Introducing Parallel Pixie Dust:
Advanced Library-Based Support for Parallelism in C++
BCS-DSG April 2011

Jason M¢Guiness, Colin Egan

ppd-bcs-dsg-11@hussar.demon.co.uk
c.egan@herts.ac.uk
http://libjmmcg.sf.net/

April 7, 2011
Sequence of Presentation.

- An introduction: why I am here.
- Why do we need multiple threads?
  - Multiple processors, multiple cores....
- How do we manage all these threads:
  - Libraries, languages and compilers.
- My problem statement, an experiment if you will...
- An outline of Parallel Pixie Dust (PPD).
- Important theorems.
- Examples, and their motivation.
- Conclusion.
- Future directions.
Introduction.

Why yet another thread-library presentation?

▶ Because we still find it hard to write multi-threaded programs correctly.
  ▶ According to programming folklore.

▶ We haven’t successfully replaced the von Neumann architecture:
  ▶ Stored program architectures are still prevalent.

▶ The memory wall still affects us:
  ▶ The CPU-instruction retirement rate, i.e. rate at which programs require and generate data, exceeds the memory bandwidth - a by product of Moore’s Law.
    ▶ Modern architectures add extra cores to CPUs, in this instance, extra memory buses which feed into those cores.
How many Threads?

The future appears to be laid out: inexorably more execution units.

- For example quad-core processors are becoming more popular.
  - 4-way blade frames with quad cores provide 16 execution units in close proximity.
  - picoChip (from Bath, U.K.) has hundreds of execution units per chip, with up to 16 in a network.
  - Future supercomputers will have many millions of execution units, e.g. IBM BlueGene/C Cyclops64.
A Quick Review of Some Threading Models:

- Restrict ourselves to library-based techniques.
- Raw library calls.
- The “thread as an active class”.
- Co-routines.
- The “thread pool” containing those objects.
  - Producer-consumer models are a sub-set of this model.
- The issue: classes used to implement business logic also implement the operations to be run on the threads. This often means that the locking is intimately entangled with those objects, and possibly even the threading logic.
  - This makes it much harder to debug these applications. The question of how to implement multi-threaded debuggers correctly is still an open question.
What Is Parallel Pixie Dust (PPD)?

I decided to set myself a challenge, an experiment if you will, which may be summarized in this problem statement:

▶ Is it possible to create a cross-platform thread library that can guarantee a schedule that is deadlock & race-condition free, is efficient, and also assists in debugging the business logic, in a general-purpose, imperative language that has little or no native threading support? Moreover the library should offer considerable flexibility, e.g. reflect the various types of thread-related costs and techniques.
Part 1: Design Features of PPD.

The main design features of PPD are:

▶ It targets *general purpose threading* using a data-flow model of parallelism:
  
  ▶ This type of scheduling may be viewed as dynamic scheduling (run-time) as opposed to static scheduling (potentially compile-time), where the operations are statically assigned to execution pipelines, with relatively fixed timing constraints.

▶ DSEL implemented as futures and thread pools (of many different types using traits).
  
  ▶ Can be used to implement a tree-like thread schedule.
  ▶ “thread-as-an-active-class” exists.
Part 2: Design Features of PPD.

Extensions to the basic DSEL have been added:

▶ Certain binary functors: each operand is executed in parallel.
▶ Adaptors for the STL collections to assist with thread-safety.
  ▶ Combined with thread pools allows replacement of the STL algorithms.
▶ Implemented GSS(k), or baker’s scheduling:
  ▶ May reduce synchronisation costs on thread-limited systems or pools in which the synchronisation costs are high.
▶ Amongst other influences, PPD was born out of discussions with Richard Harris and motivated by Kevlin Henney’s presentation to ACCU’04 regarding threads.
More Details Regarding the Futures.

The use of futures, termed *execution contexts*, within PPD is crucial:

▶ This hides the data-related locks from the user, as the future wraps the retrieval of the data with a hidden lock, the resultant future-like object behaving like a proxy.
▶ This is an implementation of the *split-phase constraint*.
▶ They are only stack-based, cannot be heap-allocated and only *const* references may be taken.
  ▶ This guarantees that there can be no aliasing of these objects. This provides extremely useful guarantees that will be used later.
  ▶ They can only be created from the returned object when work is transferred into the pool.
Part 1: The Thread Pools.

