Thread-safe and Thread-neutral Bags

A Presentation (20, 45 or 90 minute) for Intermediate to Advanced C++ Developers

Richard T. Saunders
Rincon Research Corporation
rts@rincon.com

Abstract

The bag is a fundamental container for use in threaded systems: many threads pull continually and simultaneously from a bag to get data on which to operate. The bag is conceptually simple with one key operation (`get`) but implementing the bag in such a way to be both thread-safe (the state of the bag is never inconsistent amidst multiple threads) and thread-neutral (multiple threads do not impede each other) is surprisingly difficult. This paper explores aspects of implementing the bag in a threaded environment: the number of threads accessing the bag simultaneously, the nature of the work, the type and number of processors on a machine, and the high-speed producer/consumer relationship in particular. It's clear that one bag does not fit all needs. This paper introduces two main abstractions for the bag, depending on the nature of the application: the drawer and the cupboard. These new abstractions are implemented using the the new C++11 multi-threading features to ensure thread-safe and thread-neutral bags; these implementations are vetted using both synthetic benchmarks and a real-world application.

1. Introduction

Using threads is difficult[7], but the rewards can be significant if used properly; application speedups relative to the number of processors can be achieved if care is taken. One canonical technique for using many threads is the *work crew*[3] (also known as *map-reduce*[5]) paradigm[1]. Consider the *work crew* of Figure 1. A large amount of "work" needs to be processed, where each piece of work is fairly independent. This work is divided evenly among multiple workers (or *mapped*); each worker computes its separate input in an independent thread. When all threads finish, they join and take the completed work of each thread and synthesize it (or *reduce*) into a final answer. This is the model promoted by OpenMP[4]; some of Intel Threading Building Blocks[8] use this model. [14] uses similar examples.

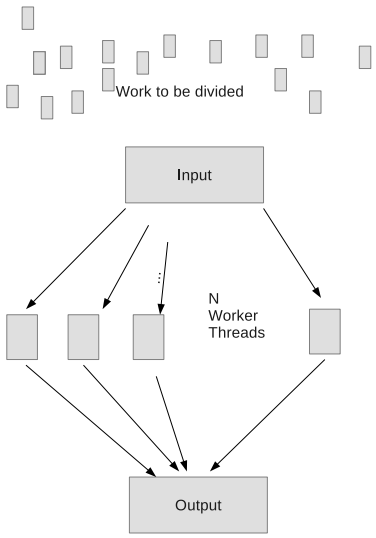

Figure 1. Work-Crew or Map-Reduce Paradigm

1.1 The Slowest Worker Problem

Applications initially start by dividing the work evenly among n worker threads and wait for all threads to finish. On an unloaded machine, this works fine as all threads typically finish at the same time, potentially giving a speedup of n for that section of code. Unfortunately, in many real-world situations, threads will finish at different times and the applications end up waiting for the slowest thread to finish. What was hopefully a speedup of n for this particular section of code with n threads would be significantly reduced; the application may be throttled by the scheduler and other work on the machine. This is the *slowest worker* problem, as the slowest worker reduces speedup.

1.2 The Bag

The *bag* is a very simple mechanism to help mitigate the effects of scheduler irregularities and the other applications running simultaneously on the machine. Rather than *apriori* divvying up the work evenly among the workers, each worker dynamically pulls from a bag whenever work is needed. Thus, if one worker gets scheduled twice as often as other workers, that worker will do twice as much work as the other workers, but all workers will finish at the same time—they should all find the bag empty and thus finish at approximately the same time. Thus, the slowest worker will not throttle the worker crew: all work will be done as fast as the machine can support.

[1] Although map-reduce is usually used across multiple processes or machines rather than multiple threads, the paradigm is still the same

C++ Now Conference 2013 May 13-17, 2013.

If the work done by each thread is excessively time-consuming, the slowest worker problem can still be an issue, even with a bag; a poorly scheduled thread can start on the last piece of work just as each all other threads finish, causing the application to wait for the one last thread (some of this effect could be alleviated by scheduling the more time-consuming work first—if there is a notion of approximately how long each piece of work would take—and the least time-consuming work at the end). The longer the time to work, the worse the slowdown. The purpose of the bag is to mitigate this problem of *apriori* division; work still needs to be fine-grained enough to be scheduled fairly regularly. The user still needs to divide the problem as fine as possible.

