The Parallel Break Construct
or How to Kill an Activity Tree

Yair I. Friedman Dror G. Feitelson Iaakov Exman
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract

Most parallel languages provide means to express parallelism, e.g. a `parallel-do` construct, but no means to terminate the parallel activities spawned by such constructs. We propose three high-level primitives for this purpose, which are defined by analogies with primitives that break out of sequential iterative constructs. The primitives are `pcontinue`, which terminates the calling activity, `pbreak`, which terminates all the activities in the construct that spawned the calling activity, and `return`, which terminates all the activities created in the current function call. These constructs are especially useful in search problems, where an activity that finds a solution can terminate other activities that are investigating inferior approaches.

Given that parallel constructs can be nested, activities form a tree rooted at the original activity that started the program. The main challenge in implementing `pbreak` and `return` is identifying the subtree of activities that should be killed. Three algorithms were designed for this purpose. Two were implemented, and experiments show that using these constructs can provide significant performance benefits. It should be noted that while the implementation was done in the context of the *ParC* language and the MAXI system, the concepts apply to any language using closed constructs to express parallelism.

keywords: forced termination, kill, pbreak, pcontinue, return, parallel activities

1 Introduction

Consider the task of finding a specific item in a large, unordered set. Using a serial search, the expected time to find the item is $N/2$ (where N is the size of the set). Given P processors, it is easy to parallelize the search, with each processor searching $1/P$ of the set. However, the typical speedup achieved would only be $P/2$, because the processor that finds the item has to wait for the others to complete the futile search of their portions of the set, which takes N/P time. In order to achieve a speedup of P, the processor that finds the item must be able to immediately terminate the searches on the other processors.

Most low-level thread packages include functions to spawn a single thread at a time, and also include functions to kill previously spawned threads (see, e.g. [9]). Therefore if the parallel search is programmed using such a low-level interface, it is possible to distribute the IDs of all the threads at the outset, and then kill them when the item is found. However, programmers typically prefer to work with higher-level interfaces, for example multithreaded parallel languages. It turns out that practically all such languages do not provide facilities for one thread to kill another [7].

The *ParC* language is an exception. Parallelism is expressed in this language by using closed constructs, similar to the well-known `parbegin/parend`. These constructs may be nested in arbitrary ways, leading to a tree of activities where all the leaves are executing in parallel (subject to the availability of processors). Three high-level primitives are provided to allow for premature termination of activities. Rather than requiring that the affected activities be identified explicitly, these primitives are based on the block structure of the language. In fact, the primitives are fashioned after the well-known C primitives used to control the execution of iterative constructs.

The next section gives a short overview of *ParC*, and introduces the three primitives for forced termination. Section 3 describes three possible implementations of the "kill-tree" operation, which is the central element in forced termination. Section 4 presents a performance evaluation based on experiments conducted on the Makbilan research multiprocessor, and Section 5 presents the conclusions.

2 Forced Termination in *ParC*

As our work was done in the context of the *ParC* language, we first provide a brief primer of *ParC*. We then describe the semantics of the three primitives for forced termination, and give examples of their use.

2.1 The *ParC* language

ParC is a superset of the *C* programming language intended to support parallel programming in a shared memory environment [3, 8]. The main additions over *C* are two block-oriented parallel 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. Note that these are closed constructs, and the parent activity is blocked until all the activities in the construct terminate. This leads to a more understandable programming style than the use of `fork` and `join` [4].

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. 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 [3].

In addition to the parallel constructs, there are three main synchronization mechanisms. The first is a fetch-and-add instruction, denoted `faa`. The second is a `sync` instruction, which implements a barrier synchronization among all the activities created by a certain parallel construct. This instruction would typically appear in all of these activities, or else activities in which it does appear are forced to wait for the termination of activities in which it does not. The third synchronization mechanism is semaphores.

2.2 Primitives for Forced Termination

ParC activities are logical structures in the language, and no support is given for mapping them to run-time entities known to the system. They are not identified by names or IDs at source-code level. Therefore it is impossible for one activity to affect another, and in particular, it is impossible to kill a specific activity.

Recall, however, that activities in *ParC* are generated by block-structured expressions. It is therefore possible to self destruct, or to kill a whole group of related activities, by jumping out of the block. This is done 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 spawned 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.