Primary method of controlling threads is thread pools, available in many different types:

- Work-distribution models: e.g. master-slave, work-stealing.
  - Size models: e.g. fixed size, unlimited.....
  - Threading implementations: e.g. sequential, PThreads, Win32:
    - The sequential trait means all threading within the library can be removed, but maintaining the interface. One simple recompilation and all the bugs left are yours, i.e. your business model code.
    - A separation of design constraints. Also side-steps the debugger issue!
Part 2: The Thread Pools.

- Contains a special queue that contains the work in strict FIFO order or user-defined priority order.
- Threading models: e.g. the cost of performing threading, i.e. heavyweight like PThreads or Win32.
- Further assist in managing OS threads and exceptions across the thread boundaries via futures.
- i.e. a PRAM programming model.
Exceptions.

Passing exceptions across the thread boundary is a thorny issue:

- Futures are used to receive any exceptions that may be thrown by the mutation when it is executed within the thread pool.
- If the data passed back from a thread-pool is not retrieved then the future may throw that exception from its destructor!
  - The design idea expressed is that the exception is important, and if something might throw, then the execution context must be examined to verify that the mutation executed in the pool succeeded.
Ugly Details.

- All the traits make the objects look quite complex.
  - typedefs are used so that the thread pools and execution contexts are aliased to more convenient names.

- It requires undefined behaviour or broken compilers, for example:
  - Exceptions and multiple threads are unspecified in the current standard. Fortunately most compilers “do the right thing” and implement an exception stack per thread, but this is not standardized.
  - Currently, most optimizers in compilers are broken with regard to code hoisting. This affects the use of atomic counters and potentially intrinsic locks having code that they are guarding hoisted around them.
    - volatile does not solve this, because volatile has been too poorly specified to rely upon. It does not specify that operations on a set of objects must come after an operation on an unrelated object.
An Examination of SPEC2006 for STL Usage.

SPEC2006 was examined for usage of STL algorithms used within it to guide implementation of algorithms within PPD.
The STL-style Parallel Algorithms.

That analysis, amongst other reasons lead me to implement:

1. `for_each()`
2. `find_if()` and `find()`
3. `count_if()` and `count()`
4. `transform()` (both overloads)
5. `copy()`
6. `accumulate()` (both overloads)
Technical Details, Theorems.

Due to the restricted properties of the execution contexts and the thread pools a few important results arise:

1. The thread schedule created is only an acyclic, directed graph. In fact it is a tree.

2. From this property I have proved that the schedule PPD generates is *deadlock and race-condition free*.

3. Moreover in implementing the STL-style algorithms those implementations are efficient, i.e. there are provable bounds on both the execution time and minimum number of processors required to achieve that time.
The Key Theorems: Deadlock & Race-Condition Free.

1. Deadlock Free:

Theorem

That if the user refrains from using any other threading-related items or atomic objects other than those provided by PPD, for example those in 19 and 21, then they can be guaranteed to have a schedule free of deadlocks.

2. Race-condition Free:

Theorem

That if the user refrains from using any other threading-related items or atomic objects other than those provided by PPD, for example those in 19 and 21, and that the work they wish to mutate may not be aliased by another object, then the user can be guaranteed to have a schedule free of race conditions.
The Key Theorems: Efficient Schedule with Futures.

Theorem

In summary, if the user only uses the threading-related items provided by the execution contexts, then the schedule of work transferred to PPD will be executed in an algorithmic order that is $O\left(\frac{n}{p} + \log(p)\right)$, where $n$ is the number of tasks and $p$ the size of the thread pool. In those cases PPD will add at most a constant order to the execution time of the schedule.

- Note that the user might implement code that orders the transfers of work in a sub-optimal manner, PPD has very limited abilities to automatically improve such schedules.
- This theorem states that once the work has been transferred, PPD will not add any further sub-efficient algorithmic delays, but might add some constant-order delay.
Basic Example using execution_contexts.

