penalty should be incurred when the parallel program is executed on a single processor.

2. **Flexibility.** The parallel language constructs should allow expression of varying degrees of granularity from loop-level to procedural-level, which reduces the percentage of execution time spent on overhead by optimizing for specific cases.

3. **Expressiveness.** The language model should allow the straightforward expression of a wide variety of parallel algorithms. An important aspect of this requirement is that a *dynamic* model of parallelism must be supported (this is explained below).

4. **Structure.** In addition to flexibility and expressiveness, the parallel language model should promote the development of well-structured programs. This goal requires that critical sections are centralized and well-identified, so that deadlock among multiple parallel threads is prevented.

By attaining these goals, we will have a parallel programming system that is efficient, easy to use, and widely applicable. The efficiency goal ensures that the system will be useful in both parallel and *multistream* execution modes, where time sharing users are limited to a single processor. High performance of the run-time system will reduce the minimum grain size of parallelism necessary so that system overhead does not dominate execution time. The flexibility goal will allow the programmer to specify parallel code that meets the practical minimum grain size requirement, so that multiple processors may be profitably exploited. Based on our approach to the efficiency goal, we expect the practical minimum grain size to be small (e.g., on the order of 10000 executed assembly language instructions), so that large amounts of parallelism may be exploited at run-time.

The expressiveness goal ensures that a wide variety of applications may be coded in the parallel language. The parallel model is intended to support general purpose parallel programming, which includes scientific computation-intensive applications as well as non-numeric applications typified by sorting, searching, and branch and bound applications. The model is is not intended to provide mechanisms that are useful for programming embedded or distributed systems, or even simulating such systems. It is also not intended for managing physical resources, and so is not useful for programming operating systems. These types of applications are better suited to *concurrent* languages (e.g., Ada, Occam, Concurrent C, Linda, etc.), which allow persistent communicating processes at the expense of run-time system overhead.

On the other hand, the model for parallelism we propose is intended to be more expressive than parallel loops or sections that are based on sequential languages, especially those constructs that may not be nested. To attain this goal, we incorporate a dynamic model of parallelism. Even though parallelizing compilers may use dynamic scheduling techniques to allocate loop iterates to processors, the structure of the program parallelism is typically static, where the main program creates a number of microtasks to execute the iterates, and then resumes control to set up the next parallel loop. In our dynamic model of parallelism, any thread of execution can dynamically create additional threads using an explicit **create** statement. This requires a run-time system capability to queue and schedule multiple heterogeneous threads or *procedure-processes.* Using this model, the bottleneck of the main program executing serially to set up each new homogeneous parallel section is eliminated. Instead, new instances of parallel code are set up and added to the run queue when they are created, and in parallel with other procedure-processes.

The structure goal is intended to encourage an easy to understand, straightforward style of parallel programming, that does not hamper efficiency, flexibility, or expressiveness. If implemented, the benefit of this requirement is that the model used for expressing parallelism prevents deadlock and "distributed critical sections". Deadlock results when a parallel program unit is suspended waiting for an event that will never occur (e.g. the unlocking of a semaphore). Distributed critical sections occur when different programming units manipulate shared resources, each within its own code section. Even though these accesses to shared resources may occur in mutual exclusion, the program is still difficult to understand and debug. These problems with semaphores were identified in [14] and [15], where proposals were made to unify synchronization and communication to aid in centralizing a given critical section. This idea was refined with the *extended rendezvous* concept of Ada, where a critical section can be embedded within a rendezvous statement which is used for synchronization and communication between tasks [16]. We propose a parallel programming model that centralizes access to data that is logically shared by multiple program units as well, but our constructs rely on parameter passing with process creation and termination instead of rendezvous. Our model also prevents deadlock in the scheduling and execution of parallel program units.

2 Overview and Background

Our approach to the development of a parallel programming system begins with the design of a suitable language and its run-time system. In order to focus on the parallel nature of the language, we extend an existing sequential language, namely C. We chose C since it has simple constructs and is easily implemented. Also, from our point of view, C has a desirable property in that functions cannot be nested statically. This is consistent with our model of parallel program units, that are also restricted from being statically nested. Our proposed language constructs can also be added to FORTRAN with only slight modification, as we have already done in order to perform several experiments.

The extensions we propose for parallel programming include the addition of a parallel procedure model to C. Each of these parallel procedures, called *procedure-processes* or, to adopt the terminology of [17], simply *paraprocs*, share parameters with the process that invoked it. Both process synchronization and communication are handled through the **create** and **merge** primitives. Procedure-processes may create other paraprocs, so that a logical tree of procedure-processes can be created that fluctuates dynamically.

This parallel procedure model bears some resemblance to other imperative languages that have been proposed previously for shared-memory multiprocessors. Examples include the extended FORTRAN for the HEP multiprocessor [18], the EPEX C environment proposed in [17], PCF FORTRAN [19], PCP [20], and generic run-time support packages such as the Uniform System [21], the Chameleon System [22], and Fast Threads [12].

The system we propose here is intended to be simpler and therefore more efficiently implemented than any of these other proposals. While several ideas from the other systems have been incorporated, much of the complexity has been stripped away. The mechanisms we propose allow procedure-processes to be created in groups, and restrict synchronization to be performed through paraproc creation and termination only. Our model also allows for parallelism to be specified without the possibility of deadlock, which is an issue not addressed in most other proposals for an imperative parallel procedure based language.

The run-time system for this parallel language is designed to provide maximum performance. In fact, the design of the parallel model of the language has been influenced considerably by run-time system performance considerations. The origin for our run-time system is the *self-scheduling* technique used in parallelizing FORTRAN compilers [18, 6], with our implementation as described in [23, 7]. By building on this basic run-time system, we can support a dynamic model of parallelism where procedure-processes are *created* in addition to being scheduled at run-time. Because parallel work can be created by any active paraproc, the simple self-scheduling technique must be extended to operate with a variable length run queue of available work. Also, due to the semantics of paraproc creation and termination, it is possible for procedure-processes to become suspended while waiting for child paraprocs to complete. This requires a blocking and resumption capability in the run-time system, in order to support the run-until-block semantics.

3 Parallel Procedure Model

The model we propose for specifying program parallelism is based on adding a process model to the C language. This process model has semantics very similar to that of the C function or FORTRAN subroutine. These procedure-processes are quite different from the processes or tasks of concurrent languages in that there are no communication ports, channels, or rendezvous type communication calls. Instead, all explicit communication is through parameters passed to the procedure-process at creation. Parameters are passed to the created paraprocs using reference semantics, so any manipulation of their value is evident to the parent paraproc. There is also a shared global memory space, but it is not protected from simultaneous access by multiple paraprocs. There are no explicit synchronization primitives provided in the language model except for the **create** and **merge** primitives for paraproc creation and termination. Furthermore, there is no data dependency checking between procedure-processes. The programmer ensures that this is not necessary by explicitly stating parallelism with procedure-processes. The rationale for our decisions regarding these language constructs is given in Sec. 4, where the run-time system implementation is described.