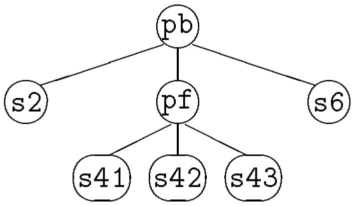
An example that clarifies the differences between these instructions is given in fig. 1. The `p` prefix in `pcontinue` and `pbreak` allows these instructions to be used in a sequential iterative construct within an activity. General `goto`s are not allowed past the boundaries of parallel constructs. This is the only restriction in *ParC* that is not compatible with *C*.

2.3 Usefulness

The need for primitives like `pcontinue`, `pbreak`, and `return` is very similar to the need for `break`, `continue` and `return` in serial programming languages. Many times a programmer

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

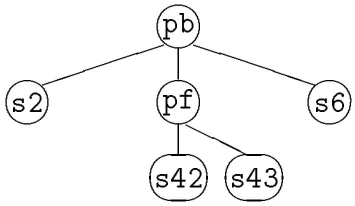
snapshot of activity tree:



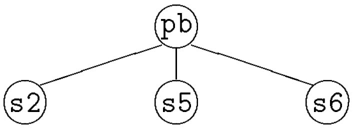
pb = `parblock`
pf = `parfor`

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

`s41` does `pcontinue`

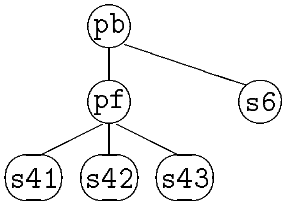


`s41` does `pbreak`



`s41` does `return`

`s2` does `pcontinue`



`s2` does `pbreak`

`s2` does `return`

Figure 1: *The effect of* `pcontinue`, `pbreak`, *and* `return` *when issued in a parallel construct. In* `parblock`, *colons are used to delimit the parallel blocks of code.*

```
search( arr, n, x )
int   arr[], n, x;
{
   parfor ( i; 1; P; 1 )
   {
      int   j;
      for (j=(i-1)*n/P; j<i*n/P; j++)
         if (arr[j] == x)
            return(j);
   }
   return(-1);
}
```

Figure 2: *Use of* `return` *in a parallel construct.*

gets to a point of the program where he knows that the computation is already finished or cannot be performed, and computing resources can be saved by scheduling them to other activities.

A clear example of such a case is given in Fig. 2. This code fragment is 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 return kills all the parallel activities that are searching other parts of the array, because once a match is found, there is no need to continue the search. One of the experiments reported in Section 4 is based on this code, and shown that on average the search time is reduced by half.

Another example that was used in our experiments is shown in Fig. 3. The MAXI system does not provide a synchronized high-resolution clock, which makes it difficult to time the execution of programs. The solution we adopted for the experiments was to dedicate one processor to count time. This is implemented as follows. Initially, two activities are spawned. The clock activity calls the `switch_off` primitive, which makes it non-preemptable. Then it synchronizes with the other activity, and starts to increment a shared variable named `clk`. Given that `clk` is a 32-bit integer, 2 billion instructions can be counted before we start having problems with overflow. The other activity waits at the synchronization point until the clock activity is started, and then initiates the execution of the parallel program. When the program terminates, a `pbreak` is used to kill the clock activity.

3 Implementation

It is easy to implement the `pcontinue` primitive: the activity just terminates itself, and other activities continue to operate normally. Therefore we will not discuss `pcontinue` further in

```
main()
{
   int   clk=0;
   parblock
   {
      switch_off;
      sync;
      while (1)
         clk++;
   :
      int   t1, t2;
      sync;
      t1 = clk;
      prog_main();
      t2 = clk;
      printf("%d",t2-t1);
      pbreak;
   }
}
```

Figure 3: *Use of* `pbreak` *to kill an activity in an endless loop.*

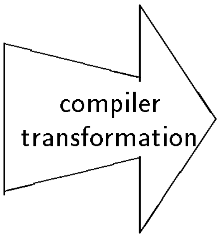
this paper. Implementing `pbreak` and `return` demands killing all activities in a certain subtree. The difficulty in doing so depends on how activities are maintained in the system. Two common structures in shared-memory machines are a global queue which is shared by all the processors [5, 10], or local queues on each of the processors [6].

