# *ParC* — An Extension of *C* for Shared Memory Parallel Processing

Yosi Ben-Asher*
Department of Mathematics and Computer Science
Haifa University
31999 Haifa, Israel

Dror G. Feitelson*
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Larry Rudolph†
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

**Summary**

*ParC* is an extension of the *C* programming language with block-oriented parallel constructs that allow the programmer to express fine-grain parallelism in a shared-memory model. It is suitable for the expression of parallel shared-memory algorithms, and also conducive for the parallelization of sequential *C* programs. In addition, performance enhancing transformations can be applied within the language, without resorting to low-level programming. The language includes closed constructs to create parallelism, as well as instructions to cause the termination of parallel activities and to enforce synchronization. The parallel constructs are used to define the scope of shared variables, and also to delimit the sets of activities that are influenced by termination or synchronization instructions. The semantics of parallelism are discussed, especially relating to the discrepancy between the limited number of physical processors and the potentially much larger number of parallel activities in a program.

**Keywords:** *ParC* language, Parallel Programming, Semantics of parallelism, Forced termination, Shared memory.

*Supported in part by an Eshkol Fellowship from the Ministry of Science and Technology, Israel
†Supported in part by a Kerin Amet grant and by France-Israel BSF Grant #3310

# Introduction

As part of a quest to make parallel programming an everyday activity, we developed an easy to use and quick to learn parallel programming language, referred to as *ParC*. *ParC* is a block-oriented, shared memory, parallel version of the popular *C* programming language. *ParC* presents the programmer with a model of a machine in which there are many processors executing in parallel and with access to both private and shared variables.

While most of the *ParC* constructs have been proposed and used before, this is to the best of our knowledge the first time that they are brought together in a cohesive form. The result is a language that allows parallelism to be expressed in a natural and simple manner. It is one of the few programming languages designed to support a shared-memory model of computation, thus making programming easier and allowing users to tap the vast base of PRAM algorithms. In addition, users can capitalize on their experience with *C*, concentrating first on writing a correct sequential program and then on efficient parallelization using the *ParC* extensions.

In this paper we describe the features and semantics of *ParC*. The rest of this section explains the motivation for designing a new language, the effect of the motivating forces on the design, and the structure of the software environment that surrounds it. The next section describes the parallel constructs and scoping rules. The exact semantics of parallel constructs when there are more activities than processors have been widely neglected in the literature. We discuss this issue and provide guidelines for acceptable implementations. We then describe the innovative instructions for forced termination, which are based on analogies with *C* instructions that break out of a construct, followed by a discussion of synchronization mechanisms. A discussion of the programming methodology of *ParC* is then given and is followed by a discussion of our experiences with *ParC*. A comparison of *ParC* with other parallel programming languages is delayed until the end of the paper, after we have described all of its features.

## Motivation and Design Guidelines

Many parallel programming languages (or language extensions) have been proposed and implemented in recent years [1, 2]. These languages are based on different philosophies and provide a wide spectrum of features. *ParC* belongs to the family of those that provide *explicit* parallelism, as opposed to functional and logic languages. In addition, it is conservative rather than revolutionary, e.g. dataflow, and is based on the well-known and widely used *C* language, making it ideal for wide acceptance by the programming community [3]. It supports the shared-memory model of computation, which is widely believed to ease parallel programming and enhance user productivity [4]. These basic characteristics reflect the belief that such a language is the best vehicle for research in parallel algorithms and their implementation on parallel architectures.

The practical usefulness of *ParC* stems from two sources. On the one hand, it is similar in flavor to the pseudo-languages that are used by theoreticians for describing parallel algorithms. Indeed, we have found that most shared-memory parallel algorithms can be directly coded in *ParC*. On the other hand, it is also conducive for the parallelization of sequential programs written in *C*. The language promotes an understanding of the problems inherent in parallel code, through the parallelization process, and allows various approaches for their solution to be expressed and compared.

The main feature of *ParC* is that the parallelism is incorporated in the block structure of the language. This encourages the use of structured program design, and also provides a high-level description of the program structure, thus furnishing a natural extension to the sequential *C* control structures. The language is not intended to be completely general in the sense that *any* parallel structure can be expressed. On the contrary, good structure means *restricting* the programmer and not allowing some constructs, as in the use of closed `while` loops instead of a general `goto` instruction. The generalization of this idea to parallelism results in the use of closed parallel constructs rather than `fork` and `join`. In addition, a structured language may include some redundancy if a number of similar constructs are useful; for example, *C* provides three different iterative constructs, `for`, `do`, and `while`, even though the first two can easily be expressed using the third. *ParC* has three redundant parallel constructs, as well as redundant synchronization mechanisms.

Parallelism is expressed in *ParC* at a high level of abstraction. The programmer need not know the exact number of processors, the relative speeds of the processors, or even which processor is executing which

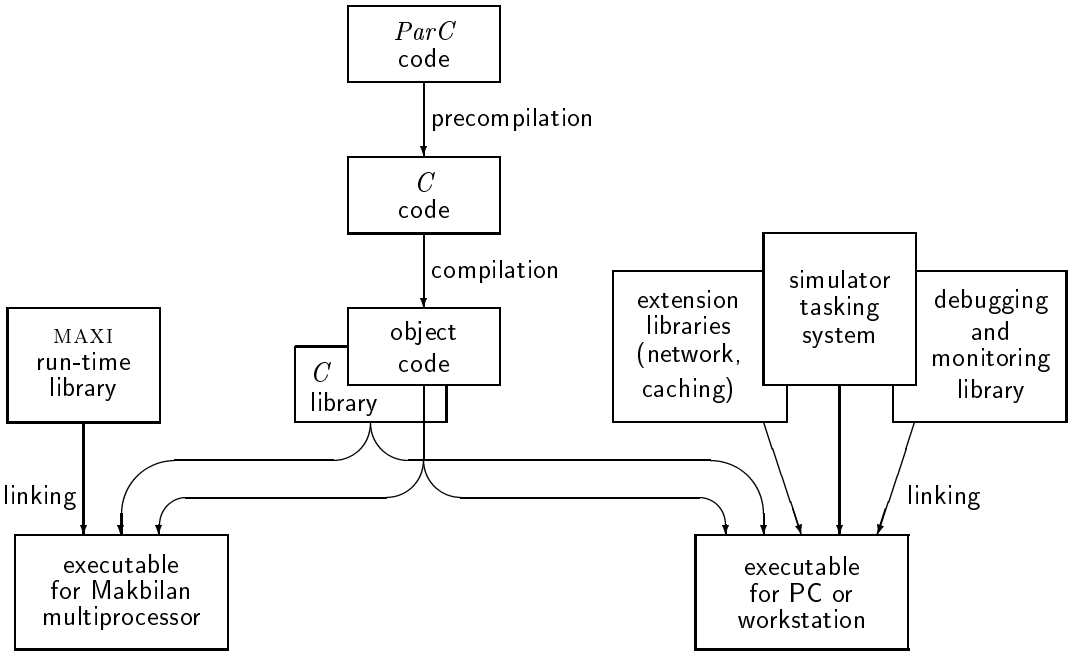



Figure 1: *The ParC software system.*

activity. This results in very portable code, and frees the programmer from worrying about too many low level details, allowing him to concentrate instead on getting the parallelism correct. On the other hand, for some applications, the resulting code may not be very efficient. In such cases, it may be desirable to "hand-tune" the code to the machine. For this purpose, *ParC* supplies concise high-level facilities to express how the program is mapped at run time. It is also possible to improve performance by restructuring the program and changing the relationship between parallel and sequential constructs. Note that this is all done within the context of the language, and there is no need for additional low-level primitives. The tuning is usually done after the application has been fully developed and performance analysis has shown the problem spots.

## System Structure

The purpose of the *ParC* system is to facilitate research on the design and implementation of parallel algorithms, and at the same time provide a testbed for research on parallel programming, debugging, and run-time systems, all in the context of a shared-memory model. The environment is composed of three parts: a precompiler that translates *ParC* code into *C*, a simulator that simulates parallel execution on a UNIX workstation or any IBM compatible PC [5], and a run-time library that supports the execution of *ParC* programs on the Makbilan research multiprocessor [6, 7] (see Figure 1).

The parallel program is represented by a single Unix (or DOS) process, but this has little significance to the programmer. Process-oriented Unix system calls may not be used in a parallel program. Unix or DOS is available to the programmer for program start-up and for all I/O operations. Thus I/O operations are serialized, and file descriptors are shared by all the parallel activities. This is true for both the simulator and the Makbilan.

The software layer of the **Mak**bilan **sy**stem is called MAXI for short [6, 7]; it runs above the Intel RMK kernel. The implementation of *ParC* constructs is done partly by code transformations in the precompiler, and partly by the simulator and MAXI library. It should be noted that the precompiler is actually a compiler, not a preprocessor. The parallel extensions in *ParC* are *language* extensions, and not merely high-level calls to system functions. The precompiler has to recognize the language and keep a symbol table in order to support the scoping rules and other features.

The simulator may be linked with libraries that extend its functionality. One set of libraries emulates

various hardware features, e.g. a certain network topology or a caching scheme [8]. Other libraries belong to different versions of the parallel debugger. These libraries contain routines that monitor the program execution, and provide data for a graphical display facility [9]. Even without these additions, the simulator gives a detailed performance prediction by counting assembler instructions.

The Makbilan research multiprocessor contains up to 16 single board computers, connected by a Multibus II. Each board has an Intel 80386 processor running at 20 MHz, an 80387 mathematical coprocessor, and 4 MB of memory. A processor can access the memory on another board through the bus, but this takes longer than a local access. Hence the Makbilan is a non-uniform memory access (NUMA) architecture [10]. Additional elements on the bus include the Unix host, a terminal server, and a peripherals interface card.

It should be noted that *ParC* does no depend on this architecture. It is fairly cost effective to assemble a small scale parallel machine using several single board computers connected by a shared bus. The Makbilan is only one instance of such an architecture. A shared memory that is accessed through a crossbar or a multistage network could also be used. *ParC* does not make any assumptions about the architecture; it just requires the ability to share the address space across all the processors.

# Creating Parallelism: Activities and Shared Variables

This section describes the essence of *ParC*: its parallel constructs, scoping rules for sharing variables, and other special constructs related to parallel processing. *ParC* is a superset of the *C* programming language and so there is no need to review any of the standard features.