3.1 Declaration and Scope

The advantages we see in choosing C as a base language for adding parallel constructs are its rules for function and variable declaration, scope, and visibility. Function declarations cannot be nested in C, and we choose to apply this restriction to procedure-processes as well. Processes are declared in a manner almost identical to C functions (see Sec. 3.3 below for an example). Using semantics similar to C keeps the model both simple and consistent with the C language.

Scope and visibility rules for variables are the same as they are for C [24]. Global variables are declared at the beginning of the main program or are visible in other files using the `extern` declaration. Variables declared within a procedure-process are not visible outside the procedure-process. Variables may be visible to a subset of processes all declared in a single file using the `static` declaration. Accesses to these variables are unprotected as is the case for global variables. Paraprocs themselves may also be declared as `static`, which restricts their visibility to the file where they are declared and prevents paraprocs declared in other files from creating instances of them.

Paraprocs must be reentrant, so that multiple instantiations of them may all execute simultaneously. All variables local to a paraproc are allocated on the stack, so all of them are deallocated when the paraproc completes, just as in the case of a procedure/function return. Static variables are not allowed in paraprocs, they must instead be declared as described above. Besides its own variables, a paraproc can only access parameters, global variables, and visible static variables. Data that is to be shared by subsets of procedure-processes is controlled by the programmer by using parameters or static variables. If the programmer requires that access to shared static or global data must be synchronized, the **create** and **merge** primitives must be used.

3.2 Creation and Termination

Paraprocs are created in groups of 1 or more in a single **create** statement. The paraproc name and parameter list are specified in the **create** statement, and a variable is specified as the destination of a procedure-process group value that is returned. If a parameter is a scalar or structure, it is passed to each paraproc created. If a parameter is an array, the value passed is indexed from the array base using the created child paraproc's instantiation number. This number is unique to each paraproc in the group, and is a value between 1 and N where N is the number of procedure-processes in the group. Each paraproc has a predefined variable **me**, which is the value of its instantiation number.

After paraprocs are created, they execute in parallel with their creator and siblings. Since any procedure-process may create additional paraprocs, the program can be thought of as a "tree" of paraprocs executing in parallel. A procedure-process terminates when it executes a **complete** statement. All paraprocs must **merge** with their child paraprocs before they can execute a **complete** statement. The **merge** statement specifies the procedure-process group value returned by the **create** statement. When all of the child paraprocs in the group terminate, the parent paraproc may continue past the merge point. This is effectively a barrier synchronization, but its scope is limited to the parent paraproc and its children rather than the entire program. A paraproc may create several different procedure-process groups during its lifetime, execute with them in parallel, and merge with them in any order it wishes.

The main thread of execution of the program is the "master" process, and it is the only scheduleable execution thread when the program begins. The program terminates when the master thread completes, and this can only happen after it merges with all of its children.

3.3 Example Syntax

The following program fragments show the proposed syntax for the parallel procedure model. The code is a parallel quicksort program adapted from a sequential version given in [25]. The code assumes two predefined functions, `findpivot` and `partition`, which are used to find the pivot value and partition the global array `A` to be partially sorted around the pivot. These sequential routines can be found in [25]. Because we require that all C functions be reentrant, `findpivot` and `partition` can be both called by many paraprocs simultaneously. This algorithm computes the correct result, because there is no data dependence between paraprocs. Any paraproc sorting the array is working only with its own subrange. This subrange is only shared with its parent paraproc, and the parent stops accessing the array before it creates any children. This is an example of using the **create** statement to synchronize access to shared data.

This algorithm demonstrates the ability of parallel procedures to specify a grain size that is large in comparison to most parallel loop bodies. The calls to the serial routines `findpivot` and `partition` provide a large number of instructions to be executed within a single parallel procedure. Because of the semantics of our language constructs, these function calls in the body of the paraproc do not inhibit parallelization on procedure-process boundaries.

In the example, the new reserved words needed for the parallel extensions are shown in boldface. The first code fragment demonstrates the syntax used for declaring paraprocs (Fig. 1). The keyword **process** denotes that the code is not a C function and instead is to be executed as a parallel thread when created by a corresponding **create** statement. The rest of the code is the same as it would be if the procedure-process was instead a function, except for the **complete** statement, which is used to terminate the paraproc.

The quicksort procedure-process also includes **create** and **merge** statements. The **create** statement is used to create a group of paraprocs using some visible procedure-process declaration. In this case, the visible procedure-process is quicksort itself. The value returned by **create** is a procedure-process group identifier, and is used later as a parameter to the **merge** primitive. The number 2 in brackets in the **create** statement specifies the number of paraprocs of the same type to be created. This value can be replaced by any integer expression. The paraproc name and actual parameters are then specified. In this example, the first created paraproc has access to `iarg[0]` and `jarg[0]`, while the second has access to `iarg[1]` and `jarg[1]`. These array values are assigned from local parameters and variables before the **create** statement. These values correspond to the partitioning of the subrange of the array `A`.

```
/* type declaration for records to be sorted */
typedef struct node {
    char     name[NAMESIZE];
    int      key;
}   RECORD;

RECORD A[N];           /* N is some predefined constant */
                       /* A is array of records */
.   .   .              /* other globals are declared here */

process quicksort(i,j)
int i(*), j(*);  /* declare type of parameters */
{                      /* * indicates array slice */
    int pivot, pivotindex, k;      /* local vars */
    iarg[2], jarg[2];
    pid sorters;           /* group handle declaration */

    /* source code for findpivot is in the Aho ref. */
    pivotindex = findpivot(i,j);
    if(pivotindex != 0)
        {
        pivot = A[pivotindex].key;
        /* source code for partition is in the Aho ref. */
        k = partition(i,j,pivot);
        iarg[0] = i; jarg[0] = k - 1;
        iarg[1] = k; jarg[1] = j;
        sorters = create[2] quicksort(iarg[*],
                  jarg[*]);
        /* creates 2 paraprocs, each gets one value of */
        /* unique index in iarg and jarg */
        merge(sorters);   /* wait for the 2 sorter */
        }                 /* paraprocs to complete */
    complete;             /* paraproc terminates */
}
```

Figure 1: Procedure-process declaration.