If all the activities reside in a global queue, that queue has to be locked and searched for activities belonging to the subtree that is being killed. This should probably be done by a single processor. The more interesting case is when activities are mapped to all the processors, and each has a local queue. It is then necessary to search through all the processors, identify the relevant activities, and kill them. In this section we describe three algorithms for killing a subtree under the scenario of multiple local queues. Note that although the queues are local, some of the algorithms assume that the processors can access each other's data structures.

3.1 The `kill_tree` Operation

By definition, the `pbreak` primitive breaks out of the parallel construct that spawned the calling activity, killing all its siblings in the process. If those siblings had in the meantime spawned additional activities, these activities are killed as well. Therefore the effect of original code:

```
ret_type func()
{
  s1
  parblock {
     s21
  :
     s22
     parblock {
        s231
     :
        s232
        return(x)
     }
  }
  s3
}
```




```
ret_type func()
{
  ret_type         ret_val;
  int              ret_flag;
  activity_handle  ret_id;

  ret_flag = FALSE;
  ret_id = get_handle();

  s1
  parblock {
     s21
  :
     s22
     parblock {
        s231
     :
        s232
        ret_val = x;
        ret_flag = TRUE;
        kill_tree(ret_id);
     }
  }
  if (ret_flag)
     return(ret_val);
  s3
}
```

Figure 4: *Code transformation to implement* `return` *in nested parallel constructs. The variables* `ret_val`, `ret_flag`, *and* `ret_id` *are shared by the spawned activities because the parallel constructs appear in their scope.*

`pbreak` is actually to kill the subtree of activities rooted at the parent of the calling activity. The parent activity then resumes its execution.

The `return` statement returns from the last function call. If the function includes nested parallel constructs, and one of the activities in them calls `return`, then all the activities created in the function have to be killed. These activities form a subtree that is rooted in the activity that called the function. That activity then resumes execution at the instruction after the function call.

Implementing `return` is somewhat more difficult than implementing `pbreak`, because the root of the subtree is not necessarily the direct parent of the calling activity. Therefore

```
kill_tree( id )
{
   if (id == 0)
      id = parent(current_activity);
   broadcast( id );
}
```

Figure 5: *Pseudocode for the* `kill_tree` *operation.*

the identity of the root has to be maintained and propagated when additional activities are spawned. Moreover, it may also necessary to pass a return value. In our implementations, both these problems are dealt with by a source-level transformation carried out by the compiler, as shown in Fig. 4. Three shared variables are declared at the beginning of the function. `ret_id` is used to store a handle to the root activity. `ret_val` is used to store the return value. `ret_flag` is used to note the fact that the subtree has terminated as a result of a `return` call, as opposed to normal termination.

To summarize, both `pbreak` and `return` can be implemented by the same `kill_tree` operation. For `pbreak`, `kill_tree` is invoked with a zero argument, implying that the root of the tree is the direct parent of the calling activity. For `return`, a handle to the root is passed as the argument. This handle is obtained via the `get_handle` call.

Note that when a subtree is terminated, the root activity may only resume after the whole subtree is dead. 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. Moreover, it is imperative that the subtree be deleted as soon as possible, to prevent situations where new activities are generated at a higher rate than existing ones are terminated. The implementation of `kill_tree` should therefore interrupt the whole machine, and initiate an algorithm to identify the relevant activities and kill them. This is done by sending a broadcast interrupt with the root ID (Fig. 5). the interrupt handler implements the said algorithm.

The following sections describe three such algorithms. 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.

```
handler( id )
{
   foreach a in table {
      if (parent(a).id == id) {
         if (a has spawned)
            broadcast( a );
         delete a
      }
   }
}
```

Figure 6: *Pseudocode of a simpleminded top-down algorithm.* `a` *denotes an activity.*

3.2 Simple Top-Down Algorithm

The first algorithm is to follow the recursive structure of the activity tree, deleting only the *direct* children of a given activity at each stage. To do so, nothing has to be done at spawn time. When `kill_tree` is called, the root activity ID is broadcast. All processors are interrupted and start scanning their activities lists. For each activity the father ID is checked, and if it matches the ID being searched, the activity is deleted. If the activity had itself spawned some children, a recursive broadcast is sent with the activity's ID. Pseudocode for this algorithm is shown in Fig. 6. Note that after all activities are killed, it is necessary to resume the root. This is not shown.

