C++ Data-flow Parallelism sounds great! But how practical is it? Let's see how well it works with the SPEC2006 benchmark suite

(Yet Another!) Investigation into using Library-Based, Data-Parallelism Support in C++(03 & 11-ish).

Jason M^{c}Guiness & Colin Egan

accuconf-12@hussar.demon.co.uk
c.egan@herts.ac.uk
http://libjmmcg.sf.net/

30th April 2012

Sequence of Presentation.

- An introduction: why I am here.
- How do we manage all these threads:
  - libraries, languages and compilers.
- An outline of a data-flow library: Parallel Pixie Dust (PPD).
- Introduction to SPEC2006 followed by my methodology in applying data-flow to it.
- Leading on to an analysis of SPEC2006.
- How well did the coding go ... some simple examples & discussion around pitfalls.
  - If time & technology allow some real-live examples!
- Some results. Wow numbers! Yes really!
- 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.

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 Data-flow and why is it so good?

"Manchester Data-flow Machine", developed in Manchester in 80s[1].

- *Not* von Neumann architecture. No PC. Large number of execution units & special memory (technically CAMs).
  - Compiler-aided:
    - Compiler automatically identifies & tags dependencies which annotate the output binary.
    - Dependencies & related instructions loaded into CAM.
    - When all of the tags are satisfied for a sequence of instructions, that is *fired*, i.e. submitted to an execution unit.
  - No explicit locks.
    - No race conditions nor deadlocks.
    - Considerable effort put into the data-flow compiler to generate provably correct code.

Is The Way Forward™.

- Why don't we use it? Need to recompile source-code & buy "exotic" hardware.
  - Inertia, nih[2]-ism. Companies hate to do this.

[1] Stephen Furber is continuing the work there.

[2]

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.
- Adapters for the STL collections to assist with thread-safety.
  - Combined with thread pools allows replacement of the STL algorithms.
- Optimisations including void return-type & 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.

Analysis of SPEC2006, 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)
    - Based upon Batcher's bitonic sort, not Cole's as recommended by Knuth.
    - Work in progress: still getting the scalability of these right.

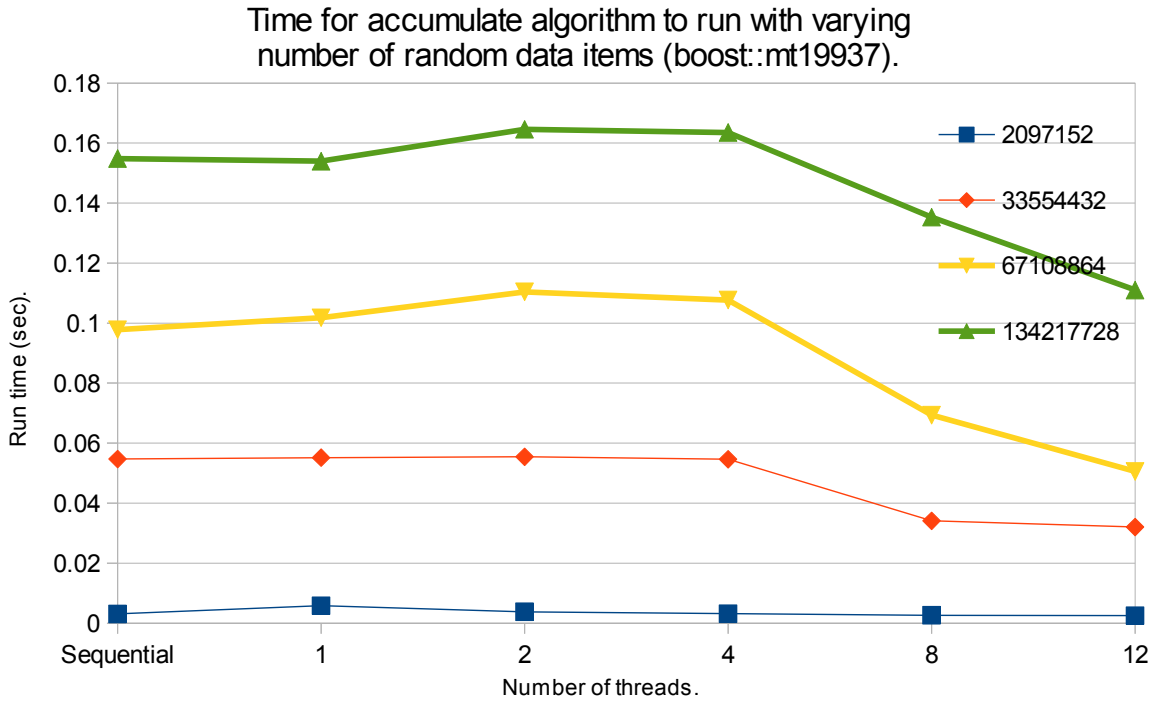
Example using accumulate().