In general, each parameter is passed (with reference semantics) to each created procedure-process, except for any parameters that are arrays with a subscript specified as a $*$. In this case, each paraproc is passed a unique entry of the array based on its unique instantiation identifier. The parameters in the paraproc to be created are declared as $*$'ed arrays also, so that the compiler may generate the proper addressing sequence in instances where it has yet to encounter the actual create statement. The array values are taken sequentially, starting with the value at subscript 0, since arrays are zero-based in C. If the array is multi-dimensional, with *'s in multiple dimensions, the indices are assigned with the right-most index varying first (this is consistent with row-major ordering). These rules are significant if the number of procedure-processes created is less than the total number of values in an array parameter. If the number of paraprocs is greater than the number of values corresponding to *'ed dimensions of an array parameter, an erroneous condition occurs. It is possible to generate code to check this condition at run time if it cannot be determined at compile time (e.g., the number of paraprocs to be created is not known at compile time). The issue of checking for erroneous conditions is discussed in [7].

The next line of quicksort shows the syntax used for merging with child paraprocs. This statement acts as a barrier synchronization by suspending the parent until all of the children referenced by the procedure-process group handle are completed. Since the semantics of the parameter passing are by reference, any values returned by the child paraprocs are available in the actual parameters after the **merge** statement. However, in this example, `iarg` and `jarg` are unaffected. The child paraprocs' updates of `A`, though, are recorded in shared memory before completion of the **merge** statement. This updating of shared memory is discussed in Sec 4.4 below.

```
main()
{          /* declare paraproc handle of predefined type pid */
    pid sorter;
    .   .   .       /* code to initialize A goes here */
    sorter = create [1] quicksort(0, N - 1);
    .   .   .  /* execution continues here after create */
    merge(sorters);
}          /* wait for sorter paraproc to complete */
```

Figure 2: Procedure-process creation in main program.

Figure 2 can be thought of as a continuation of Fig. 1, where, after declaration of the quicksort procedure-process, the main program appears. The main program creates 1 quicksort paraproc to sort the entire array. Because the creator does not suspend after the **create**, other statements may be executed before the **merge**, including more **create** statements for other procedure-processes. For example, several sorts on different keys may all proceed in parallel if the array `A` is used as a read-only variable. This would require some minor modifications to quicksort incorporating indirection through pointers to make it more general.

3.4 Structure

The structure of our parallel language encourages a programming style where procedure-processes are used as computational tasks while the parent paraproc coordinates the data and results. The **merge** primitive provides a barrier type synchronization mechanism. Because this technique is somewhat limited, some algorithms may need to create and merge with procedure-process groups several times in order to synchronize access to shared data. Because of the efficient design of our run-time system (described in Sec. 4), we believe this situation is acceptable. Furthermore, as we will show in the next section, our basic parallel extensions prevent a deadlock situation from occurring.

Alternatives to our **create** and **merge** primitives that allow processes to synchronize arbitrarily, e.g., semaphores, allow a coding style which is confusing, contains distributed critical sections, and permits a deadlock situation to occur. Deadlock can also occur if rendezvous style communication is used. Adding rendezvous capabilities also increases process weight by adding communication queues to the process state, which makes run-time system context switching more expensive. However, as stated above, some algorithms are not suitable for our proposed programming model, and they will have to utilize other languages that incorporate alternative synchronization techniques.

4 Run-Time Support

The run-time system is the key component in the implementation of a parallel programming system. It provides the interface between the hardware, operating system, and the model of parallelism at the language level. In order to describe the run-time system, we must make some assumptions about the hardware environment. Our proposal is intended for a system of one or more homogeneous processors, each having direct access to a logically global shared memory space. This memory space may be contained in a central "main memory" unit or spread across several memory units. The memory space may be cached into multiple caches or local memories that may share individual data items. In this case, the caches are assumed to be kept *consistent* [26]. Alternatively, the memory space may be spread across multiple local memories and possibly a central memory, with no two memory units both possessing a single data item. In this case, all processing elements must be able to directly access each local memory (e.g., [27]), or the operating system must create the "illusion" of shared memory (e.g., [28]). The target architecture must also supply a non-interruptible read-modify-write instruction for synchronization. This instruction should be executable by the run-time support in user mode so that overhead is kept to a minimum.

4.1 Overview

The basic structure of the run-time system is based on *microtasking*. Each processor allocated for the execution of the program begins by executing the run-time system scheduling kernel which runs in user mode (the role of the operating system is discussed in Sec. 6 below). The kernel code continuously attempts to obtain work for the processor from a global queue. When a program begins, the main unit begins execution and the run queue is empty. Only when additional procedure-processes are created does work enter the queue. All work present in the run queue is ready for scheduling, there is no need to synchronize between execution of queue entries. When work is obtained from the run queue, it is processed until the run-time system is reentered, either to obtain more work, add work to the queue, or to perform synchronization. When the run-time system is reentered, it is possible that some updating of global run-time system data structures will be performed as a result of the work just completed or the synchronization request. When all work is completed each processor will be busy looping in the kernel attempting to acquire work. When the last thread of execution terminates (the main program unit), an operating system call is made so that those processors may be reclaimed and used for another job.

4.2 Scheduling

As mentioned earlier, this approach for scheduling processors can be described as a self-scheduling style. This term has been used to describe the technique where multiple processors each obtain a unique iteration of a parallel loop they are to execute [18, 6]. The case we describe is similar in that each processor acquires an index and other basic information from the queue that determines which instantiation of which parallel procedure it is to execute. The queue structure enhances the analogy, since one queue entry is made for each *group* of procedure-processes created.

The queue is a linked list of work entry data structures, or *frames*. These frames are similar to the frames used in the Spoc run-time system [29]. However, the frames we use are simpler, and are not variable length execution frames for procedures. Instead, they contain a fixed amount of basic information that is needed to begin a procedure-process. A frame consists of the parallel procedure's starting address, the number of members in the procedure-process group, a pointer to a memory space where parameter pointers reside, and a pointer to the next frame in the work queue. The slot that holds the number of instantiations to be executed also doubles as a synchronization counter. Frames are allocated from an area in memory designated to be the frame pool. Because frames are of a fixed size, their allocation and reclamation can be performed very quickly without any interaction with the operating system.

