Run-Time Support for Parallel Language Constructs in a Tightly Coupled Multiprocessor

Dror G. Feitelson* Yosi Ben-Asher† Moshe Ben Ezra
Iaakov Exman Lior Picherski Larry Rudolph Dror Zernik‡

Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
E-mail contact: rudolph@cs.huji.ac.il

Abstract

This paper describes the run-time implementation of a parallel programming language. Unlike traditional designs, our system exploits both shared memory aspects and message passing features. It enjoys the benefits of both polling and interrupts, giving more weight to the former, i.e. processors do not interrupt each other unless absolutely necessary.

The language/system interface deals with groups of parallel activities as whole, so as not to impose unnecessary serialization on the language implementation. Parallel block-oriented and other constructs are implemented on top of a real-time operating system. Algorithms and data structures for the distribution of newly spawned activities and for the termination of activities via parallel `break` and `return` instructions are described. Performance measurements are given to compare between possible algorithms and explain the behavior of selected ones.

Keywords: groups of activities, group creation, parallel termination, dynamic load adaptation, shared memory architecture, non-uniform memory access.

*Current address: IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598

†Current address: Department of Mathematics and Computer Science, University of Haifa, 31999 Haifa, Israel.

‡Current address: Department of Electrical Engineering, The Technion, 32000 Haifa, Israel.

1 Introduction

Parallel programming is the activity of delineating the instructions to be carried out by a number of processors and their co-relations.

Low-level parallel programming facilities, such as C-threads [8], require the programmer to fork parallel activities one at a time. If many parallel activities are required, the programmer must consider, e.g. spawning them in a loop or using a logarithmic tree structure [29].

Alternatively, a high-level parallel programming language may be used. Such languages typically include constructs for spawning all the required activities at once, thus delegating the question of an efficient implementation to the runtime system.

The responsibility and delineation of system services in a parallel environment is still the subject of much research and this paper argues that they should be divided among the parallel program, a runtime system, and an operating system kernel.

In particular, the paper describes the implementation of such system calls in MAXI, the Makbilan runtime system. The Makbilan research multiprocessor is a shared-memory, bus-based machine, which is used to execute programs written in a locally defined parallel programming language, PARC[20, 4]. MAXI contains novel techniques that are widely applicable, although it is specifically tailored for the Makbilan and for the shared-memory constructs of PARC, to serve as an environment for research and comparison of alternative algorithms.

This paper discusses support for manipulations of group of threads. The operations which have to be provided include: fine grain parallelism, spawning large number of threads, quick termination of groups, and varying amount of computation of threads. Their efficient implementation is achieved by:

- *A distributed design based on polling rather than interrupts:* Each processor tries to minimize its interactions with other processors, so as not to exhaust communication facilities that might be required by the user application. Most of the interactions are done by polling shared data structures when the processor is ready, rather than having the processors interrupt each other.

- *Dynamic process creation via a global queue and multitasking via local queues:* A global queue is used for balanced distribution of the work when new activities are spawned, but local queues are used for efficient scheduling of activities that have been created

already. Processors access the global queue at a rate which is inversely proportional to the size of their local queue.

- *Dynamic adaptation to the amount of concurrency:* Since activities may be numerous and very short, it is not desirable to always create a task or process for each activity. The notion of envelopes is introduced to adapt to the right level of concurrency. An envelope is a task that is controlled by the low-level kernel and in which activities are executed. One envelope may execute many short-lived activities. On the other hand, if the activities of a parallel program are all heavy and long-lived, or if they are interdependent, then the number of envelopes will become equal to the number of activities and their scheduling will be managed by the real-time kernel.

- *Efficient support for parallel constructs that terminate groups of activities:* The closed parallel constructs of the programming language require new, non-obvious implementations to support the ability for one activity to terminate whole groups of related activities. In ParC, a `break` or `return` operation can be executed in the middle of a nested parallel construct thereby terminating the entire construct. The participating activities are identified either by checking the structure of the activity tree or by using a special coding scheme.

- *Locality in memory references:* In a NUMA machine exploiting the local memory is a key issue in achieving efficiency. The run-time system supports a variety of mechanisms to improve this aspect.

ParC and its runtime system are intended to provide a general purpose convenient environment for users. A basic guideline in the system design is the desire to shift some of the burden of parallel programming from the user to the system. User programs enjoy the full benefits of a shared-memory environment, however, they are not allowed full control over the processors and memory mapping. Instead, the programs issue high-level ParC instructions and leave the implementation details to the system (see Figure 2). For example, the execution of a `parfor` construct invokes Maxi to spawn the required number of activities and map them to processors; the program does not specify on which processor the parallel activities are to be executed. The design and evaluation of system algorithms for this and other problems is the main part of our research.

The runtime system supporting the language constructs can be implemented either in the operating system kernel, or else it can be implemented as a user-level software package

above the kernel [1, 27]. We have chosen to build our system on top of a real-time operating system kernel. The runtime system should have more control over tasks and other system resources than is traditionally permitted in multi-user, interactive, time-sharing operating systems such as Unix. Instead of modifying such an operating system kernel, a real-time system kernel provides essentially the same control and requires significantly less work. It also has superior performance.

Our focus was on the issues of activity creation, scheduling, and termination. Despite the relative ease of implementing the runtime system above an existing real-time kernel, we did not develop a full featured operating system. For example, we did not handle multiprogramming, memory management, or parallel I/O. Sequential I/O was provided by the Unix host, which also maintained the file system.

The next section provides the necessary background about the Makbilan multiprocessor and the ParC language. Section 3 introduces the underlying design of the Maxi runtime system. Sections 4 and 5 present the Maxi implementation of activity creation, and Section 6 describes the implementation of forced termination.

2 Background

Our model of parallel computation can be described as non-uniform access, shared memory, MIMD. In addition, the parallel machine is dedicated to executing a single (parallel) job.

We have designed our runtime system for a particular machine platform, but this platform is indicative of a large number of parallel machines. The specific machine architecture is outlined in the first subsection. On top of each processor is a real-time kernel that was intended for uniprocessor applications. The second subsection reviews the features that are relevant to our runtime system. Our maxi runtime system supports programs written in our local parallel language, although any language with calls to our runtime library can be supported. The relevant features of our language, ParC, are presented in the third subsection.

2.1 The Makbilan Testbed

The Makbilan parallel computer has been assembled at the Hebrew University and has been operational for several years. The features that are relevant to the runtime system are outlined below:

- The Makbilan research multiprocessor consists of up to 15 single-board computers in a Multibus-II cage[1].
- Each board has an i386 processor running at 20 MHz, providing about 4 MIPS[2]. A mathematical co-processor, a message passing co-processor, and 4MB of memory are also part of each board.
- Memory on remote boards may be accessed through the bus, thus supporting a shared-memory model. As access to on-board memory is faster than access to memory on remote boards, Makbilan is a non-uniform memory access (NUMA) machine [3]. The processors have on-board caches, but they do not cache remote references. Hence there is no issue of cache coherence.
- The box also includes one board that acts as a Unix host, and boards for a bus controller, a peripherals interface, and a terminal controller. Users log on to the Unix board, and can then load and execute ParC programs on all the other boards.

All memory addresses are translated by the paging mechanism; however, it is assumed that the whole usable address space is located in physical memory. There is no paging to disk. This not only simplifies the runtime system, it also provides for faster program execution. As 3 of the 4 MB of memory on each board are used for application data, heap, and stacks, an application executing on a 16-processor system has a total of 48 MB at its disposal.

2.2 The Real-Time Operating System Kernel

In addition to the off-the-shelf hardware, our parallel processor uses an off-the-shelf real time kernel for many of the operating system functions on each processor. The local kernel is Intel's RMK [17], which is a real-time kernel designed to use hardware support provided by the i386 and the Multibus-II. This kernel is highly optimized to provide fast task creation and context switching. In fact, context switching in RMK is faster than the context switching instruction provided in the i386 instruction set, because the kernel manipulates the hardware data structures directly [17, p. 7-35]: we clocked it at 72μs [3]. Parallel activities

[1] i386 and Multibus-II are Intel trademarks.

[2] Recently, a smaller configuration with the newer and fast i486 processors also became operational. It uses the same Maxi runtime system.

are implemented by RMK tasks (which are the RMK equivalents of Unix processes). The overhead for task creation and termination is about 1ms [3]. The scheduling time quantum was set to 50ms in MAXI.