Listing 1: Accumulate with a Thread Pool and Future.

```
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;
typedef ppd::safe_colln<
        vector<int>, lock_traits::critical_section_lock_type
> vtr_colln_t;
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>())
);
assert(*context==4);
```

- The `accumulate()` returns an `execution_context`:
  - Released when all the asynchronous applications of the binary operation complete & read-lock is dropped.
- Note `accumulate_t`: contains an atomic, specialized counter that accrues the result using suitable locking according to the API and counter-type.
  - This is effectively a map-reduce operation.

Scalability of `accumulate()`.



- Some scaling, but too small range of threads to really see.
- Smallest data-set 2,097,152 items, run-time 3ms → 2ms.
  - Scalability affected by the size of a quantum of a thread on a processor.
    - Effect ceases with over 67,108,864 data elements.
  - Need huge data-sets to make effective use of coarse-grain parallelism.
  - ~7% increase from 1 → 2 threads probably due to movement of dataset.
- One run was taken. Dual processor, 6-cores each, AMD Opteron 4168 2.6GHz, 16Gb RAM in 4 ranked DIMMs (2 per processor), NUMA mode m/b & kernel v3.2.1 on Gentoo GNU/Linux, gcc "v4.6.2 p1.4 pie-0.5.0" with *–std=c++0x* & head revision (`std::move` added).

Why SPEC2006[4]?

- Published since 1989, numerous updated versions in that time.
- "A useful tool for anyone interested in how hardware systems will perform under compute-intensive workloads based on real applications."
- Publications in ACM, IEEE & LNCS amongst many others.
- Studied in academia a great deal:
  - For example ~60% performance improvement on 4 cores and a further ~10% on 8 when examining loop-level parallelism[3].
  - Poor scalability, code was written to be *implicitly sequential*.
    - For better performance design for parallelism...
    - Is that a premature optimisation, ignoring Hoare's dictum!
- Choose something - not pulled it out of a hat; worse still something that makes me look artificially good!

[3] Packirisamy, V.; Zhai, A.; Hsu, W.-C.; Yew, P.-C. & Ngai, T.-F. (2009), Exploring speculative parallelism in SPEC2006., in 'ISPASS' , IEEE, , pp. 77-88 .

[4] `http://www.spec.org/cpu2006/press/release.html`

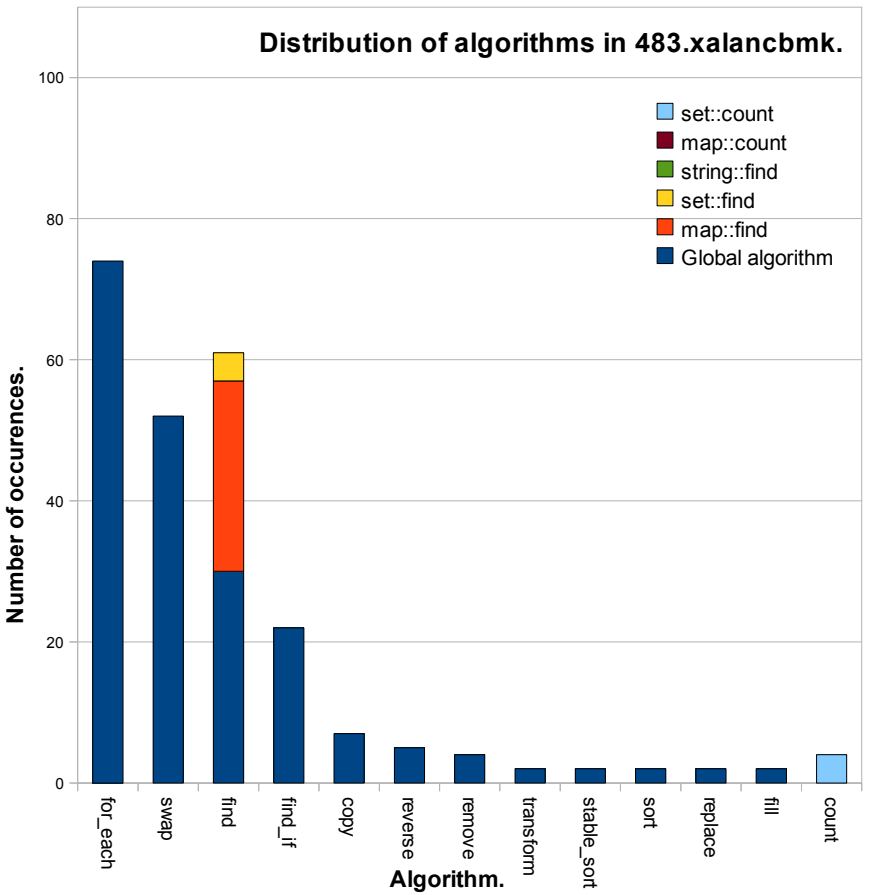
Methodology.