Listing 1: General-Purpose use of a Thread Pool and Future.

```c
struct res_t {
    int i;
};
struct work_type {
    void process(res_t &)
};
pool_type pool(2);
execution_context context(pool<<joinable()<<time_critical()<<work_type());
pool.erase(context);
context->i;
```

- To assist in making the code appear to be readable, a number of typedefs have been omitted.
- The `execution_context` is created from adding the wrapped work to the pool.
  - `process(res_t &)` is the only invasive artifact of the library for this use-case.
  - Alternative member functions may be specified, if a more complex form is used.
- Compile-time check: the type of the transferred work is the same as that declared in the `execution_context`. 
The typedefs used.

Listing 2: typedefs in the execution context example.

```
typedef ppd::thread_pool<
pool_traits::worker_threads_get_work, pool_traits::fixed_size,
pool_adaptor<
generic_traits::joinable, platform_api, heavyweight_threading,
pool_traits::normal_fifo, std::less, 1
>
> pool_type;
typedef pool_type::create1<work_type>::creator_t;
typedef creator_t::execution_context::execution_context;
typedef creator_t::joinable::joinable;
typedef pool_type::priority<api_params_type::time_critical>::time_critical;
```

- Expresses the flexibility & optimizations available:
  - pool_traits::prioritised_queue: specifies work should be ordered by the trait std::less.
    - Could be used to order work by an estimate of time taken to complete - very interesting, but not implemented.
  - GSS(k) scheduling implemented: number specifies batch size.
  - execution_context checks the argument-type of process(...) at compile-time,
    - also if const, i.e. pure, this could be used to implement memoizing - also very interesting, but not implemented.
Example using `for_each()`.

Listing 3: For Each with a Thread Pool and Future.

```cpp
typedef std::safe_colln<
    std::vector<int>, std::lock_traits::recursive_critical_section_lock_type
> vtr_colln_t;

typedef pool_type::void_exec_ctxt execution_context;

struct accumulate {
    static int last; // for_each needs some kind of side-effect...
    void operator()(int t) { last+=t; }
};

vtr_colln_t v;
v.push_back(1); v.push_back(2);
execution_context context(pool<<joinable()<<(pool.for_each(v, accumulate())));

// The `safe_colln` adaptor provides the lock for the collection,
// in this case a simple EREW lock,
// only locks the collection, not the elements in it.

// The `for_each()` returns an `execution_context`:
// The input collection has a read-lock taken on it.
// Released when all of the asynchronous applications of the
// functor have completed, when the read-lock has been dropped.
// It makes no sense to return a random copy of the input functor.
```
Example using transform().

Listing 4: Transform with a Thread Pool and Future.

typedef ppd::safe_colln<
  vector<long>, lock::rw<lock_traits>::decaying_write,
  no_signalling, lock::rw<lock_traits>::read
> vtr_out_colln_t;

vtr_colln_t v; vtr_out_colln_t v_out;

v.push_back(1); v.push_back(2);

pool_type::void_exec_ctx context(
  pool<<joinable()<<pool::transform(  
      v, v_out, std::negate<vtr_colln_t::value_type>()
      )
  ) ;
  *context;

► vtr_out_colln_t provides a CREW lock on v_out: it is re-sized, then the write-lock atomically decays to a read-lock.
  ► Note how the collection is locked, but not the data elements themselves.

► The transform() returns an execution_context:
  ► Released when all of the asynchronous applications of the unary operation have completed & the read-locks have been dropped.
  ► Similarly, it makes no sense to return a random iterator.
Example using `accumulate()`.

Listing 5: Accumulate with a Thread Pool and Future.

```cpp
typedef pool_type::accumulate_t<
  vtr_colln_t
>::execution_context execution_context;

vtr_colln_t v;
v.push_back(1); v.push_back(2);
execution_context context(
    pool.accumulate(v, 1, std::plus<vtr_colln_t::value_type>())
);
*context;
```

- The `accumulate()` returns an `execution_context`:
  - Released when all of the asynchronous applications of the binary operation have completed & the read-lock is dropped.