A real-time kernel, like RMK, has many advantages over the usual academic choice of a Unix kernel because of several reasons. The primary reason is that it is highly efficient and allows most of the CPU cycles to be devoted towards executing the parallel program. In addition, the kernel allows more involvement in the system state by the higher levels of software; in our case by the runtime system.

Task priorities, memory mapping, and stack allocation can be controlled, and a high-level interface to the interprocessor interrupts facility is available. The RMK scheduler maintains a set of task queues organized by priorities. A *fence* is specified separating priorities that allow preemptive scheduling from those that require nonpreemptive scheduling. Below the fence, tasks at the highest priority level are scheduled in a round robin, time sliced fashion. If a task above the fence becomes ready, it preempts the running task and runs until it yields the processor, or until a task with a still higher priority becomes ready. The MAXI system changes the priority of tasks to ensure that they will not be interrupted by PARC activities.

The kernel also makes it easy to provide virtual memory without paging. The MAXI system does a static allocation of virtual address to physical address at system configuration time.

Almost all of the low level interrupt handling facilities are left to the real-time kernel. This includes the basic message passing primitives, the clock functions, and the handling of error conditions. Once again, this removes much of the burden of system development. Finally, the real time kernel is small in size, both in terms of data structures and in actual code. The kernel is replicated in each processors local physical memory. The RMK kernels are local and maintain local data structures, e.g. the local run queue. At this level, there is no notion of parallelism and interaction among the processors. All such interactions are done by MAXI, and are described in the next section.

2.3 The PARC Language

PARC is a superset of the C programming language, designed and implemented by our group and intended to support parallel programming in a shared memory environment [20, 4]. A primary goal of PARC was to facilitate parallel programming by novice programmers; in particular, undergraduate students proficient in C should be able to write parallel programs

```
prefix( arr, beg, end )
int arr[], beg, end ;
{
    int m;
    if (beg ≥ end) return;
    m = beg + (end - beg + 1)/2 -1;
    parblock
        { prefix( arr, beg, m ); }
        :
        { prefix( arr, m+1, end ); }
    epar
    parfor int i; m+1; end; 1;
    {
        arr[i] = arr[i] + arr[m];
    }
    epar
}
```

Figure 1: *Implementation of parallel prefix computation using divide and conquer in* ParC, *illustrates the use of* `parfor` *and* `parblock`. *The* `epar` *statement terminates the parallel constructs.*

after one class or by reading a short document.

The main extensions to the C language are two block-oriented parallel language constructs, `parblock` and `parfor`; the first indicates that the constituent sub-blocks execute in parallel, while the second indicates that iterations of the loop body be done in parallel. Each sub-block or iteration is called an *activity*. These constructs may be nested in arbitrary ways, creating a tree of activities where all the leaves are executing in parallel. The program in Figure 1 illustrates the use of these constructs.

The activities do not need to be independent of each other, but any dependencies must be explicitly enforced by synchronization statements. There are three main synchronization mechanisms. The first is a fetch-and-add instruction, denoted `faa`, that provides an "atomic

add to memory" operation. The second synchronization mechanism is semaphores. This allows activities to suspend execution when waiting for a certain event. The third is the `sync` instruction, which implements a barrier synchronization among all the activities created by a certain parallel construct. The semantics of this instruction is that if the instruction does not appear in the code of all of the activities of the parallel construct, the activities in which it does appear are forced to wait for the termination of activities in which it does not.

The `parfor` construct has another version, called `lparfor`, which stands for "light" `parfor`. Using this construct indicates to the compiler that the activities are light-weight and independent, and allows it to generate optimized code. The activities are broken into a number of chunks, one per processor, and each chunk is executed serially without additional overhead for work distribution and activity creation. The `lprafor` is designed to provide a static tool for load balancing. Note that `lparfor` constructs can be freely intermixed with `parfor` and `parblock` constructs.

As a rule, an activity that performs a `parblock` or `parfor` is suspended until all its children terminate. `pbreak` and `preturn` statements may be used to exit these constructs prematurely, in analogy with `break` and `return`. A `pbreak` breaks out of the innermost enclosing construct, kills all the parallel activities created by the construct (and their descendants), and resumes the parent. `preturn` returns from the last function call, and kills all the activities that were created in this function.

The ParC memory model includes both shared and private data structures. The accessibility of the data structures is determined by static scoping rules. Thus a variable that is declared within the block of code that defines a certain activity is accessible only by that activity and its descendants. It is local to the activity in which it is declared. Global variables are shared by all the activities. All this is implemented by the ParC compiler, which sets up pointers between the stacks of the different activities [4]. Therefore no runtime support is needed for handling the variety of types of memory accesses.

The Makbilan is a NUMA machine since each processor has local memory, shared memory is distributed, and bus accesses are priority based. In order to allow shared memory objects to be associated with activities (which is important in a NUMA architecture), a *mapped* `lparfor` construct is provided. Recall that an `lparfor` chunks activities so that the number of chunks equals the number of processors. When the mapped version is used, the serial number of the processor that executes each chunk is guaranteed to match the serial number of the chunk. Mapped mode for `parblock` and `parfor` constructs can be provided only for

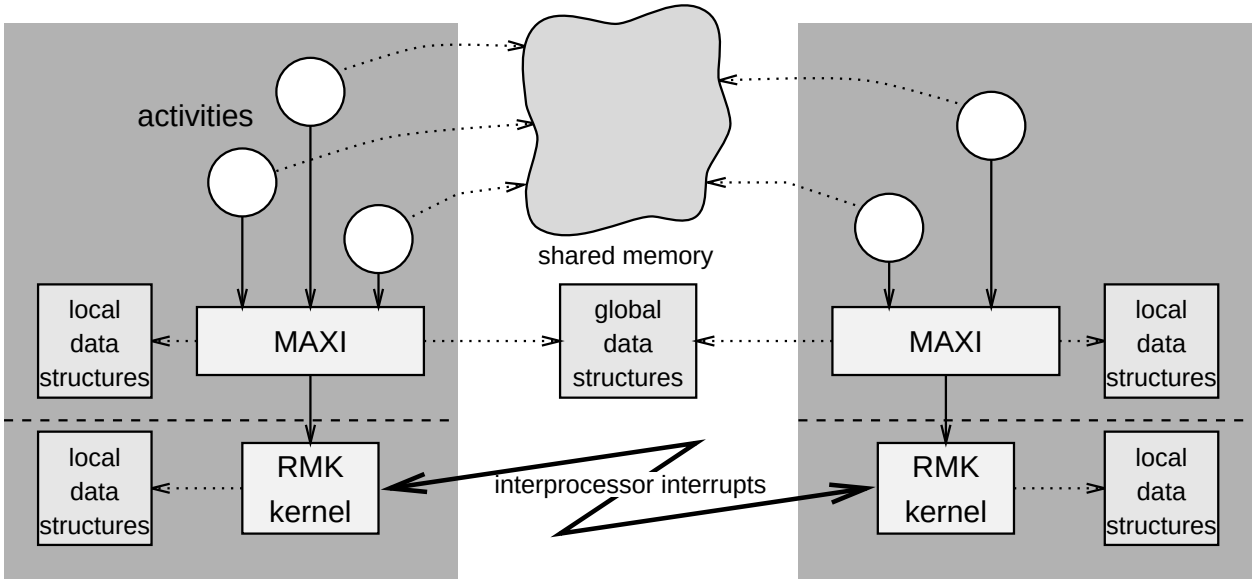

Figure 2: *Interactions among components of the application and* MAXI. *Solid vertical arrows denote system calls, and dotted arrows denote access to data structures.*

the light version `lparfor` since the other constructs may generate more activities than the available processors. `malloc` can be used to allocate shared memory which is local to the allocating activity.

3 Runtime System Design

The **Mak**bilan **Sy**stem, called MAXI for short, is the interface between the parallel program and the real-time operating system kernel executing on each processor. By and large, MAXI is in charge of creating, terminating, and mapping each of the activities specified in the parallel program. The MAXI runtime system supports PARC parallel constructs by using the services of the local RMK kernel on each board (Figure 2). The actual scheduling is also done by RMK. Note that the local RMK kernel is not an inherent part of the system: alternative kernels that support local task manipulation can be used. Because all machine specific details, including interrupt handling, are hidden in the underlying kernel, our MAXI runtime library code is fairly portable.