## Parallel Constructs

Parallel programs have two main sources of parallelism: loops and recursion. The parallel constructs in *ParC* are especially designed to handle both cases in an efficient and structured manner. The parallel "things" generated by *ParC* are called *activities*, in order to avoid overloaded terms such as "process" or "task".

### parfor

Loop parallelism is handled by the parfor construct, which is a parallel version of the conventional for loop. The syntax for this construct is:

parfor ( *index*; $e_1$; $e_2$; $e_3$ )
    stmt

The body of the loop, stmt, can be any legal statement in *ParC*, including a compound statement with possible nested parallel constructs and function calls. If it is a block (compound statement), it can declare its own private variables which won't be known outside the scope of this block, or by any of the other iterations of the block; this is elaborated below. The bodies of the distinct iterates are executed in parallel; the meaning of being executed in parallel is discussed in the section entitled "The Meaning of Parallelism".

A distinct, local copy of the loop index is created for each iterate; this is always an integer variable. The index variable can be modified without affecting other activities. By analogy to serial for loops, the number of activities is $\left\lfloor \frac{e_2-e_1}{e_3} \right\rfloor + 1$. The index variable in activity $i$ is initialized to $e_1 + (i-1)e_3$. Note that this variable is only defined within the parfor block, and does not exist outside of its scope.

Unlike the for loop, however, the expressions $e_1$, $e_2$, and $e_3$ are evaluated only once, before any activities are actually spawned. Moreover, it is illegal to directly transfer control, via a goto statement, either into or out-of the body of the parfor construct.

### lparfor

The lparfor construct is important for the common case in which the number of iterates in a parallel loop is much larger than the number of processors, $P$. If the iterates are independent, i.e. with no data dependencies,

it would be more efficient to *chunk* them and create only $P$ activities, one per processor. Each of these chunked activities would then execute $n/P$ of the iterates in a serial loop (where $n = \frac{e_2 - e_1}{e_3} + 1$ is the total number of iterates). This saves the overhead associated with spawning all the activities, and improves the possibility of using fine-grain parallelism. *ParC* has a special construct that implements this optimization, called lparfor, with the **l** prefix denoting lightweight. Using lparfor gives the compiler the opportunity to generate more efficient code. Note that this construct does not add functionality, it is useful only for optimizations. The implementation can choose to balance the load between the $P$ activities statically, by allocating $n/P$ iterates to each, or dynamically, by using chunked self scheduling [11, 12, 13]. This is a further optimization that does not change the semantics.

It was decided to add a special construct to the language instead of just adding a hint to the compiler, as found in many commercial parallel languages, because of the semantic implications. lparfor explicitly implies that the iterations are independent. A correct program using parfor constructs may not be correct if lparfor is used instead. The reverse is not true — one may always replace an lparfor with a parfor without affecting correctness (just efficiency).

parblock

The parallelism that is found in recursion, such as that of divide and conquer algorithms, is handled by the parblock construct:

```
parblock
{
    stmt-list
::
    stmt-list
}
```

(in general, more than two blocks are possible). However, this construct is more general than a specialized construct for divide and conquer, because the constituent activities are coded individually. Note that the three parallel constructs are redundant: a parblock and lparfor may be implemented using a parfor, for example. But such implementations would be bulky and degrade code quality.

### Mapped Constructs

All three parallel constructs can be qualified by the mapped keyword. This has the effect of always mapping activities to physical processors in the same way. The exact mapping is implementation-dependent; a reasonable choice is round robin by the activity index, i.e. mapping activity $i$ to processor $i \bmod P$. The key feature is that in every instantiation of a mapped parallel construct, the $i$'th activity will be mapped to the same processor. Hence these activities have a locality relationship. Advanced programmers can make use of this feature to produce more efficient code on many machines types. It gives the programmer some ability to control the allocation of activities to processors without the burden of keeping track of process IDs or explicit mapping. It also helps make code more readable by avoiding some sync statements.

In the current implementation, only the mapped lparfor construct is supported.

### Nesting of Parallel Constructs

The different parallel constructs may be nested in arbitrary ways, and also mixed with regular $C$ constructs. In all cases, the activity that executes the parallel construct is suspended until all the constituent activities have terminated. Nesting of parallel constructs thus creates a tree of activities, where only the leaves are executing while internal nodes wait for their descendents. Note that these constructs allow the user to express the spawning of a large number of activities at once, while avoiding messy implementation and mapping details [14]. Such details are delegated to the runtime system.

General gotos are not allowed past the boundaries of parallel constructs. This is the only restriction in *ParC* that is not compatible with *C*.

```
int  sorted[n];

void quicksort( arr1, arr2, left, right )
{
    int  i, l, r, n, splitter;

    n = right − left + 1;
    if (n == 1)
        sorted[left] = arr1[left];
    if (n <= 1)
        return;

    splitter = arr1[left];
    l = left;
    r = right;
    for (i=left+1; i<=right; i++) {
        if (arr1[i] < splitter)
            arr2[l++] = arr1[i];
        else
            arr2[r−−] = arr1[i];
    }

    sorted[l] = splitter;

    quicksort( arr2, arr1, left, l−1 );
    quicksort( arr2, arr1, r+1, right );
}
```

```
int  sorted[n];              /*shared array */

void quicksort( arr1, arr2, left , right )
{
    int  l, r, n, splitter;    /*no declaration for i */

    n = right − left + 1;
    if (n == 1)
        sorted[left] = arr1[left];
    if (n <= 1)
        return;

    splitter = arr1[left];
    l = left;
    r = right;
    lparfor ( i; left+1; right; 1 ) {
        if (arr1[i] < splitter)
            arr2[ faa(&l,1) ] = arr1[i];
        else
            arr2[ faa(&r,−1) ] = arr1[i];
    }

    sorted[l] = splitter;

    parblock {
        quicksort( arr2, arr1, left, l−1 );
    ::
        quicksort( arr2, arr1, r+1, right );
    }
}
```

Figure 2: *Sequential and parallelized versions of quicksort.* faa *is an atomic fetch-and-add operation (see the section on synchronization). Note that in the parallel version* l *and* r *are shared by all activities in the* lparfor*, but each recursive call will create new copies of* l *and* r*. There is only one copy of the array* sorted *and it is shared by all activities.*

### An Example: Parallel Quicksort

As an example, consider the parallelization of a *C* program that implements the quicksort algorithm (left of Figure 2). Two arrays are used alternately to copy elements that are smaller or larger than the first element in the segment. The first element is copied to the output array at a location that is between the set of smaller values and the set of larger values. The recursion ends when the segment is empty or includes a single element; if there is a single element, it is copied to the output array.

The parallel version in *ParC* is given in the right of Figure 2. An lparfor is used to compare the elements against a splitting value in parallel, dividing them into those that are larger and those that are smaller. As this implies parallel access to the indices l and r, the atomic fetch-and-add (faa) instruction is used to increment and decrement them. Then a parblock is used to perform the two recursive calls in parallel.

## Scoping Rules

An important feature of a programming language is its scoping rules. In parallel languages, scoping is often replaced by explicit declarations that variables are either private or shared. We find that explicit declarations are unnecessary, as the scoping naturally leads to a rich selection of sharing patterns. This is so because *ParC* is a real parallel language, not just a sequential language with calls to a parallel run-time support system, and the precompiler understands the program structure.

```
parfor ( i; 0; N−1; 1 ) {
    static int s;        /* shared by all N activities */
    int t;               /* private to each activity */

    faa(&s,1);           /* must be updated atomically */
    t++;                 /* private variables need not be protected */
}
```

Figure 3: *A variable declared as static within the construct will actually be shared by all the activities of the construct!*

### Sharing Patterns in Nested Blocks

The scoping rules of the *C* language come in two flavors: local to a procedure and global. Within a procedure there is a hierarchical scoping of variables, since each block of code may contain a data declaration section. This is reminiscent of the hierarchical scoping of Pascal. The scope of global variables can be limited to just those procedures within the same file by the static statement.

*ParC* allows parallel constructs to be freely nested inside of other parallel constructions. By preserving the normal *C* language scoping rules, variables declared within a block (no matter if the block is parallel or sequential) can be accessed by any of its nested statements (parallel or sequential). In particular, if a nested statement is a parallel construct, all the spawned activities will share the variables declared in the enclosing blocks. Global variables are shared among all the program activities; static variables and externals are also considered global. The general rule is: if you can see it, you can use it. If more than one activity can see a variable, then that variable is shared, otherwise it is private. Every variable within a *ParC* program can potentially be shared. A private variable can become shared when a shared variable points to it.

Local variables declared within a block are allocated on the activity's stack. The scoping rules therefore imply a logical "cactus stack" structure for nested parallel constructs. Note, however, that this is the case only for nested parallel constructs in the same procedure. Code in one procedure cannot access variables local to the procedure that called it unless they are passed as arguments. It cannot modify them unless they are passed by reference.

An example of the hierarchical scoping rules in *ParC* was given already in Figure 2. The index variable i in the lparfor is local in each activity, so there are many (specifically, right − left) distinct copies of it. The variables l and r, however, are shared and must be protected during their modification. This is done by the atomic faa (fetch-and-add) operation.

### static Declarations

The *C* language was designed in the early '70s in the context of writing the Unix operating system on a PDP 11. It is remarkable that most of its features can be used in *ParC* on parallel machines with no ill-effects. A major exception is the static declaration. In *C*, static variables are persistent across function invocations. The implementation is therefore to allocate storage for such variables from the global heap space, rather than on the stack. In *ParC*, static variables declared in a parallel block become shared by all the activities, rather than being private to the declaring block (see Figure 3).

## Forced Termination

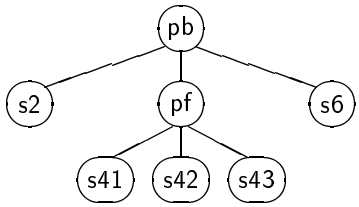
*ParC* activities are logical structures that are implicitly specified by the parallel constructs and consequently are not identified by names or IDs. Therefore it is impossible for an activity to directly kill or terminate any other activity.

