Parallel Language Constructs for Efficient Parallel Processing

By

R.M. Clapp and T.N. Mudge

CSE-TR-66-90

THE UNIVERSITY OF MICHIGAN

COMPUTER SCIENCE AND ENGINEERING DIVISION
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
Room 3402 EECS Building
Ann Arbor, Michigan 48109-2122
USA

Parallel Language Constructs for Efficient Parallel Processing

Russell M. Clapp and Trevor N. Mudge

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

June, 1990

Abstract

For many applications, commercial multiprocessors are emerging as lower cost alternatives to mainframes and supercomputers. However, in order to outperform high speed uniprocessor machines with MIMD parallel processing, effective parallel programming environments are needed. While many such environments have been proposed for shared-memory multiprocessor systems, most can be considered to be in a state of infancy. The approaches used for these parallel programming systems has spanned the range from automatically parallelized sequential programs to concurrent languages such as Ada. In this paper, we propose an alternative approach, defining 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 motivation for these goals, and then present an overview of our proposed model for parallel procedures, or *paraprocs*. We then proceed with detailed descriptions of the language construct's syntax and semantics, and discuss their use in a sample program. The following section describes the design of the run-time system that supports the parallel procedure model. The run-time design discussion explains the motivation for the language model design decisions, as well as providing details about the possibilities for optimization in code generation, the role of the operating system, and the transparent support in the run-time system for synchronization and communication. The final section highlights the design goals and describes our initial experimentation with 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 is generally poor. 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 the 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.

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 [5, 6, 7, 8], 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. 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 since 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 [9]. This model implies that the granularity of 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 [10]. Our approach to parallel programming also addresses this need for larger grain parallelism.

An efficient implementation of a parallel language should also allow parallel programs to execute on a single processor without a significant penalty of excess run-time overhead. In addition to enhancing portability and allowing development on single processor systems, this ability also provides reasonable efficiency when the program is run on one processor in a multiprocessor system. While this may not seem significant at first, it is important to note that most shared memory multiprocessors used in a commercial setting today are multiprogrammed as time-shared systems that allocate one processor per user program. This mode of operation has been termed as *multistream* execution [11]. In a situation where there are more parallel jobs than there are processors in the system, it may be desirable to restrict all jobs to a single processor. In this case, the speed of the program when run on a single processor must not be compromised by parallelization.

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 procedures should be as low as possible, with a minimum penalty 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. This allows larger grain parallelism to be specified, which reduces the percentage of execution time spent on overhead.

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.

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 and 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. 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 1000 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 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 to be used, without hampering efficiency, flexibility, or expressiveness. The benefit of this requirement is that the model used for expressing parallelism is made to prevent 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. This conclusion was also reached in [12], and is one reason why semaphores are not directly supported in the Ada language. We propose a parallel programming model that centralizes access to data that is logically shared by multiple program units, as well as preventing 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.

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 simply *paraprocs*, share parameters with the procedure-process that invoked it. Both paraproc 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 is created that fluctuates dynamically.

This parallel procedure model bears some resemblance to others that have been proposed previously. One such approach was the extended FORTRAN used for the HEP multiprocessor [13]. This system allowed individual procedures to be created and executed in parallel. Synchronization was performed using a "full/empty" bit for each datum. A read of an "empty" variable was suspended until a producer filled it.

Another system that includes parallel procedures is the EPEX C environment described in [14]. The model proposed there is similar to ours and is based on many of the same goals. However, the syntax and semantics of the parallel procedures are very different. The techniques for synchronization and memory allocation are also different. Some of these differences can be attributed to the target architecture, the IBM RP3, which supports both local and global memory areas. The EPEX C system also provides language constructs for parallel loops and parallel "sections". These constructs have their run-time support based on the support for parallel procedures. We believe that parallel loops may be supported more efficiently than parallel procedures, and we base the run-time support for parallel procedures on that for parallel loops. These techniques are discussed in Sec. 4 below.

Other run-time system packages have been developed that enable parts of a sequential language program to be executed in parallel. These "system" calls are usually language independent, and require explicit calls to be inserted into the original program (e.g., The Uniform System [15]). This approach

is similar to providing support for parallelism at the operating system level, which we believe requires excessive overhead. The generality of these run-time systems also prevents many assumptions about program structure to be made, which in turn prevents speed enhancing optimizations from being made. An overview of these run-time environments is given in [3].

Another parallel language proposal that we have made uses the parallel procedure idea for distributed-memory multiprocessors (DMMs) [16]. However, in that system, additional constructs are provided in the language as an abstraction of the message passing hardware and operating system. We consider the procedures in this case to be more like "full-fledged" processes, since they have the capability to send and receive messages with other processes. Also, as a performance consideration, the location of processes within the DMM can be controlled at the language level.

