Picothreads for operating system code

Simon Peyton Jones

January 10, 2000

1 Introduction

Paul Barham outlined for us:

(a) The way he'd like to write operating system code. The code looks like a sequential program, but if it is executed like that it will run dreadfully slowly because it interacts synchronously with remote machines and devices.

(b) The way he is forced to write it. The code is split into a myriad fragments, driven off events such as returning messages from remote machines or device interrupts.

In practice, it is so difficult to get the event-driven code (b) to work correctly that people often use (a) and accept that its performance sucks.

This paper makes a first stab at defining a language that combines the programming convenience of (a) with the performance of (b). Indeed, to a first approximation, the language compiler simply transforms an (a) program into a (b) program.

My purpose in writing this is to put up an Aunt Sally for you all to shoot at. I'm sure I have multiple misconceptions about current practice and the nature of the problem we are trying to solve, for example. By writing this I hope to get those misconceptions out in the open, where they can be corrected.

There's lots of related work that I mention in Section 6.

2 The problem

The problem is this. *Code that is easy to understand runs like a dog; code that runs fast has a totally different structure, and is much harder to write and debug.*

In this section we explain why this is so.

2.1 Overlapping long lantency operations

We want to write a procedure that (say) reads a file from a disk like this:

```
read( x,y ) {
   int r1,r2,ans;

   r1 = readFromDisk1( x );
   r2 = readFromDisk2( y );

   ans = combineResults( r1, r2 );

   return( ans )
}
```

Here I'm supposing that the `read` operation requires getting information from two separate disks and somehow combining the results. The problem with this program is, of course, that if the call to `readFromDisk1` is synchronous, the two reads take place sequentially.

An obvious "solution" is to fork a thread for `readFromDisk1`, and have some sort of synchronisation before `combineResults`:

```
read( x,y ) {
   int r1,r2,ans;

   r1 = readFromDisk1( x );  /* Spawn */
   r2 = readFromDisk2( y );  /* Spawn */

   sync;  /* Wait for the reads to complete */

   ans = combineResults( r1, r2 );

   return( ans )
}
```

Why don't OS authors do this? I think because it's just too expensive to fork a thread. I think that the really expensive aspect is having to allocate a stack of unknown size, which involves fiddling with the memory management hardware. [**Note:** *I'd like to amplify what the problem really is.*]

To overlap the two disk reads, we are forced to write something like this:

```
read( x,y ) {
   sendReadToDisk1( x );
   sendReadToDisk2( y );
   go to sleep
}

event_loop() {
   e = getNextEvent();
```

```
   if e is reply-from-disk1 then
     r1 = result(e)
     if disk2 has replied already,
     ans = combineResults( r1,r2 )
     wake up with( ans )

   else if e is reply-from-disk2 then
     ...similar code

   else
     ...other events
}
```

[**Note:** *Is this code anything like accurate?*]

The call `sendReadToDisk1` probably allocates a little "work item", a packet of data that will be sent to the disk unit (or whatever). The event loop coordinates the return of completed work items.

The code above is a total mess. It's a bit like re-implementing a multi-tasking operating system each time you want to write a new application! [**Note:** *If this claim is true, it would be worth amplifying.*]

2.2 Spatial locality

So far I have concentrated on gaining performance by overlapping long-latency operations. But there is another reason for wanting to divide a complex task into small "work items". It is described with clarity, wit, and authority by Parkes and Vicik in their paper "A new architecture for high-performance pipeline servers"; here I will simply summarise briefly.

The key idea is this. When performing a complex task for many separate clients, the obvious structure of having one thread per client gives absolutely rotten spatial locality in many applications. (Parkes and Vicik discuss the Microsoft SQL server.) Much better is to divide the task for each client into multiple work items, and have a thread to execute each pool of similar work items. In this way, the thread gets to suck a chunk of global database state into the cache, and use it repeatedly to do a similar operation on behalf of many clients. By providing infrastructure to support this style of programming they get big performance improvements.

From this standpoint, this paper aims to generalise their "pipeline server" and make it easier for programmers to use.

2.3 Heterogeneous and distributed processing

There is yet another reason that we may be forced to split up a complex task into "work items": it may be an absolute requirement that different parts of the task are executed by different processors, or at different places. For example, a file read in a network file system involves at least the client OS and the server, and maybe a name server on the way. At the lowest level, we create work items for hardware devices, like disks and netowrk interfaces, and they are clearly different "processors" than the main CPU.