The design philosophy of MAXI differs from traditional runtime designs for parallel machines which have exploited either the shared memory aspect of the machine or the efficient

message passing features but never both.

Parallel operating systems for shared memory, MIMD parallel machines with uniform access to the shared memory usually maintain system data structures in the shared memory. Each processor accesses and updates these structures as necessary. Of course, appropriate locks are required. For example, process scheduling is done via a central global task queue where processors insert and delete tasks during context switch operations [10]. Dynamic memory allocation schemes also work in a similar fashion.

On the other hand, operating systems for parallel machines with distributed memory and efficient message passing mechanisms usually employ a distributed processing approach. That is, each processor maintains private system data structures. Processors communicate via message passing and inter-processor interrupts. In many cases, these facilities are used to provide high-level remote procedure calls (RPC) [26]. This allows the kernel itself to be parallelized [27], so that one processor to directly influence other processors' actions by issuing remote procedure calls. For example, threads (activities) can be created, deleted, suspended, or resumed on a remote processor. Note that the affected kernel may be required to field many such remote requests at once. Moreover, these requests might interfere with the local computation carried out on the processor. The detrimental effect of such interference exists even in high performance parallel machines which allow only one process per processor, because much coordination is still needed for operations such as load balancing and parallel I/O.

In our case, the target architecture is a nonuniform-access shared-memory system. It is significantly cheaper for each processor to access the memory situated on its own board than to access memory situated on another processor's board. Thus, it is preferable to maintain private data structures. Similarly, it is desirable for activities to be scheduled on the processor that has fast access to the activity's data structures, e.g. its stack. That is, once an activity is assigned to a processor, it should not migrate.

The MAXI design is based on the observation that if all of the processors are already busy, they should not interrupt each other: it is easier and just as efficient to let each processor work locally, and check for external requests when it is ready to service them. This is a legitimate approach since all activities have equal priorities. There is no reason to give a request from a remote activity precedence over the computation of a local activity. If certain processors are idle, they continuously check for external requests, and efficiency is not compromised.

System	Location	Object	Section
MAXI	global	list of new spawn descriptors	4.1
		active spawn descriptors used for `sync`	4.2
		per-processor lists of terminated spawn descriptors	4.2
	local	Process Control Blocks and Tables	4.2
		lists of spawn descriptor representatives	5.2
RMK	local	run queue	2.2
		interrupt vector	6.1

Table 1: *Main data structures in* MAXI *and* RMK

Interrupt mechanisms are expensive in terms of performance. Thus, it is preferable to use a polling scheme for communication between processors. Under MAXI, a processor that generates a service request places information in an agreed upon location; other processors periodically check this location and as a result may take appropriate action and update their local data structures. As a consequence, MAXI code need not be reentrant and does not need to maintain a large number of locks.

However, polling is not always adequate. MAXI makes use of the RMK interprocessor interrupt facility so that one processor can cause all the others to perform some action in unison. The main use of this facility is to `pbreak` out of a parallel construct or to `exit` the program altogether.

The main data structures used in MAXI are listed in Table 1, where they are classified as global or local. The main RMK data structures are also noted, mainly to show that they need not be part of MAXI. These data structures are further explained in the following sections.

4 Mapping and Scheduling of Activities

PARC specifies the activities that may be executed in parallel and RMK provides the task management on each processor, but it is the job of MAXI to map activities to processors. Recall that with the `parfor` construct, it is possible to create many activities, each executing essentially the same code but with a different activity index. Also recall that the parent

activity is suspended until all of the child activities terminate. Thus, the main runtime support required for such constructs is the distribution and execution of the activities, as well as reporting back upon their termination. This section describes the simple mechanism to achieve these goals; the next section then discusses some optimizations.

There is a wide spectrum of choices in the scheduling and allocating of activities to processors, especially when the number of activities may be much larger than the number of processors. Of course each has its own advantages. One extreme possibility is to use *self-scheduling* from a single global workpile[3]. Each processor executes an activity for a single time quantum and then replaces the activity back onto the workpile and chooses the next activity from the workpile [10, 19]. This approach has been applied in the past to parallel machines with uniform access times to memory. So, the fact that the mapping of activities to processors is not preserved — it is highly unlikely that an activity will be executed by the same processor for two consecutive time quantums — is not a drawback. The advantage of this scheme is that it provides an efficient load sharing mechanism that does not allow any processor to be idle while there is work to be done. On the other hand, the global workpile can easily become a serial bottleneck. Moreover, in a NUMA architecture, it is difficult to take advantage of the memory structure.

At the other extreme is *local scheduling* with each processor maintaining it own *local workpile*, and the activities are mapped to processors via program command [23]. The advantages are numerous. At the system level, local queues are considered more efficient because activities do not move from one board to another. Thus state information such as the activity's stack is always local and does not have to be copied. Local queues also reduce contention for global locks, which could degrade performance [2]. At the application level, the fact that activities do not migrate also opens the possibility for associating data structures with activities, so that these data structures are always in local memory (this is important for any NUMA architecture). For example, a program can create a shared array such that different parts of the array are local to different activities executing on distinct processors.

MAXI leans towards this second approach in that there is a static mapping of activities and local workpiles are used. However, the mapping happens at run-time and depends on the system state and load. The advantages of local queues also mesh nicely with the fact

[3]We use the term *workpile* instead of the more common term *task queue* since parallel access to the workpile violates the strict definition of a FIFO queue.

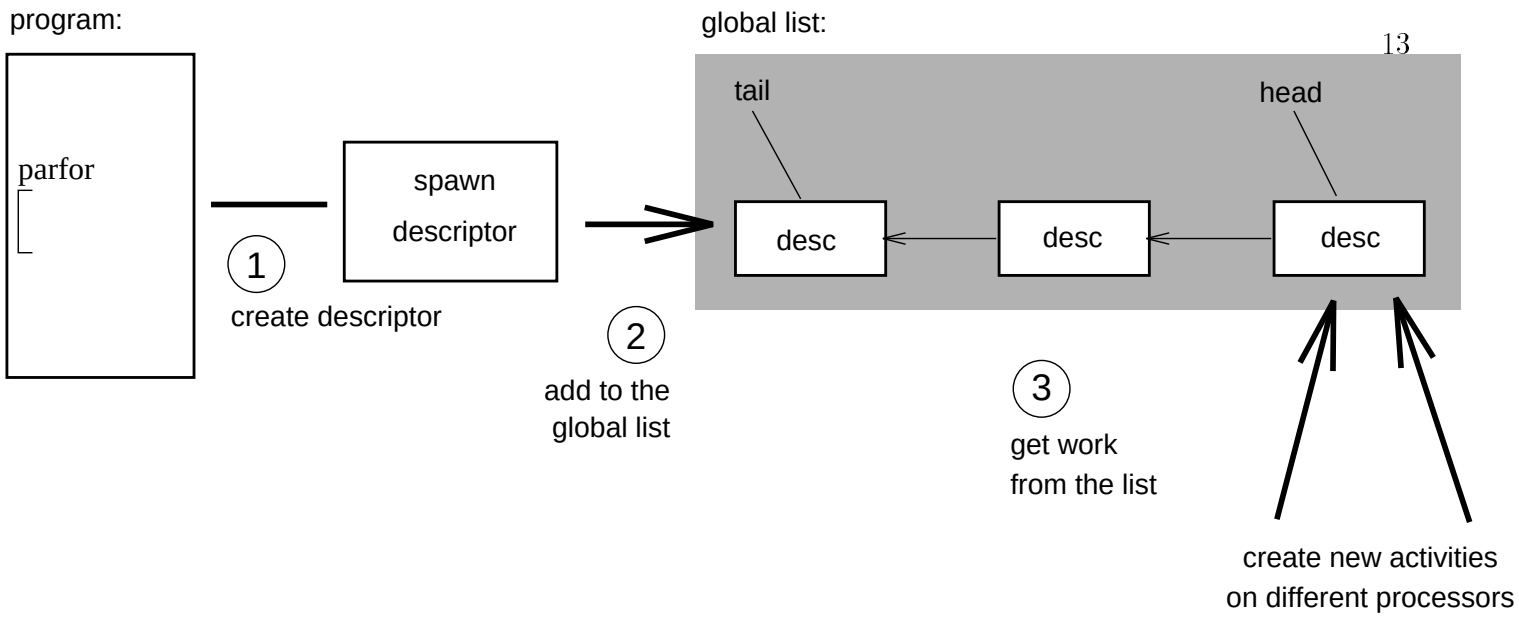

Figure 3: *Implementing spawns by a global list of descriptors.*

that RMK is based on local queues, and add impetus to its use. Had we decided to use a global queue we would have been unable to use RMK, or impose severe restrictions on its usage. A closely related approach appears in [11], but in MAXI we provide a scheduling scheme which is more sensitive to the load on each processor.