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.

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 baseline for our run-time system is the *self-scheduling* technique used in parallelizing FORTRAN compilers [13, 9]. 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 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.

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

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 [17]. 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 paraprocs 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[1]. 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 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 it applies to the parent paraproc and its children only, and not 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 [18]. The code assumes two predefined functions, **findpivot** and **partition**, which are used to find the pivot value and

[1] As an optimization, it may be possible to allocate locals in registers. This is discussed in Sec. 4.6 below.

partition the global array **A** to be partially sorted around the pivot. These sequential routines can be found in [18]. 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**.

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 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 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 paraproc's

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

RECORD A[N];                /* N is some predefined constant, A is array of records */
. . .                       /* other global are declared here */
process quicksort(i,j)
int i, j;                           /* declare type of parameters*/
{
    int pivot, pivotindex, k;       /* local variable declaration */
    iarg[2], jarg[2];               /* variable declaration for child paraproc parameters */
    pid sorters;                    /* variable declaration for paraproc group handle */

    pivotindex = findpivot(i,j);
    if(pivotindex != 0)
        {
        pivot = A[pivotindex].key;
        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: Paraproc declaration.

```
main()
{
    pid sorter;                    /* declare paraproc handle of predefined type pid */
    . . .                          /* 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: Paraproc creation in main program.

updates of **A**, though, are recorded at the merge point.

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 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. However, due to the efficient design of our run-time system (described in Sec. 4), we believe this situation is acceptable as long as run-time overhead is kept to a minimum. 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 System

The run-time system is the key component in the implementation of a parallel programming system. It provides the interface between the hardware, the 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* [19]. Alternatively, the memory space may be spread across multiple local memories and 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., [20]), or the operating system must create the "illusion" of shared memory (e.g., [21]). 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. 4.7 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 [13, 9]. 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 [22]. However, the frames we use are simpler, and are not 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*[2] 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 paraproc group represented by the frame at the head of the run queue.

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

This global value is initialized by reading the count of paraprocs 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[3]. 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 Sec. 4.7 below.

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,

[3] In some systems (e.g., the Astronautics ZS series), the processor may idle until the global register becomes positive. This avoids busy waiting.

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 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[4] . 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 (an unlikely restriction), then the program must abort. Any other chance for deadlock to occur comes from the actions of the operating system.

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. Parameters allow 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[5]. 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. 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

[4] 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.

[5] Possibilities for optimizing parameter passing are discussed in Sec. 4.6 below.

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. 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.

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.

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.

4.6 Overhead, Optimization, and Extensions

While our system for suspending and resuming a merging paraproc 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 paraproc 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 paraproc stacks, eliminating the need to reload stack pointers from and allocate stack areas in PCB's.

Our proposed system provides several opportunities for optimization. The techniques for parameter passing and local stack variable allocation are prime targets for optimization. Because the run-time system assumes no data dependence between procedure-processes, parameter values may be copied into registers, as long as they are written back to their original locations in memory before the next **create** or **merge** statement. Local variable allocation may occur in registers and avoid the stack entirely if they are not to be passed as parameters.

These optimizations can also be used in generating code for parallel loops, commonly referred to as **doall** and **doacross** loops [9]. In this case, it may be possible for the generated code and run-time support to revert back to the baseline self-scheduling system. If the compiler can determine that no other paraprocs may be active, frame allocation may be bypassed, and the global registers or well-known memory locations may be initialized directly. If other paraprocs may be present, frame allocation and initialization are required, but parameter passing is not needed. The shared variables are referred to directly as globals, and registers may be used to hold their values between synchronization points. These techniques can also be applied to procedure-processes that do not create any children, since they can be executed inline.

Because optimized code can be generated for parallel loops and integrated into our run-time system, we believe that a **doall** construct should be included in our parallel language proposal. This would enable the programmer to specify parallel loops, and eliminates the burden placed on the compiler to detect them. The compiler may, however, coalesce nested loops to increase grain size and reduce scheduling points [23]. By adding parallel loop constructs to the language, we can allow this type of parallelism to be expressed naturally, without forcing the programmer to use procedure-processes. Furthermore, by adding a **doacross** looping feature, inter-iteration synchronization may be introduced and handled by the code generator. If we assume that the compiler generates the correct sequence of instructions for synchronization, our argument for deadlock prevention is still valid (provided that the microtasks are scheduled in the order specified by the **doacross**, and that processors do not switch context at these synchronization points). Parallel loop constructs have been proposed and incorporated into several FORTRAN [24, 25] and C [14] dialects. By adapting one of those proposals into our C-based language, the procedure-processes can be reserved for larger grain parallel processing and recursive algorithms.

