Introducing Parallel Pixie Dust

Jason McGuiness

http://libjmmcg.sourceforge.net/

20th April 2009

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 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.

Why do We Need Multiple Threads?

The memory wall is a major influencing factor:

- A definition of the memory wall is the effect that one gets when the rate at which a processor may retire instructions exceeds the bandwidth of the memory sub-system.
- The use of Amdhal's law has been highly effective in concealing this issue: 90% of the time only 10% of the code (or data) is used.
- Another factor is locality of data both in space and time.
  - This means that cache hierarchies work very effectively.
  - But cache hierarchies do not overcome the latency of memory access, just the bandwidth.
  - As the gap between memory bandwidth and processor instruction retirement rates has widened, caches have become larger and the hierarchies deeper to cope.
  - So they have become more expensive.

How to Achieve Greater Instruction-Retirement Rates?

The increase in cost has meant alternatives have become viable:

- Such as the use of multiple processors & multiple cores.
  - Which may be logically viewed as multiple execution units retiring instructions.
- We now have *n*-cores retiring instructions, et voilà, we have increased the instruction retirement rate.
  - *If we can keep the n execution units running.*
- So now we have some sort of thread of execution, either architectural or OS-level, or a hybrid.
- These threads need to be managed. Therein the problems lie...

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 Cyclops.

The Problems with Threading: a Brief Review.

Let's leave aside the maxim: "The first rule of optimization is don't do it. The second rule is don't do it yet."

We'll presume we need to create threads. Creating threads is easy!!! (`_beginthreadex()`, `pthread_create()`, etc...)

- But synchronizing access to the data they mutate is hard:
  - Problem: race-conditions or deadlocks may arise. i.e. ensuring that the threaded code will run correctly for all "reasonable" data inputs is problematic, and effectively impossible.
  - Ensuring that the threaded code runs an efficient schedule is usually just ignored.
- Folklore not just in computer-science but in general programming, states that writing correct multi-threaded programs is a **black art**.

I do not disagree!

How do We Solve these Problems?

- There are various attempts to solve the threading problem:
  - Language level:
    - e.g. HPF and UPC or Threaded-C: highly successful but restricted domains.
  - Library level:
    - e.g. PThreads, Win32, Open-MP and MPI: very low-level, usually supporting very primitive threading-related operations.
    - Intel Thread Building Blocks and Concurrent Collections - very interesting, but not the subject of this talk.
    - Boost.Thread - influenced by the C++0x standardization effort, and Anthony Williams' excellent work, amongst others.
  - Auto-threading compilers:
    - e.g. MSVC and IBM VisualAge currently support a limited form of Open-MP relating to loop scheduling.
    - e.g. HPF, EARTH-C. These generate guaranteed correct and optimal schedules! But few outside the scientific community use them. EARTH-C even solves the hardest problems to parallelize, but for the IBM EARTH-MANNA architecture[1].

[1] X. Tang, "Compiling for Multithreaded Architectures." PhD thesis, University of Delaware, Newark, DE, Apr. 1999.

A Quick Review of Some Threading Models:

- Restrict ourselves to library-based techniques.
- Raw library calls.
- The "thread as an active class".
- The "thread pool" containing those objects.
- 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.

Problems Arising from these Models.

In the face of tens, let alone millions, of threads this approach is untenable:

- Testing the business logic is a nightmare.
- Testing the threading in the face of maintenance upgrades is a practical impossibility - the code-base becomes locked into libraries and compilers.
- Performance can become sub-linear due to excessively conservative locking!
- The business cost of maintaining these balls of mud becomes untenable.
- We face the failure of using multiple execution units as a technique to overcome the memory wall.

And there's no other commercial architecture out there to compete...... *YIKES!!!!*

What am I looking for?

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 optimal (mathematically), 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.

Further Constraints I set Myself.

- The imperative language used is C++: I know it, and it doesn't have the library that I want, yet.
- I focused on *general purpose threading*:
  - The situation when loop-bounds or conditions are not known at compile-time.
    - If these are statically computable, then conditions may be removed, and loops may be statically unrolled, rescheduled, etc as is well-known in the literature and well implemented in HPF, much more efficiently.
  - 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.
  - I specifically avoided investigating issues and threading relating to asynchronous I/O.