The rest of this section first describes how activities are allocated to processors at run-time. It then addresses several straightforward but important implementation details.

4.1 Activity Allocation

In designing a strategy for activity allocation to processors, we wanted to ensure that in the extreme situations, the allocation strategy did the correct mapping. For example, when the number of activities is not greater than the number of processors and the activities are computationally heavy, then each activity should be mapped to its own processor. In addition, when the number of activities is much greater than the number of processors, and each activity is short (e.g. less than a time quantum), then the system should provide good load balance.

As we do not use run-time migration for load balancing, it is important to balance the initial mapping of activities to processors. The MAXI implementation achieves load balancing in a straightforward manner, by using a global list of spawned activities, from which all the

processors take additional work. To accomplish this, each processor has a special activity, called `get_work`, that belongs to the run-time system. Whenever it is scheduled, it takes one new activity from the global list. As the `get_work` activity competes with other activities for the processor, the rate at which it is scheduled depends on the load on the processor. Thus it will be scheduled more often on lightly loaded processors, causing them to take more additional work [21].

Individual activities are not placed in the global list as this would incur a large overhead when a large number of light activities are spawned at once. Instead, each item on the list is a data structure called a *spawn descriptor* (Figure 3). It specifies the number of activities that should be spawned, the code that they should execute, and possibly a pointer to an arguments list that should be passed to them. The descriptor also includes a counter of the number of activities that have terminated, additional data structures used in the implementation of synchronization primitives, and a pointer to the parent activity (this is used to resume the parent when the last activity terminates). Whenever a parallel construct is executed, a new spawn descriptor is added to the tail of the list. New activities are generated from the descriptor at the head of the list. The head and tail are locked independently, allowing spawns and activity creation to proceed concurrently. As activities are always generated from the oldest descriptor, the activity tree is executed in breadth-first order[4]. This ensures a degree of fairness in the execution [4].

The effectiveness of the load balancing scheme is demonstrated by our measurement results from the following experiment. The experiment was performed on a 10-processor configuration. A total work of 100,000,000 assignments to the same global variable was divided equally among different numbers of activities. In the five decompositions shown in Figure 4, the program divided the assignments among either 10^2, 10^3, 10^4, 10^5, or 10^6, activities. For each number of activities used, the figure shows the total time required and the way the activities came to be distributed among the processors. The important point to notice is that the best performance is achieved when the number of activities on different processors are not the same.

When there were only 100 activities, each doing 1,000,000 assignments, the load was divided equally: each processor executed exactly 10 activities. However, the total execution time was about 100 seconds, which was sub-optimal, as witnessed by the fact that when there were a few thousand activities the execution time was reduced by 15%, requiring about 85

[4]Once again, due to the parallelism, it is not strictly breadth-first, but the parallel analog to it.

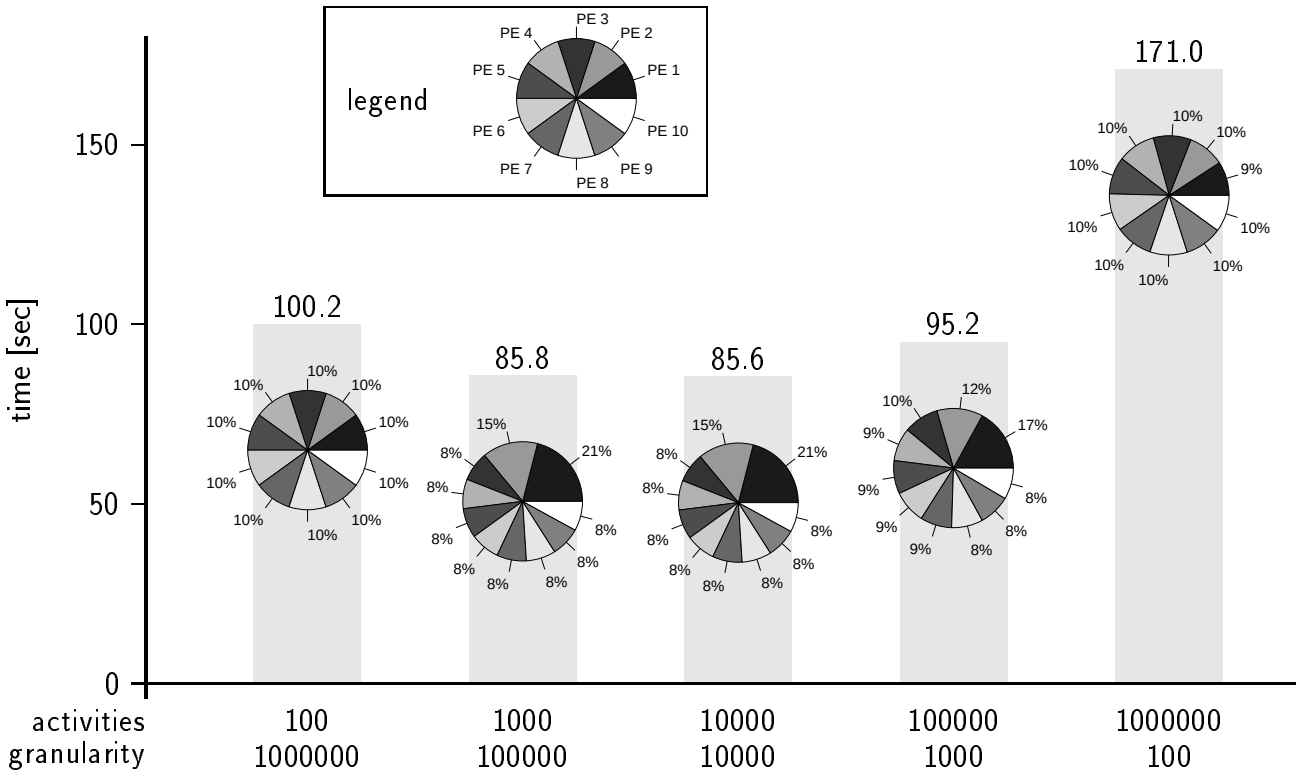

Figure 4: *Results of load balancing experiment. The time is the parallel completion time. Percents do not always add up to 100 because of rounding error. It is therefore best to divide 100,000,000 operations into several thousand activities when 10 processors are used.*

seconds (the minimum occurs for about 1900 activities). The distribution of the work shows that this reduction was the result of non-equal loads: processor #1 executed about $2\frac{1}{2}$ times as many activities as processors 3 through 10. The reason for this behavior is that processor #1 is more efficient, because both the spawn descriptor and the global variable happen to be in its local memory. Such a non uniform cost of accessing a variable is typical for NUMA architectures, and must be considered at the system design level as well. When the pool of activities is big enough, processor #1 takes more of them, executes them faster, and thus reduces the total execution time. Processor #2 is more efficient than the others because a processor's bus priority depends on its serial number.

The higher execution time for small numbers of activities results from two effects. If only 10 activities are spawned, each processor executes one of them. Processor #1 might finish his earlier then other processors, but then it is left with nothing useful to do because all the other activities have been picked up already. But the measurement for 100 activities shows

that actually processor #1 is not more efficient than the others in this case. The reason is that all the other processors are executing activities that access the global variable in the first processor's memory at a high rate. This high volume of remote accesses slows the first processor down. When there are thousands of other activities, the other processors take more time off for activity creation and termination, thus giving the first processor a chance to get ahead.

