Picking Patterns for Parallel Programs

Anthony Williams

Just Software Solutions Ltd
http://www.justsoftwaresolutions.co.uk

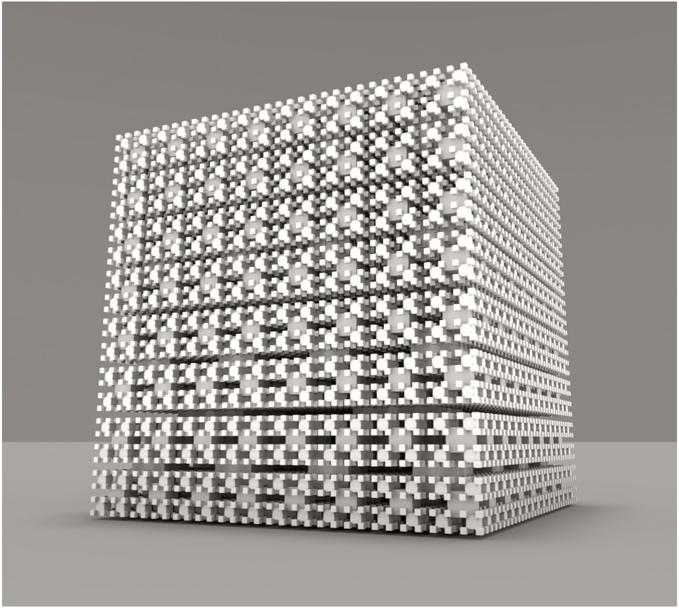
16th April 2011

- Buzzword Bingo
- Structural Patterns
- Communication Patterns
- Choosing Patterns

Buzzword Bingo

Actors	Agents	Message Queues
Tasks	CSP	Dataflow
Map-reduce	Locks	Continuations
Lock-free	False Sharing	Fork/Join
Work-Stealing	Pipelines	SPMD

Structural Patterns



Loop Parallelism

Loop Parallelism (I)

- Apply the same operation to many independent data items
- Great for **Embarrasingly Parallel** problems
- Frameworks commonly provide a `parallel_for_each` operation or equivalent

Loop Parallelism (II)

```
std::vector<some_data> data;
parallel_for_each(data.begin,data.end(),process_data);

#pragma omp parallel for
for(unsigned i=0;i<data.size();++i) {
  process_data(data[i]);
}
```

Fork/Join

Fork/Join (I)

- Subdivide into parallel tasks, and then wait for them to complete
- Often used recursively
- Works best when used at the top level
- Need to watch for uneven workloads

Fork/Join (II)

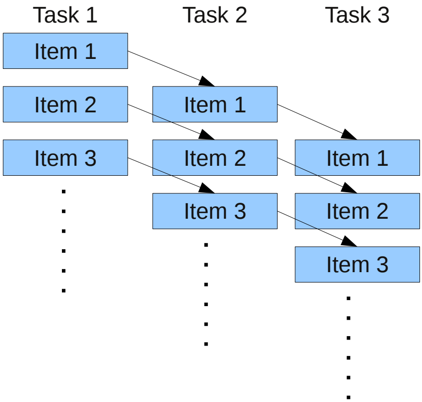
```
template<typename Iter,typename Func>
void parallel_for_each(Iter first,Iter last,Func f) {
    unsigned long const length=std::distance(first,last);
    if(length<minimum_split_length) {
        std::for_each(first,last,f);
    } else {
        Iter const mid_point=first+length/2;
        auto top=std::async([=]{
              parallel_for_each(first,mid_point,f);});
        auto bottom=std::async([=]{
              parallel_for_each(mid_point,last,f);});
        top.wait(); bottom.wait();
    }
}
```

Pipelines

Pipelines (I)

- A set of discrete tasks that must be applied in sequence
- The same sequence of tasks need to run over a large data set
- An alternative to loop parallelism, where the order of the data is important

Pipelines (II)



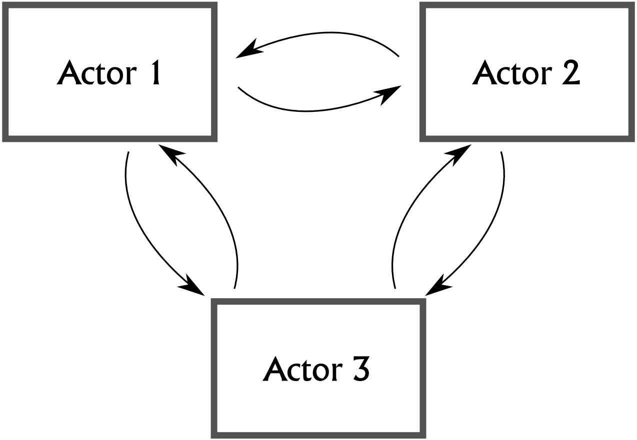
Pipelines (III)

