Implementing Batcher’s Bitonic Sort in C++
An Investigation into using Library-Based, Data-Parallelism Support in C++.
October 2011

Jason McGuiness, Colin Egan & Russel Winder

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

October 5, 2011
Sequence of Presentation.

- An introduction: why I am here.
- How do we manage all these threads:
  - libraries, languages and compilers.
- Very brief introduction of parallel sorting algorithms.
- An outline of Parallel Pixie Dust (PPD).
- Outline of implementation using PPD.
- Conclusion.
- Future directions.
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.
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.
Parallel Sorting...

Parallel sorting has a history:

- The three Hungarians (Ajtai, Komlos & Szemeredi, AKS) presented an algorithm that takes $O(\log(n))$ time with $n$ processors in 1983, but has impractically large complexity constants.

- Cole’s parallel merge sort was presented in 1986 and takes $O(\log(n))$ time with $n$ processors, but has much smaller complexity constants.

- Batcher’s bitonic sort was presented in 1968 and takes $O\left(\log^2(n)\right)$ time, but according to Knuth when comparing it to AKS: "Batcher’s method is much better, unless $n$ exceeds the total memory capacity of all computers on earth!"

- Research has indicated that Batcher’s algorithm has better complexity constants than Cole’s, so we shall focus on Batcher’s algorithm.
An Examination of SPEC2006 for STL Usage.

SPEC2006 was examined for usage of STL algorithms used within it, sorting is important to one of the benchmarks, so a potential target for parallelism.
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.
  ▶ Gives rise to important properties: efficient, dead-lock & race-condition free schedules.
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.
The STL-style Parallel Algorithms.

That analysis, amongst other reasons lead me to implement in PPD:

1. `for_each()`
2. `find_if()` & `find()`
3. `count_if()` & `count()`
4. `transform()` (both overloads)
5. `copy()`
6. `accumulate()` (both overloads)
7. `fill_n()` & `fill()`
8. `reverse()`
9. `max_element()` & `min_element()` (both overloads)
10. `merge()` & `sort()` (both overloads)
Basic Example of `sort()`.

Listing 1: Parallel `sort()` with a Thread Pool and Future.

```cpp
typedef ppd::safe_colln<vector<long>, lock::recursive_critical_section_lock_type> vtr_colln_t;

vtr_colln_t v;
v.push_back(2); v.push_back(1);
execution_context context(
    pool<<joinable()<<pool.sort<ascending>>(v)
);
*context;
```

▶ To assist in making the code appear to be readable, a number of typedefs have been omitted.
▶ 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 `sort()` returns an `execution_context`:
  ▶ The input collection has a read-lock taken on it.
  ▶ Released when the asynchronous sort has completed, when the read-lock has been dropped.
  ▶ It makes no sense to return a random iterator.
The typedefs used.

Listing 2: typedefs in the sort() 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::void_exec_cxt execution_context;
typedef execution_context::joinable joinable;

▷ 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.
Implementing Batcher’s Bitonic Sort with PPD.

Listing 3: Schematic of the sort() method.

```cpp
const in_iterator middle( boost::next( begin, half_size_of_portion ));
execution_context sort_lhs_ascending(
    pool<<joinable()<<(sort<ascending>)(begin, middle, compare)
);
execution_context sort_rhs_descending(
    pool<<joinable()<<(sort<descending>)(middle, end, compare)
);
*sort_lhs_ascending;
*sort_rhs_descending;
execution_context bitonic_merge_all(
    pool<<joinable()<<(merge<direction>)(begin, end, compare)
);
*bitonic_merge_all;

- As the target processor may be more sophisticated than a comparator, we could use a serial sort for a minimal sub-range, based upon threading costs of the target architecture, terminating the recursion early.
  - We’d avoid using quicksort, as it is $O(n^2)$ for nearly-sorted inputs, which a bitonic sequence is.
Implementing Batcher’s Bitonic Merge with PPD.

Listing 4: Schematic of the merge() method.

```cpp
cont const out_iterator middle(boost::next(begin, half_size_of_portion));
cont const out_iterator output(middle);
execution_context shuffle(
    pool<<joinable()<<(pool.swap_ranges(
        begin, middle, output, swap_pred<direction>(compare)
    )
    )
); shuffle;
execution_context merge_lhs(
    pool<<joinable()<<(merge<direction>(begin, middle, compare)
    );
execution_context merge_rhs(
    pool<<joinable()<<(merge<direction>(middle, end, compare)
    );
*merge_lhs;
*merge_rhs;
```

▶ Again the target processor may be more sophisticated than a simple comparator, we could use a serial sort in a similar manner.

▶ Although this might increase the complexity with an $O(n \log(n))$ factor.
Some Thoughts Arising from the Examples.

- Internally PPD creates sub-tasks that recursively divide the collections into $p$ equal-length sub-sequences, where $p$ is the number of threads in the pool, generating a tree of depth $O(\log(\min(p, n)))$.

- The comparison operations on the pairs of 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 5: 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.

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.

- A study of the scalability of the sort & merge operations would be interesting - this is to be done.
  - I believe the algorithmic complexity of the `merge()` and `sort()` operations is \( O \left( \log^2 \left( \frac{n \log(n)}{p} + \log(p) \right) \right) \).
- PPD assists with creating efficient, deadlock and race-condition free schedules.
- The business logic and parallel algorithms are sufficiently separated to allow relatively simple, independent testing and debugging of each.
- 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!