When the number of activities is very large (1,000,000 activities each doing only 100 assignments), memory contention again degrades the performance of the first processor. In this case the contention is for the spawn descriptor, not the global variable. The additional overhead for spawning activities is also considerable, and therefore the performance deteriorates dramatically. In particular, processor #1 now manages a bit less work than the others, because the contention for the spawn descriptor loads its local memory. Indeed, the main drawback of the global list is that it has to be locked when activities are added to or deleted from it, thus serializing the process of activity creation to some degree. Note, however, that the serialization is not complete because allocating local data structures for the activity can be done in parallel. Moreover, this drawback is well worth it because the global list allows the processors to balance their loads based on their relative performance, rather than just based on the number of activities that each executes.

The main lesson from this experiment is that in complex systems it is extremely hard to anticipate the interaction among the various factors that affect performance. In our case, these factors included the non-uniform memory access times, the contention for the bus and the memory module, the different bus priorities, and the detrimental effect of remote accesses on local memory accesses. Although different machine architectures will have different sets of complex interactions, they all share the feature that the effects of the complex interactions are usually very difficult to predict.

Striving for load balancing by dishing out equal numbers of activities to the different processors is not enough, because the above factors cause the processors to have widely varying efficiencies. In our opinion, a better solution is to use a dynamic and adaptive system, that does the balancing at run time based on performance, rather than just keeping the number of activities equal. But in order to be effective, the system must be given enough activities to distribute; specifically, the application must include more than the minimal number of activities, that is more than one per processor. However, the extreme of too many very fine-grained activities should also be avoided. As a rule of thumb, the length of

each activity should be at least several times longer than the overhead required to generate it, which is of the order of a thousand shared memory accesses.

4.2 Other Implementation Details

We present a few more ParC construct implementation details in order to show our use of polling and low run-time system overhead. The first example is the implementation of the mapped `lparfor` construct, the second how we implement barrier synchronization, and the last is how the parent activity is revived after all the child activities terminate.

ParC provides for finer control of the mapping of activities to processors to allow the programmer to optimize the use of local memory. The mapped `lparfor` construct states that the system will always use the same function for mapping activities to processors each time this construct is invoked. To meet this constraint, the `get_work` task cannot just pick the first new activity it sees; it must ascertain that the activity may be run on the same processor.

The implementation is trivial. A "mapped" field is added to the spawn descriptor. If the spawn descriptor represents a mapped `lparfor`, and only in this case, the first bit of this field is set. The rest of the bits are used as a bit mask to show which activities have been created already. When `get_work` sees that the construct is mapped, it checks its bit in this mask. If it is set, this activity has been created already, so `get_work` abandons this spawn descriptor and looks for the next one. If the bit is not set, the activity is created and the bit is set for future reference.

The second example is that of barrier synchronization. The ParC construct is simply the statement `sync`, and its semantics require that no activity continues its execution past this construct until all activities have executed the `sync` construct. Uniprocessor and distributed processing systems generally use an event queue to implement the barrier synchronization. That is, as soon as a process executes a barrier synchronization, it is suspended and placed on an event queue associated with the particular barrier synchronization. Tightly coupled parallel systems with at most one process per processor almost always use a busy waiting solution.

Following our design philosophy, event queues are not used. Moreover, pure busy waiting is not employed. Instead, a shared data structure consisting of two pointers and a flag indicating which to use is updated and polled by the activities. This data structure is located in the spawn descriptor from which the synchronizing activities were generated, so

they all have pointers to it. Each activity that executes a `sync` first updates the shared counter. Then it checks to see it the counter indicates that all activities are ready. If not, it performs a `yield` instruction. This allows any other activities mapped to the same PE to execute its code. An activity mapped to a process with few activities will poll more frequently than one that is mapped to a PE with many activities. When the last activity performs the `sync`, it also resets the counter for the next `sync` operation. The two counters are used alternately, to avoid race conditions.

Finally consider support for the resumption of the parent activity. When a certain activity spawns a set of child activities, it is suspended until the termination of all the child activities. This removes the activity from the RMK run queue, but it is still accessible through the MAXI PCB table[5]. The spawn descriptor contains two counters: one of how many children have to be created, and the other of how many have not yet terminated. Both are initialized to the number of activities in the `parfor` or `parblock` construct. As each child is created, it decrements the counter of how many still have to be created. When this counter reaches zero, the spawn descriptor is unlinked from the global list. However, all the activities that were created from it maintain pointers to the spawn descriptor.

Each child that terminates uses this pointer to access the spawn descriptor and decrement the counter of non-terminated children. The last child to terminate, as identified by the termination counter reaching the zero, should resume the parent. This requires a kernel call on the processor on which the parent was running. If the parent was running on a different processor than the one used by the child, the child cannot resume it directly.

The solution is to create a set of lists of activities that should be resumed — one such list for each processor. The elements in these lists are the same spawn descriptors used to generate the children: the last child that terminates links the spawn descriptor onto the termination list associated with the processor on which the parent ran. This list is examined by the `get_work` task just before it examines the spawn list, and if any descriptors are found on it the tasks that created them are resumed. To support this, the spawn descriptor includes fields for the task identifier of the parent and for the processor number.

[5]The PCB table contains information about the tasks that have been created on the processor (PCB stands for "process control block"). Actually, the RMK PCB table is embeded in the MAXI PCB table.

5 Reducing Task Management Overhead

Reliance on a dynamic and adaptive system for mapping activities to processors is a good idea only if it does not incur a large overhead. Particular attention is required when there is a large number of fine-grained parallel activities. This situation is of real concern since it is often convenient to express a parallel algorithm as being composed of a very large number of fine-grain parallel activities. Much research has been done on the mapping and scheduling of such fine-grain activities so as to achieve high performance. Dataflow architectures use extensive hardware support to achieve this goal [9, 25]. The common approach on conventional multiprocessors is to increase the granularity at compile time based on an analysis of the program structure (see, e.g., [22]). We advocate the support of ultra-light-weight tasks in the runtime system, and contend that efficient support is possible at run time without previous knowledge about the program. In fact, we hope to relieve the programmer from the need to investigate how to partition the work into activities in order to achieve good performance.

The naive approach to activity generation maps each activity to an RMK task. When there are very many independent and fine-grain activities, this approach suffers from the overhead involved in creating these tasks. In addition, the system must maintain large data structures to cope with all the tasks.

In this section, we present several dynamic schemes to reduce this overhead. MAXI attempts to create and destroy a minimal number of RMK tasks by "reusing" these tasks. We refer to a reusable task as an *envelope*. Overhead associated with access to shared structures is also reduced by creating local *representatives*. Finally, overhead is reduced by optimizing the implementation for the situation in which activities perform explicit synchronization.

5.1 Envelopes: Support for Multiple Fine-Grained Activities

Our suggested optimization is based on the possibility that a single RMK task executes multiple activities, saving overhead and data structures. A new RMK task is not created for every activity, but rather only if an activity suspends or executes for longer than a scheduling time quantum. In effect, the granularity is increased on-line when possible. This is actually an automatic on-line version of the `lparfor` construct.

A similar approach has also been suggested for functional programming [18]. That work enjoys the benefits of freedom from side-effects, which makes all the tasks independent and

```
get_work:
forever {
   if (global list not empty)
      create an envelope task
   yield processor
}
```

```
envelope:
while (global list not empty) {
   reset timer
   remove and execute one activity
}
delete the envelope task
```

Figure 5: *High level description of the* `get_work` *task and of an envelope. Each processor has one* `get_work` *task in its local queue, and it creates additional envelopes as necessary.*

therefore makes it possible to combine them together in any way. Our mechanism is more general and flexible, and correctly handles situations in which one activity depends on another.

The details are as follows. As already described, the `get_work` task on each processor looks at the global list, and creates a new task if it finds work (Figure 5). However, the created task does not directly execute a single activity; rather, it executes what we call *envelope* code (the code that allows a task to be reused). An envelope serially executes activities. That is, it transfers control to one activity at a time (Figure 5). Whenever an activity terminates, control returns to the envelope code which goes back to the global list to get another (new) activity to execute.

Only when the global list is empty does the envelope delete itself. The timer is reset to a full time quantum before each activity is started. Thus, if the activities are indeed independent and fine-grained, the envelope will not be preempted. Therefore there is a good chance that one RMK task per processor will execute all of them, without the overhead for generating additional RMK tasks and without the overhead of context switching. If, however, an activity suspends execution due to synchronization constraints, or executes for longer than a time quantum, the envelope *is* preempted (as part of the standard RMKmultitasking facility). A new envelope is then created by `get_work` to deal with the rest of the activities. In the extreme case, this degenerates to the naive algorithm where a separate RMK task was created for each activity at the outset.