Now, you can regard the creation of a remote "work item" as a sort of long-latency operation (and thus a special case of Section 2.1); or you could say that the reason it must be performed remotely is because spatial locality requires it (which is like Section 2.2).

Nevertheless, it helps to reinfoce the point that we are definitely *not* in the conventional parallel-programming world in which there is (logically) a single pool of runnable threads, any of which can be run by any processor.

2.4 IO Request Packets

The Windows NT input/output architecture has the *IO Request Packet* (IRP) as a fundamental component. An IRP records the parameters of a requested IO operation; it records how far that operation has progressed; it even has a little fixed-size stack in which layered drivers can record their part of the state when the pass the request down to lower drivers.

An *IO Manager* is responsible for scheduling IRPs for execution. It is important that the IO Manager is entirely independent of the IO operations themselves; it glues them together.

These IRPs are clearly "work items" of the sort I have informally suggested above. But programming device drivers using IRPs is pretty complicated, involving explicit stack manipulation, call-backs, and much else besides.

2.5 The Holy Grail

Here's a summary of what we are trying to achieve.

- We want overlap long-latency operations without destroying the structure of our programs.
- We want to exploit spatial locality by grouping together similar chunks of work and executing them in batches.
- We want to build on the performance-related insights embodied in the Pipeline Server, and in IO Request Packets, making it much easier to achieve the performance they offer.
- We aren't allowed to create lots of fully-fledged threads with stacks of unknown sizes.

3 Towards a design

Here is how one would write the program in the language I'm tentatively proposing:

```
forkable read( x,y ) {
   int r1,r2,ans;

   par {
      r1 = readFromDisk1( x );
   ||
      r2 = readFromDisk2( y );
   }

   ans = combineResults( r1, r2 );

   return( ans )
}
```

The idea is this:

- The `par` construct forks one or more *work units*, and then waits for them all to complete. So execution only reaches the call to `combineResults` when both work units have completed.
- Each work unit is specified by a statement that looks like a procedure call (e.g. `r1 = readFromDisk1(x)`).
- The `par` construct actually allocates a little block of memory for each work unit — we will call this block a "work unit" too — containing the procedure to call (e.g. `readFromDisk1`), its argument (`x`), and a pointer to where to deposit its result (`&r1`).
- This work unit is somehow scheduled for processing by the appropriate disk or disk driver. When it is completed, it writes its result back into the specified place (e.g. `r1`).
- The `forkable` keyword `read` indicates that it can be invoked by a `par` in some other procedure. It cannot be called as an ordinary C procedure.

From a programming language point of view, there is nothing exotic about this; par-begin has been around since *Ask Tony for an appropriate reference.* `par` was directly supported in Occam and the Transputer.

Indeed the "right" way to think about a work unit is to think of it as a thread: it has state (parameters, local variables); there is some scheduler that decides when to execute it; it can spawn other work units (threads); it can suspend in the middle of its execution.

What is new here is the *limitations* we will place on the scheme, so that we can get a very efficient implementation. In particular, a work unit is represented by a *fixed size* block of memory, whereas a conventional thread has a thread control block (TCB) *plus an arbitrary-sized stack of activation records (stack frames).* The contents of a work unit is essentially the activation record of a single procedure.

The language is designed to be C (or perhaps C++), with a few extra constructs layered on top. A compiler for the language simply slurps in the program, deals with the extra constructs leaving the rest alone, and spits out a vanilla C/C++ program. You can think of the compiler as a preprocessor for C/C++. But it can't be done with `cpp`!

3.1 Service queues and servers

Work units waiting for service are put in a *service queue* of of related work units. I call it a "queue", but perhaps "pool" would be a better term. There is no built-in first-in first-out property; that is under the programmer's control.

To take a low level example (suggested by Paul), a work unit to read a disk block might be put in the service queue of work for a particular disk arm. A service queue is usually a linked list of work unit blocks, but it might also be implemented in hardware. That is, a work unit block might be physically allocated in a harware device, and implicitly queued. [**Note:** *Details vague in the extreme, but you get the idea.*]

A service queue is served by a *server*. A server is something like a real live OS thread, but it might also be a device. In the disk arm case, I'm not sure whether the queue of requests is snarfed off the disk arm's service queue by software or hardware; either way I'll call it a "server".