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, The cost of a `kill_tree` operation is proportional to the number of activities in the system multiplied by the number of internal nodes in the subtree, because it requires all the activities to be scanned again for each internal node that is found. This repeated scanning requires complicated coordination between the processors, to ensure that the whole subtree is indeed deleted and that the root activity is not resumed too soon. We therefore chose not to implement this algorithm.

3.3 Search and Mark Bottom-Up Algorithm

The second algorithm uses the bottom-up approach: all the activities in the system are scanned once, and the parent pointers for each one are followed up the tree. If the root of the subtree is found along the way, the activity is deleted. If the root of the whole tree (called "main") is reached, it means that the activity is not in the subtree that is being terminated.

The overhead of this algorithm at spawn time is limited to maintaining the parent pointers, if they are not already maintained anyway. The `kill_tree` operation is implemented as

```
handler( id )
{
   foreach a in table {
      if (a is main) a.mark = main;
      if (a.id == id) a.mark = root;
      else a.mark = unknown;
   }
   sync;
   foreach a in table {
      p = a;
      while (p.mark == unknown)
         p = parent(p);
      a.mark = p.mark;
   }
   sync;
   foreach a in table {
      if ((a.mark == root) && (a.id != id))
         delete a
   }
}
```

Figure 7: *Pseudocode of the bottom-up algorithm.*

above (Fig. 5).

Pseudocode for the handler is given in Fig. 7. The handler operates in three phases: first some initialization, then all the activities in the subtree are marked, and finally they are deleted. All the handlers synchronize between the phases, because they access each other's data structures when they follow the parent pointers and check the markings.

In the initialization, all activities are marked as unknown except for two: the root of the subtree being killed, and the root of the whole tree (if they are the same, it should be marked `root`. Otherwise, the root of the whole tree is marked `main`). The synchronization at the end of this phase ensures that no handler will see an uninitialized marking.

The bulk of the work is done in the second phase. Each handler scans all the activities on its processor, and follows the parent pointers up the tree until it finds a marked activity. it then copies the marking. As a result, all the activities in the subtree will end up marked `root`, and all those out of the subtree will end up marked `main`. Note that by going up the tree until the first marked activity is found we quickly spread the marks downward, thus alleviating the contention for activity markers near the top of the tree. The synchronization at the end of this phase ensures that all parent pointers remain valid as long as necessary, and no activities are deleted prematurely.

```
handler( id )
{
   foreach a in table {
      if ((a.b>=id.b) && (a.t<=id.t))
         delete a
   }
}
```

Figure 8: *Pseudocode of the coding algorithm.*

In the third phase, those activities marked with `root` are deleted. The only exception is the root activity itself, which resumes execution.

The cost of this approach is proportional to the number of activities in the system multiplied by the average number of activities that have to be passed until a marked one is found. This factor is bounded by the depth of the whole activity tree. The main drawback of this algorithm is that it links data structures in distinct processors, forcing many remote references to be made. And, despite the propagation of markers downwards, there is also a danger that the links in the top levels of the tree would become hot-spots, and the contention for them would reduce system performance.

3.4 Coding Algorithm

The third algorithm is to use a coding scheme that allows all the descendants of a given activity to be identified at once. In our code, activities are identified by *ranges* of unsigned integers, written as $[b, t]$ (where t is *not* included in the range). 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 $[b, t-1]$ is divided into k; The last number in the range is the unique identifier for the parent. The first child is then identified by $[b, b + \frac{t-1-b}{k}]$, the second by $[b + \frac{t-1-b}{k}, b + \frac{2(t-1-b)}{k}]$, and so on. Activities in a subtree rooted at $[b, t]$ are then easy to identify: they have ranges that are a subrange of $[b, t-1]$. The root is also easy to find and awake because it is the only activity whose range ends with t.