When a processor schedules a paraproc, an indivisible *fetch&decrement*[1] operation is performed on a global register or well-known memory location that contains the number of procedure-processes yet to be created for the procedure-process group represented by the frame at the head of the run queue. This global value is initialized by reading the count of procedure-processes to be created (which is also the synchronization counter) from the corresponding frame when it is moved to the head of the run queue. The global value is read before the fetch&decrement operation to assure that it is greater than zero. If it is not, the run-queue is empty, and the value is reread in a tight loop until it is greater than zero, indicating that the fetch&decrement can proceed. If the number returned by the decrement is greater than zero, the processor begins execution of the procedure-process indicated by the current frame with the unique instantiation index returned from the fetch&decrement operation (this index is the value of the **me** variable described above). The parameter pointer is used to obtain access to the parameters passed to the procedure-process. The synchronization counter in the frame is decremented when the process executes the **complete** statement.

If the value returned from the fetch&decrement operation is not greater than zero, one of two operations takes place. If the value is negative, the scheduling kernel is reentered to reread the value in a tight loop until it is positive[2]. When the value becomes greater then zero, the fetch&decrement is performed again. If the value returned from the fetch&decrement is equal to zero, the last procedure-process of the current frame has been scheduled and the global run queue pointer to this frame must be updated. The processor that assumes this task must wait until all other processors that scheduled one of the current paraprocs has read the starting address and frame pointer before these run-time system globals can be changed. These values are then updated from the next frame in the queue and the queue head pointer is set to reference this next frame of work. If no new work is available, the processor must wait for a new set of paraprocs to be created by entering the tight loop mentioned above. It may also be possible for the run-time system to call the operating system to relinquish the processor instead of waiting for more work. This can only be done under certain conditions, and is discussed in [7].

4.3 Synchronization

Synchronization among multiple procedure-processes is expressed at the language level using the **create** and **merge** primitives. There are no other synchronization primitives provided, but mutual exclusion is observed by the run-time system when needed to perform its services. Because **create** may specify parameters, communication between parent and child is possible, and the **merge** primitive is used to synchronize access to this data. Communication between sibling paraprocs must be coordinated by the parent, and is done by passing the same parameters to more than one child paraproc. Further possibilities for communication are discussed below, in Sec. 4.4.

Synchronization in the run-time system is performed by directly manipulating the synchronization hardware or using the assembly level synchronization instructions. Mutual exclusion is necessary in the run-time system to protect the integrity of run-time system data structures that are shared by all processors running in the kernel. The synchronization performed in the run-time system allows the high-level synchronization statements of the language to be supported, thus eliminating the need for programmers to use low-level synchronization routines such as semaphores.

The barrier style synchronization of the **merge** statement is supported by decrementing the synchronization counter in a frame on behalf of a completing procedure-process. The merging parent checks this value to see if it is zero. If it is not, that paraproc must block, and a new one is scheduled from the run queue with control being transferred in a manner similar to a procedure call. When that procedure-process completes, the synchronization counter for the blocked parent paraproc is checked, and if it is still not zero, another paraproc is scheduled on that processor. If it is zero, control is returned to the parent paraproc, in a manner similar to a procedure return.

Because of the simple structure of synchronization in the language, it is not possible for 1) a blocked parent paraproc to be waiting for the completion of a child paraproc *and* 2) all other processors are trying to schedule more work *and* 3) no new paraprocs are available to be scheduled. This

[1] If the hardware does not support fetch&op, the operation is performed non-atomically using the provided synchronization primitive to ensure mutual exclusion.

[2] In some systems (e.g., the Astronautics ZS series), the processor may idle until the global register becomes positive. This avoids busy waiting. Also, the processor may check its stack to see if it has a blocked paraproc that may be resumed. This is discussed further below.

deadlock condition cannot occur because conditions 2) and 3) together imply that any children created by the parent in condition 1) must have completed, and the parent may resume execution. The only way that child paraprocs may block and in turn block their parent is if they themselves have created more paraprocs[3] This, however, violates conditions 2) and/or 3) for deadlock. There is no other way that paraprocs can block waiting for other events, since there are no other synchronization primitives.

The only possibility for deadlock within the run-time system is when the frame pool becomes exhausted. If the pool cannot be enlarged dynamically, then the program must abort. However, a simple operating system call should trivially accomplish this task. Any other chance for deadlock to occur comes from the actions of the operating system. This issue is discussed below in [7].

4.4 Communication

As stated above, communication between paraprocs is primarily through parameters that are passed at paraproc creation. Data can also be shared through the use of global variables. Programmers use **create** and **merge** to synchronize access to shared data as stated above. A benefit of this approach is that a model of *weak ordering* [30] or even *release consistency* [31] is provided at the language level. As a result, the compiler may generate code that stores data which is potentially sharable by multiple procedure-processes in registers, as long as shared memory is updated by the procedure-process at each **create** and **merge** statement it encounters. Sharing data through parameters also allows parent paraprocs to shared different variables with different child paraprocs. For example, the syntax described above demonstrated how to split up the elements of an array over a group of child paraprocs.

Parameters are passed with reference semantics to improve efficiency. Since we assume a shared memory space, reference parameters are more efficient than making unnecessary copies of variables. This approach also relieves the programmer from passing pointers to objects that will be modified, as must be done when parameters are passed by value. Passing pointers with value parameters would also complicate the syntax proposed in Sec. 3 above.

In addition to reducing the number of variable copies, another aspect of the run-time system that provides for increased efficiency is the fixed size of frames. This is done to make frame allocation and initialization very fast. It requires, though, that a set of parameters be passed via a single address. This address refers to an area where the addresses of all the parameters reside. In order to implement this approach, there must be a quick way to allocate local memory for a procedure-process.

4.5 Stack Management

In order to provide this local memory, we assume that each processor has a chunk of sequential memory locations that it can efficiently manage as a local stack. This stack is used in the classic way for procedure (C function) calls and operating system calls. For parallel procedure-processes, it is used for parameter passing with a pointer to the parameter space placed in the corresponding frame.

[3] Any created paraproc is ready for execution due to the dynamic nature of the run queue. That is, there is NO work placed in the run queue unless it is ready for execution.

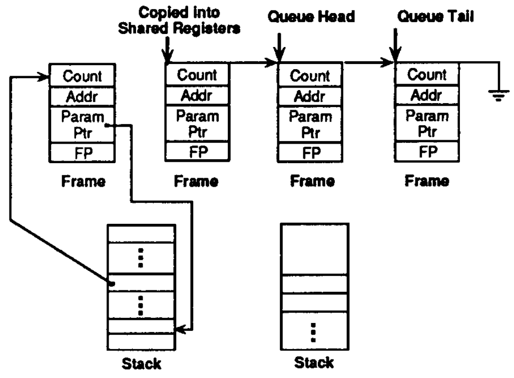

Figure 3: Processor stacks and frames in run queue.

