# EFFICIENT PARALLEL RECURSION

**PER BRINCH HANSEN**

**(1995)**

**A simple mechanism is proposed for dynamic memory allocation of a parallel recursive program with Algol-like scope rules. The method is about as fast as the traditional stack discipline for sequential languages. It has been used to implement the parallel programming language SuperPascal.**

## 1 Introduction

I will describe a memory allocation scheme for block structured programming languages that support unbounded activation of parallel processes and recursive procedures. This technique has been used to implement the parallel programming language SuperPascal (Brinch Hansen 1994).

Three decades ago, Dijkstra (1960) proposed the standard method of dynamic memory allocation for recursive procedures in block structured, sequential languages, such as Algol 60 (Naur 1963), Pascal (Wirth 1971) and C (Kernighan 1978).

The scope rules of Algol-like languages support stack allocation of memory for sequential programs. All variables are kept in a single stack. When a block is activated, an *activation record* (a data segment of fixed length) is pushed on the stack. The activation record holds a fresh instance of every local variable of the block. At the end of the activation, the activation record is popped from the stack. Since each activation creates a new instance of the local variables, stack allocation works for both recursive and nonrecursive procedures. The crucial assumption behind stack allocation is that dynamically nested block activations always terminate in last-in, first-out order.

After two decades of research in parallel programming languages, there is still no efficient standard method for dynamic memory allocation of parallel

P. Brinch Hansen, Efficient parallel recursion, *SIGPLAN Notices 30*, 12 (December 1995), 9–16. Copyright © 1995, Per Brinch Hansen.

recursion. When you add parallelism to a block structured language, the variable instances form a tree structured stack with branches that grow and shrink simultaneously. If dynamic parallelism is combined with unbounded recursion, the number and extent of the stack branches are unpredictable.

In a parallel recursive program, there is no simple relationship between the order in which blocks are entered and exited. So, you cannot use the traditional last-in, first-out allocation. This makes it more difficult to reclaim and reuse the memory space of activation records efficiently.

With few exceptions, language designers have ignored the thorny problems of parallel memory allocation by outlawing recursion and restricting parallelism to the point where it is possible to use static memory allocation.

In many languages, it is impossible to reclaim the memory space of parallel processes. These include Concurrent Pascal (Brinch Hansen 1975), Simone (Kaubisch 1976), Modula (Wirth 1977), Distributed Processes (Brinch Hansen 1978), Pascal Plus (Welsh 1979), StarMod (Cook 1980), SR (Andrews 1981), Concurrent Euclid (Holt 1983), Planet (Crookes 1984) and Pascal-FC (Davies 1990).

CSP (Hoare 1978), Edison (Brinch Hansen 1981), and occam (Inmos 1988) support process activation and termination, but only of a fixed number of parallel nonrecursive processes determined during compilation.

Static memory allocation is adequate for many parallel computations (Fox 1988). However, parallel recursion is the natural programming tool for parallel versions of divide-and-conquer algorithms, such as quicksort, the fast Fourier transform and the Barnes-Hut algorithm for n-body simulation (Fox 1994).

Parallel recursion requires dynamic allocation and release of activation records in a tree structured stack. B6700 Algol (Organick 1973) and Mesa (Lampson, 1980) demonstrate that it is possible to support both parallelism and recursion in systems programming languages. The substantial overhead of parallel processes in these languages is acceptable in operating systems, which support slowly changing configurations of user processes. It is, however, too inefficient for highly parallel computations.

Is there a memory allocation method that makes parallel recursion as efficient as sequential recursion for all systems and user programs? I don't know any. Parallel recursion can probably only be implemented efficiently at the expense of some generality.

As a reasonable compromise, I will confine myself to the problem of allocating activation records of different lengths for a single parallel program in a memory of fixed size. The proposed technique is more ambitious than previous methods in the following sense: *it succeeds in making the activation and termination of parallel processes and recursive procedures equally fast!*

Joyce (Brinch Hansen 1989) was my first attempt to simplify memory allocation for parallel recursion. The multiprocessor implementation of Joyce uses a stack-like scheme for parallel block activation in a single memory heap. On entry to a block, an activation record is allocated at the top of the heap. On exit from the block, the activation record is marked as free. Free space is reclaimed only when it is at the top of the heap. This method works well for many parallel recursive programs. However, it fails if a program continues to demand space for parallel block activations before previously released space can be reclaimed. In that situation, the heap grows until it runs out of memory.

The occasional failure of the Joyce heap made me look for a more robust memory allocation for SuperPascal. After solving this problem, I found that I had reinvented a simplified version of the *Quick Fit* allocator, which was used for heap management in the sequential programming language Bliss (Weinstock 1988).

The main contribution of this paper is the discovery that Quick Fit is an efficient memory allocator for a parallel recursive language that requires an *unbounded, tree structured stack* of activation records. The consistent omission of efficient parallel recursion in previous block structured languages shows that this insight only seems obvious once you know the solution.

## 2 Assumptions

I will state the assumptions behind the method in general terms. However, I will use the implementation of block structured parallel languages to motivate the assumptions.