One limitation of the bag model described here is that it does *not* allow work to be placed in the bag once the workers have started; this is not an uncommon model: for example, this is what OpenMP embraces. In OpenMP, a program is a generally series of map-reduce points in a program. A more general model that would allow dynamic puts to a bag would more like a client-server relationship or producer/consumer relationship; more discussion of those can be found in [3][10][14]. Even though the bag model described here is more limited, it does still cover many real-world applications: the Cross-Ambiguity Function described in *Section 6: Real World Results* was the impetus for the particular bags described here.

One of the nice features about dividing up the work *apriori* among the workers is that there is no need for any of the workers to communicate and/or synchronize while they are working: each worker is independent because the work is known. It's a well-known that excessive synchronization can significantly reduce the speed-up of multiple threads[2][3][6][10][13]. By introducing the bag into the work crew, it's possible that this will be introducing synchronization overhead that significantly slows the application. By carefully implementing the bag using the techniques of [10] and [11] and the new features of C++11, a bag can be achieved that causes no collateral damage to the performance of multiple threads. Unfortunately, there are many kinds of work crews, and different techniques are needed for different types of applications.

2. Bag of Integers

The first idea of most bags is to implement a class which contains all the work that needs to be doled out. This seems to be the decision of the Microsoft `ConcurrentBag<T>` class. The interface is complex as the classes support multiple ways to add work, remove work, etc. A key simplification is that the bag only needs to dole out integers from 0 to $n-1$ where n is the number of pieces of work. The work can then be placed in a standard `vector` using standard `vector` operations that are familiar to STL programmers. Thus, the only interface the bag needs is the *get*:

```
// Return false if the bag is empty.  Otherwise,
// return true and pull some integer from the
// bag into result
bool get (uint32_t& result);
```

Using a bag of integers with a `vector` would look something like:

```
vector<string> work{ "work1", "work2", "work3"};
BAG_OF_INTS bag(0, work.length());
                        // Bag of {0,1,2}

// Main loop, get work from bag and operate
uint32_t work_index;
while (bag.get(work_index)) {
  string& single_piece_of_work=work[work_index];
  doSomething(single_piece_of_work);
}
```

Restricting the bag to contain only integers simplifies the implementation and the interface. Interface is important: making it relatively straight forward to use means it's more likely to be used. Implementation is important: using only integers for the bag takes advantage of some of the atomic integer operations of C++11 to help make the bag thread-neutral and thread-safe.

If desired, a templatized convenience class containing the bag of integers and the `vector` can easily be created to wrap the two ideas above.

3. The Drawer

The fundamental thread-safe bag is the `iDrawer`. All of the bags mentioned here starts with "i" to emphasize they only dole out integers. The basic abstraction is called a *drawer* where the actual C++ class is called `iDrawer`; occasionally the two notions will be mixed, but it should always be clear from context which is which. The drawer is built out of two integers: the current value and the upper bound. Once the current value is greater than the upper bound, then the drawer is empty. The upper bound is only set at construction time and it is only ever read: it is never set (thus can be cached without fear of changing). The current value is the only value that changes: whenever a *get* happens, it is atomically incremented. The increment uses either the C++11 atomic primitives (which are based on the atomic read-increment instructions of the machine) or GNU primitives.

```
class iDrawer {
   public:
      iDrawer (uint32_t starting_index,
               uint32_t length) :
        current_(starting_index),
        upperBound_(starting_index+length)
      { }

      bool get (uint32_t& thing_from_bag)
      {
        if (current_>=upperBound_) return false;
        uint32_t temp = current_.fetch_add(1)
        if (temp>=upperBound_) return false;
                    //double-checked lock pattern

        thing_from_bag = temp;
        return true;
      }

   protected:
      // Current value
      std::atomic<uint32_t> current_;

      // Upper bound
      // Read-only, no need for atomic access
      uint32_t upperBound_;
};
```

Note that when `current_` is initialized, it is initialized to what what would be the next from the bag. Thus, only when `current_` is atomically incremented would the current value be returned.

If the atomic increment is implemented using the atomic instructions of the machine, a get from the bag becomes very cheap. The *ticket lock* of the Linux kernel uses similar mechanisms to implement cheap locks.