The overhead of the algorithm at spawn time involves the calculation of the new range: essentially a couple of additions and a divide. The implementation of `kill_tree` is again the same as that shown in Fig. 5, provided we interpret `id` as an ordered pair of numbers rather than as a scalar (`get_handle` should therefore also return an ordered pair in this case). The handler invoked to handle the broadcast interrupt is extremely simple, as shown in Fig. 8. The cost of this scheme is proportional to number of activities in the system. It should be noted that the run time of this algorithm is not dependent on the size of the subtree that is being terminated.

This scheme is much simpler than the previous ones, in the sense that it is easy to

```
typedef struct {
   int   m[I];
   int   b, t;
} ID;

/* parent is the ID of the spawning activity */
if (parent.t - parent.b < k) {
   for (i=0; i<I; i++)
      if (parent.m[i] == 0) break;
   if (i == I) return( FAIL );
   parent.m[i] = parent.t;
   parent.b = 1;
   parent.t = MAXINT;
}
/* now split into k subranges */
```

Figure 9: *Allocation of codes in the multi-word coding algorithm.*

maintain the ranges and the search time is proportional to the number of activities. Its main drawback is that it limits the size of the activity tree that can be supported. However, because we are using ranges 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 longwords are used for the range boundaries, these numbers become 6 and 64 levels respectively. Note that by using any integers for range bounds, rather than using bit positions, the restrictions on the activity tree are reduced.

Multi-word codes

If the restriction on the tree size is a concern, the coding scheme can be extended to accommodate larger trees. The original coding uses two words to hold the values of b and t, giving a range of 32 bits. Each additional word that is added adds another 32 bits.

The multi-word coding works as follows. Initially, b and t are set as before, and all the additional words are set to zero (the number of additional words is denoted by I).. As activities are spawned, and original range is divided between them. If an activity tries to spawn k children, but its range is too small (i.e. $t - b < k$), it adds another word to the code (see Fig. 9). This word is set to the activity's t value, and then b and t are reset to the full range of 32 bits again (note that the range starts from 1 rather than 0). Pictorially, the tree is partitioned into parts that can be handled with 32-bit codes. The additional code words specify which leaf in the higher tree is the root of the next tree.

The procedure to check whether an activity is in the subtree identified by the ID of its

```
is_in_subtree( id, root )
{
   int   i = 0;
   while ((i<I) && (root.m[i]!=0)) {
      if (id.m[i] != root.m[i]) return(FALSE);
      i++;
   }
   if ((i<I) && (id.m[i]!=0)) {
      if ((id.m[i]>=root.b) && (id.m[i]<=root.t))
         return( TRUE );
      else return( FALSE );
   }
   if ((id.b>=root.b) && (id.t<=root.t))
      return( TRUE );
   else return( FALSE );
}
```

Figure 10: *Procedure for checking if an activity belongs to a specified subtree, using the multi-word coding algorithm.*

root is slightly more complicated than before (Fig. 10). The activity is in the subtree only if it has no less additional code words than the root, and all of them match. If the activity has more additional code words, the first additional one must be within the range indicated by $[b, t]$ in the root. otherwise, the $[b, t]$ range of the activity must be a subrange of the $[b, t]$ range of the root.

4 Experimental Results

4.1 The Makbilan Testbed

The implementation and experiments were conducted within the framework of the Makbilan research multiprocessor and the MAXI system.

The Makbilan research multiprocessor consists of up to 15 processor boards in a Multibus-II cage (the experiments were run on a 12-processor configuration). Each board has an Intel 386 processor, a 387 mathematical co-processor, a message passing co-processor, and 4MB of memory. 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 [2]. The box also includes one 486 board that acts as a Unix host. Users log on to the Unix board, and