4.7 Operating System Issues

The role of the operating system (OS) in our parallel processing system is simple and straightforward. We still require the OS to support I/O requests, virtual memory, and library routines (e.g., timing support). The concepts of time and process priorities are omitted from our proposed language, because they complicate the implementation. On the other hand, we believe that a predefined dynamic memory allocation routine (i.e., **malloc**) should be supported by the run-time system, in order to avoid unnecessary OS interference. This routine must account for the fact that multiple paraprocs may be calling it simultaneously. Separate memory areas for the frame pool and dynamic allocation should be acquired from the OS at program startup.

The method we propose for OS scheduling of the parallel program is the *virtual processor* approach. The program is allocated some number of OS processes, which it uses as virtual processors. The virtual processors begin execution of the run-time system as a user program. The OS manages the scheduling of virtual processors on physical processors. Clearly, a policy of allocating a number of virtual processors equal to the number of available physical processors should reduce OS level context switching overhead. In order to ensure that deadlock will not occur in the program, we require the OS scheduler to prevent starvation of any virtual processor. However, if a parent paraproc is not blocked at a merge point on a given virtual processor, that OS process may be relinquished by the run-time system at a scheduling point, decreasing the program's allocation of virtual processors by 1. This is desirable in the presence of increasing system load, where the number of OS processes due to other jobs may increase to be far greater than the number of physical processors.

The virtual processor approach to an OS interface has been used successfully by several compilers

(e.g., the Ada compilers for the Sequent Symmetry and Encore Multimax) and parallel language environments [7]. In addition to scheduling virtual processors, the OS must also ensure that these OS processes can share their virtual memory space. This system enables the run-time system to run in user mode and manage the program level parallelism, even though it is transparent to the programmer.

5 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 sections describing the run-time system, its interface to the operating system, and the opportunities for optimization are intended to demonstrate the efficiency of our proposed system. We expect the performance of the run-time system to allow reasonable execution speeds for parallel programs on a single processor, and we have verified this in our prototype system (described below). 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 efficiency, flexibility, expressiveness, and structure have been met, but only repeated usage of a completed system can determine this for sure.

We have begun to construct this parallel programming system by developing the run-time system. The baseline system that supports non-recursive parallelism without parameters or need for frames has been implemented. This represents the most optimized form of generated code and run-time support for our parallel model, and we have used it to support the execution of parallel **doall** loops. For maximum efficiency, we have developed this 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 [26], an architecture containing high speed decoupled-access-execute (DAE) processors. The machine has been designed to be configured with as many as 16 processors, each possessing its own 128K bytes of local memory. 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 our 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 interpreter-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.

Depending on the data alignment of global variables, the grain size of parallelism, and the different options for used for self-scheduling, we have observed near linear speedups in some tests using as many as 16 processors [27, 28]. The grain sizes used in these tests were on the order of 3500 assembly language instructions for a simple matrix multiply parallel loop. We believe that the use of parallel-procedures with more complex algorithms will increase this grain size significantly, allowing near linear speedups for larger numbers of processors. Our next step is to complete the entire run-time system, and run some tests using the full functionality of the parallel procedure model.

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, Mass., 1988.

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

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

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

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

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

[8] M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "Design rationale for Psyche, a general-purpose multiprocessor operating system," in *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 255–262, August 1988.

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

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

[11] T. Lovett and S. Thakkar, "The Symmetry multiprocessor system," in *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 303–310, August 1988.

[12] 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.

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

[14] 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, February 1987.

[15] R. H. Thomas and W. Crowther, "The Uniform System: An approach to runtime support for large scale shared memory parallel processors," in *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 245–254, August 1988.

[16] R. M. Clapp and T. Mudge, "A parallel language for a distributed-memory multiprocessor," in *Proceedings of The Fourth Conference on Hypercube Concurrent Computers and Applications*, pp. 515–522, Monterey, CA, March 1989.

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

[18] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structure and Algorithms*, Addison-Wesley, Reading, Massachusetts, 1983.

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

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

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

[22] A. J. Musciano and T. L. Sterling, "Efficient dynamic scheduling of medium-grained tasks for general purpose parallel processing," in *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 166–175, August 1988.

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

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

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

[26] J. E. Smith et al., "The ZS-1 central processor," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pp. 199–204, October 1987.

[27] 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 *Workshop Proceedings of the 16th International Symposium on Computer Architecture*, Kluwer Academic Publishers, Norwell, MA, 1989. (to appear).

[28] R. M. Clapp and T. N. Mudge, "Parallel loops on a multiprocessor mini-supercomputer," Advanced Computer Architecture Lab, Department of Electrical Engineering and Computer Science, The University of Michigan, April 1990. submitted for publication.