- Maximum parallelism is the number of tasks in the pipeline
- Can mess with cache locality:
  - If each **task** is always run on the same core then the data must be moved between cores
  - If **data** is kept on the same core, then the task code must be reloaded for each step

Actors




Actors (I)



Actors (II)

- Each actor runs entirely isolated (no shared state)
- The only communication between actors is via message queues
- In some languages these rules are enforced. In C++ it is **your** responsibility to follow them.

Actors (III)

Upsides:

- Each actor can be analysed independently
- Data races are impossible (if you follow the rules)
- Can be easier to reason about

Actors (IV)

Downsides:

- Not good for short-lived tasks
- Message passing isn't always the best communication mechanism
- Scalability is limited to the number of actors

Building an ATM with Actors

3 actors:

- User Interface
- Core ATM Logic
- Communicating with the bank

Actors (VI)

```
struct ping { jss::actor_ref sender; };
struct pong {};
jss::actor pp1([] {
        jss::actor::receive().match<ping>(
            [](ping p) {
                p.sender.send(pong());
            });
    });
jss::actor pp2([&] {
        pp1.send(ping{jss::actor::self()});
        jss::actor::receive().match<pong>(
            [](pong){});
    });
```

Speculative Execution

Speculative Execution (I)

Make use of available concurrency:

- Perform tasks that **might** be needed
- Execute multiple algorithms to obtain a result
- Execute dependent tasks with predicted data

Speculative Execution (II)

- Can reduce **latency**
- Increases total work done
- Cancelling speculative tasks can be tricky
- Speculative tasks must be **side-effect free**

Speculative Execution (III)

```
template<typename Iter,typename Match>
Iter find(Iter first,Iter last,Match M,unsigned N) {
  unsigned long const D=std::distance(first,last);
  std::atomic<bool> done(false);
  std::promise<Iter> p;
  std::vector<std::future<void>> v(N);
  for(unsigned i=0;i<N;++i) {
    Iter const end=(i==(N-1))?last:(first+D/N);
    v[i]=std::async(Searcher<Iter,Match>(first,end,M,done,p
    first=block_end;
  }
  for(auto&f:v) { f.wait(); }
  return done?p.get_future().get():last;
}
```

Speculative Execution (IV)

```
template<typename Iter,typename Match>
struct Searcher {
  Iter first; Iter last; Match const& M;
  std::atomic<bool>& done; std::promise<Iter>& p;
  Searcher(...); // obvious constructor impl

  void operator()() {
    for(Iter it=first;it!=last && !done;++it) {
      if(*it==M) {
        try { p.set_value(it); }
        catch(std::future_error) {}
        done=true; return;
      } } }
};
```

Map/Reduce


PEDNANDREA
Tel Ex
Coun
Off
Nancherrow
Terrace
PW
BOSWEDDEN ROAD
CHAPEL ROAD
CHAPEL STREET
NORTH ROW
B3306
Recn Gd
Sch
Cape Terrace
Venton East Hill
PW
A3071
CHURCH ST
PRINCESS ST
QUEEN ST
PLACE
F Sta
MARKET ST
Pleasant
Terrace
Sch
WEST
LA
FROWDA
PO
Bethany
Place
New Downs
Liby
FORE STREET
St Just
BOSORNE ST
TERR
CLOSE
ROAD
REGENT T
PW
BOSORNE
Slaughter Hous
W
Pol
Sta
SOUTH PLACE
Football
Ground
Recn Gd
CARRALLACK
M
Murray
Villas
Carn Bosavern
Carrallack
Carrallack
Farm
W
W

→

Map/Reduce (I)

An algorithm in two halves:

- Map: `x[i]` $\rightarrow$ `f(x[i])`
- Reduce: Accumulate the results (e.g. $\sum$`f(x[i])`)

(Aside: Google's MapReduce operates on key/value pairs)

Map/Reduce (II)