- Note the use of the `accumulate_t` type: it contains a specialized counter that atomically accrues the result using suitable locking according to the API and type.
  - This is effectively a map-reduce operation.
    - `find()`, `find_if()`, `count()` and `count_if()` are also implemented, but omitted, for brevity.
Example using `logical_or()`.

Listing 6: Logical Or with a Thread Pool and a Future.

```cpp
struct bool_work_type {
    void process(bool &) const {}
};
pool_type pool(2);
pool_type::bool_exec_cxt context(
    pool.logical_or(bool_work_type(), bool_work_type())
);
*context;
```

- The `logical_or()` returns an `execution_context`:
  - Each argument is asynchronously computed, independent of each other:
    - the compiler attempts to verify that they are pure, with no side-effects (they could be memoized),
    - no short-circuiting, so breaks the C++ Standard.
  - Once the results are available, the `execution_context` is released.
- `logical_and()` has been equivalently implemented.
Example using `boost::bind`.

Listing 7: Boost Bind with a Thread Pool and a Future.

```cpp
struct res_t {
    int i;
};
struct work_type {
    res_t exec() {return res_t();}
};
pool_type pool(2);
execution_context context(
    pool<<joinable()<<boost::bind(&work_type::exec, work_type())
); context->i;
```

- Boost is a very important C++ library: transferring `boost::bind()` is supported, returning `execution_context`:
  - No unbound arguments nor placeholders in the call to `boost::bind()`.
  - Result-type of the function call should be copy-constructible.
    - Contrast with the reference argument result-type, earlier.
  - The cost: stack-space for the function-pointer & bound arguments.
  - Completion of mutation releases the `execution_context`. 
Some Thoughts Arising from the Examples.

- The read-locks on the collections assist the user in avoiding operations that might invalidate iterators on those collections.
  - Internally PPD creates tasks that recursively divide the collections into $p$ equal-length sub-sequences, where $p$ is the number of threads in the pool, at most $O(\log \min(p, n))$.
- The operations on the elements should be thread-safe & must not affect the validity of iterators on those collections.
- The user may call many STL-style algorithms in succession, each one placing work into the thread pool to be operated upon, and later recall the results or note the completion of those operations via the returned `execution_contexts`.
Sequential Operation.

Recall the `thread_pool` typedef that contained the API and threading model:

**Listing 8: Definitions of the typedef for the thread pool.**

```cpp
typedef ppd::thread_pool<
    pool_traits::worker_threads_get_work, pool_traits::fixed_size,
    generic_traits::joinable, platform_api, heavyweight_threading,
    pool_traits::normal_fifo, std::less, 1
> pool_type;
```

To make the `thread_pool` and therefore all operations on it sequential, and remove all threading operations, one merely needs to replace the trait `heavyweight_threading` with the trait `sequential`.

- It is that simple, no other changes need be made to the client code, so all threading-related bugs have been removed.
- This makes debugging significantly easier by separating the business logic from the threading algorithms.
Other Features.

Other parallel algorithms implemented:

▶ unary_fun() and binary_fun() which would be used to execute the operands in parallel, in a similar manner to the similarly-named STL binary functions.

▶ Examples omitted for brevity.

Able to provide estimates of:

▶ The optimal number of threads required to complete the current work-load.

▶ The minimum time to complete the current work-load given the current number of threads.

This could provide:

▶ An indication of soft-real-time performance.

▶ Furthermore this could be used to implement dynamic tuning of the thread pools.
Conclusions.

Recollecting my original problem statement, what has been achieved?

- A library that assists with creating deadlock and race-condition free schedules.
- Given the typedefs, a procedural way to automatically convert STL algorithms into efficient threaded ones.
- The ability to add to the business logic parallel algorithms that are sufficiently separated to allow relatively simple debugging and testing of the business logic independently of the parallel algorithms.
- All this in a traditionally hard-to-thread imperative language, such as C++, indicating that it is possible to re-implement such an example library in many other programming languages.

These are, in my opinion, rather useful features of the library!