Also, the stack is used for saving contexts of merging paraprocs and accessing synchronization counters at scheduling points. Thus, the stack is shared concurrently among all paraprocs scheduled on a particular processor that have yet to complete. Only one paraproc at a time is active on any processor, so the sharing of the stack is very similar to the sharing that goes on between the caller and callee of a function.

When a procedure-process begins execution, it allocates space for its local variables using the stack in the usual way, but it does not save the current register set or any subset thereof. The address in the frame that refers to the parameter space is available to the paraproc, and several strategies are possible for using this address to refer to the parameters. These techniques may involve use of the stack. When a new group of paraprocs is to be formed, the stack is used to allocate a parameter space area and an address referring to this area is placed in the new frame. After paraproc creation, the creator continues execution, and the stack may continue to grow in size. When a paraproc terminates, it returns the stack top pointer to reflect the top of the stack when the paraproc began execution. Figure 3 shows a logical organization of stacks and frames in the run queue.

If a paraproc executes a **merge** statement and the child paraprocs have not all yet terminated, the parent paraproc must block. Any registers that must be saved by the parent paraproc are pushed on the stack in addition to a return address, as is similar to the case of a procedure call. Also, the last item to be pushed on the stack is the address of the synchronization counter that the blocking paraproc is waiting on. The processor then jumps to the run-time system's self-scheduling code to obtain more work.

This approach relieves paraprocs from saving registers on the stack before they may begin execution. Since a blocking paraproc saves its own context[4] on the stack, it need only save the registers that it will use after the merge point. In the programs we have implemented with this technique, the amount of registers to be saved at a merge

[4] Essentially its stack space and general purpose registers, not special purpose registers or other information managed at the operating system level.

point is usually less than 4, as opposed to saving all general purpose registers upon entry. Furthermore, this technique enables a paraproc context to be transferred to another processor (or virtual processor) via a stack pointer. Although this type of migration has not yet been implemented, its usefulness and exact implementation are discussed in [7].

When a paraproc completes, it returns the stack to its original location as described above. A check is then made to see if the top of the stack is a valid pointer to a synchronization counter that is now equal to zero. If it does, the counter reference is popped and a procedure-style return is executed so that the blocked parent may resume. If the synchronization value is non-zero, a jump to the run-time system's self-scheduling code is executed instead.

Because the stack is used as a link to parameters passed to child paraprocs, the parent paraproc is required to merge with its children before completing. This is to prevent the stack top from being returned to a point that deallocates the parameter space. We believe this a not a serious restriction, since it is likely for parent paraprocs to require synchronization with their children so that shared data can then be manipulated safely. This restriction also permits the use of fixed length frames, since the stack is used for the variable length portion of the logical execution frame.

While our system for suspending and resuming a merging procedure-process may seem quite unconventional, we believe that this technique saves us considerable execution overhead. An alternative scheme would be to save the entire state of the suspending process in a process control block (PCB), and link all such blocks in a queue of blocked processes. This approach would increase the overhead of scheduling, since a check of the suspended process queue would then be necessary. We believe the procedure call and return technique to be more efficient, as calling and returning sequences for procedure calls are well understood, and can be implemented with only a few instructions. This approach also allows local stack areas to be used for process stacks, eliminating the need to reload stack pointers from and allocate stack areas in PCB's.

5 Experimental Implementation

5.1 The ZS Simulator Testbed

For maximum efficiency, we have developed the run-time system code in assembly language, and added it to the sequential assembly code generated by both FORTRAN and C compilers. Our initial target is the Astronautics ZS multiprocessor series [32], an architecture containing high speed decoupled-access-execute (DAE) processors [33, 34]. The machine has been designed to be configured with as many as 16 processors, each possessing its own 128K bytes of local memory and connected to an 8-way interleaved main memory of 128M bytes using a crossbar interconnect. Cache consistency is not provided in hardware, but this feature was added to the simulator testbed described below. One feature of this architecture which we have found beneficial is the set of shared "semaphore" registers which support the fetch&op synchronization primitive. These registers hold the run-time system globals and provide for fast synchronization.

Because the ZS-1 system is configured with only 1 CPU, we have used it to develop and debug our code for a single processor. For larger configurations, we have used an execution-driven register-transfer level simulator which uses ZS executable binaries as input. In uniprocessor tests, we have found the simulator to accurately reflect the speed of the hardware.

The use of the simulator also allowed us to effectively change the architecture, by adding a cache consistency scheme. A full-map central directory-based hardware cache consistency scheme was implemented based on the technique described in [35]. This technique allows multiple caches to hold the same block until a write occurs, invalidating all other copies. We refer to this technique as "shared" hardware consistency. With the modified simulator, we were able to turn hardware cache consistency off, which we refer to as "assumed" consistency, and thus determine an amount of overhead for keeping caches consistent. Other cache consistency techniques were also tested, and full details of the experimental testbed can be found in [7].

The implementation of parallel procedure run-time support for the ZS architecture was straightforward. The frames allocated and initialized as part of paraproc creation were assembled into a linked list as the run queue. The head frame of the run queue was placed in the semaphore registers, so that scheduling could be performed quickly without the need for chasing pointers. Only one processor was needed to update the list and copy the head frame into the semaphore registers once all paraprocs of the current frame had been scheduled. The assembly code for frame allocation and initialization, scheduling and run queue management, and paraproc blocking and resumption can be found in [7].

5.2 Source Programs

TRFD: The first program used in our tests to measure performance of our run-time system is based on the TRFD code from the Perfect Club Benchmark Suite. This program is a "kernel" of the calculation used in a two-electron integral transformation [36]. The code is based on a restructured mathematical equation which requires on the order of n^5 floating point operations where n is the number of basis functions used in the transformation. The code for the TRFD program used in our experiments appears in [7], and its structure is summarized by Fig. 4. The parallelization was detected by hand at the loop level, and was based primarily on the findings reported in [37]. There are 5 parallel loops in total, two in the `INTGRL` subroutine and 3 in the `OLDA` subroutine, each being a `doall` loop, where loop iterations may execute in any order and in parallel. The mainline of the code is a `DO` loop which repeats the entire calculation (one call each to `INTGRL` and `OLDA`) for different values of n. It has been modified to remove statements associated with timing, computing FLOP rates, and printing the results. However, our uniprocessor tests using the parallelized code did include these other statements, in order to help verify the code's correctness.