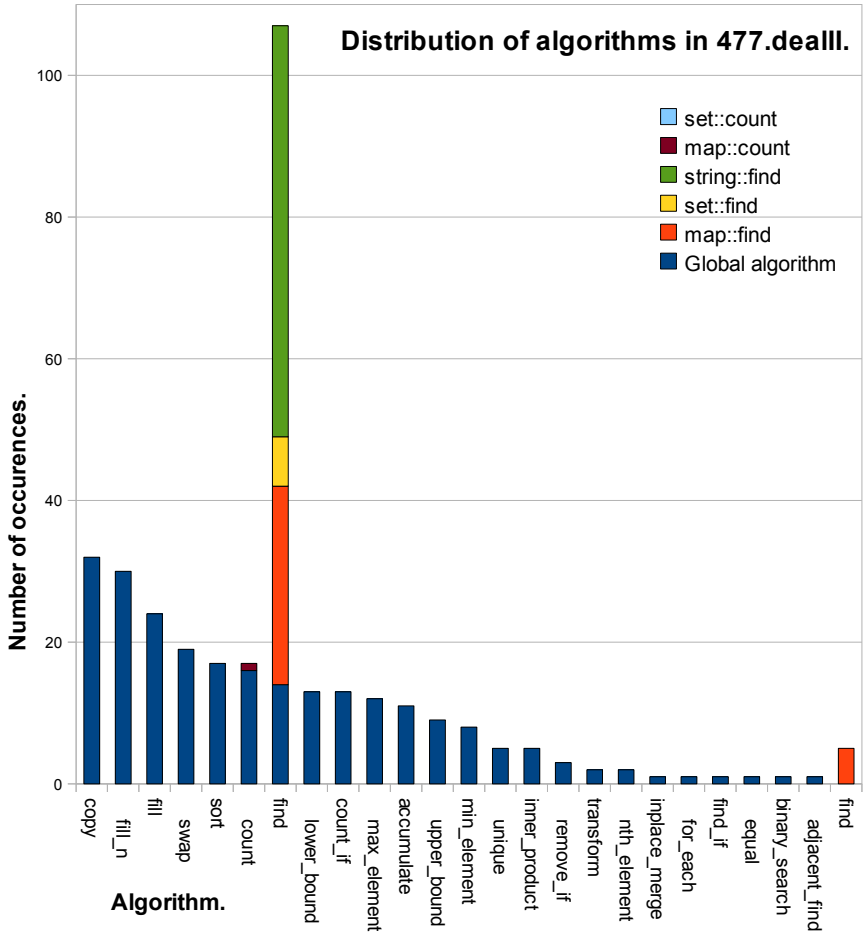
Naive approach...

- Examine by how much is the performance affected by a simple analysis of a C++ code-base and subsequent application of high-level data-parallel constructs to it.
  - Will give a flavour of the analysis done & a subjective report on how well the application seemed to go.
- Did not:
  - run a profiler to examine hot-spots in the code,
  - attempt to understand algorithms and program design,
  - do a full, detailed, critical analysis of the code,
  - do a full rewrite in a data-flow style.
  - No time, but of course one would really do these.

An Examination of SPEC2006 for STL Usage.

SPEC2006 was examined for usage of STL algorithms used.