Note that envelopes are more flexible than previous proposals for pre-creation of tasks (e.g. in [5]), because the number generated is adapted to the characteristics of the program. When there are many short independent tasks, the overhead is on the order of a procedure

call. When independent tasks are very long, the envelopes introduce a small overhead relative to the total execution time. Therefore, the envelopes are still competitive with an optimal execution that would not create redundant tasks. The main advantage of the envelopes mechanism, though, is that it does not require any user intervention.

It should also be noted, however, that envelopes are less flexible than the thread management mechanisms used in various thread packages (e.g. [13, 28]). These mechanisms use their knowledge of the application to perform fine-grain thread scheduling without operating system intervention. In effect, they replicate operating system services within the application. We feel this type of implementation is overly complex, especially since the RMK kernel we are using is rather efficient in its own right. Interestingly, the often-cited problem of handling thread blocking is turned from a disadvantage into an advantage in our design. Other thread packages need to be able to block a user thread without blocking the kernel thread that implements it, so that the same kernel thread can go on to execute other user threads. This complicates their implementation. In our design, blocking signals the fact that the activities are interdependent, and therefore additional RMK tasks should be created for them.

5.2 Representatives: Improving Locality

To further reduce the overhead, there should be no interprocessor interaction involved in the execution of independent fine-grain activities. In particular, possible contention for the global list should be avoided. This can be achieved by partitioning the activities into disjoint sets, which are associated with the different processors. The relationship between the different components of the activity execution mechanism is then as follows:

- When a set of activities should be spawned, a spawn descriptor is placed in the global list as before.
- When the `get_work` task on any processor finds a spawn descriptor in the global list, it creates a local *representative* of the descriptor. A certain fraction of the activities are moved from the spawn descriptor to the representative. All local representatives are linked together, creating a local list.
- Envelopes loop and take activities for execution from the local representatives. As only one envelope can execute at a time on a certain processor, no locks are needed. The envelopes are non-preemptable when accessing the local queue.

code	*time [s]*		
	default	*envelopes*	*rep's*
`parfor 10000 delay(1ms);`	4.02	3.36	2.02
`lparfor 10000 delay(1ms);`	2.02	2.02	2.02
`parfor 1000 sync;`	4.95	5.12	5.02
`parfor 100 parfor 100 sync;`	7.50	2.85	2.50

Table 2: *Experimental results using envelopes and representatives. A five-processor configuration was used.*

The main question that remains is how to divide the activities between the representatives. Equal division may not be optimal if the activities are not identical. A promising approach is to always allocate $1/P$ of the remaining activities, as was suggested for guided self-scheduling of parallel loop iterations [19].

5.3 Experimental Results

Experimental results using envelopes and representatives are presented in Table 2. More detailed results can be found in [24]. Four different codes are used as examples (the code given in the table is shorthand that captures the essence of the real code, but is not real ParC syntax). In the first, 10000 activities are spawned, and each performs some local work that takes 1ms. Using envelopes is somewhat faster than the default version, but the real advantage comes from using representatives. In fact, with representatives the elapsed time is reduced to within 1% of the actual computation time (10000 activities $\times$ 1ms / 5 processors = 2sec).

The optimality of representatives is also shown by the second experiment, which is identical to the first except for using the `lparfor` construct. This construct tells the compiler that the spawned activities are independent, and that they should be chunked into P equal chunks, one per processor. Thus the compiler substitutes the user directive to spawn 10000 activities by a directive to spawn just P activities, and in addition generates code by which

these activities each perform $10000/P$ of the original assignments. This eliminates any unnecessary overhead. Using envelopes or representatives doesn't change anything because there is actually only one RMK task on each processor.

The third example concerns 1000 activities that perform a barrier synchronization. In order to do so all of the activities have to be spawned, so the envelopes gain no advantage. However, if there are multiple sets of synchronizing activities, as in the last example, envelopes can be reused for different activities. With representatives, this leads to a speedup of 3.

5.4 Support for Synchronizing Activities

A bug in the initial implementation of envelopes exposed another optimization which we had not thought of. The bug caused MAXI to initially create an envelope task for each activity (precisely what we had hoped to avoid), rather then creating them one at a time in the hope that some task creation can be avoided. To our surprise, this led to a significant improvement in the execution time of the third example, where all the created activities synchronize with each other. The reason is that creating all the envelopes at once saved the overhead of switching to `get_work` and checking the global list for each one.

Further analysis showed that if all the activities barrier synchronize (`sync`) shortly after being spawned, the time to create them is quadratic in the number of activities (Figure 6). In contrast, the cost of just spawning activities or just performing a `sync` is linear. The quadratic factor is a result of the way that the `sync` is implemented, and specifically, a result of using busy waiting with a `yield` instruction. The system then operates according to the following scenario. First, one activity is spawned by the `get_work` task. This activity tries to `sync`, finds that its siblings have not arrived yet, and `yield`s. `get_work` runs again, and spawns another activity. This activity too tries to `sync`, fails, and `yield`s. The first activity is then rescheduled, tries again, fails again, and `yield`s again. This pattern is repeated until all the activities are spawned: after each one, *all* the existing activities try to `sync`. Thus the time to spawn and `sync` N activities is proportional to the sum of N times the spawn overhead, plus $(N/P)^2/2$ times the overhead to perform a context switch and check the sync condition.

Rather than always create a table full of envelopes, which also has non-negligible overhead, it is easy to implement an optimization that solves this problem in a selective manner. Recall that the shared data structures used to implement barrier synchronization among sib-

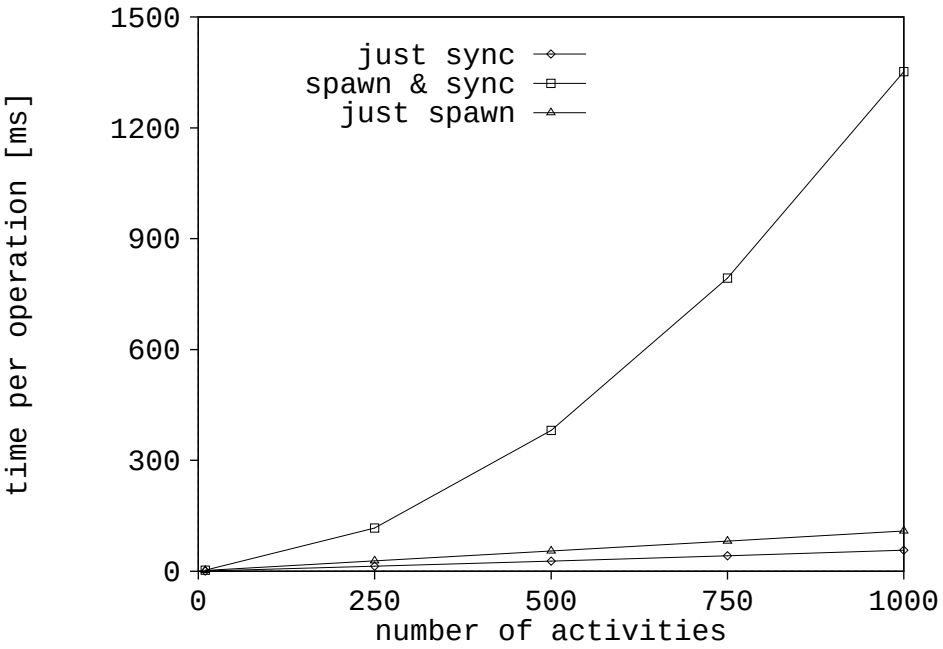

Figure 6: *Time for* `sync` *instruction by a large number of activities. Performance degrades sharply if the activities have to be spawned.*

ling activities are stored in their spawn descriptor. When `get_work` spawns a new envelope, it can check the barrier count in the spawn descriptor at the head of the list. If it is not at its initial value, this means that one of the sibling activities that has been created already has reached a barrier synchronization point. The nature of barrier synchronization then implies that all the siblings are probably going to synchronize, and therefore they are not independent. Therefore `get_work` should not create just one envelope, but rather should create enough envelopes for all the activities that may be expected to execute on this processor.