3.1 Double-Check Lock Pattern

The check against the upper bound occurs twice: once before the atomic add and once after. Why? Much like the *Double-Check*

Lock Pattern used for Singletons (see Alexandrescu 6.9.1[1]), it's possible for other threads to sneak in (because multiple threads hit this code at about the same time) after the initial check—the state may need to be checked again for consistency. In particular: it's possible for some thread x to see current value less than the upper bound, have another bunch of threads sneak in and perform the atomic increment, then by the time thread x performs its atomic increment, the bounds have already been exceeded. So before the value obtained is returned, it is checked to see if its still within bounds. Note that because the fetch and increments are atomic, there will never be a invalid value in range. Also, all values in the bag will be seen as well (no repeats), as the atomic increment guarantees that.

The double-check is only needed when the drawer is nearly empty—a bunch of threads increment the current value too optimistically, but only a few get a valid number. The rest of the threads have overestimated, but the double check takes care of that. There is a potential bug lying dormant here: if the number of threads (workers) plus the actual upper bound would ever exceed the maximum value of an `uint32_t`, this method will not work (as the `uint32_t` would roll-over). In practice, this is not a real problem—the bag can be emptied and re-seeded before this error condition would happen.

3.2 Pre C++11 Compilers

Unfortunately, compilers that support all the C++11 features are still not quite ubiquitous, so the atomic types mentioned above may be not be available for all platforms. The GNU compiler (and Intel compiler, because they tend to support the GNU features) support a similar construct atomic instruction to the `fetch_add`. With two minor changes (below), a pre C++11 compiler can still use a drawer.

```
// Atomic fetch_add for GNU
uint32_t temp=__sync_fetch_and_add(current_,1);

// Change defintion of current from
// atomic<uint32_t>
volatile uint32_t current_;
```

These simple changes preserve the same speed of implementation of the C++11 atomics.

3.3 Thread-Safe vs. Thread-Neutral

At this point, the drawer is arguably *thread-safe*: the bag will dole out the integers correctly in the presence of multiple threads without ever having inconsistent state. But how fast is it? Multiple threads reaching into the bag may slow down other threads trying to reach into the bag at the same time. One of the goals of the bag implementations discussed here is to preserve *thread-neutrality*[11]. When one thread interferes with the running time of another thread, that thread is said to cause collateral damage: [11] defines thread-neutrality as the lack of collateral damage. In other words, if the bag won't impede the progress of another thread, the bag is thread-neutral.

To implement bags that are thread-neutral means showing that multiple threads accessing the bag don't slow down other threads reaching into the bag at the same time. The drawer is thread-safe and simple and fast. Making a thread-neutral bag, however, depends on the nature of the worker threads and the work done per thread.

There are two questions that are fundamental when addressing the thread-neutrality of the bag: How much work is done per *get* of the bag? How many workers are getting from the bag at the same time?

Question 1: How much work is done per *get*? For a synthetic benchmark, the work gets characterized in one of three ways:

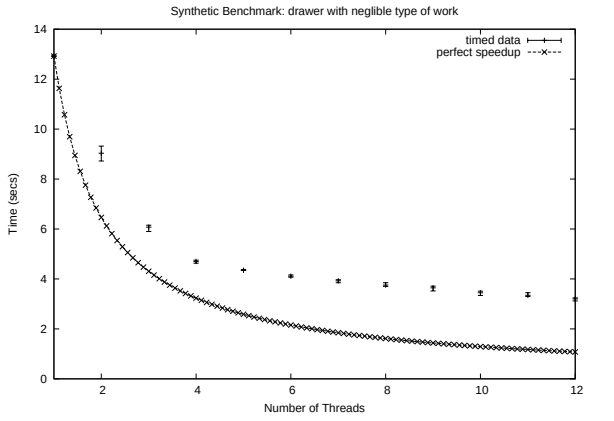

Figure 2. Demonstrating a drawer on a *negligible* work type

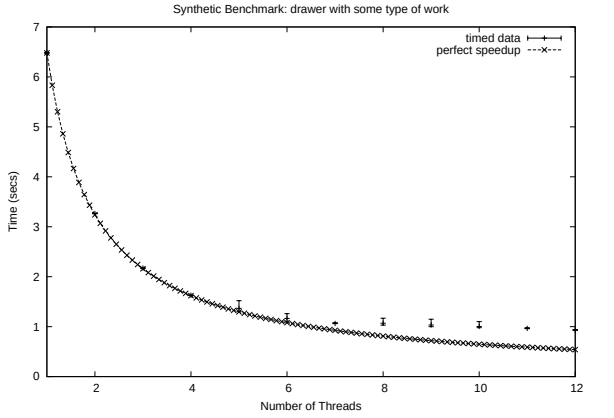

Figure 3. Demonstrating a drawer on a *some* work type

- Negligible Work: Each thread pulls data out of the bag as fast as it can. Each thread does little to no work with the item obtained—the bag is being strained to its limits to keep up. This model is reminiscent of the High-Speed Producer/Consumer model of [10].
- Some Work: Each thread pulls work quickly, but there is some work before it pulls again.
- Ample Work: A thread rarely does a *get*, as it is too busy doing real work to look in the bag very often.

The synthetic benchmark simulates work by adding and subtracting from an arbitrary sized integer in a loop. For *negligible* work, there is exactly one add. In *some* work, the add/subtract is performed one hundred times. In the *ample* work, the add/subtract is performed one million times.

Question 2: How many workers are there? The more workers there are, the more chance they can get in each others way.

3.4 Performance

For these experiments, a 6-CPU machine with hyperthreading (so it looks like 12 CPUS) was used. Hyperthreading has its own issues which will be addressed later[10]. A standalone program simulates the different kinds of workloads (negligible, some, ample) with different numbers of threads. In a perfect world, the amount of time would decrease directly proportional to the number of workers: that's what the *perfect speedup* line represents (thus 12 seconds

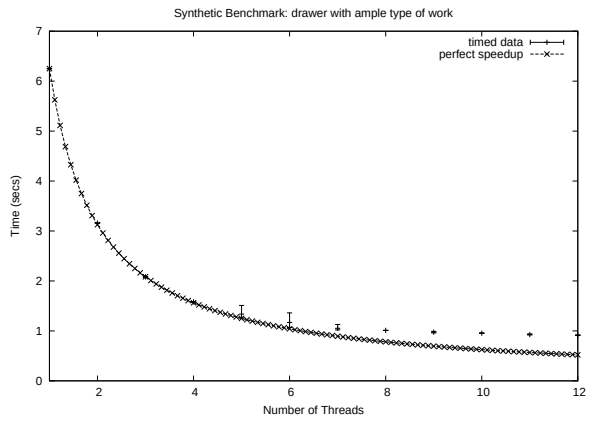

Figure 4. Demonstrating a drawer on a *ample* work type

with 1 worker would be 6 seconds with 2 workers, 4 seconds with 3 workers, etc). In other words, smaller numbers are better in the graphs, as they are run-times of the synthetic benchmark.

The drawer works exceptionally well when there is some work (Figure 3) or ample work (Figure 4) for a few number of threads, but works horribly in the neglible workload (Figure 2). The drawer has the advantage of the simple interface, but unfortunately, the drawer works poorly as the number of threads rises and the work per thread drops. The problem is that each thread tends to be stepping on each other: all the threads are trying to access the same memory location (the current value) and even though the atomic add is fast, it stalls the bus as all the other threads who are trying to do atomic adds at the same time. This collateral damage slows the application.

4. The Cupboard

The cupboard is an abstraction built on top of the drawer. From the previous section, it's apparent that the drawer simply had too much contention when there are too many threads trying to get into the one drawer at once. Part of the problem is the drawer has no way of knowing how many workers are trying to pull from it, so it has no way to level out the load among the threads. The cupboard is based on the metaphor of a kitchen cupboard with multiple drawers. By having multiple drawers, each worker can stay out of the way of other workers by simply looking at one drawer most of the time. When a drawer is empty, a worker will have to start looking at other drawers, but by then, the majority of work will be have been done.