However, since activities in *ParC* are generated by block-structured expressions, *ParC* does allow some limited control over the execution of the activities within the construct. It is possible for an activity to self destruct or to kill a whole group of related activities, by "jumping out" of a parallel construct. This is done

Sample code:

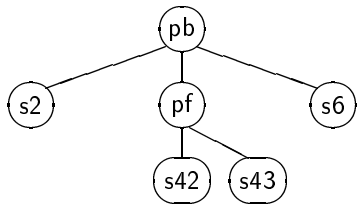
```
main()
{
   m1
   f()
   m2
}
void f()
{
   s1
   parblock
   {
      s2
   ::
      s3
      parfor (i; 1; 3; 1)
         s4i
      s5
   ::
      s6
   }
   s7
}
```

Snapshot of activity tree:

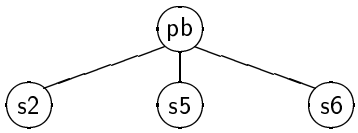



The effect of different instructions executed by activity s41 or activity s2 at this moment are:

s41 does pcontinue

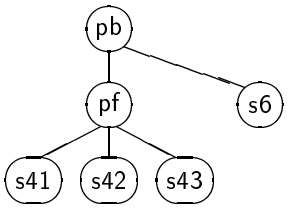



s41 does pbreak




s41 does return


s2 does pcontinue




s2 does pbreak


s2 does return


Figure 4: *The effect of* pcontinue, pbreak, *and* return *when issued in a parallel construct.*

by the following *ParC* instructions, whose semantics when executed inside a parallel construct are based on analogies with serial constructs:

pcontinue — deletes the activity that executes it, without any effect on its siblings. This is analogous to a jump to the end of the activity's block of code.

pbreak — terminates the activity with all its siblings and their descendents, effectively ending the parfor or parblock that generated them. As in *C*, this breaks out of the construct and is analogous to a jump to the first instruction after the construct.

return — returns from the last function call, terminating all the activities generated within this function.

```
search( arr, n, x )
int  *arr, n, x;
{
    lparfor ( i; 1; n; 1 )
        if (arr[i] == x)
            return(i);
    return(−1);
}
```

Figure 5: *An example of using a* return *construct within an* lparfor *construct. When a match is found, all the other activities are terminated.*

Likewise, setjmp and longjmp allow to return to a previous state across any number of function calls.

The p prefix in pcontinue and pbreak is required so that these instructions can be used unambiguously even within a sequential iterative construct nested inside a parallel construct.

An example that clarifies the effect and highlights the differences between these instructions is given in Figure 4. Note that when a subtree is terminated, the root activity may only resume after all the constituent activities have terminated. It is not enough to start an asynchronous mechanism that will cause the subtree to terminate (or, in the extreme case, just let it execute to termination in parallel with the root), because the activities in the subtree have access to local variables in the root. Indeed, it is implied that the whole machine be interrupted and the subtree terminated as fast as possible, to prevent situations where new activities are generated at a higher rate than existing ones are terminated.

The code in Figure 5 shows the use of the return statement within a parfor construct in a code fragment of a parallel search on all the elements of a large array. When a match is found, the search is terminated by the return instruction that returns the index of the matching cell. The lparfor construct is used to reduce the overhead since the size $n$ of the array may be much larger than $P$, but only $P$ activities will be spawned and each will loop over $n/P$ elements.

## Synchronization

The closed parallel constructs are not sufficient for all the synchronization needs of parallel programming. There is no wide-spread agreement as to what constitutes satisfactory high-level synchronization mechanisms for parallel languages. Some researchers emphasize data abstraction and mutual exclusion, and advocate the use of monitors or similar constructs. Others emphasize the relationship between synchronization and communication, and suggest mechanisms that are based on message passing (e.g. the Ada rendezvous). A third group suggests the use of events as the main mechanism for process synchronization.

The approach adopted for *ParC* is not to choose a specific high-level primitive. Instead, a number of low-level synchronization primitives with different characteristics are provided. This allows the programmer to create various synchronization schemes, and to control the program behavior and the resulting performance. It is certainly possible that some high-level primitives will be added to the language if the programming experience indicates that they are useful for a large class of applications.

The three basic synchronization mechanisms in *ParC* are:

- Fetch-and-add, denoted faa(&i,exp), is an atomic read-modify-write operation on integers. It has been shown to be useful in a large number of algorithms [15, 16]. Specifically, faa can be used to implement wait-free interactions, where a number of activities operate on shared data structures simultaneously without having to wait for each other. In addition, it can be used for various synchronization schemes based on busy waiting. An example of its use appears in the quicksort program of Figure 2. faa is used there to allocate distinct cells in the array arr2 to different activities that all increment the index variables at the same time. The atomic nature of this instruction alleviates the need for mutual exclusion, and prevents unnecessary serialization.

```
int  a[N];

parfor (i; 0; N−1; 1)
{
    int  t, tmp;

    a[i] = init_val(i);
    sync;

    for (t=0; t<T; t++) {
        tmp = f(a[i−1],a[i],a[i+1]);
        sync;
        a[i] = tmp;
        sync;
    }
}
```

```
int  t, a[N], tmp[N];

lparfor (i; 0; N−1; 1)
    a[i] = init_val(i);

for (t=0; t<T; t++) {
    lparfor (i; 0; N−1; 1)
        tmp[i] = f(a[i−1],a[i],a[i+1]);

    lparfor (i; 0; N−1; 1)
        a[i] = tmp[i];
}
```

Figure 6: *Two examples of the initialization and use of private variables. In the first, a* sync *statement in embedded in the loop. In the second, the two parallel constructs clearly show the sequentiality. The use of the* lparfor *should be efficient for large values of $n$, but it requires temporaries to be defined as shared arrays.* lparfor *cannot be used on the left because the* sync *creates a dependency between the iterations.*

- Semaphores complement the wait-free capability of faa by providing an interface through which an activity can suspend itself waiting for an event. Variables of type semaphore may be declared, and the P and V operations may be applied to them with the usual semantics [17].
- It is evident that barrier synchronization is a very useful primitive for parallel programming [18]. We therefore added the sync instruction to *ParC*. This instruction implements a barrier that involves all the activities spawned by a certain parallel construct. Thus an activity that executes a sync is suspended until all its live siblings also execute a sync. If only a subset of these activities perform a sync instruction, while the others loop forever or wait at a semaphore, deadlock will ensue. However, if a subset of these activities terminate, only the remaining activities have to perform a sync. In particular, if all the awaited activities terminate, the activities waiting at the sync may proceed.

It should be noted that the sync instruction is redundant, in the sense that it can be implemented by using semaphores. However, its use is so common that it is appropriate to provide a special instruction for it, and relieve users from the implementation details.

A common use is to ensure that all variables that might be shared have been initialized before they are accessed (Figure 6 left). In this example, an array is initialized and then each cell is "relaxed". Alternatively, this could be done with separate parallel constructs, thus emphasizing the sequential nature of the operation (Figure 6 right). However, using separate constructs requires shared arrays for temporary values that have to be maintained across constructs. We note in passing that it is doubtful that any machine could support a naive implementation of the code on the left of this figure (with the parfor construct) for large values of $n$ as it would require a large number of activities to be maintained.

Two additional instructions allow users to influence the execution of activities on processors, bypassing the high level of abstraction provided by the other language constructs. For example, these instructions allow the programmer to achieve effects similar to co-routines. They are:

yield — An activity executing this instruction explicitly yields its processor, causing a context switch to another ready activity.

switch_on, switch_off — This allows the user to create nonpreemptable activities. It is used to change the semantics of the parallel constructs as defined below. The reasons for this option are discussed in the section entitled "The Meaning of Parallelism". A nonpreemptable activity may nevertheless be preempted if it is forced to wait for some event. This happens if the activity performs a sync instruction,

a P on a semaphore, a yield, a blocking I/O command, or spawns additional activities. Note that this instruction cannot be used to guarantee mutual exclusion, because it has no effect on activities runing on other processors.

As an example illustrating the use of these instructions consider a global time-stamp mechanism which is useful for program monitoring and debugging. The idea is to initially create two activities: a non-preemptable activity, accomplished via the switch_off construct, which increments a "clock" variable in an infinite loop, and a regular activity which starts the parallel program. Whenever an interesting event occurs, it can be labeled with the current clock value. This provides a full ordering of the events that occur during program execution. When the parallel program terminates, a pbreak can be used to stop the clock activity. As another example, the yield instruction can be used to implement a two-phase blocking mechanism, which might be more efficient than busy-waiting [19].

# Implementation Issues

*ParC* is a parallel programming language that, hopefully, will allow programmers to get high performance from parallel computers. It is therefor important that the programmer have some idea as to how *ParC* implements its constructs and how the data structures get mapped. On the other hand, the goal of an easy to use language implies that the programmer need not know a whole lot about the implementation nor the underlying machine. In this section, we try to give the appropriate amount of information to meet these two conflicting goals.

Similarly, it is important for the programmer to understand the meaning of the parallelism provided by *ParC*. The language itself gives the programmer some control over how the activities are actually executed. Explicit details of the runtime or operating system are therefore not needed.

## Mapping Activities and Data Structures

In architectures with non-uniform memory access, such as the Makbilan, it is very important that a large part of the memory references be directed to local memory. Therefore the programmer must have some means to coordinate the mapping of activities to processors and the mapping of data structures to memory modules. In *ParC*, this is done by the scoping rules, the mapped lparfor construct, and memory allocation procedures.

### Threaded Stack

Variables and data structures that are declared within an activity are not known to other activities, except for descendents of the declaring activity. It is therefore reasonable to require that such variables reside in a memory module close to the processor that runs that activity. As the variables are allocated on the activity's stack, this implies that the stack should be allocated in the memory module adjacent to the processor that runs the activity. If run-time migration is used, the stack must be moved as part of the activity context. As the stack must retain its virtual address, this means that the page tables must also be modified.

The scoping rules are implemented by the *ParC* precompiler, by changing the references to variables declared in surrounding blocks to indirect references. Thus the run-time system does not have to chain stacks to each other explicitly to create a full-fledged cactus stack, and it does not have to search the stack to find the referenced variables [20]. Instead, direct pointers to the variables location (typically on another activity's stack) are available. We call the resulting structure of stacks with mutual pointers into each other a *threaded stack*.