- Scales well with multiple cores
- Used in OpenMP and MPI
- Covers many parallel algorithms
- The map and reduce operations must be thread-safe
- The overhead depends on the data granularity

Map/Reduce (III): Word count

```
typedef std::map<std::string,unsigned> map_type;
map_type count_words(word_list const& words) {
  map_type counts;
  std::string word;
  while(words.get_next(word))
    ++counts[s];
  return counts;
}
map_type combine_maps(
  map_type counts,map_type const& other) {
  for(auto const& entry: other)
    counts[entry.first]+=entry.second;
  return counts;
}
```

Dataflow

- You specify the relationships, and the runtime handles the parallelization
- Has a “Functional Programming” style feel, even in imperative languages

Dataflow (II): Simple Variables

```
int main() {
  jss::dataflow::variable<int> x,y,z;

  z.task([&]{return x.get()+y.get();});
  x=23;
  y=19;
  assert(z.get()==42);
}
```

Dataflow (III): Channels

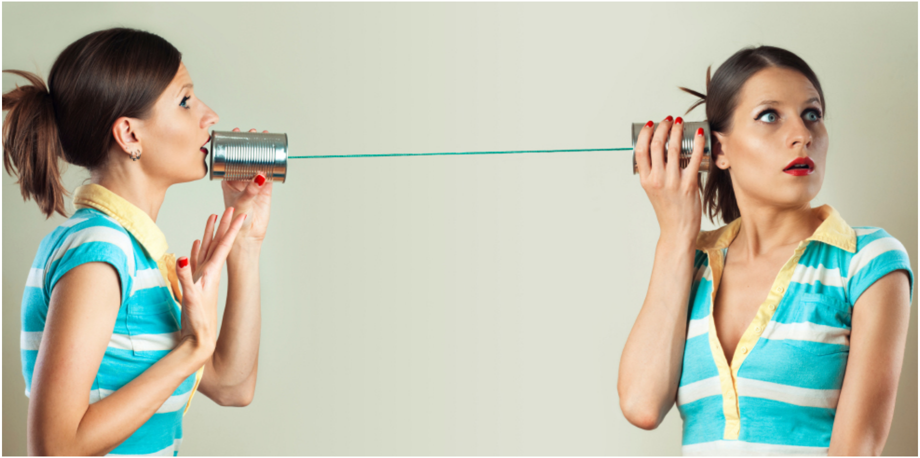
```
int main() {
  int const count=100;
  jss::dataflow::channel<int> x,y,z;
  jss::dataflow::task([&]{
    while(true) { z<<(x.pop()+y.pop()); }});
  jss::dataflow::task([&x]{
    for(int i=0;i<count;++i) { x<<rand()%20; }});
  jss::dataflow::task([&y]{
    for(int i=0;i<count;++i) { y<<rand()%20; }});

  for(int i=0;i<count;++i) { std::cout<<z.pop()<<'\n'; }
}
```

Dataflow (IV)

- Dataflow concurrency is an extension of the pipeline pattern to a general DAG
- The runtime provides facilities to split, filter and combine channels

Communication Patterns



Communication Patterns

- Read Only Data
- Mutexes and Locks
- Futures
- Queues and Channels
- Other Data Structures Designed for Concurrent Access

Mutexes and Locks

Mutexes and Locks (I)

- Explicitly limit concurrency!
- Low level mechanism
- Good for migrating sequential code
- Wrappers such as `synchronized_value<T>` can avoid correctness issues

Mutexes and Locks (II): synchronized_value<T>

```
void foo(jss::synchronized_value<std::string>& s) {
    std::string local=*s;
    s->append("foo");
    jss::update_guard<std::string> guard(s);
    unsigned pos=guard->find("f");
    if(pos==std::string::npos)
        *guard += "bar";
    else
        (*guard)[pos] = 'b';
}
```

Futures

Futures (I)

- Synchronization is internal to library
- Read and write operations are explicit
- One-shot communication
- Every task framework has some form of future

Futures (II)

In C++0x, futures are used with:

- `std::async`, as in the fork/join example
- `std::promise`, as in the search example
- `std::packaged_task`, a building block for task queues and thread pools

Queues and Channels

Queues and Channels (I)

- Allow multiple data items to be transferred
- Fundamental to message-passing systems
- Myriad of choices:

  SP/MP, SC/MC, bounded/unbounded, etc.