The `iCupboard` is implemented a vector of `iDrawers`: each thread gets its own drawer. The integers are divided up evenly (as possible) into the drawers of the cupboard so that each drawer contains about the same number of integers. The interface to the cupboard is slightly clumsier, as each *get* has to specify which drawer to look in (and deal with the return drawer in which work was found). Consider the following example, where the cupboard is set-up globally: Note that the first argument to the cupboard is the length of the sequence (*i.e.,* the length of the work vector) and the second argument is the number of workers. This is slightly different than the drawer where the first argument is the starting index and the second argument is the length of the sequence.

```
// Set-up
vector<string> work{ "work1", "work2", "work3"};
iCupboard bag(work.length(), NUMBER_OF_WORKERS);
                          // Bag of { 0, 1, 2 }
```

Thus a number of worker threads is created, where each worker thread is assigned a number from 0 to *number of worker threads*-1:

```
// Each thread gets its own thread number:
//      0..NUMBER_OF_WORKERS-1
// Main loop for each thread, getting work from
// bag and operating on it
void thread_main_loop (int thread_number)
{
  int starting_drawer = thread_number;
  int ending_drawer = -1;
  uint32_t work_index;
  while (bag.get(starting_drawer,
                 work_index, ending_drawer)) {
     string& single_work = work[work_index];
     doSomething(single_work);
     starting_drawer = ending_drawer;
  }
}
```

Most of the time, the starting drawer will be the same as the ending drawer. As the cupboard approaches being empty, the ending drawer is the drawer where the item was found. Once all the drawers are empty, the cupboard is empty.

Internally, the *get* of the cupboard is built on the get of the drawer:

```
// Get an item from SOME drawer: start looking
// from the given drawer.  If we find something,
// return true with the item and which drawer we
// found it in.  Otherwise, return false
// (meaning cupboard is empty).
bool get (int starting_drawer,
          uint32_t& item, int& ending_drawer)
{
  const int len = drawers_.length();
  int drawer = starting_drawer;
  for (int ii=0; ii<len; ii++) {

    if (drawers_[drawer].get(item)) {
      // Found a drawer with stuff!
      ending_drawer = drawer;
      return true;
    }
    if (!protect_) break;
    drawer = (drawer+1) % len;
  }
  return false;
}
```

4.1 Initial Timings and False Sharing

Intuitively, it makes sense that the cupboard would perform better than the drawer. Why is it not any better for the *negligible* workload as shown in Figure 5? The problem is that the cupboard is implemented as a vector of drawers, but the drawers are all next to each other in memory. The drawer data structure is so small that when the processor loading a cache line, it picks up the drawer next to it. Thus, *every single access* via a `get` causes the cache line to be refilled: this cache line contains the drawer of the worker and the drawer of its neighboring worker, so this will cause cache contention between processors (*i.e.,* collateral damage) as the processors try to access the drawers concurrently. This is *false sharing* and can cause significant slowdowns.

The purpose of the cupboard was to distribute the work. A useful metaphor is to imagine the drawers of a cupboard are too close together, so that multiple workers run into each other when

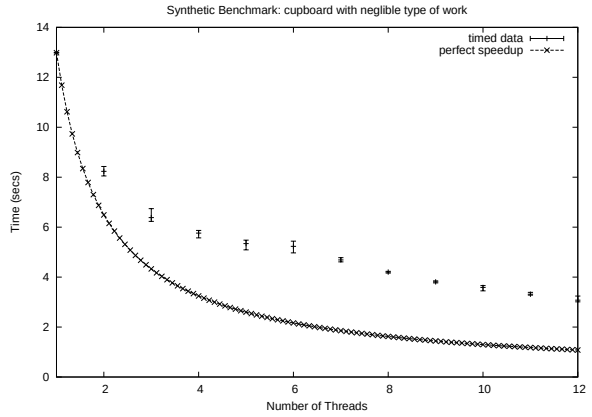

Figure 5. Demonstrating an initial cupboard implementation on a *negligible* work type: false sharing a major issue