An example is given in Figure 7. The two activities in the parblock share the variable i declared before the parblock. Each activity is transformed by the precompiler into a procedure that gets a pointer to i as its argument. The function names and the arguments are passed as parameters to the do_parblock function. This function is part of the run-time library. When it is called, it creates new activities on different processors. Each activity is complete with its own call stack. An activation record for the activity's function is constructed on the stack, and the appropriate arguments are copied to it. The activity can then commence, and behaves as if the function was called on a remote processor.

**original code:**

```
main()
{
    int  i;
    parblock
    {
        int  x;
        x = i;
    ::
        int  y;
        i = y;
    }
}
```

precompilation

```
main()
{
    int  i;
    do_parblock(f1,&i,f2,&i);
}

f1 (i)
int *i;
{
    int  x;
    x = *i;
}

f2 (i)
int *i;
{
    int  y;
    *i = y;
}
```

**at run time:**

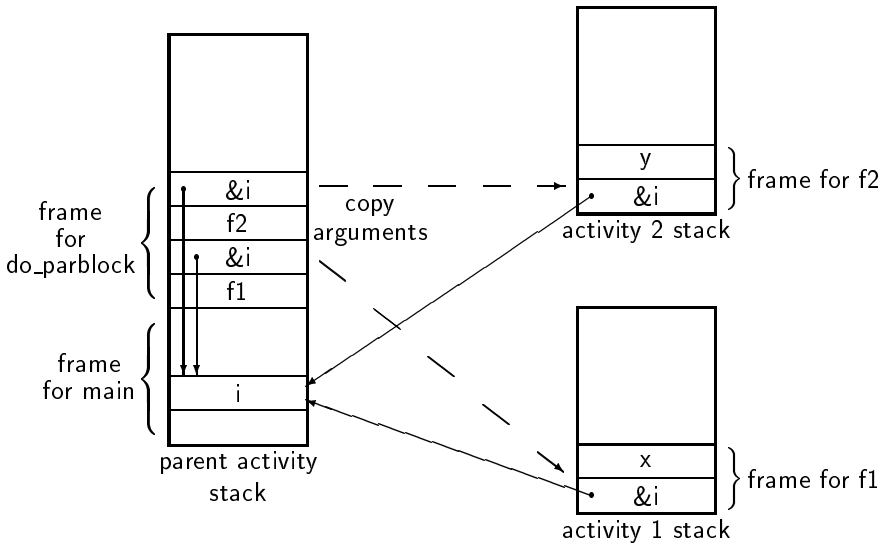



Figure 7: *Implementation of the scoping rules with threaded stacks.*

### Persistent Partitioned Data Structures

Dataparallel programming languages have a deficiency in specifying the control structure, i.e. which processor computes what. Likewise, control parallelism languages, of which *ParC* is an instance, are often deficient in expressing the data distribution. In many applications, declaring local variables is not enough. Numerous parallel algorithms call for the use of large shared data structures, which are *partitioned* among a set of activities. This means that each activity "owns" part of the data structure, and that most of its computation relates to this part. However, activities can also access parts belonging to other activities. For example, a parallel image processing application may partition a large image into blocks. A separate activity would be responsible for each block, but it would have access to neighboring blocks if necessary.

Another problem with local variables is that they only exist during the lifespan of the activity that declared them. It is not possible for one set of activities to deposit data in a set of local data structures, and for another set of activities to subsequently retrieve the data with the same locality properties. What we need is the ability to partition data structures according to a *persistent* pattern.

A *ParC* programmer can overcome these problems by using the mapped lparfor construct and the malloc system call. As explained, the lparfor construct creates no more than $P$ activities and when using the mapped variant, corresponding activities in distinct constructs will be mapped to the same processor. If the user wants to create exactly one activity on each processor, the predefined global variable proc_no may be used.

A persistent, two dimensional, shared array that is partitioned across the processors, may be created as shown in the code on the right of Figure 8. Its elements are referenced as any two dimensional array, and it fits in well with the *C* notion of a vector of pointers to vectors. First, a global array of pointers is declared. Then a set of activities is spawned by a mapped lparfor. Each activity allocates persistent local memory using the *C* malloc function; the definition of *ParC* guarantees that malloc use a local heap. The addresses of the local memory blocks are assigned to the global array. Recall that although the heap is local, it is accessible by all activities. Also note that this code works for any number of processors; the mapped construct always performs the same mapping of activity to processor.

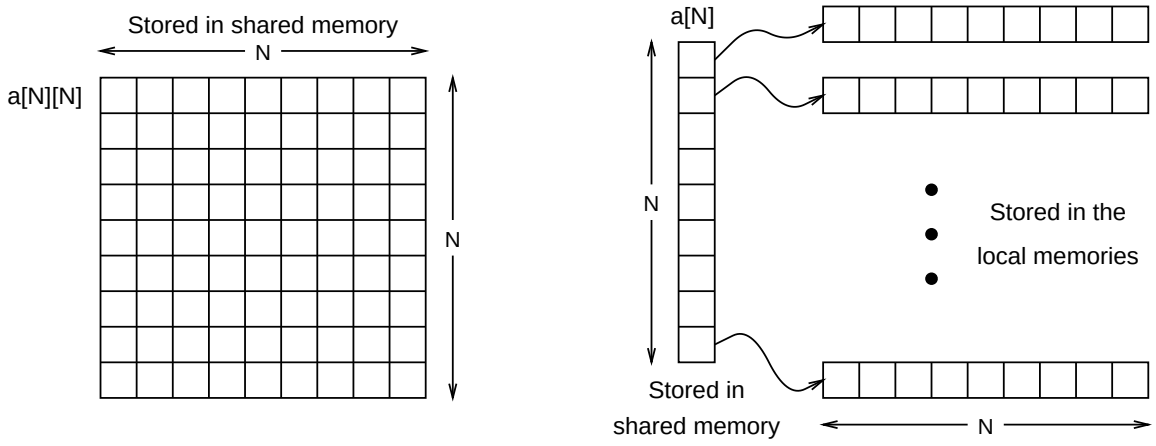
The implementation described here is not optimal; in fact, every array access involves a non-local access to the global pointer array. This is easily fixed by copying the pointers to local memory. We are currently experimenting with this and other implementations of partitioned data structures, and plan to define new language constructs to make their use easier based on the results.

## The Meaning of Parallelism

Although the *ParC* constructs identify activities that may run in parallel, a more precise definition is required in order to fully define the semantics as well as giving some indication as to the performance characteristics of the constructs. The phrase "execute in parallel" is used in many different ways, and most parallel languages leave the exact semantics up to the implementation, obtaining an *ad hoc* operational definition. We feel that the issue is far too important to be thus ignored. The ambiguities arise from the fact that the number of physical processors in any machine is limited and the fact that in MIMD mode the processors may execute at varying speeds. To resolve them, we must characterize the way in which *ParC* activities are scheduled. Thus we define which implementations give legal operational semantics.

The problem is especially severe in a shared memory model like that assumed by *ParC*. Computations in functional languages, for example, also create a tree of activities. But due to the fact that functional languages prohibit side effects, there are no interactions between these activities. Therefore the scheduling decisions may have some effect on performance, but do not affect the outcome of the computation. Computations based on explicit message passing let the run-time system know when one process is waiting for another. With shared memory, on the other hand, interactions are mediated by side effects. The run-time system cannot know that a busy-waiting process is not doing useful work. Good programs can avoid undesired scenarios by using synchronization mechanisms supplied by the system to regulate the interactions. But it is also important to define what will happen if the synchronization is omitted, or implemented by the programmer.

It is now fairly common for parallel machines to adhere to the dictates of *sequential consistency* in order to facilitate our understanding of parallel programs [21]. This requirement states that the results of parallel execution of a program are the same as the results that would be obtained had the instructions from distinct




```
/* whole array declared; inefficient */
int  a[N][N];

mapped lparfor ( k; 0; N−1; 1 )
{
   initialize(a[k]);
}

mapped lparfor ( k; 0; N−1; 1 )
{
   int  i;
   /* some use of owned array row */
   if (k == 0 || k == N−1) pcontinue;
   for (i=1; i<N−1; i++) {
      a[k][i] = f(a[k−1][i], a[k][i−1],
                  a[k+1][i], a[k][i+1]);
   }
}
```

```
/* partitioned array; efficient */
int  *a[N];

mapped lparfor ( k; 0; N−1; 1 )
{
   a[k] = (int *) malloc( N*sizeof(int) );
   initialize(a[k]);
}

mapped lparfor ( k; 0; N−1; 1 )
{
   int  i;
   /* some use of owned array row */
   if (k == 0 || k == N−1 ) pcontinue;
   for (i=1; i<N−1; i++) {
      a[k][i] = f(a[k−1][i], a[k][i−1],
                  a[k+1][i], a[k][i+1]);
   }
}
```

Figure 8: *The left side of the figure illustrates the usual two dimensional array in shared memory. In the right side the array is partitioned among the processors' memories. This is achieved as shown in the ParC code. Its elements are referenced as any two dimensional array and fits in well with the C notion of vector of pointers to vectors. The mapped version of* lparfor *ensures that the indexed activities will always be mapped to the same physical processor.*

parallel activities been interleaved and executed in some serial ordering. So if instructions $A$ and $B$ execute in parallel, the results are as if either $A$ was executed before $B$ or $B$ before $A$.

The ideal semantics of a parallel construct would completely define the interleaving. This would lead to a deterministic execution of shared-memory parallel programs. For example, we could require that the interleaving emulate a priority CRCW PRAM: this is done by executing one instruction at a time from each activity in a cycle, with the activities ordered lexicographically. Regrettably, it is not practical to require such semantics. The overhead involved in enforcing them would by far outweigh the benefits of parallelism.

The problem is that if the interleaving is not completely defined, the program execution is indeterminate. As a consequence, distinct executions of the same program may lead to different results, and even to different behaviors. For example, one execution may spawn many more activities then another, or one execution may terminate with a result while another enters an infinite loop. It is therefore impossible to specify the exact semantics of *ParC* programs. In the absence of formal semantics, we make do with a specification of a set

```
int  x=1;
parblock
{
    while (x==1) /* do nothing */
        ;
::
    x = 0;
}
```

Figure 9: *An example of an extreme case where it is critical that the programmer understand the underlying execution model. In a nonpreemptive system, this code might never terminate.*

of rules to guide the implementation of a *ParC* system.