The general problem is to allocate and release segments of different lengths in a memory of fixed size under the following assumptions:

- *Each segment occupies a contiguous memory area of fixed length.*

In a block structured program, the unit of memory allocation is an activation record of fixed length that holds the local variable instances of a single activation of a block.

- *A segment is never relocated in memory.*

During program execution, the activation records in use are linked by pointers representing variable parameters, nested blocks, and activation sequences. Dynamic relocation of linked activation records would be complicated and time-consuming.

- *A segment is released only when no other segment in use points to it.*

The scope rules enable a compiler to check that the local variable instances of a block activation are accessed only during the activation. Consequently, the corresponding activation record can safely be released on exit from the block.

- *Segments are generally allocated and released in unpredictable order.*

The nondeterministic nature of parallel recursion complicates the dynamic memory allocation considerably.

- *There is a fixed number of segment lengths.*

A block structured program consists of a fixed number of blocks. (In SuperPascal, a block is either a process statement or a procedure.) Each activation of the same block allocates an activation record of the same fixed length.

- *A program tends to use segments of the same lengths repeatedly.*

This is a plausible hypothesis about any program that uses the same procedures numerous times to transform different parts of large data structures sequentially or in parallel. The measurements in Section 4 strongly support this assumption.

The above assumptions are satisfied by a single block structured program that runs in a fixed memory area. However, they are not realistic for an operating system, which allocates an unbounded number of segments, most of which are unique to particular user jobs.

## 3 Implementation

Algorithm 1 defines the allocation of activation records for a parallel program that runs on a single processor in a memory area of fixed size. On a multicomputer with distributed memory, each processor must manage its own

```
var pool: array [1..limit] of integer;
  memory: array [min..max] of integer;
  top: integer;

procedure initialize;
var index: integer;
begin
  for index := 1 to limit do
     pool[index] := empty;
  top := min − 1
end;

procedure allocate( index, length: integer;
  var address: integer);
begin
  address := pool[index];
  if address <> empty then
    pool[index] := memory[address]
  else
    begin
      address := top + 1;
      top := top + length;
      assume top <= max
    end
end;

procedure release( index, address: integer);
begin
  memory[address] := pool[index];
  pool[index] := address
end;
```

**Algorithm 1** Memory allocation.

memory for local processes. On a multiprocessor with shared memory, the allocation and release of activation records must be indivisible operations.

I assume that an operating system allocates a fixed amount of memory for the execution of a parallel program. The allocation method used by the operating system is beyond the scope of this discussion. My only concern is the algorithms used by a running program to allocate activation records within its own memory.

A dynamic boundary divides the program memory into two contiguous parts. One part is the heap, which holds all past and present activation records. The rest is free space. During program execution, the heap can only grow, and the free space can only shrink. A register holds the current top address of the heap.

The blocks in a program have consecutive indices and fixed activation record lengths determined by a compiler. For each block, a running program maintains a pool consisting of all free activation records reclaimed after previous activations of the block. Each pool is represented by an address, which either denotes an empty pool or is the first link in a list of free activation records of the same length.

Initially, the entire memory is free and every pool is empty.

On entry to a block with a given index and length, an attempt is made to allocate a free activation record from the corresponding pool. If the pool is empty, a new activation record of the given length is allocated in the free space, which is reduced accordingly.

On exit from the block, the activation record is released and added to the corresponding pool.

The algorithms for allocating and releasing an activation record are not intended to be implemented as separate procedures. They are part of the machine code executed at the beginning and end of every process statement and procedure. An activation record is allocated or released in constant time. Most processors can perform these simple operations by executing three or four machine instructions.

When the execution of a program ends, its memory area is still divided into pools of free activation records and the remaining free space. However, that does not matter, since the operating system will reclaim the entire memory area as a single unit.

## 4 Performance

The heap allocation method described here has been used to implement the block structured parallel language SuperPascal. So far, I have written parallel SuperPascal programs for a dozen standard problems in computational science (Brinch Hansen 1995).

Table 1 shows the ability of the heap allocator to recycle previous activation records during the execution of three parallel programs on a single processor.

**Table 1** Measurements.

| Parallel program | Quicksort tree | N-body pipeline | Laplace matrix |
|---|---|---|---|
| Number of blocks | 16 | 24 | 28 |
| Process activations | 11 | 300 | 25,609 |
| Procedure activations | 18,120 | 513,553 | 67,156 |
| New activation records | 51 | 27 | 64 |
| Reused activation records | 18,080 | 513,826 | 92,701 |

The quicksort tree uses both parallel recursion (to create a binary tree of processes) and sequential recursion (to quicksort in parallel). The program consists of 16 blocks which are activated a total of 18,131 times (eleven process activations plus 18,120 procedure activations). These block activations create 51 new activation records, which are reused 18,080 times.

The n-body pipeline is a parallel nonrecursive program that repeatedly recreates a pipeline to perform force calculation for n gravitational bodies. During an n-body simulation the program activates parallel processes 300 times and procedures 513,553 times. These activations are handled by reusing the same 27 activation records over and over again.

The Laplace matrix is a highly parallel nonrecursive program. It creates parallel processes 25,609 times and calls procedures 67,156 times. These 92,765 block activations require only 64 activation records.