Analysis of 483.xalancbmk

Classified as an integer benchmark using a version of the Xalan-C XSLT processor.

- Large, not a toy: over 500000 lines in over 2500 files.
- Single global `thread_pool`.
- Effort: at least 5 days.
  - Lot of effort for little reward.
- Small function bodies make it hard to easily increase parallelism by moving independent statements so they overlap.
- We cannot wantonly replace algorithms with parallel ones...

Parallelising 483.xalancbmk

9 of the over 200 identified uses of STL algorithms were actually easily parallelised:

- `for_each()`: none parallelisable:
  - 57 used for deallocation: needs parallel memory allocator!
  - Accounts for over half of all of the applications of `for_each()`.
  - Remainder unlikely to be replaceable: the functor must have side-effects, further locking required for correct execution.
- `swap()`: right type of algorithm is important; better implemented by swapping internal pointers, not the data elements themselves. None parallelisable.
- `find()` & `find_if()`: replaced 5 & 2 respectively.
- `copy()` & `reverse()`: none replaced.
- `transform()`: none replaced.
- `fill()` & `sort()`: replaced 1 each

Analysis of 477.dealII

Classified as a floating-point benchmark: "numerical solution of partial differential equations using the adaptive finite element method".

- Large: over 180000 lines in over 483 files.
- Actually contains a hybrid implementation of data-flow for parallelism, but I didn't test it.
- Again, single global `thread_pool`.
- Old version of boost included: got odd errors, so had to remove that!
  - Be sure to use compatible versions of libraries, beware that they can be hidden anywhere!
- Effort: at least 2 days.
  - Excludes "fiddling" time.

Parallelising 477.dealII

Of the over 230 identified uses of STL algorithms 33 were actually easily parallelised:

- `copy()`: replaced none: usage requires full treatment of arbitrary ranges.
- `fill()` & `fill_n()`: replaced none & 8 respectively.
- `swap()`: none, as per xalancbmk.
- `sort()`: replaced 9.
- `count()` & `count_if()`: replaced 3 & 5 respectively.
- `find()` & `find_if()`: replaced none: mainly member-functions.
- `min_element()` & `max_element()`: replaced 2 & 3 respectively.
- `accumulate()`: replaced 3.
- `transform()`: replaced none.

Reflect Upon Original Analysis of Benchmarks.

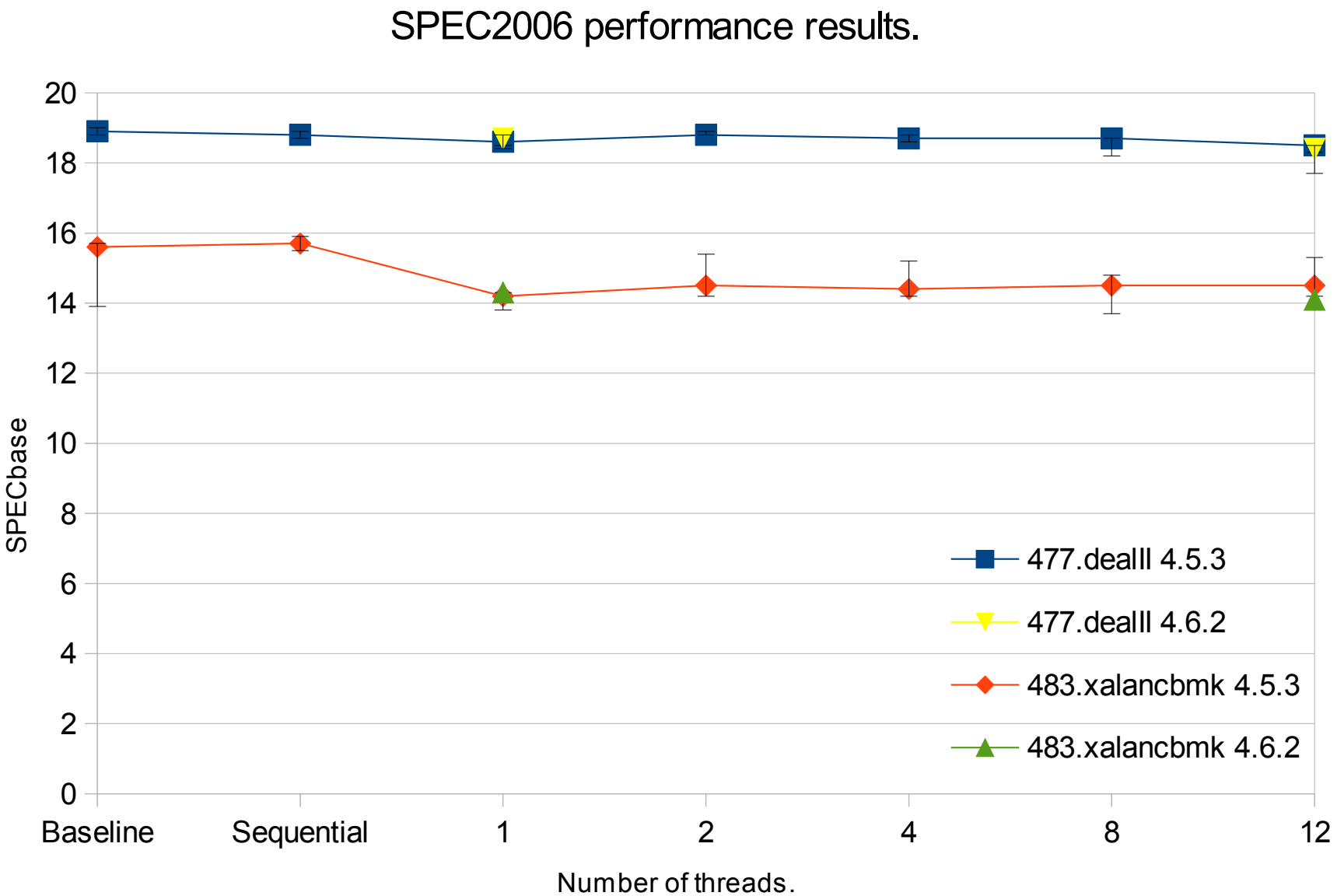
Very few of the initially identified algorithms could be replaced.