There may be lots and lots of service queues. Some of them might simply be "work to do with X", and the OS might periodically decide to deal with X by processing X's service queue. So service queues are the way in which work gets batched up. However, a service queue does not necessarily contain work units of a uniform shape. Each work unit contains its program counter, that determines *what* is done; the service queue determines *when* it is done, and *which server* does it.

The fact that there are lots of service queues is a big difference from conventional multi-threading substrates, which typically support a single pool of runnable threads. This pool might be physically distributed, but it's usually the case that there's a single logical pool.

The most common case is that the server is implemented in software. In that case, the server is essentially a conventional thread. It has a program counter, a stack, registers, etc. It services the queue like this:

```
service( q ) {
  while not_empty(q) {
    wu = pop(q);
    (wu->work)(wu);
} }
```

It repeatedly pulls a work unit off the queue, and calls the procedure recorded in the work item, passing the work item itself as the first argument (so that the procedure can get at the parameters recorded there). Of course, a server is free to stop before it has emptied the queue. Indeed, servers aren't built in all; the programmer writes them. (They are pretty simple, after all!).

The procedure in a work unit must not block the server. (Indeed, a server is likely to be a privileged OS thing, so I'm not sure how it *could* block.) This guarantees that a server can simply chomp up work units until it wants to do something else.

3.2 The big limitation

As mentioned earlier, I propose a much more limited model than the one supported by fully-fledged OS threads. The basic design choice is that each work unit is represented by a fixed-size block of memory. Here are the limitations this imposes:

- You can only fork a procedure with `par` (such as `readFromDisk1`) that you have identified as forkable with the `forkable` keyword. Forkable procedures can only be called by `par`; they cannot be called with ordinary C procedure calls.
- A forkable procedure can call an ordinary C procedure; the call to `combineResults` is an example. There is no implementation problem here, because the call will take place on the stack of the server.

- *However, ordinary C procedures cannot use* par. Why not? Because then we might need to suspend (while the par waits for its children), and now we have to capture an unknown amount of suspended procedure activation records.

This is a big limitation. It represents a regrettable loss of abstraction: we can't wrap up a common pattern of pars in a procedure. But we can still wrap up the pattern in a macro (expanded inline) or in a forkable procedure (with some runtime cost).

My main reason for supposing that this is an acceptable limitation is that I believe that, when you look really closely, existing high-performance systems embody essentially the same constraint.

There must be some way to get things started, presumably by providing a way to fork a forkable procedure from a standard OS-supported thread, with some way to synchronsise on its result.

3.3 parfor, and inlet code

A simple extension supports forking a statically-unknown number of threads:

```
parfor (i=0; i<n; i++) {
  r[i] = foo(i) ;
}
```

As before, the par construct only completes when all its children have completed.

It is quite common to want to do a little processing when the result comes back. For example, maybe we want to add the results together instead of putting the results in an array. To do this, we write a little *inlet code* (so-called for historical reasons [*et al*91]) after the forked procedure call:

```
tot = 0;
parfor (i=0; i<n; i++) {
  r = foo(i) ;       /* Forked call */
  tot = tot + r;     /* Inlet code */
}
```

The inlet code can contain arbitary C code, but must not use par. Why not? Because the work unit is already suspended waiting for all the threads to complete. We don't want the inlet code to suspend somewhere else, or we will end up with two logical suspended control points in a single work unit! The inlet code must run to completion without suspension.

The idea of inlet code that integrates a result into an a suspended computation is quite well known: I think TAM used the term first [*et al*91], and Cilk does the same thing [FLR98].

par and parfor can be nested (just as in Occam). Such nests are really just a way of specifying a single big par. For example:

```
par {
  parfor (...) {
    r = foo(i);
    tot += r;
  }
||
    r = baz();
    tot += r
}
```

I don't think parwhile makes sense.

[**Note:** *It's not clear how to implement inlet code that mentions a variable that isn't constant for all members of the* par*. For example, should we allow this?*]

```
parfor (...) {
  r = foo(i);
  tot[i/2] += r;
}
```

[**Note:** *The returning value must somehow convey* i *to the inlet code. I don't know how to do this.*]

3.4 Abortion

Paul tells me that it is sometimes important to wait for just a subset of the forked calls to complete. So in the inlet code we might want to say "if enough is now done, terminate the par". We use the keyword breakpar for this purpose. Example:

```
n = 0;
parfor (i=0; i<N; i++) {
  r = foo(i);
  ...record r...
  if (++n > N/2) breakpar ;
}
```

In this example, we break out of the parfor when more than half the forked work units have completed.