When these parallel program solve larger problems, the two nonrecursive programs run longer, but do not require more activation records. The number of activation records used by the quicksort tree increases slightly when the depth of the sequential recursion increases.

If no procedure is activated recursively or in parallel, the heap allocation uses the same amount of memory as static allocation (one activation record per block). In general, each block requires separate activation records for all activations of the block that may be in progress simultaneously (due to recursion or parallelism, or both).

## 5 Conclusions

I have described a simple heap mechanism for dynamic memory allocation of a parallel recursive program with Algol-like scope rules.

The mechanism has the following *advantages*:

- The heap allocation supports unbounded dynamic activation and termination of parallel processes and recursive procedures.
- The activation and termination of parallel processes and recursive procedures are equally fast.
- The heap allocation for parallel recursion is as efficient in reusing memory as the traditional stack discipline for sequential recursion.
- On a multicomputer with distributed memory, heap allocation is about as fast as stack allocation.

In its simplest form (presented here), the method has only two *limitations*:

- An activation record used to activate a block can only be reused by activating the same block again. This compromise makes it easy to release and reallocate the memory space of block activations.
- On a multiprocessor with shared memory, the need to lock and unlock the heap twice during a block activation makes the method less attractive.

Both limitations can probably be removed by more complicated variants of the basic idea. I leave that as an exercise for the reader.

**Acknowledgements**

It is a pleasure to acknowledge the comments of Art Bernstein, Ole-Johan Dahl, Ric Holt, Butler Lampson, Peter O'Hearn, Ron Perrott, and Jørgen Staunstrup.

## References

Andrews, G.R. 1981. Synchronizing resources. *ACM Transactions on Programming Languages and Systems 3*, 4 (October), 405–430.

Brinch Hansen, P. 1975. The programming language Concurrent Pascal. *IEEE Transactions on Software Enginering 1*, 2 (June), 199–207.

Brinch Hansen, P. 1978. Distributed processes: A concurrent programming concept. *Communications of the ACM 21*, 11 (November), 934–941.

Brinch Hansen, P. 1981. Edison—a multiprocessor language. *Software—Practice and Experience 11*, 4 (April), 325–361.

Brinch Hansen, P. 1989. A multiprocessor implementation of Joyce. *Software—Practice and Experience 9*, 6 (June), 579–592.

Brinch Hansen, P. 1994. The programming language SuperPascal. *Software—Practice and Experience 24*, 5 (May), 467–483.

Brinch Hansen, P. 1995. *Studies in Computational Science: Parallel Programming Paradigms.* Prentice Hall, Englewood Cliffs, NJ, (March).

Cook, R. 1980. ∗Mod—a language for distributed programming. *IEEE Transactions on Software Engineering 6*, 6 (November), 563–571.

Crookes, D. and Elder, J.W.G. 1984. An experiment in language design for distributed systems. *Software—Practice and Experience 14*, 10 (October), 957–971.

Davies G.L. and Burns, A. 1990. The teaching language Pascal-FC. *Computer Journal 33*, 147–154.

Dijkstra, E.W. 1960. Recursive programming. *Numerische Mathematik 2*, 312–318.

Fox, G.C., Johnson, M.A., Lyzenga, G.A., Otto, S.W., Salmon, J.K. and Walker, D.W. 1988. *Solving Problems on Concurrent Processors*, Vol. I. Prentice Hall, Englewood Cliffs, NJ.

Fox, G.C., Messina, P.C. and Williams, R.D. 1994. *Parallel Computing Works!* Morgan Kaufman, San Francisco, CA.

Hoare, C.A.R. 1978. Communicating sequential processes. *Communications of the ACM 21*, 8 (August), 666–677.

Holt, R.C. 1983. *Concurrent Euclid, the Unix Operating System and Tunis.* Addison-Wesley, Reading, MA.

Inmos Ltd. 1988. *occam 2 Reference Manual.* Prentice Hall, Englewood Cliffs, NJ.

Kaubisch, W.H., Perrott, R.H. and Hoare, C.A.R. 1976. Quasiparallel programming. *Software—Practice and Experience 6*, 3 (July–September), 341–356.

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

Lampson, B.W. and Redell, D.D. 1980. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (February), 105–117.

Naur, P. 1963. Revised report on the algorithmic language Algol 60. *Communications of the ACM 6*, 1 (January), 1–17.

Organick, E.I. 1973. *Computer System Organization: The B5700/B6700 Series.* Academic Press, New York.

Weinstock, C.B., and Wulf, W.A. 1988. Quick Fit: an efficient algorithm for heap storage management. *SIGPLAN Notices 23*, 10 (October), 141–148.

Welsh, J. and Bustard, D.W. 1979. Pascal-Plus—another language for modular multiprogramming. *Software—Practice and Experience 9*, 11 (November), 947–957.

Wirth, N. 1971. The programming language Pascal. *Acta Informatica 1*, 1, 35–63.

Wirth, N. 1977. Modula: a programming language for modular multiprogramming. *Software—Practice and Experience 7*, 1 (January–February), 3–35.