As noted above, the manner of interleaving instructions from different activities may have an effect on program termination, in addition to the obvious effect on access to shared variables. The implementation rules attempt to reduce the probability of adverse effects such as deadlock and non-termination. They try to fit the intuitive understanding of what "executing in parallel" means, thus (hopefully) reducing the chance of unpleasant surprises. We show that this is equivalent to a requirement for fairness.

**Preemption of Executing Activities**

Using a multiprocessor with $P$ processors, only $P$ activities can execute at any given time. The pool of ready activities, however, can be expected to be larger than $P$. The implementation rules define when and in what order activities join and leave the set of $P$ currently executing activities.

There are two options for the removal of an activity from the executing set: (i) voluntary, letting it run until it suspends, requests to be removed explicitly, or terminates, or (ii) preemptive, forcing it to relinquish the processor after a certain time quantum. Preemption ensures a degree of fairness between the activities. It is equivalent to a situation in which all the activities execute all the time, but use asynchronous processors. Thus it is guaranteed that none are delayed indefinitely while others proceed. The arguments for and against preemption are the following:

*For preemption*

- The user should perceive each activity as a virtual processor, and should be able to create interactions and interdependencies between these virtual processors. An extreme example is given in the code segment of Figure 9: Without preemption, code such as this might deadlock. Note that all the software algorithms for mutual exclusion are like this [17].
- Correctness (e.g. termination) should not depend on the number of processors that are available, even though performance may be effected. The user should not be forced to take this into account.
- The goal is to shift the burden from the programmer to the compiler and system. For example, the above code segment will be efficient if gang scheduling is used [22].
- Naive programmers might expect too much from a system that guarantees that activities "run to completion". For example, lack of preemption does not guarantee mutual exclusion in parallel systems. Minor changes such as adding print statements for debugging might cause activities to be preempted unexpectedly, surprising the users. Such effects lead the designers of Amoeba to comment that "probably the worst mistake in the design of Amoeba 4.0 process management mechanisms was the decision to have threads run to completion, that is, not be preemptable" [23].

*Against preemption*

```
int  x=1;
main()
{
    parblock{
       f(&x)
    ::
       x = 0;
    }
}
void f(int *y)
{
    if (*y==1)
       parblock {
          f(y);
       ::
          f(y);
       }
}
```

```
int  x=1;

main()
{
    f(x);
}


void f(int y)
{
    if ( y<1000 )
       parblock {
          f(y+1);
       ::
          f(y+1);
       }
}
```

Figure 10: *Examples of codes that support the use of breadth-first execution (on the left) and depth-first execution (right). In either case, using the other execution method might lead to an explosion of activities.*

- Only users knows what is going on, so they should have full control over the system. The system should not surprise the user by suddenly preempting one task and scheduling another, because then the user would not be able to analyze the performance of the program.
- As the model is asynchronous, the programmer must use explicit synchronization to regulate the execution. Code without explicit synchronization need not be supported as there is no formal notion of correctness [24].
- Preemption introduces additional overhead due to context switching, and degrades cache performance [25].

It seems that preemption would provide a friendlier environment for the naive user, but might annoy a seasoned parallel programmer. The implementation rules for *ParC* are therefore a compromise. The default rule is that preemption be used. However, sophisticated users that want full control may override this rule by using the yield, switch_on, and switch_off instructions described in the section on synchronization.

### Activity Selection

The second question is the order in which activities join the executing set. If the number of activities exceeds $P$, some activities will have to wait before they can start execution. The pool of waiting activities can be organized as a FIFO queue or a LIFO stack. Viewing the tree of activities that are generated, this choice will determine whether the tree is traversed in a breadth-first or a depth-first order[1].

The implications of the order of execution are subtle. Consider the two code segments shown in Figure 10. The argument for breadth-first ordering is based on the code on the left. This code might deadlock despite the use of preemption if depth-first ordering is used, because new activities are generated all the time and they prevent the execution of the only activity that can terminate the program. Actually this scenario will probably cause a system failure due to an infinite recursion. The argument for depth-first ordering is

[1] Note that this is an abuse of terminology, as $P$ branches of the tree are traversed in parallel in any case.

also based on problems in dealing with deep recursion. For example, the code on the right would require $2^{1000}$ activities to be started if it is done breadth first, but only a thousand if it is done depth first. Hence in this case it is breadth-first that will probably lead to system failure.

While both of these examples lead to the same effect (the generation of huge numbers of activities), there is a difference. The code on the right is looking for trouble by explicitly asking for the creation of $2^{1000}$ activities; using depth-first is a system optimization that might get it out of trouble despite itself. The code on the left, on the other hand, is perfectly reasonable if we use a natural extension to the notion of fairness that motivates the use of preemption. The extension requires that fairness be maintained not only between independent activities, but also between branches in the tree of activities. In other words, an activity is charged not only for its own execution but also for the execution of its descendents. When a certain activity and its descendents execute for too long, the whole branch is preempted, giving other branches a chance to execute. Therefore the breadth-first approach is advocated for *ParC*.

Note that fairness is advocated as a practical matter, not a theoretical issue. We extend the work done on fairness in nondeterministic systems, such as CSP, and claim that it is important also for indeterministic systems[2]. Regrettably, even with fairness it is impossible to define the semantics of an indeterministic shared-memory program. However, in many cases fairness will improve program behavior and ease the task of parallel programming.

# Experience with *ParC*

*ParC* has been in use for several years at Hebrew University and has served as the vehicle for the annual course on parallel algorithms. In this section, we review this experience, and the strengths and weaknesses of the language.

*ParC* has lived up to its charter of providing a platform for undergraduate students to gain experience with parallel programming. The parallel algorithms course has successfully trained both graduate and undergraduate computer science students. The main exercises in parallel programming usually could be accomplished by identifying the most intensive inner loops of an application and replace the for loop with the parfor construct. While this simple modification rarely leads to significant speedups, it reveals the challenges of parallel processing. One is immediately faced with the problem of collecting the partial results of each iteration. The faa primitive makes this task easier. For example, the conjugate gradient method has been programmed and achieved a speedup between 3 and 5 using 8 processors using this simple method. The large variance in speedup is due to the sparsity of the input matrix and to the different convergence speeds from the parallel and sequential versions.

## Two detailed examples

*ParC* is both simple enough to quickly get a working program with reasonable performance and rich enough to allow a knowledgeable programmer to squeeze out additional performance. The following examples show how performance-enhancing transformations may be applied to a parallel program. The main idea is to start with a simple parallelization, and then to refine it to get better performance. In the first example the simple parallelization possible with *ParC* gives good performance to begin with, and extensive additional optimizations accrue only limited benefits. In the second example the simple parallelization is not enough, and additional work is needed to improve memory locality. In both cases, it is easy to express the optimizations using features of *ParC*.

```
int A[N], j;
for (j=lg(N)−1; j>=0; j−−) {
    parfor ( i; 0; 2**j−1; 1 )
        add( &A[i], A[i+2**j] );
}
```

Figure 11: *A simple ways to sum N numbers in ParC. The final sum is contained in A[0]. Since the iterates are independent, an* lparfor *construct should be used.*

```
int A[N];
parfor ( i; 0; N/2−1; 1 ) {
    int j;                          /* private due to scoping rules */
    for (j=lg(N)−1 ; j>=0 ; j−−) {
        if (i>=2**j) pcontinue;     /* terminate top half */
        add( &A[i], A[i+2**j] );
        sync;                       /* explicit barrier each iteration */
    }
}
```

Figure 12: *An optimization to avoid the explicit generation and termination of activities during each of the lg(N) iterations. But many systems are unable to maintain a large number of waiting activities.*

**Vector Summation Example**

Consider the calculation of the sum of the elements of an array. For simplicity, assume that both the size of the array N, and the number of processors P, are powers of two[3]. The basic parallel algorithm is to sum element pairs, then pairs of pairs, and so on, resulting in a tree structure. It is easier to express this algorithm if the pairs are not taken as adjacent elements, but rather as elements that are half an array apart. The code presented in Figure 11 is a *ParC* version of this procedure (with relaxed notation, e.g. use of $2{**}j$ for exponentiation).

This formulation is convenient in that it includes an implicit synchronization at the end of each iteration, because a new parfor is spawned for each one. But this costs the overhead associated with the parfor, which is much higher than the overhead of synchronization alone. An lparfor construct can be used to reduce this overhead.

Another potential transformation is to avoid the creation and termination of the activities in each iteration. This is achieved by reorganizing the code and putting the sequential for loop *inside* the parfor, adding an explicit synchronization at the end of each iteration. This version is shown in Figure 12. Note that now an lparfor cannot be used, because the sync introduces a dependency between the activities.

The number of activities is halved in each iteration and the sync instruction "knows" the number of participating activities. However, in many cases the array is much larger than the number of processors and so, since an lparfor construct cannot be used, it is then advisable to *chunk* the computation by hand. This is done by spawning only P activities and have each one take care of a number of values. Indeed, for large values, the code in Figure 12 could not execute on the Makbilan as there were too many activities for the system to maintain.

The code in Figure 13 implements the "chunking" optimization. The original lg(N) iterations are now

[2] Nondeterminism occurs when a certain language construct allows an arbitrary choice between a number of options. Such constructs exist in several message passing languages, e.g. the select statement in Ada and the ALT construct in Occam. Indeterminism is when the choices are not explicit, and the program behavior can change at any moment due to differences in the execution rates of different processes. This may happen in shared memory programs that do not synchronize accesses to shared variables.