- Functional-decomposition nature of the code breaks the basic blocks into too small a scale to readily identify parallelism.
  - Larger function bodies & style of use make it easier to increase parallelism by overlapping independent statements.
- Uninvestigated potential side-effects confound the ready replacement of many algorithms with parallel versions.
- Use of aspect-style programming would have greatly eased replacement of types.
  - Ease conversion of collections into those that are adapted to parallel use, return types, etc.
- Complicated compilation options used to test robustness of build (gcc "v4.5.3-r2 p1.1 pie-0.4.7"):

```
-pipe -std=c++98 -mtune=amdfam10 -march=amdfam10 -O3
-fomit-frame-pointer -fpeel-loops -ftracer -funswitch-loops
-funit-at-a-time -ftree-vectorize -freg-struct-return -msseregparm
```

  - Would be very sensible to move to C++11.

Performance of Modified Benchmarks.



- "ref" dataset. The median of five runs was taken. Error bars: max & min performance.

Commentary upon the results.

xalancbmk:

- General lack of scaling. Not enough parallelism exposed, given it is not very scalable.
- Sequential performance is effectively the same as the baseline, so no harm to use it speculatively.
- ~5% performance reduction when using threads.
  - Shows that the threads are actually used, just not effectively.
  - Roughly constant, so independent of number of threads, likely to be overhead of the implementation of data-flow in PPD.

dealII:

- No cost to speculatively use parallel data-flow model.
  - I did check that the parallel code was called!
- Other code costs swamp costs of threading, unlike xalancbmk.
  - Don't sweat the O/S-level threading construct costs.
- Design of algorithm critical to scalability.
  - Up-front design cost higher as more care required.

Conclusions & Future Directions.

- Benchmarks were not written with parallelism in mind, so the parallelism revealed is poor.
  - When designing an algorithm that may become parallel, vital to target a programming model.
    - Ensure option of running sequentially: no cost if it is not used.
    - Different models scale differently on the same code-base.
    - So carefully choose a model that will reflect any inherent parallelism in the selected algorithm.
    - This is hard: look around on the web.
- Better statistics including collection sizes to assist in identifying opportunities for parallelisation.
  - Yes, profiling is vital, good profiling better!
    - Perfectly parallelised code runs at the speed of the sequential portion of it, which is asymptotically approached as the parallel portion is enhanced. (One of Amdhal's Laws?)
- *No golden bullet: post-facto bodging parallelism onto a large pre-existing code-base is a waste of time & money.*