What happens to the others? One possibility is to leave them to go their merry way, but arrange that their inlet code is subverted to simply discard their results.

But in general we probably need a way to abort a work unit. Certainly, a great deal of effort in Win NT IRPs goes into dealing with abortion. So probably we need a way to abort a work unit, and breaking out of a par should (somehow) abort the incomplete work units.

Timeouts can be handled in this way, by forking a "call" to a special timeout forkable work unit, whose inlet code aborts any un-completed work units.

[**Note:** *Abortion requires that the parent keeps track of its children, so that it can abort them. It's unclear how to do this in general. Maybe the children are simply linked together, and have an* abort *entry point. But that increases the overhead per* par.]

3.5 Blocking

Urk! If a work unit can't block, how do we implement the fact that par must await the completion of its children? Simple: the par simply returns (to the server, which gets to work on the next work unit). When the par'd work units are all complete, the work unit is rescheduled by putting it on some suitable service queue.

While the work unit is blocked, it isn't on any service queue (though we might want to keep it on some queue just so we can find work units that have got blocked for too long).

3.6 Scheduling

When we fork a work unit, or reawaken one when a `par` completes, we are specifying a chunk of work to be done "sometime". But there are two little bits of work have to be done right away:

- When forking, we must allocate and initialise the work unit.
- When forking or joining we must *schedule* the work unit — that is, put it in an appropriate service queue.

It's pretty clear (to me) that both of these actions should be controllable by the programmer. "Built-in" choices are not going to work. For example:

- We definitely might want to specify which of the many service queues is the Right Place for this work unit
- The Right Place might depend on the parameters of the `fork` call. For example, a remote file access might want to go into a queue that depend on which network interface is to be used.
- We might want to allocate a work unit in some hardware queue instead of `malloc`ing it.

Quite how to make all of this programmer controllable isn't so clear. One fairly simple possibility is to allow the programmer to specify a scheduler when s/he defines a `forkable` procedure:

```
forkable read( x,y ) schedule {
      /* The scheduler code */
      addToQueue( theRightQueueFor(x), # )
} {
     ...code for read as before...
}
```

One can put arbitrary C code as the scheduler code. The symbol '`#`' is replaced by a pointer to the work unit (syntax arbitrary of course) Here `addToQueue` is some standard queue manipulating procedure, while `theRightQueueFor` looks at `x` to decide which queue to add the work unit to.

Even for scheduling, one might want to specify a scheduler independently for each `sync`, as well as for the initial `fork`:

```
forkable read( x,y ) schedule {
      /* The scheduler code */
      addToQueue( theRightQueueFor(x), # )
} {
   int r1,r2,ans;
   join j; initJoin(j);

   par {
      r1 = readFromDisk1( x );
   ||
      r2 = readFromDisk2( y );
   } schedule {
      addToQueue( ReadResultsQueue, # )
   };

   ans = combineResults( r1, r2 );

   return( ans )
}
```

3.7 Allocation

Under this scheme, the *allocation* of the work unit is still implicit; it's the *scheduling* that is controlled by the programmer. Making allocation explicit too seems more gross, but it too is useful, I think. For example, one might allocate directly in the hardware. Or one might want to allocate remotely: that is, send a message to some remote machine that will cause it to allocate and then schedule the work unit.

Syntax?

4 Implementation

How is all this to be implemented? We just do the standard object-oriented thing. We compile each forkable procedure `p` to:

- `p`'s work-unit structure, `struct_p`, which gives the layout of the work unit. The high level idea is that a work unit corresponds to a thread, but the low level view is that it is an object with a method suite that knows how to manipulate that object.
- `p`'s forker, `fork_p`, a procedure that allocates a work unit for `p`.
- `p`'s scheduler, `sched_p`, a procedure that deals with the work that has to be done when `p` is `fork`'d or reawakened.
- `p`'s worker, `work_p`, a procedure that contains the code that the programmer wrote.