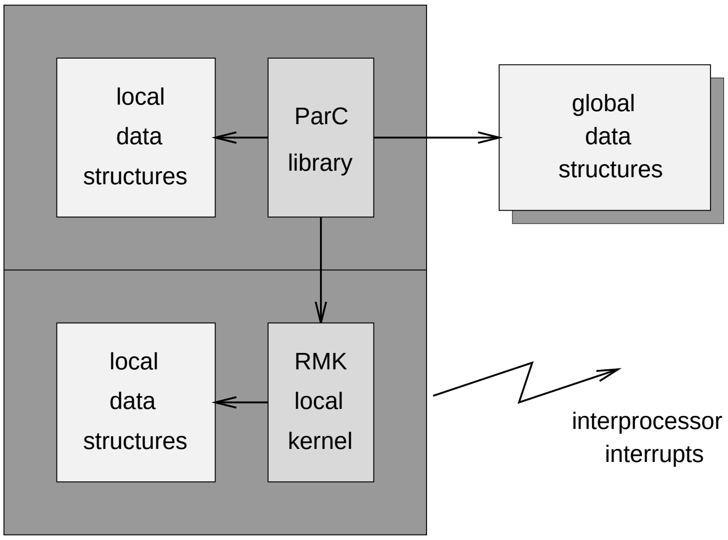


Figure 11: *MAXI system configuration on each board.*

can then load and execute *ParC* programs on all the other boards.

MAXI is an acronym for the **Mak**bilan **Sy**stem [6]. The system may be partitioned into two main layers: A runtime library that supports the *ParC* constructs, and a local kernel on each board. The local kernel is Intel's RMK [1], which is a real-time kernel designed to use hardware support provided by the 386 and the Multibus-II. This kernel is highly optimized to provide fast task creation, termination, and context switching. *ParC* activities are implemented by RMK tasks (the RMK equivalents of Unix processes). Note that RMK is not an inherent part of the system: alternative kernels that support local task manipulation could be used.

The implementations of the `kill_tree` operation were part of the MAXI runtime library. It should be noted that the MAXI design emphasizes asynchronous distributed operation without unnecessary interdependencies between boards. Thus each board has a local copy of the runtime library, complete with local data structures (fig. 11). Global data structures in shared memory are used only when activities executing on one board need to influence the execution of other activities on other boards. The broadcast interrupts needed to implement the `kill_tree` operation are provided by RMK and the Multibus II.

4.2 Overhead Evaluation

We first designed a set of experiments to measure the overhead involved in using the `kill_tree` facility. This has two components: the increased overhead when activities are spawned, and the overhead for the `kill_tree` operation itself.

	default	*coding*	*bottom-up*
time [s]	59.656	63.406	72.007
degradation	0%	6%	21%

Table 1: *Increase in the cost of spawning, per activity spawned.*

Spawn overhead

We spawned a full binary-tree of depth 12 (i.e. a total of 4095 activities) for 60 times. The time we measured is the time given by the system clock. The results are given in table 1. As it can be seen from the table the coding algorithm has a reasonable overhead. The bottom-up algorithm on the other hand has a relatively large overhead. We think that the cause for the large overhead is the locking of the global spawn list, since it is a doubly linked list in this version of the code.

`kill_tree` *overhead*

Based on the design of the algorithms presented in Section 2, we expect that the cost of killing a tree will be the sum of three terms: a constant overhead for context switching and the broadcast interrupt, a cost involved in the killing of each activity, and a cost proportional to the height of the tree (in the case of bottom-up traversal of the tree). A number of experiments were carried out to evaluate these terms. In all cases, a tree of activities was created by nested spawns, and then all of it was killed using a `pbreak` or `return`.

To find the constant overhead a flat tree is sufficient. Since the system clock was too rough we didn't measure time. Instead we spawned twelve activities (one for each processor). When we are certain that the tree is spawned, activity number one raises a flag and issues a `pbreak` command. The other activities start counting immediately on a local counter. The result is the value of the counter for each processor. The average was 342.07 for the coded library, and 340.18 for the pointers library. We have not discovered any difference between the different PEs. This means that the constant overhead is about 340 instructions, for both algorithms.

In another experiment we checked the total time to kill a tree as a function of its depth. We expected that the coded version will not be influenced by the depth of the tree, while the bottom-up version should be affected when running on very deep trees.

Again since the system clock was too rough we counted instructions instead. We performed the experiments on a special version of the runtime libraries. In this version processor number 12 does not participate in spawning and killing the tree. Instead it waits for0 processor number one to raise a flag (signalling the begining of the kill-tree operation, and then it starts counting. When the kill-tree operation is finished, the father of the tree raises another flag thus telling processor 12 to stop counting. We spawned several full trees with different node degrees on the other eleven processors. Each tree had about 4000 nodes, so that the only difference is the height. To eliminate the effect of small deviations in the number of nodes, the results are normalized by the number of nodes in each case.