What is Parallel Pixie Dust (PPD)?

The main design features of PPD are:

- It is based upon futures: in fact it implements a software-simulation of data-flow.
- Although "thread-as-an-active-class" exists, the primary method of controlling threads is thread pools of many different types (think traits).
  - Futures and thread pools can be used to implement a tree-like thread schedule.
- Adaptors are provided for the STL collections to assist with providing thread-safety.
  - By using these adaptors and thread pools, the STL algorithms can be replaced with multi-threaded versions.
- Amongst other influences, PPD was born out of discussions with Richard Harris and motivated by Kevlin Henney's presentation regarding threads given circa 2004.

PPD: 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*.
- The futures are also key in passing exceptions across thread boundaries.
- The void return-type optimization has been implemented.
- They may be used to delete that work from the thread-pool.
- They are only stack-based, cannot be heap-allocated and only `const` references may be taken.
  - This guarantees that there can be no aliassing 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.
    - There are no (Boost) promises!

PPD: 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!
  - 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.
- i.e. a PRAM programming model.

PPD: 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.
  - This idea goes further:
    - All OS threads managed by the thread pool are destroyed when the pool terminates. The implication is that if there was no execution context to pass the exception into, then that thread will re-throw the exception from its destructor.
    - Ugly as hell, but otherwise what happens? Swallow the exception? That's ***uglier***.

PPD: The Concurrency Assistance.

I'm really trying hard to not say "thread safe" as it isn't!

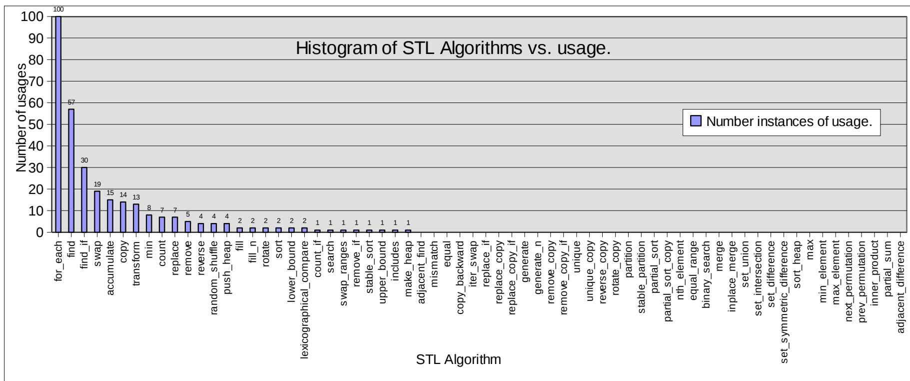
- The provision of adaptors for the STL collections assist with providing thread-safety.
  - The adaptors are used to provide the locks required by the thread pool library to maintain thread safety of the adapted collections.
  - In technical terms the type of lock supplied allows me to specify if the collection supports EREW or CREW operations.
- STL algorithms are neat ways to express lots of potential parallelism also the size of the collection is dynamic - known only at run-time.
  - Otherwise use HPFs auto-parallelizing loops!
  - These collections also assist in making PPD appear to be "drop in" to sequential code.
    - Naturally one would profile to determine which areas PPD might assist with!

PPD: 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.
  - 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 (the lock).

An Examination of Sample Code for STL Usage.

A large code-base that has been developed for over 6 years was examined for the distribution of STL algorithms used within it to guide implementation of algorithms within PPD. The target code-base consisting of over 180 projects was examined for usages of 66 of the STL algorithms. The distribution is:



PPD: The STL-style Parallel Algorithms.

That analysis, amongst other reasons lead me to implement:

1. `for_each`
2. `count_if` and `count` (which is trivially based on `count_if`)
3. `transform` (both overloads)
4. `copy` (trivially based upon `transform`)
5. `accumulate` (both overloads)

Which means I should cover over 45% of the cases in the code-base examined.

- Why no `find`/`replace`/`sort`/`merge`?
  - I haven't got around to implementing optimal versions of them yet:
    - `merge` is easy for implementing an optimal algorithm,
    - `find`/`replace`/`sort` are harder because there is a lot of literature regarding implementing these.
- So I'm missing an important 30% of cases.