[3] *ParC* provides the number of processors in a variable named proc_no. We use P here for short.

```
int  A[N], P=proc_no;                    /* the number of actual processors */
parfor ( i; 0; N/2−1; N/2P ) {
   int  j, k, cnk=N/2P;                  /* private due to scoping rules */
   for (j=lg(N)−1; j>=0; j−−) {
      if ((j<lg(P)) && (i>=2**j))
         pcontinue;                      /* terminate top half in last lg(P) iterations */
      for (k=i; k<i+cnk; k++)            /* loop on private chunk */
         add( &A[k], A[k+2**j] );
      if ( j > lg(P) ) {                 /* shift to lower part of the array in first phase */
         i = i/2;
         cnk = cnk/2;
      }
      sync;                              /* explicit barrier each iteration */
   }
}
```

Figure 13: *The final hand-optimized code. It does slightly better than the first* lparfor *version.*

| | Time (Speedup) | | | | |
|---|---|---|---|---|---|
| Number of PEs | 1 | 2 | 4 | 8 | 16 |
| Sequential | 2620(1.00) | | | | |
| Sum1 (parfor) | 6380(0.41) | 3315(0.79) | 1885(1.39) | 950(2.76) | 485( 5.40) |
| Sum1 (lparfor) | 2660(0.98) | 1360(1.92) | 865(3.03) | 440(5.95) | 235(11.15) |
| Sum3 | 2635(0.99) | 1340(1.96) | 845(3.10) | 425(6.16) | 220(11.91) |

Table 1: *The result of adding together 4096 numbers (with an expensive add operation to compensate for the memory conflicts). The first line is the time using a simple* for *loop. The two lines labeled "Sum1" are from the code in Figure 11, and "Sum3" is from the code in Figure 13.*

split into two phases. In the first lg(N)−lg(P) iterations, there are more activities than array elements. Each activity then loops on the array elements that are assigned to it (this is the loop with index k). The assignment changes from one iteration to the next, as the partial sums are created in the lower half of the array. In the final lg(P) iterations, the number of activities is halved with each iteration, and the assignment does not change any more, as it was in the previous example.

We implemented these four variants on the Makbilan parallel machine. In order to compensate for the problems with memory contention, we increased the complexity of the addition operation (a dummy loop of 500 iterations was the overhead per each call to the add routine). The experiments consisted of summing together 4096 numbers. The second version could not run because of the large number of activities that had to be maintained. The results are shown in Table 1. We see that the simplest parfor version was indeed rather slow, but the lparfor version came quite close to the sequential time for 1 processor and with 16 processors got a speedup of more than 11 times the best sequential time (2620/235). The hand optimized version with chunking only did slightly better than that, indicating that using lparfor can save significant amount of work.

### A Matrix Multiplication Example

As another example, consider matrix multiplication where the arrays are partitioned between the local memories of the processors. Using the mapped version of lparfor, a number of arrays may be manipulated in parallel.

```
int **A, **B, **C;
void init()
{
    A = (int **) malloc(N); B = (int **) malloc(N); C = (int **) malloc(N);
    mapped lparfor (i; 0; N−1; 1) {
        int j;
        A[i] = (int *) malloc(N); B[i] = (int *) malloc(N); C[i] = (int *) malloc(N);
        for (j=0; j<N; j++) {
            A[i][j] = init_A(i,j); B[i][j] = init_B(j,i);
        }
    }
}
```

Figure 14: *Initialization of three partitioned arrays for the matrix multiplication example.*

```
mapped lparfor ( i; 0; N−1; 1 )
{
    int  k, t, sum, *a, *b, *c;

    a = A[i];
    c = C[i];

    for (k=0; k<N; k++) {
        b = B[k];
        sum = 0;

        for (t=0; t<N; t++)
            sum += a[t] * b[t];
        c[k] = sum;
    }
}
```

```
mapped lparfor ( i; 0; P−1; 1 )
{
    int  j, k, t, sum, *a, *b, *b_rem, *c;
    int  bot, top;

    bot = i*N/P;
    top = bot+N/P;
    b = (int *) malloc(N);

    for (k=bot; k!=(bot−1)%N; k=(k+1)%N) {
        b_rem = B[k];
        for (j=0; j<N; j++)
            b[j] = b_rem[j];

        for (j=bot; j<top; j++) {
            a = A[j];
            c = C[j];
            sum = 0;
            for (t=0; t<N; t++)
                sum += a[t] * b[t];
            c[k] = sum;
        }
    }
}
```

Figure 15: *Two versions of matrix multiplication. The first assumes nothing about the machine, using the* mapped lparfor *construct. The second knows that it is better to exploit and reuse local memory. It allocates just one activity per physical processor, and copies each column of B to local memory before using it.*

| | Time (Speedup) | | | | |
|---|---|---|---|---|---|
| Number of PEs | 1 | 2 | 4 | 8 | 16 |
| MM (lparfor N) | 32740(1.00) | 20580(1.59) | 14045(2.33) | 8080(4.05) | 8175( 4.00) |
| MM (lparfor P) | 31990(1.00) | 16210(1.97) | 10215(3.13) | 5075(6.30) | 2570(12.45) |

Table 2: *The result of matrix multiplication with partitioned arrays. The first row shows the behavior of the simple coded version (left side of Figure 15). Improved performance can be attained through better use of local memory and attempts to avoid memory conflicts (right side of Figure 15). Note that the speedups are relative to the execution on a single PE, not relative to a separate sequential version.*

Figure 14 shows how the partitioned arrays are allocated. Note that the mapped lparfor construct ensures that partition $i$ will be local to activity with index $i$. Matrices $A$ and $C$ are stored in row-major, and each partition is interpreted as a band of rows. Matrix $B$ is stored in column major.

We present two versions of the body of the matrix multiplication code. The first assumes nothing about the machine, using the mapped lparfor construct. The results (first row of Table 2) show some improvement in run time as the number of processors increases, but it levels of at a speedup of 4 for 8 PEs and then starts to degrade. The reason is that while all accesses to $A$ and $C$ are local, accesses to $B$ are almost always remote.

The second version exploits and reuses local memory by allocating just one activity per real processor (the initialization must be similarly modified, but is not shown). Then, each column of the array $B$ is copied from shared memory to a local array. It is then used for the entire slice of the array $A$ that is local. Note that the code is parametrized by P, the real number of physical processors, and thus adjusts to the available parallelism. Significant improvements result, as seen from the second row of Table 2; a speedup of almost 12.5 for 16 processors.

## Other Applications

Students have also gotten significant speedups on more complex applications, e.g. a speedup of 10.9 using 16 processors for an FFT application. Here again, the implementation was fairly simpleminded: a butterfly network was simulated in shared memory and an activity was created for each of the nodes in the network. Message passing between the nodes was done in shared memory with flags used to indicate the presence of data.

As expected, sorting programs were popular among students. One interesting variant on sorting tried to avoid the bottlenecks on the shared bus of the Makbilan. The *ParC* program limited its parallelism to the parallelism inherent in the system, i.e. the number of processors. Each processor radix sorted its assigned data and then only a few were allowed to distribute their buckets to the other processors. In the mean time, the other processors continued radix sorting with their buckets, so that the data transferred would be more sorted. Surprisingly, this method was not noticeably better than allowing all the processors access to the bus. This is due to the fact that the local work is so large.

Another interesting application parallelized the usual fractal demonstration program. Several different methods were tried. Initially, each process generated its own strip. But due to the differing amounts of work, load balancing was added to the program. Each process calculated its rate of progress and compared it to the global average. If it was too slow, it split its remaining work and placed part of it in a global workpile. When a process was finished with its work, it would take extra work from the global workpile. This scheme was compared to the automatic balancing within the *ParC* runtime system by simply spawning off a new activity with the extra work. Because of context switching overheads, it turned out to be better to do the load balancing within the program. *ParC* made it very easy to explore these alternatives.

Most students finished when they had working, (mostly) correct implementations. A few continued to investigate improvements. These improvements mostly were in exploiting memory locality, programming load balancing, and avoiding excessive process creation and termination. Finally, since the course was attended by computer science students, there was a stronger desire to develop programming and environmental tools

for parallel programming rather than to run applications themselves. Computer scientists rarely care about the result of a program — the program execution holds more interest than its output. As a consequence, there were debuggers, visualizers, runtime systems, and automatic load balancing systems developed within the *ParC* environment.

Alas, debugging *ParC* programs was not an easy task, although we believe that there are features that make it easier to debug *ParC* programs than parallel programming languages based on message passing. Many *ParC* programs can be easily serialized: simply replace the parfor statement with a for statement. Most of the bugs have been simple ones, but were hard to track down because students suspected subtle timing bugs or strange parallel execution possibilities, overlooking the simple possible causes. For the most part, the common *ParC* bugs have to do with confusion between what is local and what is global — e.g. having a for loop with a global index variable inside a parallel construct.

# Comparison With Other Languages

There are many parallel machines in existence and various ways of programming them. Many parallel languages do not use an imperative style; rather, the parallelism is implicit in the way the computation is carried out, as in functional and logic languages [26, 27], or else is is left as a challenge for the compiler [28]. As *ParC* is an imperative language with explicit control parallelism, we will only compare it with other languages of the same type.

In a nutshell, *ParC* may be characterized as a language providing a shared-memory, MIMD, asynchronous model of computation, with dynamic spawning of parallel activities, and hierarchical scoping rules that provide a sense of locality. We know of no other language with all these features. The following paragraphs list various parallel languages and point out the differences between them and *ParC*.

## Programming Model

Most of the imperative parallel languages are designed for distributed architectures rather than for shared memory machines [1]. The Occam [29], Ada [30], Joyce [31], Concurrent C [32], and Cosmic C [33] languages, for example, provide facilities for synchronous or asynchronous message passing between parallel processes. This allows for efficient implementations on both shared-memory and message-passing architectures. However, programs written in these languages cannot utilize the full possibilities afforded by shared-memory architectures, because they do not allow shared variables.

Languages like Linda [34] or Swarm [35] provide a shared data space, which is like *ParC*, but they lack scoping and a sense of locality. This is partly due to the fact that these languages support a content-addressable associative memory, rather than a traditional location-addressable shared memory as in *ParC*. Split-C is another language that supports a non-traditional shared memory model [36, 37]. Access to the shared address space is mediated by active messages, and specifically asynchronous "put" and "get" of memory blocks to and from other PEs. The accessing PE can then check (of busy wait) on a local flag to see when the put or get completes. Hence assignment from a remote location is a split-phase operation.

The pC language uses the Mach task/thread model: threads in the same task share a flat address space, but cannot access the address spaces of other tasks [38]. This gives a two-tier memory. Concurrent C allows users to use the available shared memory on uniprocessor and shared-memory multiprocessor implementations, for fear that users would refuse to use the system if this was prohibited [39]. No language support such as scoping is given, and worse, implementations that lack shared memory (e.g. LAN-based) cannot execute programs that rely on this feature.