6 Forced Termination of Activity Groups

Because ParC uses closed constructs to form a serial/parallel graph of activity execution, special features are needed to support premature termination. We first review the semantics of the type of termination, then give two implementation schemes, and finally present our experimental results.

6.1 The `pbreak` and `preturn` instructions

Parallel activities in ParC programs are defined by blocks of code in parallel constructs and are created and terminated dynamically at run time. There is no equivalent to the Unix `fork` style, which tells the system to "create an additional activity" nor is there any notion of an activity identifier or `ID` at the language level. Moreover, there is no way to explicitly kill an activity or send it a signal. However, it is possible to terminate all the activities that were generated by the same construct, together with any descendants created by nested constructs. This happens when one of the activities tries to jump out of the construct, by issuing a `pbreak` or `preturn` instruction. For example, such behavior is useful when a set of parallel activities are spawned to speed up a search through a large data structure. The activity that finds the required item then `pbreak`s, terminating the search of all the other activities [4].

Terminating a subtree of activities can be done in synchronous or asynchronous styles. The synchronous style means that the whole system is interrupted and immediately takes steps to terminate the relevant activities. The asynchronous style means that a notice about terminating the subtree is posted in shared memory, and each processor does its part at its convenience. We advocate the synchronous style, because the whole point of the `pbreak` and `preturn` statements is to stop various activities from doing spurious work. It is therefore imperative that the subtree be deleted as soon as possible. This also prevents situations where new activities are generated at a higher rate than existing ones are terminated. Note that this is a situation where polling cannot be used, and we must resort to using inter-processor interrupts.

`pbreak` and `preturn` in parallel constructs are implemented by sending a broadcast interrupt to all the processors. The interrupt handler that is installed in the RMK interrupt vector is a Maxi function that scans the processor's PCB table, and deletes those belonging to the subtree that is being terminated. The main challenge is to identify the required activities. This is complicated by the fact that the activities in each level of the subtree are spread across all the processors, and the only link between an activity and its grandparent that happen to be on the same processor may be a third activity on another processor. Implementing `preturn` is further complicated by the fact that the root of the subtree is not necessarily the direct parent of the activity that performs the `preturn`, and it is also necessary to pass a return value. However, these problems are easily dealt with by a source-level transformation, which is implemented by the ParC compiler [12].

6.2 Algorithms for Activity Identification

We have designed two algorithms for identifying the related activities that should be terminated. The considerations in comparing them involve the overhead that is required when activities are spawned, and the resulting efficiency of terminating a subtree. If one assumes that forced termination is rare, it is better to reduce the overhead of spawning at the expense of more costly forced termination. If, on the other hand, forced termination happens frequently, it should be optimized at the expense of more bookkeeping during spawning.

First, however, consider the straightforward implementation. When a termination construct is executed, it is possible to traverse the recursive structure of the activity tree, deleting only the *direct* children of a given activity at each stage. To do so, it is enough to broadcast the parent activity's ID[6] to all the processors. If any of these children have additional descendants, a recursive broadcast is sent with the child's ID. This scheme is completely general, conceptually simple, and can deal with any tree structure. It also does not add any overhead when activities are spawned. However, it requires all the activities to be scanned again for each internal node in the subtree when the subtree is being terminated. This repeated scanning requires complicated coordination between the processors, to ensure that the whole subtree is indeed deleted and that the parent activity is not resumed too soon.

Alternatively, we propose a bottom-up version of this approach. Again, the activity ID of the root of the subtree that is being terminated is broadcast to all the processors. The activities on each processor are scanned, and the parent pointers are followed from each activity up to the root of the activity tree[7]. If the subtree root is encountered on the way, the activity is marked to be deleted. The cost of this approach is proportional to the number of activities in the system multiplied by the depth of the whole activity tree. Its main drawback is that it links data structures in distinct processors, forcing many remote references to be made. There is also a danger that the links in the top levels of the tree would become hot-spots, and that contention for them would reduce system performance. However, this can be avoided by marking each activity that is identified as being either in or out of the subtree, and searching only until a marked activity is found rather than up to the root.

The third algorithm is to use a coding scheme that allows all the descendants of a given

[6] While activities do not have IDs at the language level, they do at the system level. This is a system-wide unique integer.

[7] Note that the PCB table must be globally accessible for this to work.

algorithm	*increase in spawn overhead*	*average operations per killed activity*
coding	6%	6.7–8.4
bottom-up	21%	12.5–13.5

Table 3: *Comparison of the performance of two implementations of the kill-tree operation. A number of different trees with about 4000 activities were used.*

activity to be identified at once. For example, activities may be identified by *ranges* of (unsigned) integers, written as $[b,t]$. The first activity in the program is identified by the whole range from $b=0$ to $t=2^{32}-1$. When an activity identified by $[b,t]$ spawns k children, the range is divided into k; the first child is then identified by $\left[b, b+\frac{t-b}{k}\right]$, the second by $\left[b+\frac{t-b}{k}+1, b+\frac{2(t-b)}{k}\right]$, and so on. Activities in a subtree rooted at $[b,t]$ are then easy to identify: they have ranges that are a subrange of $[b,t]$.

This scheme is much simpler than the previous one, in the sense that it is easy to maintain the ranges and the search time is only proportional to the number of activities. It main drawback is that it limits the size of the activity tree that can be spawned. However, there is a nice tradeoff between the depth and width of the tree. With 32-bit words, for example, only 3 levels of nesting are possible if a thousand activities are spawned each time. But if only two are spawned, 32 levels of nesting are supported. If 64-bit words are used for the range boundaries, these numbers become 6 and 64 levels respectively; this is already available on DEC's Alpha microprocessor, and is expected to be available in other microprocessors shortly. Note that by using any integers for range bounds, rather than using bit positions, the restrictions on the activity tree are reduced. It is expected that in most cases system tables will be the limiting resource, and not the coding scheme.

6.3 Implementation Results

Run-time libraries implementing the last two algorithms have been written [12]. The performance of the two approaches is tabulated in Table 3. As expected, the coding scheme is more efficient. However, considering that the coding limits the structure of the activity tree, we decided to provide users with both options. Users who know that their program does not spawn too many activities may use the coding scheme. Otherwise, the more general bottom-up scheme should be used.

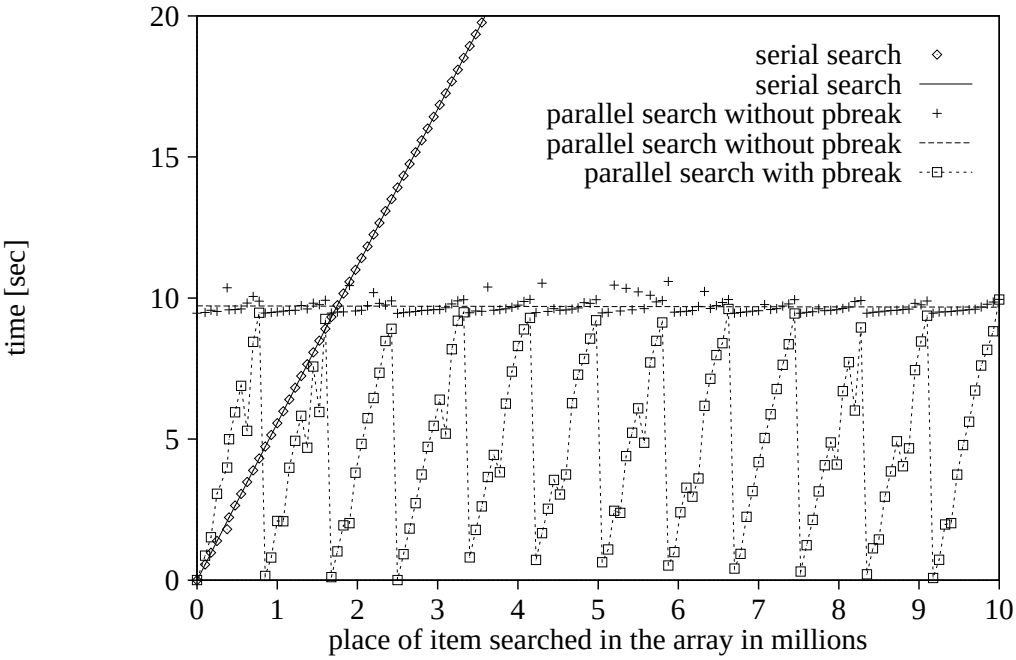

Figure 7: *Results of searching for an element in a large array.*

The question still remains as to whether using `pbreak` and `preturn` is beneficial at all, or maybe the overhead is too large. To resolve this question, we wrote a small test program that searches for a certain element in a large array. When the search is done in parallel, the activity that finds the correct element can terminate the futile search of its siblings by breaking out of the parallel construct. The results are shown in Figure 7. For a sequential search, the time to completion depends on the location of the item in the array. For a parallel search without using `pbreak`, the search time is nearly constant. This is the time required by activities that do not find anything to finish searching their allocated part of the array. When `pbreak` is used, however, the search time varies widely. If any activity finds

the element at the beginning of its part of the array, it terminates the search at once. But if the element is found near the end of an activity's part, there is no saving. For large arrays (like the one in the figure) the overhead for the `pbreak` is small enough so that on average the search time is cut in half.