PPD: 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***.[2]
3. Moreover in implementing the STL-style algorithms those implementations are optimal, in a mathematical sense, i.e. there are provable bounds on both the execution time and minimum number of processors required to achieve that time[3].

---

[2]Interested in the details? Ask me later, in the bar. A paper will be published in some journal on this, possibly CVu or Overload, until then you can download a draft from:
http://libjmmcg.sourceforge.net/accu09_draft_paper.pdf

[3]Much of this has come from "Efficient Parallel Algorithms", Alan Gibbons & Wojciech Rytter, for which I am grateful to Martin Waplington.

PPD: 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 26 and 28, then they can be guaranteed to have a schedule free of race conditions.

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 26 and 28, 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.

PPD: The Key Theorems: Optimal Schedule with Futures.

Theorem

If the user only uses the threading-related items provided by the execution contexts, then the schedule of the work transferred to PPD will be executed in no worse an algorithmic order than that of the schedule implemented by the programmer.

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

PPD: A Corollary: Optimal Schedule with STL-style algorithms.

Corollary

If the user uses the adaptors, for the STL collections, with CREW-PRAM semantics, and uses the STL-style algorithms, for example 28, then the scheduling algorithm implemented within PPD guarantees the following bounds will hold on that schedule:

1. *That the optimal time in which n-items of work on m-processors will be executed is:* $\left\lceil \frac{n}{m} \right\rceil - 1 + \log m$
2. *That the optimal number of processors with which to execute that work is:* $\left\lceil \frac{n}{\log n} \right\rceil$
3. *That using at least the optimal number of processors, the optimal time is:* $\lceil \log n \rceil$

PPD: A Result of the Optimality Theorems.

If PPD is used for scheduling multi-threaded code then:

Corollary

Using any of the features of PPD will add at worst a $\lceil \log n \rceil$ *time to the schedule, which is also the optimal time, then using any other additional threading schedule is likely to make that aggregate schedule worse.*

- i.e. PPD will not make your chosen parallel schedule sub-optimal.
- If you can express your parallel schedule in PPD, then those portions expressed using PPD will use an optimal schedule.
  - But that doesn't mean that just because you use PPD, your chosen parallel schedule *will be* optimal.

That's enough theory after lunch, I'll stop waving my hands! What about using PPD...

PPD: Basic Example using execution_contexts.

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

```
struct res_t {
        int i;
};

struct work_type {
        typedef res_t result_type;
        void process(result_type and) {}
};

pool_type pool(2);
async_work work(creator_t::it(work_type(1)));
execution_context context(pool<<joinable()<<time_critical()<<work);
pool.erase(context);
context->i;
```

- Note the use of a number of typedefs to assist in making the code appear to be readable.
- The execution_context is created from adding the wrapped work to the pool.
  - The work must be wrapped due to a lack of compiler support: the res_t, work_type and process() are some invasive artifacts of the library.
    - Not restricted to process(), alternative member functions may be specified.

PPD: The typedefs used.

Listing 2: Definitions of the typedefs in the execution context example.

```
typedef ppd::thread_pool<
        pool_traits::worker_threads_get_work, pool_traits::fixed_size
>::type<
        pool_adaptor<generic_traits::joinable>::type<
                platform_api, heavyweight_threading, pool_traits::normal
        >
> pool_type;

typedef pool_type::async_work async_work;
typedef pool_type::joinable joinable;
typedef pool_type::create<res_t> creator_t;
typedef pool_type::priority<
        pool_type::api_params_type::time_critical
> time_critical;
typedef pool_type::execution_context::type<res_t> execution_context;
```

- Partial specializations weren't available when I started to implement the library, so they look more complex.
- These express the flexibility and optimizations available in the library: this is the primary cause for the apparent complexity of the `typedef` for the `thread_pool`.

PPD: Example using `for_each()`.

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

```
typedef ppd::safe_colln<
        vector<int>, exception,
        lock_traits::critical_section_lock_type
> vtr_colln_t;
typedef pool_type::execution_context<void> execution_context;
struct accumulate {
        static int last;
        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());
*context;
```

- The `safe_colln` adaptor provides the lock for the collection, in this case a simple EREW lock.
- 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.