In fact, languages designed explicitly for shared memory MIMD machines seem to be rather scarce. Many installations use simple thread packages, that allow multiple threads that share the whole address space. On the other hand, there do exist a number of SIMD languages that effectively allow data sharing through pointers or array indexes [40, 41, 42, 43]. The main limitation in these languages is that all the parallel computations are identical, and must proceed in lockstep. This is relaxed in some recent languages such as HPF [44], where the computations are only loosely synchronous.

Finally, some parallel languages are tailored very closely to a specific architecture. This is especially typical in languages designed for vector machines, where the desire is usually for high vector performance.

This implies very simple and regular structures that fit the machine's vector registers. Clarity, portability, and ease of programming are sacrificed [45, 46]. While this can result in efficient code for loops, it is not a general approach, and specifically does not support the parallelism of divide-and-conquer algorithms.

### Expressing the Sharing Patterns

As noted above, some parallel languages for the shared memory model allow all activities to access the full address space. Therefore there are no means to define sharing patterns. Examples include HPF [44] and other dataparallel languages, and well as the Linda Tuple Space [47, 34].

Some partitioning of the address space is provided by the two-level task/thread scheme used in pC [38]. All threads created within the context of the same task share that task's address space. However they cannot access the address spaces of other tasks. In addition, accessibility is not necessarily correlated with locality. Split-C exposes the locality by providing special operations (put and get) to access remote memory [36, 37]. A number of systems go a step further and introduce special annotations that tag variables as private or shared [48, 49]. This is useful as an indication of where the variable should be stored in a NUMA architecture. For example, on the Cedar system the degree of sharing dictates whether a variable is stored in a cluster memory or in the global memory [50, 51].

An alternative approach is to initially allow all variables to be shared, but during execution to limit the access to certain activities as needed. Obviously this is practiced wherever mutual exclusion is used, but here we are talking of having special constructs for it in the language. One example is the use of check-in/check-out operations [52]. Another example is the "with" construct in Jade [53, 54].

Scoping rules like those used in *ParC* were proposed for programming the NYU Ultracomputer [55].

### Expressing the Parallelism

It is often desirable that parallel programs should express the parallelism inherent in the algorithm and not what is physically available, to allow efficient implementations on different architectures [59]. Therefore languages should be capable of expressing massive parallelism. This can be done by parallel loop constructs and recursion. Surprisingly, many parallel languages place various limitations on the spawning of parallel activities. Occam does not allow recursion, making it practically impossible to code divide-and-conquer algorithms [29]. Those languages that do have a parallel loop construct do not use it to define scoping boundaries. For example, Occam provides closed parallel constructs that are similar to those of *ParC*, but it does not support shared variables. Fortrans with a parallel-do construct typically assume that there are no interactions between iterations (like *ParC*'s lparfor) [45].

Some languages have low-level unstructured support for parallelism. Dijkstra has observed that closed parallel constructs (such as those provided by *ParC*) promote a structured style, just like the closed control structures advocated for sequential languages [17]. While this idea has caught on in sequential languages, and the use of goto has all but disappeared, most parallel languages still supply fork and join primitives instead of closed constructs; examples include Ada [30] and Concurrent C [32]. In many cases, such as the C-threads package [56] and MPC [14], the support for parallelism is limited to a direct interface to the run-time system.

Many other systems have no direct support for expressing parallelism within the language — they assume a copy of the program is executed on each processor (the SPMD paradigm) [57]. For example, this is the case in Concurrent Pascal [58] and in Split-C [36, 37], where it is required because the use of active messages depends on the fact that all nodes have the same memory image.

Two systems that provide capabilities somewhat similar to those of *ParC* are pC and the split-join model. pC has a parfor construct with exactly the same syntax as the *C* for loop, which is more general that the parfor in *ParC*. The price is that the implementation must first execute the control sequentially to figure out how many iterations there are, and then spawns the threads [38]. The split-join model takes the opposite approach. Rather than using parallel constructs to create more activities, such constructs are used to split the existing activities into groups that will work on different tasks [60]. The total number of activities is constant and limited to the number of processors. When the tasks are finished the groups re-join to create the original larger group.

*ParC* is unique in the way it deals with shared memory, and integrates scoping, synchronization, and forced termination with the block structure that is used to express the parallelism. Note that the *ParC* constructs are part of the block structure of the language, just like serial loops or conditionals. Thus each activity is just a block of code (a compound statement), as opposed to languages in which parallelism is created by defining processes or modules (e.g. Ada [30] and Joyce [31]). Thus *ParC* is suitable for the expression of very fine-grain parallelism. Of course, activities may also be quite large and may call functions.

### Expressing synchronization

Synchronization mechanisms in shared memory environments typically fall into two categories. One is mechanisms associated with specific access to a specific shared data structure, in order to guarantee consistency. Examples include all the various mechanisms to ensure mutual exclusion, such as semaphores [17], monitors [58], locks [61, 62], check-in/check-out [52, 63], and with [53, 54]. The other category is mechanisms used to delimit program phases, once it is known that there are no data races within each phase. Examples include the barrier synchronization [18, 64], rendezvous [30, 65], and join [66].

Another classification distinguishes among synchronization mechanisms according to their effect on the activity that uses them. Specifically, there is a dichotomy between those mechanisms that cause the activity to block, and those that allow the activity to just probe the synchronization condition. If the condition is not met, the activity can then proceed with some other work. Many blocking mechanisms were imported from concurrent languages that were designed to ease the programming of multitasking uniprocessor systems [67, 14]. In such systems blocking is the only reasonable alternative. Recently there is much interest in wait-free primitives, which are more suitable to truly parallel systems [68]. Examples include various read-modify-write instructions such as test-and-set [69], fetch-and-add [15], and compare-and-swap [70].

*ParC* provides a repertoire of low level primitives, that cover *different* synchronization behaviors, both blocking (semaphores and barrier) and wait-free (fetch-and-add). High-level primitives may be added if programming experience indicates that certain constructs are especially useful.

## Conclusions

*ParC* is a promising *C* language extension for coding shared memory parallel algorithms. Unlike various macro packages that just provide access to lightweight processes supported by the system, *ParC* integrates the parallelism into the block structure of the language. This results in a natural pattern of shared and private variables, based on the conventional scoping rules of *C*, and also allows for high-level instructions to terminate whole branches of a parallel program and to synchronize a set of activities. To summarize, the salient features of *ParC* are:

- Closed parallel constructs that can be freely nested.
- Control over allocation and mapping of activities that does not involve detailed knowledge of the underlying architecture.
- No explicit declarations of shared variables — variables are shared when parallel activities can see them.
- High level primitive for activity and group termination.
- Ability to exploit distributed, shared, NUMA memory of many MIMD machines.

The ideal semantics for a *ParC* program are that all the activities actually execute in parallel on distinct processors. However, the number of activities in a program may surpass the number of physical processors that are available. When some activities are blocked from execution, deadlock may ensue if there are interdependencies between executing and blocked activities. We advocate the use of preemption and breadth-first execution of the activity tree to provide a degree of fairness that would avoid such situations.

The basic features of *ParC* were first defined in 1985, and a simulator that provides support for them has been in use since 1986 (for historical reasons, the actual syntax of *ParC* as recognized by the original

compiler is marginally different from that described here). This system includes a capacity to link libraries and *C* code, a graphical debugger, execution control, and exact time measurements. A run time system that supports all the *ParC* features described in this paper on the Makbilan research multiprocessor has been operational since 1991. It is being used for research on runtime systems and as a programming platform for parallel algorithms classes.


### Acknowledgements

*ParC* was first developed by Yosi Ben-Asher. Some extensions were later added by Izhar Matkevich and Dana Ron. The compiler and simulator were written by Yosi Ben-Asher, Marcelo Bilezker, and Itzik Nudler. The initial port to the Makbilan multiprocessor was done by Omri Mann and Coby Metzger. An improved run-time environment on the Makbilan was implemented by Dror Feitelson, Moshe Ben Ezra, and Lior Picherski. Forced termination was implemented by Yair Friedman. Dror Zernik and Danny Citron wrote a package for monitoring and graphical display of program execution. Martin Land has kept the hardware up and running. Larry Rudolph has supervised the project since its inception. Lively discussions between the authors and Dror Zernik, Moshe Ben Ezra, Lior Picherski, and Sharon Broude, all members of the Makbilan research group, were material in defining the semantics of various parallel constructs.


## References


[1] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, 'Programming languages for distributed computing systems'. *ACM Comput. Surv.* **21**, 261–322 (1989).

[2] C. M. Pancake, 'Multithreaded languages for scientific and technical computing'. *Proc. IEEE* **81**, 288–304 (1993).

[3] T. Merrow and N. Henson, 'System design for parallel computing'. *High Perform. Syst.* **10**, 36–44 (1989).

[4] G. C. Fox, 'Applications of parallel supercomputers: scientific results and computer science lessons'. In *Natural and Artificial Parallel Computation*, M. A. Arbib and J. A. Robinson (eds.), chap. 4, MIT Press, 1990.

[5] Y. Ben-Asher, G. Haber, and L. Rudolph, 'Improving parctical parallel code using SIMPARC: a parallel C simulator'. Manuscript, Institute of Computer Science, The Hebrew University, Jerusalem (1992).

[6] D. G. Feitelson, Y. Ben Asher, M. Ben Ezra, I. Exman, L. Picherski, L. Rudolph, D. Zernik, 'Issues in run-time support for tightly-coupled parallel processing'. In *Symp. Experiences with Distributed and Multiprocessor Systems (SEDMS) III*, 27–42 (1992).

[7] L. Rudolph, D. G. Feitelson, I. Exman, D. Zernik, Y. Ben Asher, M. Ben Ezra, and L. Picherski, 'Run-time support for parallel language constructs in a tightly coupled multiprocessor'. Technical report manuscript, Institute of Computer Science, The Hebrew University, Jerusalem (1993).

[8] B. Shaibe, 'Performance of cache memory in shared-bus multiprocessor architectues: an experimental study of conventional and multi-level designs'. Master's Thesis, Institute of Computer Science, The Hebrew University, Jerusalem (1989).