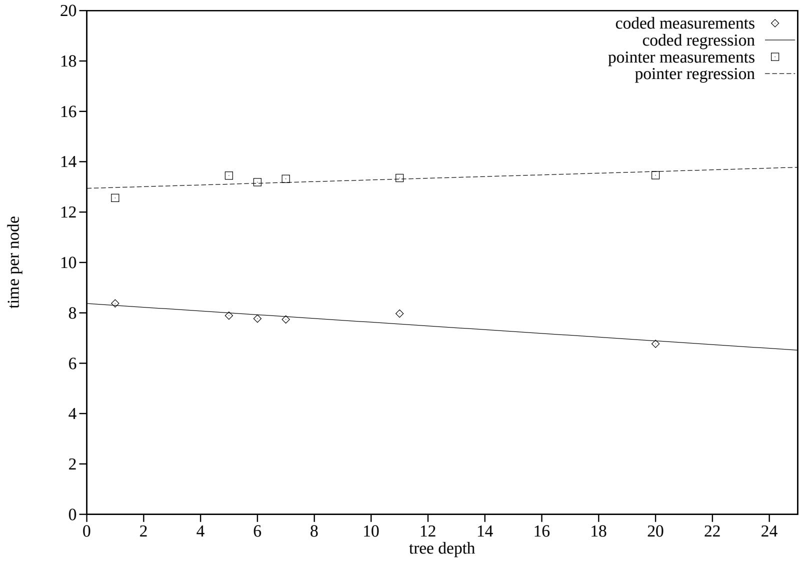


Figure 12: *Overhead as function of tree height.*

The results are shown in fig. 12. We can see that neither algorithm is influenced by the depth of the tree. The time until the kill-tree operation is done is nearly constant (per process). This implies that the killing time is not wasted on searching the subtree, but on other operations such as broadcasting and unlinking the descriptors from the lists. In particular, there seems to be no congestion near the top of the tree when using the bottom-up algorithm.

In another experiment we checked the total run time of the kill-tree operation as a function of the number of nodes in the tree. We used again the same special version of the library we used for the previous experiment. This time we spawned flat trees with depth 2 and varying number of nodes, so that the tree height will not influence the results. The results are shown in fig. 13. They reveal a constant overhead that is larger than that measured for the broadcast alone, and nearly double in the bottom-up algorithm. In addition, the overhead per node in the bottom-up algorithm is also higher than in the coding algorithm, as witnessed by the steeper slope of the graph.

4.3 To `pbreak` or not to `pbreak`?

The question of whether or not it is beneficial to use `pbreak` and `return` depends not on the absolute overhead, but on the relation between the overhead and the performance improvement. To gauge this relation, we conducted an experiment that used these constructs in the setting of a real application.

The application chosen was a parallel search, like that shown in Fig. 2. In the experiment, `P` was 12 and `n` was 10 million. The results are shown in Fig. 14. Using a serial search, the