So, `read` might generate `struct_read` thus:

```
struct {
  void *methods;       /* Address of work_read */
  workunit *parent;    /* Calling work unit */
  int inlet;           /* And inlet number */
  workunit *link;      /* Link field */


  int pc;         /* Entry point */
  int x,y;        /* Arguments */
  int r1,r2,ans;  /* Locals */
  int j;          /* Number of pending results */
} struct_read;

void *read_methods[] = {
  work_read,   /* Main code */
  in_read_r1,  /* Inlet for r1 */
  in_read_r2   /* Inlet for r2 */
}
```

The first four fields are the same in all work units; the rest can vary on a per-procedure basis. The fork_read procedure allocates, initialises, and schedules the work unit.

```
fork_read( p,i, x,y,pr, ) {
  *struct_read r = malloc( sizeof(struct_read) )
  r->work = work_read; r->sched = sched_read;
  r->parent = p; r->inlet = i;

  r->pc = 0;
  r->x = x; r->y = y;

  addToQueue( theRightQueueFor(r->x), r );
  return;
}
```

The main routine, work_read embodies multiple entry points by a standard trick: represent the "program counter" by an integer, and dispatch on it to the appropriate label. All this routine's state that is live across syncs is stored in its work unit, r. (One could load this all into local variables at the start and store it before each return; a good C compiler should generate good code for that.)

```
work_read( *struct_read r ) {
  switch r->pc {
    0: goto lbl0;
    1: goto lbl1;
  }

  /* Main entry point */
  lbl0:
    r->j = 2;
    fork_readDisk1(x, &r, 1);
    fork_readDisk2(y, &r, 2);
    r->pc = 1;
    return;

  /* Return from sync */
  lbl1:
    ans = combineResults(r->r1,r->r2);
    (r->parent->methods[inlet])(r->parent, ans);
    return;
}
```

The inlet procedures simply set the appropriate result, check whether enough results have been returned, and reschedule the work unit if so.

```
in_read_r1( *struct_read r, int val ) {
  r->r1 = val;
  if (-- (r->j) /= 0) return;
  addToQueue( theRightQueueFor(r->x), r );
}

in_read_r2( *struct_read r, int val ) {
  r->r2 = val;
  if (-- (r->j) /= 0) return;
  addToQueue( theRightQueueFor(r->x), r );
}
```

Notice that if we override fork_read somehow, that gives us a great deal of control. One could interact directly with a device, for example, or send a message to a remote machine asking it to allocate and schedule the work unit.

There's some standard signal routine that decrements the count in a join cell, and schedules the work unit if the count has gone to zero.

5 Random thoughts

- The methods pointer in each work unit points to its method suite. But the same pointer should give access to
  - The size of the work unit.
  - Maybe some debugging information.

  Essentially all the stuff that is true about all the instances of this "class".

6 Related work

6.1 Cilk

Cilk has a lot in common [FLR98, BJK+95]. It is a preprocessor for C, and mangles the code of each procedure much as described for work_read above (multiple entry points, activation frames explicitly allocated).

Cilk's emphasis is very different, though. The purpose of the parallelism is to exploit the availability of multiple processors in an SMP. Cilk therefore places heavy emphasis on only forking when necessary; it generate *two* copies of every procedure, a "fast clone" a "slow clone". Work is stolen from the *base* of the stack. It's all very, very cunning and impressive.

In our setting, I think that when we say fork we mean fork. We know there's a long latency operation, and we definitely want to fork it.

Cilk has a single, simple sync operation, and no join cells. sync waits for all locally-forked threads to terminate. I think we want more precise control than that.

6.2 Cid

join cells are stolen diectly from Cid [Nik94], which in turn abstracted them from the work on dataflow in which Nikhil was deeply involved. Cid's main goal is latency-hiding in a distributed memory machine, so we share that in common. But Cid assumes an underlying very-lightweight threads packages; threads can block with an arbitrary sized stack. There is no provision for handling stack overflows.

Cid focuses a lot of attention on global pointers, and primitives to support fetching remote data. That's not so appropriate in our setting.

Acknowledgements

Paul Barham gets the credit for starting this hare running. He also pointed me to the Pipeline Server paper.

Many thanks to Tony Hoare and Butler Lampson for an illuminating debate. I changed the design significantly in

the light of our discussion; in particular, replacing join cells with `par`.

References

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, July 1995.

[*et al*91] DE Culler *et al.* Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *Architectural support for Programming Languages and Operating Systems, Santa Clara*, pages 164–175. ACM, April 1991.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM Conference on Programming Languages Design and Implementation (PLDI'98)*, volume 33, pages 212–223, Atlanta, May 1998. ACM.

[Nik94] RS Nikhil. Cid: a parallel "shared-memory" C for distributed-memory machines. In *Proc Workshop on Languages and Compilers for Parallel Computing*, volume 892. Springer Verlag LNCS, August 1994.