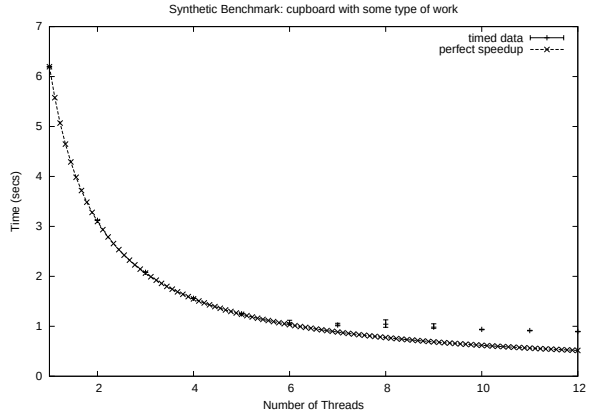

Figure 6. Demonstrating an initial cupboard implementation on a *some* work type

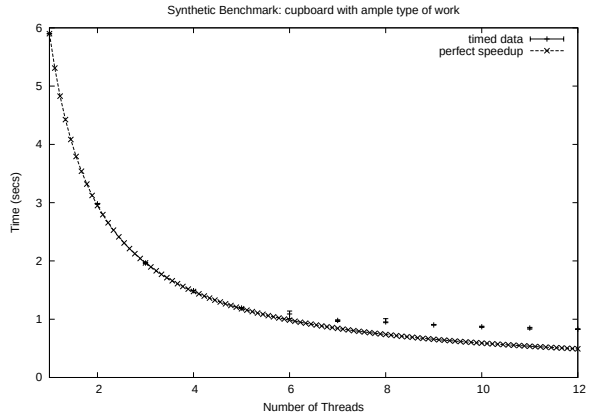

Figure 7. Demonstrating an initial cupboard implementation on a *ample* work type

they try to open the drawers. The workers need "elbow room" so they can work together and not intefere with each other.

A very simple way to eliminate false sharing is to add some space between drawers:

```
class iDrawer {
  ...
  // Current value
  std::atomic<uint32_t> current_;

  // eliminate false sharing between
  // current and upperBound: processors
  // can cache upperBound as it set once
  // at construction
  char padding_between_frequent[64];

  // upper bound
  uint32_t upperBound_;

  // eliminate cache line sharing
  // or false sharing BETWEEN drawers
  char padding_between_drawers[64];
}
```

Note that there's two places padding has been added to prevent false sharing: the padding between current and upper bound is to allow upper bound to be cached without being blown in and out of cache as current is updated. The padding at the end of the class is to prevent adjacent drawers from sharing cache lines. The padding between current and upper bound is overkill, but it can make some difference in limited situations.

The "magic number" 64 is a reasonable upper bound for the size of a cache-line on the particular processor. Anything smaller would probably not eliminate false sharing on more modern CPUs. There are other dynamic techniques for finding the cache-line size[10] that are outside the scope of this paper.

4.2 Final Timings

The cupboard behaves much better than the single drawer with multiple threads once false sharing has been eliminated; Figure 8 in particular shows the first time any implementation of a bag has been able to achieve speedup approaching the perfect speedup for the *negligible* work type. The cupboard, across all worktypes, exhibits significantly less collateral damage from multiple threads when the number of worker threads increases; better performance was obtained as the amount of speed up scaled with the number of full CPUs in all cases. Figure 9 and Figure 10 show that the final implementation preserved the near perfect speedup of initial version of the once the false sharing issue was resolved.

With the cupboard, thread-neutrality is basically achieved up to 6 threads (for this machine). Unfortunately, it would appear that the cupboard is not fully thread-neutral for a large number of threads: this will be discussed in the next section.

5. Hyperthreads

The nature of the synthetic experiment was to compare different workloads with many different threads. The machine in question is a six core Intel(R) Core(TM) i7-3960X CPU running at 3.30GHz. The machine appears to have twelve processors due to hyperthreading (also known as simultaneous multi-threading), but for the type of work in this experiment, the hyperthreads do not realize any speedup. Similar results have been seen in [9] and [10] where hyperthreading has little to no effect (or worse, potential slowdowns).

Given the nature of the experiment, it is not surprising that the hyperthreads offered no extra speedup, as [9][10] suggested. Since the work done by each thread was so similar, the functional units

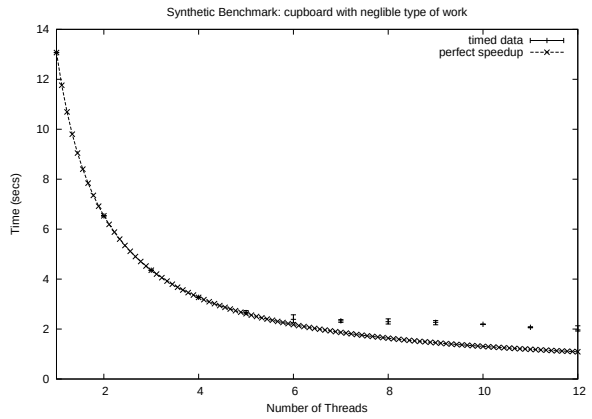

Figure 8. Demonstrating final cupboard implementation on a *negligible* work type: false sharing issue resolved

