Process-Oriented Building Blocks

Adam Sampson
Institute of Arts, Media and Computer Games
University of Abertay Dundee
Threading Building Blocks

• Open-source C++ library from Intel
• Intended to replace OpenMP (more or less) in compute-heavy parallel applications – e.g. parallel_for(0, 10, some_function);
• ... but it's actually quite sophisticated underneath:
  - Built on non-preemptible tasks
  - Work-stealing multicore scheduler
  - API for submitting raw tasks, groups, etc.
  - Portable atomics, threads
You can abuse the scheduler to do **concurrent** (rather than **parallel**) programming...

... by writing in continuation-passing style

So you can fairly easily knock up an implementation of channels on top of TBB...
PROC element (CHAN INT this?, next!) class Element {
    INITIAL INT token IS 1:
    WHILE token <> 0
        SEQ
            this ? token
            IF
                token > 0
                    next ! (token + 1)
                TRUE
                    next ! token
        :
}

class Element {
public:
    Element(pobb::channel<int>& cthis,
        pobb::channel<int>& cnex)
    : cthis_(cthis), cnex_(cnex),
        token_(1) {
    }

tbb::task *start() {
    if (token_ != 0) {
        return cthis_.read(&token_,
            pobb::make_cont(*this,
                &Element::handle));
    } else {
        return 0;
    }
}

tbb::task *handle() {
    if (token_ > 0) {
        return cnex_.write(token_ + 1,
            pobb::make_cont(*this,
                &Element::start));
    } else {
        return cnex_.write(token_,
            pobb::make_cont(*this,
                &Element::start));
    }
}

private:
    pobb::channel<int>& cthis_;  
    pobb::channel<int>& cnex_; 
    int token_; 
};
PROC element (CHAN INT this?, next!) class Element {

  INITIAL INT token IS 1:
  WHILE token <> 0
    SEQ
      this ? token
      IF
        token > 0
          next ! (token + 1)
        TRUE
          next ! token
  :

  public:
    Element(pobb::channel<int>& cthis,
            pobb::channel<int>& cnext)
      : cthis_(cthis), cnext_(cnext),
        token_(1) {
    }

    tbb::task *start() {
      if (token_ != 0) {
        return cthis_.read(&token_,
                           pobb::make_cont(*this,
                                           &Element::handle));
      } else {
        return 0;
      }
    }

    tbb::task *handle() {
      if (token_ > 0) {
        return cnext_.write(token_ + 1,
                             pobb::make_cont(*this,
                                             &Element::start));
      } else {
        return cnext_.write(token_,
                             pobb::make_cont(*this,
                                             &Element::start));
      }
    }

  private:
    pobb::channel<int>& cthis_; 
    pobb::channel<int>& cnext_; 
    int token_; 
};

Eww.

It works, but I wouldn't want to have to do this for anything non-trivial...
• This kind of translation is pretty mechanical... someone must have done this already, right?
  – (Pretty close to what SPoC does, in fact)

• Enter Gabriel Kerneis' **Continuation-Passing C**
  – Adds cps annotations to C: this function might be descheduled – so translate to CPS
  – Portable translator to ANSI C
  – Includes a very simple, single-core scheduler

```cpc
void server_process(int fd) {...}

cpc void main_loop() {
    while (true) {
        int fd = accept(sock);
        cpc_spawn(server_process(fd));
    }
}
```
I replaced CPC's scheduler with a wrapper around TBB's low-level interface

- ... pretty straightforward
- CPC's generated code is thread-safe, but many applications aren't – assume only one thread

I reimplemented (some of!) KRoC's “CIF” library of process-oriented primitives – Chan0ut, etc.

- ... can just translate CCSP's algorithms
PROC element (CHAN INT this?, next!) cps void element (Channel *this, Channel *next) {
    int token = 1;
    while (token != 0) {
        ChanInInt (this, &token);
        if (token > 0) {
            ChanOutInt (next, token + 1);
        } else {
            ChanOutInt (next, token);
        }
    }
}
static struct cpc_continuation *element(struct cpc_continuation *cpc_cont)
{
    struct element_arglist {
        int this __attribute__((__aligned__(16))) ;
    };
    inline static struct cpc_continuation *element_push(int this, struct cpc_continuation *cpc_cont_2732);
    struct element_arglist *cpc_arglist1;
    {
        cpc_arglist1 = (struct element_arglist *)cpc_alloc(& cpc_cont_2732, (int )sizeof(struct element_arglist ));
        cpc_arglist1->this = this;
        cpc_cont_2732 = cpc_continuation_push(cpc_cont_2732, (cpc_function *)(& element));
        return (cpc_cont_2732);
    }
}
struct cpc_continuation *__element_pc22(struct cpc_continuation *cpc_cont)
{
    struct __element_pc22_arglist {
        int next ;
        int this ;
        int *token __attribute__((__aligned__(16))) ;
    };
    inline static struct cpc_continuation *__element_pc22_push(int next, int this , int *token, struct cpc_continuation *cpc_cont_2737);
    struct __element_pc22_arglist *cpc_arglist1;
    {
        cpc_arglist1 = (struct __element_pc22_arglist *)cpc_alloc(& cpc_cont_2737, (int )sizeof(struct __element_pc22_arglist ));
        cpc_arglist1->next = next;
        cpc_arglist1->this = this;
        cpc_arglist1->token = token;
        cpc_cont_2737 = cpc_continuation_push(cpc_cont_2737, (cpc_function *)(& __element_pc22));
        return (cpc_cont_2737);
    }
}
struct cpc_continuation *__element_pc23(struct cpc_continuation *cpc_cont)
{
    struct __element_pc23_arglist {
        int next ;
        int this ;
        int *token __attribute__((__aligned__(16))) ;
    };
    // ... and so on
That I can live with...

I've ported over most of the CCSP benchmark suite – including **agents**
Conclusions

- You can write concurrent programs in **portable** C
- Use TBB's multicore work-stealing scheduler
- Easy to add new synchronisation objects
- Combine with TBB's existing smart data-parallel functions
- Performance *seems* good – no proper analysis
  - … which is why this is a fringe presentation!
- Minor downside: I ran into some problems with CPC's translator – but should be fixable
- If this sounds interesting, talk to me...