7 Conclusions

Most multiprocessor operating systems are straightforward adaptations of uniprocessor systems. This forces parallel language implementations to be based on primitives that were originally designed for time-sharing systems, such as `fork` or `kill`. As a result unnecessary serialization may occur.

The MAXI runtime library that supports PARC programs is based on an interface that deals with groups of activities at once. Thus efficient parallel implementations are possible. This poses new challenges and opportunities in the area of parallel system design. Algorithms for the creation and management of groups of new activities and for the forced termination of subtrees of activities were developed in response to this challenge. The underlying principle of our design is the use of polling rather than interrupts whenever possible, so as to reduce overhead and complexity. Parallel aspects of the runtime support are essentially handled by a user-level library. The actual task management activities are done on each processor independently of all the others, using a real-time kernel.

This design should be contrasted with other parallel runtime systems, in which the kernel itself is parallelized. Such systems allow one processor to directly influence other processors' actions by issuing remote procedure calls. For example, threads (activities) can be created, deleted, suspended, or resumed on a remote processor. Note that the affected kernel may be required to field many such remote requests at once. Moreover, these requests might interfere with the local computation carried out on the processor. In contrast, the MAXI design is based on the observation that if all of the processors are already busy, they should not interrupt each other: it is easier and just as efficient to let each processor work locally, and check for external requests when it is ready to service them. If certain processors are idle, they continuously check for external requests, and efficiency is not compromised. Our activity allocation schemes are sensitive to the actual load on the processors, and support mapping constructs which are essential for the utilization of a NUMA machine.

MAXI is a research environment. Several versions and some support tools have been

developed, while the system itself is also used for research on parallel algorithms and for parallel programming courses. The various versions support gang scheduling, (rather than scheduling each activity individually) [15], monitored mode [6], and deadlock detection mode [14].

Acknowledgments

The ParC language 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 simulator's task system was ported to the Makbilan multiprocessor by Omri Mann and Coby Metzger. The new system, which is described in this paper, was developed and implemented by Moshe Ben-Ezra, Lior Picherski, and Dror Feitelson. Yair Friedman implemented the kill-tree operation, and Constantine Shteiman implemented envelopes and representatives. Dror Zernik, Larry Rudolph, Yosi Ben-Asher, and Sharon Broude took part in the discussions of various design alternatives. Martin Land kept the hardware up and running. Larry Rudolph has been in charge of the whole project since its inception, and Iaakov Exman replaced him when on sabbatical.

The parallel processors hardware was funded by a grant from the Israel Vatat Foundation and a generous equipment grant from Intel Corporation. The Maxi system was supported in part by grants from the US-Israel BSF 88-0045 and France-Israel BSF.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "*Scheduler activations: effective kernal support for the user-level management of parallelism*". *ACM Trans. Comput. Syst.* **10(1)**, pp. 53–79, Feb 1992.

[2] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "*The performance implications of thread management alternatives for shared-memory multiprocessors*". *IEEE Trans. Comput.* **38(12)**, pp. 1631–1644, Dec 1989.

[3] 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, Oct 1991.

[4] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph, "*ParC — an extension of C for shared memory parallel processing*". *Software — Pract. & Exp.*, to appear.

[5] P. Brinch Hansen, "*The programming language Concurrent Pascal*". *IEEE Trans. Softw. Eng.* **1(2)**, pp. 199–207, Jun 1975.

[6] D. Citron, I. Exman, and D. Feitelson, *MKMONITOR - A Parallel Monitor, or What You See is What You Program.* Technical Report 91-19, Dept. Computer Science, The Hebrew University of Jerusalem, Dec 1991.

[7] R. F. Cmelik, N. H. Gehani, and W. D. Roome, "*Experience with multiple processor versions of Concurrent C*". *IEEE Trans. Softw. Eng.* **15(3)**, pp. 335–344, Mar 1989.

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

[9] J. B. Dennis, "*Data flow supercomputers*". *Computer*, pp. 48–56, Nov 1980.

[10] J. Edler, A. Gottlieb, and J. Lipkis, "*Considerations for massively parallel UNIX systems on the NYU Ultracomputer and IBM RP3*". In *EUUG (European UNIX system User Group) Autumn '86 Conf. Proc.*, pp. 383–403, Sep 1986. Another version appeared in *Proc. Winter USENIX Technical Conf.*, pp. 193–210, Jan 1986.

[11] P.A. Emrath, M.S. Anderson, R.R. Barton, and R.E. McGrath, "*The Xylem operating system*", *Intl. Conf. on Parallel Processing*, vol I, pp 67–70, Aug 1991.

[12] I. Exman, D. G. Feitelson, and Y. I. Freidman, *How To Kill an Activity Tree.* Technical Report 91-20, Dept. Computer Science, The Hebrew University of Jerusalem, Dec 1991.

[13] J. E. Faust and H. M. Levy, "*The performance of an object-oriented threads package*". In *Object-Oriented Prog. Syst., Lang., & Appl. Conf. Proc.*, pp. 278–288, Oct 1990.

[14] D. G. Feitelson, "*Deadlock detection without wait-for graphs*". *Parallel Computing* **17(12)**, pp. 1377–1383, Dec 1991.

[15] D. G. Feitelson and L. Rudolph, "*Gang scheduling performance benefits for fine-grain synchronization*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 306–318, Dec 1992.

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

[17] Intel Corporation, *iRMK I.2 Real-Time Kernel Reference Manual.* 1988. Order number 462660-001.

[18] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., "*Lazy task creation: a technique for increasing the granularity of parallel programs*". *IEEE Trans. Parallel & Distributed Syst.* **2(3)**, pp. 264–280, Jul 1991.

[19] C. D. Polychronopoulos and D. J. Kuck, "*Guided self scheduling: a practical scheduling scheme for parallel supercomputers*". *IEEE Trans. Comput.* **C-36(12)**, pp. 1425–1439, Dec 1987.

[20] L. Rudolph and Y. Ben-Asher, *The PARC System.* Technical Report CS-88-8, Leibniz Center for Research in Computer Science, Hebrew University of Jerusalem, Aug 1988.

[21] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, "*A simple load balancing scheme for task allocation in parallel machines*". In 3rd *Symp. Parallel Algorithms & Architectures*, pp. 237–245, Jul 1991.

[22] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[23] M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "*Design rationale for Psyche, a general-purpose multiprocessor operating system*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 255–262, Aug 1988.

[24] K. Shteiman, D. Feitelson, L. Rudolph, and I. Exman, "*Envelopes in adaptive local queues for MIMD load balancing*". In *CONPAR'92/VAPP-V*, Sep 1992. To appear.

[25] V. P. Srini, "*An architectural comparison of dataflow systems*". *Computer* **19(3)**, pp. 68–88, Mar 1986.

[26] B. H. Tay and A. L. Ananda, "*A survey of remote procedure calls*". *Operating Systems Rev.* **24(3)**, pp. 68–79, Jul 1990.

[27] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "*Mach threads and the Unix kernel: the battle for control*". In *Proc. Summer USENIX Technical Conf.*, pp. 185–197, Jun 1987.

[28] A. Tucker and A. Gupta, "*Process control and scheduling issues for multiprogrammed shared-memory multiprocessors*". In 12th *Symp. Operating Systems Principles*, pp. 159–166, Dec 1989.

[29] 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*, vol. I, pp. 171–180, Jan 1989.