Performance of the parallel loop version of TRFD is reported in [7]. However, there are two major differences in the parallel procedure version which significantly impact performance. First, each parallel loop is treated as if it were a parallel procedure. This transformation is one that could be performed by a compiler to make parallel loop code suitable for our parallel procedure run-time system. However, some minor code changes are necessary at the assembly language level in order to use the paraproc run-time system. Instead of passing values through semaphore

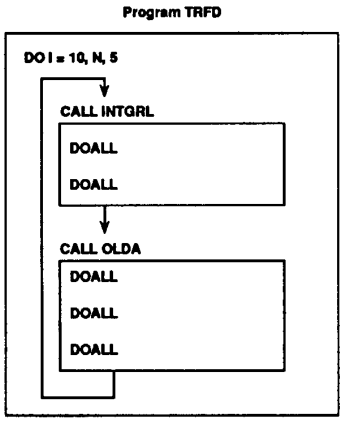

Figure 4: Structure of TRFD code.

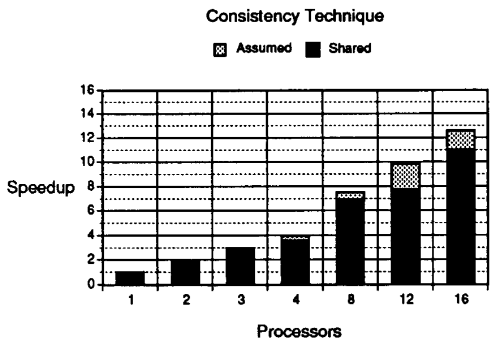

Figure 5: TRFD speedups for $n = 10$.

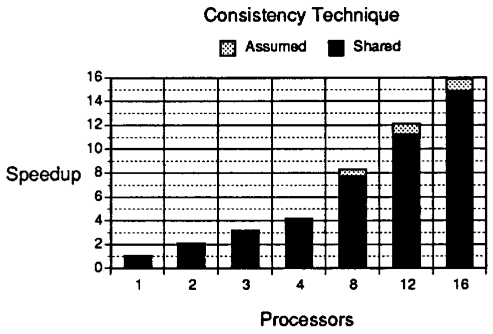

Figure 6: TRFD speedups for $n = 15$.

registers to parallel code loop bodies as done in the code of [7], these values are passed as parameters using stack space and the frame structure described above. Also, each paraproc body is passed the frame pointer of the parent code, so that read only stack variables may be shared. This last change may be thought of as an enhancement that enables a reduction in parameter space for paraproc calls when values are used in a read-only fashion.

The second major change for the paraproc version of the code is the inclusion of nested parallelism. In order to limit overhead, the baseline self-scheduling kernel used to support parallel loops in [7] does not support nested parallel constructs. The inclusion of nested parallelism for the paraproc run-time system allowed parallelizing of the outermost loop of the program and the first nested loop in the final parallel loop of the OLDA subroutine. The TRFD program overlays the memory space for several arrays in one large common block. Because the outermost loop of the program repeats the entire calculation for different values of n, the memory space in this common block had to be expanded to avoid unnecessary data dependencies between the calculations associated with different values of n. This required an expansion from 16.8M bytes of the parallel loop version to 36M bytes with the procedure version that can simultaneously accommodate values of n equal to 10, 15, 20, 25, 30 35, and 40. However, in our simulations with $n = 10$ and $n = 15$, only 4M bytes were needed.

The addition of the nested parallel loop in the OLDA subroutine increased the number of microtasks for this last loop from n to $n * (n + 1)/2$. The parallelization of the outermost loop of the entire program did not increase parallelism for the test where $n = 10$. However, when $n = 15$, the program also repeats the calculation for $n = 10$. In this case, the entire calculations involving all parallel loops may execute simultaneously for both $n = 10$ and $n = 15$. The code for this version of TRFD can be found in [7].

Quicksort: The parallel procedure code for the quicksort program is based on the recursive sequential code found in [25] and used above to demonstrate the syntax and semantics of our language extensions. For our experiments, though, the array of records to be sorted was made up of floating point numbers which were also used as keys. The program was also parallelized at the assembly language level after compiling sequential portions written in FORTRAN. The FORTRAN version of the parallel procedure quicksort program can be found in [7].

5.3 Performance

TRFD: Figures 5 and 6 show the speedups for the parallel procedure version of TRFD. Each bar in these graphs is made up of overlays of bars for each cache consistency technique. That is, for example, the area shaded for assumed consistency shows the additional performance gain over the central directory-based hardware consistency.

For $n = 10$ with assumed cache consistency, we see that speedup is linear for up to four processors, and shows an efficiency rating, $E = S/P$ where S is speedup and P is processors, of greater than 80% for 16 processors. When cache consistency is introduced, the efficiency rating drops to 69% for 16 processors, but speedup is still nearly linear for 1 to 4 processors. When comparing this data to the results for the parallel loop version reported in [7], a performance gain of 20% to 25% is observed. Thus, the addition of nested parallelism in the last loop of the OLDA subroutine offset any increase in run-time system overhead. However, the limiting factors in performance in this case are still load balancing and the overhead for run-time scheduling and synchronization. These factors account for the 20% loss of efficiency for 16 processors

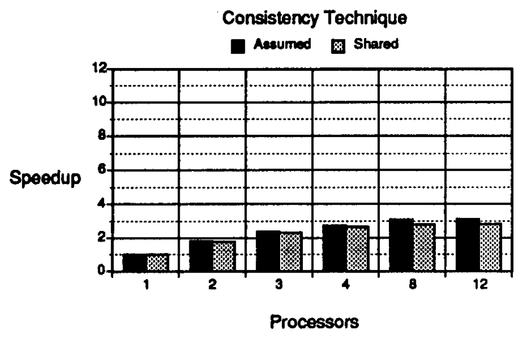

Figure 7: Quicksort speedups for $n = 6400$.

while cache consistency accounted for 11%.

For $n = 15$, the speedups are up to 25% better for both assumed and shared hardware consistency. However, in this case, the speedup is linear for assumed consistency and nearly linear for shared hardware consistency even for 16 processors. The decrease in efficiency for shared hardware consistency is less than 10% for 16 processors. These results show that the slowdown due to load balancing and overhead for run-time scheduling and synchronization has been overcome by the increased parallelism brought about by making the outermost loop of the program into two paraprocs. This demonstrates that, with sufficient parallelism, cache consistency overhead is the major barrier to achieving linear speedup. However, even the overhead for cache consistency was reduced for $n = 15$, due to increased grain sizes for some paraprocs represented by the `doall` loops in `OLDA`. A more detailed account of the performance of this program using the different run-time systems and cache consistency protocols can be found in [7].