PPD: Example using transform().

Listing 4: Transform with a Thread Pool and Future.

```
typedef ppd::safe_colln<
        vector<int>, exception,
        ppd::lock::rw<lock_traits>::decaying_write,
        ppd::lock::rw<lock_traits>::read
> vtr1_colln_t;

vtr_colln_t v;
v.push_back(1);
v.push_back(2);
vtr1_colln_t v_out;
execution_context context(
        pool<<joinable()<<pool.transform(
                v, v_out, std::negate<vtr_colln_t::value_type>()
        )
);
*context;
```

- In this case `vtr1_colln_t` provides a CREW lock on the collection. `v_out` is re-sized, then the write-lock atomically decays to a read-lock.
- 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.

PPD: Example using `accumulate()`.

Listing 5: Accumulate with a Thread Pool and Future.

```
typedef pool_type::count_t<
        vtr_colln_t, long
>::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<typename vtr_colln_t::value_type>())
);
context->get();
```

- The `accumulate()` returns an `execution_context`:
  - Note the use of the `count_t` type: it contains a specialized counter that atomically accrues the value using suitable locking according to the API and type.
    - PPD also implements `count()` and `count_if()`, but I omitted examples, for brevity.
  - Released when all of the asynchronous applications of the binary operation have completed & the read-lock is dropped.

PPD: Some Thoughts Arising from the Examples.

- That the read-locks on the collections assist the user in avoiding operations on the collections that might invalidate iterators on those collections, when they should use EREW locks.
- That the operations applied to the elements of the collections by the algorithms are themselves thread-safe in some manner and do 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`.
  - This allows the user to fill the thread pool with work and maintain high throughput.
  - But this implies that the work that is done on each iterator is more costly than the synchronization and thread-management operations that are incurred. This is unlikely for such simple algorithms as `count()` and `copy()`.

PPD: Sequential Operation.

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

Listing 6: Definitions of the typedef for the thread pool.

```
typedef ppd::thread_pool<
        pool_traits::worker_threads_get_work, pool_traits::fixed_size
>::type<
        pool_adaptor<generic_traits::joinable>::type<
                platform_api, heavyweight_threading, pool_traits::normal
        >
> 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.

PPD: Other Features.

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.

PPD: Limitations.

1. The test cases are trivial:
   - They have some coverage, but needs more sophisticated test cases - library may not be bug-free.
2. The implementation of all of the various types of thread pool are not complete.
3. The missing STL algorithms:
   - The lack of `find()`, `merge()`, `sort()`, `merge()` etc., algorithms means that PPD misses covering some 30% of instances of usage of these algorithms in the example code-base.
4. Linking to a good-quality multi-threaded memory manager would be useful, such as Hoard, but this is optional.

PPD: Cross-Platform Issues.

Not actually cross platform!

- Different qualities of C++ compilers (G++ vs MSVC++) conspire to defeat this ability (lack of two-phase look up, compiler bugs, repeated re-compilation against alternate platforms, etc, etc).
  - Always a moving target, upgrading sometimes means regressions in the compilers.
  - This has been a major stumbling block, requiring almost continual retesting.
    - I should not have attempted to design the library with a large code-base of cross-platform code, and a thin layer to normalise the APIs. I should have reimplemented the library for each API.
    - This is a quality of implementation issue relating to the compilers, not an issue relating to the C++ language.

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 automatically to convert STL algorithms into optimal 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!

Future Directions.

Given the issues and limitations of PPD there are further areas that it would be useful to investigate:

- Most of the library is concerned with CREW-PRAM, but what about EREW-PRAM constraints?
- GSS(k), or baker's scheduling, would be useful to reduce the synchronisation costs on thread-limited systems or pools in which the synchronisation costs are high.
- A performance analysis of using PPD vs. other threading models using a standard benchmark suite[4].
- Some investigation of how PPD might support asynchronous I/O threading operations.
- How does or even should PPD interact with:
  - The C++ 0x standardisation efforts regarding threading.
  - The excellent work in Boost.Threads?

[4] For example SPEC2006: http://www.specbench.org/cpu2006/Docs/