Queues and Channels (II): Bounded vs Unbounded

- **Bounded** Queues limit the number of items in the queue

  $\Rightarrow$ Producer blocks if the queue is full

- **Unbounded** Queues have no such limit

  $\Rightarrow$ May consume a lot of memory if producer runs faster than consumer

Queues and Channels (III): Bounded vs Unbounded

- Unbounded queues handle “bursty” data better
- Bounded queues even out the processing

Queues and Channels (IV): SPSC and MPSC

- **S**ingle **P**roducer **S**ingle **C**onsumer

  ⇒ one-to-one channel between 2 threads, actors or tasks

- **M**ultiple **P**roducer **S**ingle **C**onsumer

  ⇒ standard "message passing" queue, such as an Actor's mailbox

Queues and Channels (V): SPMC and MPMC

- **S**ingle **P**roducer **M**ultiple **C**onsumer

  $\Rightarrow$ broadcast channel

- **M**ultiple **P**roducer **M**ultiple **C**onsumer

  $\Rightarrow$ general purpose queue

Queues and Channels (VI)

Special-purpose queues may have additional properties

- Work-stealing queues
- Dataflow channels
- Priority queues

Other Data Structures Designed for Concurrent Access

Other Data Structures Designed for Concurrent Access (I)

- More than one thread can access the data structure concurrently, without either one waiting for the other
- There may be restrictions on which operations can be called concurrently

Other Data Structures Designed for Concurrent Access (II)

- Stacks
- Hash tables
- Other variations on lists, sets, maps and queues

Concurrent Hash Tables (I)

- Allow concurrent queries and modifications
- **May** allow concurrent removal
- **May** allow concurrent iteration

Concurrent Hash Tables (II)

- Use for lookup tables — e.g. find user information by ID
- Cache results — e.g. DNS queries
- Can be faster than map/reduce on SMP systems

Concurrent Hash Tables (III): Word count

```
typedef jss::concurrent_map<
  std::string,std::atomic<unsigned>> map_type;

void count_words(
  map_type& counts, word_list const& words) {
    std::string word;
    while(words.get_next(word)) {
      std::pair<map_type::iterator,bool>
        value=counts.insert(s,1);
      if(!value.second)
        ++(value.first->second);
    }
}
```

Other Concurrent Data Structures

- Skip lists: Java provides `ConcurrentSkipListMap` and `ConcurrentSkipListSet`
- TBB has `concurrent_vector`
- More complex data structures end up with a mutex lock

Choosing your patterns

Choosing your patterns (I)

What style of problem is it?

- event-driven
- recursive
- embarassingly parallel

Choosing your patterns (II)

What level of the application are we at?

- top level
- background processing
- inner loop

Choosing your patterns (III)

How can we best split the tasks and data?

- few/many tasks
- small/large amounts of data
- simple/complex interactions

Choosing your patterns (IV)

What scale are we at?

- Single multi-core processor
- Multiple multi-core processors
- Local cluster
- Globally distributed

Further Reading

- Patterns for Parallel Programming, Timothy G. Mattson, Bervely A. Sanders, Berna L. Massingill, Addison Wesley.
- Programming Scala, Dean Wampler, Alex Payne, O'Reilly Media (Available online)
- The GPars Project — Reference Documentation (Available online)
- Concurrent Programming in Erlang, Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams, Prentice Hall

just::thread provides a complete implementation of the C++0x thread library for MSVC 2005, 2008 and 2010, and g++ 4.3, 4.4 and 4.5 for Ubuntu/Debian/Fedora. MacOSX support coming soon.

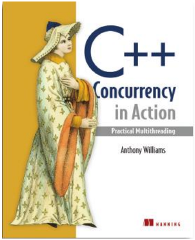
For a 50% discount go to:

http://www.stdthread.co.uk/accu2011

Just::Thread **Pro** also coming soon, with support for many of the high level facilities shown in this presentation. Find out more at:

http://www.stdthread.co.uk/pro

My book



C++ Concurrency in Action: Practical Multithreading with the new C++ Standard, currently available under the Manning Early Access Program, and due to be printed this summer.

http://www.stdthread.co.uk/book/

Photo credits

The images used in this presentation are courtesy of:

Daniela Hartmann	Florian Seroussi
Mary Harrsch	nicolas genin
Eric Gilliland	Barbara Doduk
Danie Ware	Pascal
Paul Hughes	ClayOgre
Travis	subblue
Ordnance Survey OpenData	