[9] D. Zernik and L. Rudolph, 'Animating work and time for debugging parallel programs - foundation and experience'. In *ACM ONR Workshop on Parallel and Distributed Debugging*, 46–56 (1991).

[10] Y. Ben-Asher and D. G. Feitelson, 'Performance and overhead measurements on the Makbilan'. Technical Report 91-5, Dept. Computer Science, The Hebrew University of Jerusalem (1991).

[11] C. D. Polychronopoulos and D. J. Kuck, 'Guided self scheduling: a practical scheduling scheme for parallel supercomputers'. *IEEE Trans. Comput.* **C-36**, 1425–1439 (1987).

[12] S. F. Hummel, E. Schonberg, and L. E. Flynn, 'Factoring: a method for scheduling parallel loops'. *Comm. ACM* **35**(8), 90–101 (1992).

[13] T. H. Tzen and L. M. Ni, 'Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers'. *IEEE Trans. Parallel & Distributed Syst.* **4**, 87–98 (1993).

[14] D. Vrsalovic, Z. Segall, D. Siewiorek, F. Gregoretti, E. Caplan, C. Fineman, S. Kravitz, T. Lehr, and M. Russinovich, 'MPC - multiprocessor C language for consistent abstract shared data type paradigms'. In 22nd *Ann. Hawaii Intl. Conf. System Sciences*, I:171–180 (1989).

[15] A. Gottlieb, B. Lubachevsky, and L. Rudolph, 'Basic techniques for the efficient coordination of very large numbers of cooperating sequential processes'. *ACM Trans. Prog. Lang. Syst.* **5**, 164–189 (1983).

[16] C. P. Kruskal, 'Algorithms for repalce-add based paracomputers'. In *Intl. Conf. Parallel Processing*, 219–223 (1982).

[17] E. W. Dijkstra, 'Co-operating sequential processes'. In *Programming Languages*, F. Genuys (ed.), pp. 43–112, Academic Press, 1968.

[18] P. O. Frederickson, R. E. Jones, and B. T. Smith, 'Synchronization and control of parallel algorithms'. *Parallel Computing* **2**, 255–264 (1985).

[19] J. K. Ousterhout, 'Scheduling techniques for concurrent systems'. In 3rd *Intl. Conf. Distributed Computing Systems*, 22–30 (1982).

[20] S. F. Hummel and E. Schonberg, 'Low-overhead scheduling of nested parallelism'. *IBM J. Res. Dev.* **35**, 743–765 (1991).

[21] L. Lamport, 'How to make a multiprocessor computer that correctly executes multiprocess programs'. *IEEE Trans. Comput.* **C-28**, 690–691 (1979).

[22] D. G. Feitelson and L. Rudolph, 'Gang scheduling performance benefits for fine-grain synchronization'. *J. Parallel & Distributed Comput.* **16**, 306–318 (1992).

[23] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum, 'Experiences with the Amoeba distributed operating system'. *Comm. ACM* **33**(12), 46–63 (1990).

[24] S. V. Adve and M. D. Hill, 'Weak ordering - a new definition'. In 17th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, 2–14 (1990).

[25] J. C. Mogul and A. Borg, 'The effect of context switches on cache performance'. In 4th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, 75–84 (1991).

[26] P. Hudak, 'Conception, evolution, and application of functional programming languages'. *ACM Comput. Surv.* **21**, 359–411 (1989).

[27] E. Shapiro, 'The family of concurrent logic programming languages'. *ACM Comput. Surv.* **21**, 413–510 (1989).

[28] D. Klappholz, A. D. Kallis, and X. Kong, 'Refined C: an update'. In *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua (eds.), 331–357, MIT Press (1990).

[29] INMOS Ltd., *Occam Programming Manual.* Prentice-Hall, 1984.

[30] United States Department of Defence, *Reference Manual for the Ada Programming Language.* ANSI MIL-STD-1815, 1983.

[31] P. Brinch Hansen, 'A multiprocessor implementation of Joyce'. *Software — Pract. & Exp.* **19**, 579–592 (1989).

[32] N. H. Gehani and W. D. Roome, 'Concurrent C'. *Software — Pract. & Exp.* **16**, 821–844 (1986).

[33] C. L. Seitz, 'Concurrent architectures'. In *VLSI and Parallel Computation*, R. Suaya and G. Birtwistle (eds.), chap. 1, Morgan Kaufmann Publishers, Inc., 1990.

[34] S. Ahuja, N. Carriero, and D. Gelernter, 'Linda and friends'. *Computer* **19**(8), 26–34 (1986).

[35] G-C. Roman and K. C. Cox, 'Implementing a shared dataspace language on a message-based multiprocessor'. In 5th *Intl. Workshop Software Specification & Design*, 41–48 (1989).

[36] T. von Eiken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, 'Active messages: a mechanism for integrated communication and computation'. In 19th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, 256–266 (1992).

[37] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, 'Parallel programming in Split-C'. In *Supercomputing '93*, 262–273 (1993).

[38] R. Canetti, L. P. Fertig, S. A. Kravitz, D. Malki, R. Y. Pinter, S. Porat, and A. Teperman, 'The parallel C (pC) programming language'. *IBM J. Res. Dev.* **35**, 727–741 (1991).

[39] R. F. Cmelik, N. H. Gehani, and W. D. Roome, 'Experience with multiple processor versions of Concurrent C'. *IEEE Trans. Softw. Eng.* **15**, 335–344 (1989).

[40] J. R. Rose, 'C*: a C++-like language for data parallel computation'. In *Usenix C++ Papers*, 127–134 (1987).

[41] W. D. Hillis, *The Connection Machine*. MIT Press, 1985.

[42] R. H. Perrott, D. Crookes, and P. Milligan, 'The programming language ACTUS'. *Software — Pract. & Exp.* **13**, 305–322 (1983).

[43] J. T. Kuehn and H. J. Siegel, 'Extensions to the C programming language for SIMD/MIMD parallelism'. In *Intl. Conf. Parallel Processing*, 232–235 (1985).

[44] D. B. Loveman, 'High Performance Fortran'. *IEEE Parallel & Distributed Technology* **1**, 25–42 (1993).

[45] A. H. Karp, 'Programming for parallelism'. *Computer* **20**(5), 43–51 (1987).

[46] A. H. Karp and R. G. Babb II, 'A comparison of 12 parallel Fortran dialects'. *IEEE Software* **5**(5), 52–67 (1988).

[47] D. Gelernter, 'Generative communication in Linda'. *ACM Trans. Prog. Lang. & Syst.* **7**, 80–112 (1985).

[48] H. F. Jordan, 'The Force'. In *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass (eds.), 395–436, MIT Press (1987).

[49] M. A. Nichols, H. J. Siegel, and H. G. Dietz, 'Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler'. *IEEE Trans. Parallel & Distributed Syst.* **4**, 222–234 (1993).

[50] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua, 'Restructuring Fortran programs for Cedar'. In *Intl. Conf. Parallel Processing*, I:57–66 (1991).

[51] U. Meier and R. Eigenmann, 'Parallelization and performance of conjugate gradient algorithms on the Cedar hierarchical-memory multiprocessor'. In 3rd *Symp. Principles & Practice of Parallel Programming*, 178–188 (1991).

[52] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, 'Cooperative shared memory: software and hardware for scalable multiprocessors'. In 5th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, 262–273 (1992).

[53] M. S. Lam and M. C. Rinard, 'Coarse-grain parallel programming in Jade'. In 3rd *Symp. Principles & Practice of Parallel Programming*, 94–105 (1991).

[54] M. C. Rinard, D. J. Scales, and M. S. Lam, 'Jade: a high-level, machine-independent language for parallel programming'. *Computer* **26**(6), 28–38 (Jun 1993).

[55] J. Edler, A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Teller, and J. Wilson, 'Issues related to MIMD shared-memory computers: the NYU Ultracomputer approach'. In 12th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, 126–135 (1985).

[56] E. C. Cooper and R. P. Draves, *C Threads*. Technical Report CMU-CS-88-154, Dept. Computer Science, Carnegie-Mellon University, Jun 1988.

[57] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, 'A single-program-multiple-data computational mode for EPEX/FORTRAN'. *Parallel Computing* **7**, 11–24 (1988).

[58] P. Brinch Hansen, 'The programming language Concurrent Pascal'. *IEEE Trans. Softw. Eng.* **1**, 199–207 (1975).

[59] L. G. Valiant, 'A bridging model for parallel computation'. *Comm. ACM* **33**(8), 103–111 (1990).

[60] E. D. Brooks, III, B. C. Gorda, K. H. Warren, and T. S. Welcome, 'Split-join and message passing programming models on the BBN TC2000'. Technical Report UCRL-ID-107022, Lawrence Livermore National Laboratory (1991).

[61] L. Rudolph and Z. Segall, 'Dynamic decentralized cache schemes for MIMD parallel processors'. In 11th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, 340–347 (1984).

[62] M. Heuser, 'An implementation of real-time thread synchronization'. In *Proc. Summer USENIX Technical Conf.*, 97–105 (1990).

[63] D. A. Wood, S. Chandra, B. Falsafi, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, S. S. Mukherjee, S. Palacharla, and S. K. Reinhardt, 'Mechanisms for cooperative shared memory'. In 20th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, 156–167 (1993).

[64] R. Gupta, 'The fuzzy barrier: a mechanism for high speed synchronization of processors'. In 3rd *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, 54–63 (1989).

[65] N. Gammage and L. Casey, 'XMS: a rendevous-based distributed system software architecture'. *IEEE Software* **2**(3), 9–19 (1985).

[66] R. S. Nikhil, G. M. Papadopoulos, and Arvind, '*T: a multithreaded massively parallel architecture'. In 19th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, 156–167 (1992).

[67] G. R. Andrews and F. B. Schneider, 'Concepts and notations for concurrent programming'. *ACM Comput. Surv.* **15**, 3–43 (1983).

[68] M. Herlihy, 'Wait-free synchronization'. *ACM Trans. Prog. Lang. & Syst.* **13**, 124–149 (1991).

[69] J. E. Burns, 'Mutual exclusion with linear waiting using binary shared variables'. *SIGACT News* **10**, 42–47 (1978).

[70] H. S. Stone, *High-Performance Computer Architecture*. Addison-Wesley, 2nd ed. (1990).