Quicksort: Figure 7 shows the speedup for the parallel quicksort program for 6400 floating point numbers. Similar results were also observed for 1000 and 50,000 numbers. While the speedups for greater numbers of processors is somewhat disappointing for this program, it is important to note that the speedups are at least respectable for 2, 3, and 4 processors.

The overall limit to speedup for this program is due to the nature of its parallelization. The `findpivot` and `partition` routines consume a large percentage of the execution time for each paraproc. Also, the running time for these routines is dependent on the size of the interval they are working on. At the beginning of the program, before the first recursion, there is only one paraproc executing and the interval to be processed is the entire array `A`. After the first recursion, there are only 2 paraprocs executing, each with an interval roughly one-half the size of `A`. Clearly, there is a large portion of execution time spent with less than 4 scheduleable paraprocs. Also, as parallelism increases, grain size decreases.

In addition to the reasonable speedups for 2, 3, and 4 processors, there is further evidence that the run-time support code is not the limiting factor in speedup. In an attempt to compare our paraproc quicksort program to "the best sequential algorithm" in order to compute speedup, we tested a sequential FORTRAN version taken from the Numerical Recipes [38]. This program contained loops and `GO TO` statements together with a stack-like data structure to implement recursion at the FORTRAN source level. This code structure inhibited loop-level parallelization due to data dependencies. Also, in an attempt to avoid a slowdown with quicksort in the case where the data set is already sorted [25], a random number generator was incorporated to aid in partitioning the intervals.

In our tests with this program, it ran between 38% and 76% slower than our paraproc version on 1 processor. This indicates that the "retrofit" of recursion at the source level combined with the random number generator required more overhead than our paraproc run-time system and the procedure calls to `findpivot` and `partition`.

6 System Issues

For all of our performance tests described above, there are no operating system calls, I/O statements, or OS-level context switching. As a result, virtual processors are in effect physical processors, and the program's allocation does not vary at run-time. This is not unreasonable for these tests, since all program running times are less than 1 second, a reasonable amount of time observed between interrupts for an OS-level process on a shared-memory multiprocessor[5]. However, in order to make our system feasible in a realistic setting, it must handle OS interaction and be able to vary processor allocation dynamically. For example, the parallel quicksort program is one which could benefit from acquiring additional processors only after several recursions. The following paragraphs briefly describe the issues involved in merging our run-time system with a compiler and multiprocessor operating system. As always, the goal is to provide additional capabilities without unnecessarily increasing overhead.

The major compilation issues involve the incorporation of a loop-based model of parallelism into the language, the development of strict guidelines for code generation, and identification of areas for optimization. A loop-based model of parallelism should be added, as mentioned above, to allow our system to benefit from work done in detection and restructuring of parallel loops [5, 39, 6]. This model can be supplied either with compiler-automated detection or an explicit set of language constructs such as those found in [40, 19, 17]. Strict code generation guidelines are necessary to enforce conventions required for the parallel object code. For example, all procedures, parallel and sequential, must be reentrant. Finally, optimization effort should be spent to improve the performance of parameter passing. Implementation dependencies such as shared registers or register windows may aid in this effort. A more detailed description of each of these issues can be found in [7].

The role of the operating system in our parallel processing system is simple and straightforward. We still require the OS to support I/O requests, shared virtual memory, and library routines (e.g. timing support). The concepts of time and process priorities are omitted from our proposed lan-

[5]Although an interrupt may occur as often as once every millisecond to check for scheduleable OS-level processes, these interrupts may only be felt by one processor, possibly idle and not allocated for use by a parallel program. Hence, an observed time between interrupts of 1 second is not unreasonable.

guage, because they complicate the implementation. The major areas for defining an OS interface to our run-time system are virtual processor support, I/O, and scheduling. Our run-time system is best supported by an OS that provides multiple OS-level processes or threads as virtual processors. The number of virtual processors should be less than the number of physical processors, and this number can vary over the life of the program. I/O support is also required from the OS, but language-level synchronization by the programmer should be used to prevent chaos with shared file descriptors.

Scheduling needs to be reconciled with the run-time system for asynchronous system calls and interrupts. An asynchronous system call returns before the request is serviced, so that work may be done in the interim. This could be treated as a merge point by the paraproc run-time system. Interrupts can seriously disrupt the progress of a parallel program when synchronization is required. In order to avoid a situation where many threads are waiting for a blocked thread, some researchers have proposed *gang scheduling* [41,42] or *coscheduling* [10], where an interrupt to any virtual processor stops the entire program. However, we believe that this approach can be avoided if virtual processors are preempted only at a merge point. This avoids blocking a synchronization operation, and enables a stack's context of blocked paraprocs to be managed in a queue of stack pointers by the run-time system for later rescheduling. This approach could be supported by an interrupt being posted to the entire program, with one virtual processor responding at the next merge point, which is reasonable in a multiprocessor time-shared system. A complete discussion of each of these issues and their impact on deadlock can be found in [7].

7 Summary and Future Directions

In this paper, we have proposed parallel procedure language extensions and described a run-time system to support them. These parallel processing extensions have been designed to meet the goals stated in Sec. 1. The descriptions of the dynamic model of parallelism, its simplicity, its ability to support nested **create** statements and recursion, and its relation to parallel loops are intended to demonstrate the flexibility and expressiveness of the proposed parallel programming system. The structure promoted by the parallel programming extensions is discussed in terms of centralizing access to shared data and preventing deadlock among parallel procedures. We believe that the goals of flexibility, expressiveness, and structure have been met, but only repeated usage of a completed system can determine this for sure.

Our results from experimentation with the paraproc runtime system with the TRFD program suggest that we have met our goal for efficiency as well. In the tests with $n = 15$, cache consistency overhead was on the only barrier preventing linear speedup, as the results with assumed consistency demonstrate. Additional work described in [7] confirms that the cache consistency overhead is due to program references to shared cache blocks, and not run-time system references, which are mostly shared through the semaphore registers in the ZS implementation.

We also demonstrated speedup potential for recursive algorithms using our language extensions and run-time system. However, the experiments with the parallel quicksort program also exposed the limits of speedup due to the nature of the parallel algorithm itself. Although enhancements to the run-time system and its operating system interface may overcome this difficulty, it is also possible to improve speedup efficiency by taking advantage of the language features to specify more parallelism, including heterogeneous parallelism. For example, the quicksort program could be modified to sort using pointers to records with multiple fields and keys. Then, several sorts on different keys could proceed in parallel. Also, since paraproc creation does not block the parent until a merge point, other paraprocs could be created to perform other operations on the same shared data. For example, a set of student records could be processed to determine a curve for test scores in parallel with sorts based on student names and identification numbers.