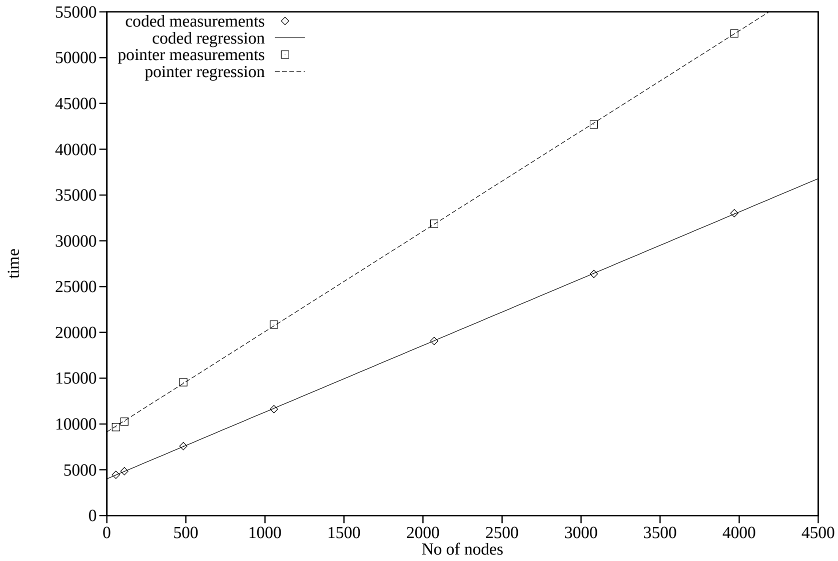


Figure 13: *Overhead as function of number of nodes.*

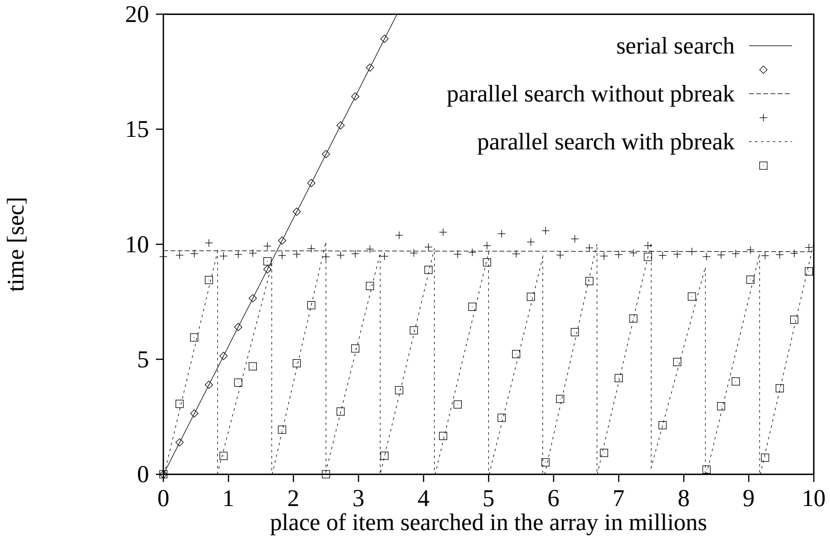


Figure 14: *Search time as a function of the location of the item.*

searching time was proportional to the location of the item for which we are searching. Using a parallel search without killing all the activities, the search time was constant: the activity that found the item might terminate before the others, but the 11 others would nevertheless continue to search through the 833,333 array cells allocated to each one of them. Using a parallel search with the activity that finds the item killing all the others, the search time varied widely. If the item happened to be at the beginning of an activity's segment of the array, it was found quickly and the search terminated. If it was near the end of a segment, the search time was essentially the same as with no killing. On average, the search time was halved.

With the array size used in this experiment, the overhead for killing the tree was very small compared with the search time, so the improvement was significant. With smaller arrays, the relative cost of the overhead would be bigger. To quantify this effect, we ran the experiment with different array sizes and found the break-even point. Using the coding algorithm we found that even for an array with 8250 elements (750 elements per activity) it gave a better running time when `return` was called. The bottom-up algorithm needed an array with 11550 elements (1050 elements per activity) before it gave better running time.

5 Conclusions

Killing a tree of activities created by nested spawns is a useful operation for parallel programming. Two constructs that perform this function were defined and implemented: `pbreak`, which breaks out of the innermost parallel construct, and `return`, which returns from the last function call and kills all the parallel activities spawned within that function. These constructs are naturally integrated with the block structure of the programming language, and circumvent the need for low-level primitives. While our work was done in the context of the *ParC* language, the concepts apply to other languages as well.

The main challenge in implementing the `kill_tree` operation is to identify the relevant activities. We developed, implemented, and experimented with two algorithms for this purpose. Our results are promising in the sense that the overhead involved is relatively low. Thus use of the new constructs can lead to significant performance benefits.

References

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

[2] 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.

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

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

[5] 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.*, pp. 126–135, 1985.

[6] D. G. Feitelson, Y. Ben-Asher, M. Ben Ezra, I. Exman, L. Picherski, L. Rudolph, and D. Zernik, "*Issues in run-time support for tightly-coupled parallel processing*". In 3rd *Symp. Experiences with Distributed & Multiprocessor Syst.*, pp. 27–42, USENIX, Mar 1992.

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

[8] 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.

[9] 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.

[10] S. Thakkar, P. Gifford, and G. Fielland, "*Balance: a shared memory multiprocessor system*". In 2nd *Intl. Conf. Supercomputing*, vol. I, pp. 93–101, 1987.