The next step of this work is to extend the run-time system to account for the interface to a compiler and multiprocessor operating system as introduced in Sec. 6 above. This will enable us to perform additional experimentation with parallel programs executing in a true multiuser, time-shared environment. This experimentation, coupled with an effort to code more algorithms using our proposed language constructs, will enable a complete evaluation of the usefulness of our proposed shared-memory parallel programming system.

References

[1] A. H. Karp, "Programming for parallelism," *IEEE Computer*, pp. 43–57, May 1987.

[2] R. G. Babb II, ed., *Programming Parallel Processors*, Addison-Wesley, Reading, MA, 1988.

[3] A. H. Karp and R. G. Babb, "A comparison of 12 parallel FORTRAN dialects," *IEEE Software*, pp. 52–67, Sept. 1988.

[4] M. Kallstrom and S. S. Thakkar, "Programming three parallel computers," *IEEE Software*, pp. 11–22, Jan. 1988.

[5] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1982.

[6] C. D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, Norwell, MA, 1988.

[7] R. M. Clapp, *Run-Time Support for Parallel Programs*, PhD thesis, Department of Electrical Engineering and Computer Science, The University of Michigan, 1991.

[8] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for Unix development," in *Proc. of the USENIX 1986 Summer Technical Conference*, pp. 193–210, June 1986.

[9] J. Edler, J. Lipkis, and E. Schonberg, "Process management for highly parallel Unix systems," in *Proc. of the USENIX Workshop on Unix and Supercomputers*, 1988.

[10] B. Beck and D. Olien, "A parallel-programming process model," *IEEE Software*, pp. 63–72, May 1989.

[11] M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "Design rationale for Psyche, a general-purpose multiprocessor operating system," in *Proc. of the 1988 Int. Conf. on Parallel Processing*, pp. 255–262, Aug. 1988.

[12] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," Technical Report 90-04-02, The Department of Computer Science and Engineering, University of Washington, Oct. 1990.

[13] Z. Li and P.-C. Yew, "Efficient interprocedural analysis for program parallelization and restructuring," in *Proc. of Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS 1988)*, pp. 85–97, July 1988.

[14] C. A. R. Hoare, "Communicating sequential processes," *Comm. of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[15] P. Brinch Hansen, "Distributed processes: A concurrent programming concept," *Comm. of the ACM*, vol. 21, no. 11, pp. 934–941, Nov. 1978.

[16] J. D. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Breckner, O. Roubine, and B. A. Wichmann, "Rationale for the design of the Ada programming language," *ACM SIGPLAN Notices*, vol. 14, no. 6, , June 1979.

[17] A. Norton and W. L. Chang, "Self-scheduling in the runtime environment," Technical Report RC 12572 (#56256), IBM T. J. Watson Research Center, Yorktown Heights, NY, Feb. 1987.

[18] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system," in *Real-Time Processing IV, Proc. of SPIE*, pp. 241–248, 1981.

[19] B. Leasure et al., "PCF FORTRAN: Language definition," Technical Report Version 1, The Parallel Computing Forum, Aug. 1988.

[20] E. D. Brooks III, "PCP: A parallel extension of C that is 99% fat free," Technical report, Lawrence Livermore National Laboratory, 1988.

[21] R. H. Thomas and W. Crowther, "The Uniform System: An approach to runtime support for large scale shared memory parallel processors," in *Proc. of the 1988 Int. Conf. on Parallel Processing*, pp. 245–254, Aug. 1988.

[22] G. A. Alverson, *Abstractions for Effectively Portable Shared Memory Parallel Programs*, PhD thesis, University of Washington, 1990. Department of Computer Science and Engineering.

[23] R. M. Clapp, T. N. Mudge, and J. E. Smith, "Performance of parallel loops using alternative cache consistency protocols on a non-bus multiprocessor," in *Proc. of the Cache and Interconnect Workshop, 16th Int. Symp. on Computer Architecture*, pp. 131–152, Kluwer Academic Publishers, Norwell, MA, 1989.

[24] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[25] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[26] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, Sept. 1982.

[27] R. Rettberg and R. Thomas, "Contention is no obstacle to shared-memory multiprocessing," *Comm. of the ACM*, vol. 29, no. 12, pp. 1202–1212, Dec. 1986.

[28] M. Beltrameti, K. Bobey, and J. R. Zorbas, "The control mechanism for the Myrias parallel computer system," *ACM Computer Architecture News*, Aug. 1988.

[29] A. J. Musciano and T. L. Sterling, "Efficient dynamic scheduling of medium-grained tasks for general purpose parallel processing," in *Proc. of the 1988 Int. Conf. on Parallel Processing*, pp. 166–175, Aug. 1988.

[30] S. V. Adve and M. Hill, "Weak ordering - a new definition," in *Proc. of the 17th Int. Symp. on Computer Architecture*, pp. 2–11, June 1990.

[31] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proc. of the 17th Int. Symp. on Computer Architecture*, pp. 15–26, May 1990.

[32] J. E. Smith et al., "The ZS-1 central processor," in *Proc. of the Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pp. 199–204, Oct. 1987.

[33] Astronautics Corporation of America, *ZS Central Processor – Architecture Reference Manual*, Madison, WI, 1988.

[34] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Trans. on Computer Systems*, vol. 2, no. 4, pp. 289–308, Nov. 1984.

[35] L. M. Censier and P. Feautrier, "A new solution to coherence," *IEEE Trans. on Computers*, vol. C-27, no. 12, pp. 1112–1118, Dec. 1978.

[36] "Perfect Club Benchmark Suite 0 documentation," Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Mar. 1989.

[37] C. M. Grassl and J. L. Schwarzmeier, "Performance of applications programs on supercomputers: Results from the Perfect Benchmarks," Technical report, Cray Research, Inc., Mendota Heights, MN, Apr. 1990.

[38] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes*, Cambridge University Press, Cambridge, England, 1986.

[39] M. Wolfe, "Multiprocessor synchronization for concurrent loops," *IEEE Software*, pp. 34–42, Jan. 1988.

[40] M. D. Guzzi, D. A. Padua, J. P. Hoeflinger, and D. H. Lawrie, "Cedar FORTRAN and other vector and parallel FORTRAN dialects," in *Proc. of Supercomputing '88*, pp. 114–121, Nov. 1988.

[41] D. G. Feitelson and L. Rudolph, "Distributed hierarchical control for parallel processing," *IEEE Computer*, pp. 65–77, May 1990.

[42] D. L. Black, "Scheduling support for concurrency and parallelism in the Mach operating system," *IEEE Computer*, pp. 35–43, May 1990.