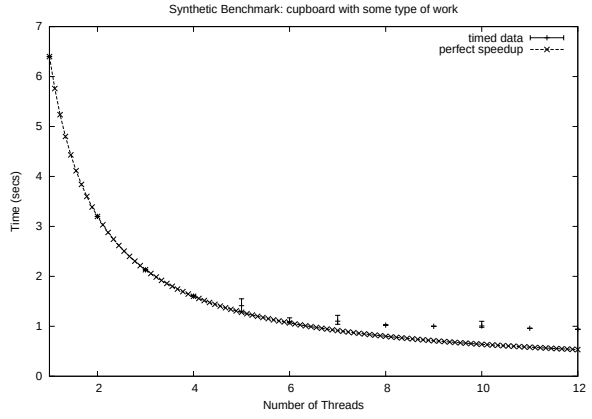

Figure 9. Demonstrating final cupboard implementation on a *some* work type

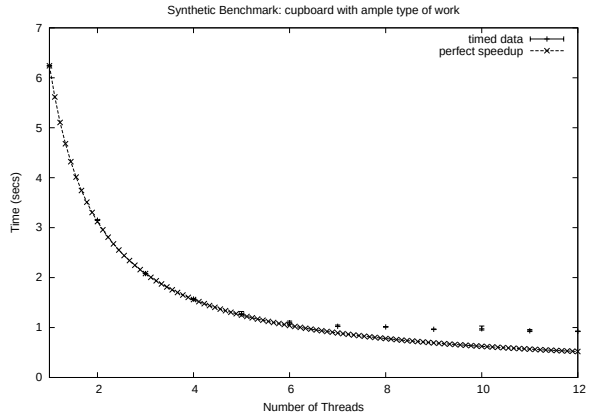

Figure 10. Demonstrating final cupboard implementation on a *ample* work type

being used by the processors were always the same, so there were never any unused functional units inside the machine that could be exploited by hyperthreading. Had the workload been (say) a mixture of floating point and integer arithmetic, the hyperthreading may have been more useful.

It was encouraging to see that for the 6 full processors on the machine under test, the drawer (for some and ample worloads) and the cupboard (under all workloads) were able to achieve perfect speedup for up to 6 processors. The trends of the graphs suggest that the extra hyperthreads gave no extra benefit, but perhaps more important, the extra threads didn't deleteriously change the performance.

In other words, taking into account the hyperthreaded nature of the machine and the nature of the experiment, the cupboard was able to achieve true thread-neutrality.

6. Real World Results

6.1 The Cross Ambiguity Function

Many problems work well in the work crew paradigm, see [3][5][6]. One such application of the work crew is computing a the Cross-Ambiguity Function (the CAF): the CAF is a embarassingly parallel computation used heavily in Digital Signal Processing. A CAF is a two-dimensional structure of TDOA (time difference of arrival) vs. FDOA (frequency difference of arrival): it's embarassingly parallel as each line of the CAF can be computed (TDOA line) independently of any other line. CAF computation is also potentially very expensive as each line of the CAF involves a fast-fourier transform per line (strictly speaking, an inverse FFT). There are many ways to compute a CAF as well as many types of CAFs, but this study concentrates on the frequency domain CAF using fourier transforms. More information about CAFS can be gleaned in [12]. Computing a CAF with multiple threads fits the work crew paradigm well, as it is typically a compute intensive problem that divides well into separate pieces of work.

6.2 Performance

The performance numbers of the tests seen so far reflect a synthetic benchmark where all work is a simple compute-intensive for-loop. Real world performance of the bag in a running application is a critically important metric. The tests below showing the results of using a bag structure in a real-world Digital Signal Processing (DSP) application using a CAF. The DSP application contains multiple threads (besides the threads of the CAF) that compete for system resources, although the CAF computation is typically the bottleneck of the application. The work types represented by the real-world application are negigible (for very small sized CAFs: FFT sizes of 512), some (for moderate sized CAFS: FFT sizes of 16384), and ample (for very large CAFs: FFT sizes of 32000000).

What is seen and expected in Figure 11 and Figure 12 is that using the bag instead of the *apriori* division of work has little effect on the final running time (perhaps the bag gives slightly better performance in Figure 12). Certainly it would be better if the performance had increased, but it's important to note that the bag (which now has significantly more synchronization per line) has not slowed down the application. To be fair, (for small and moderate sized CAFs), the slowest worker problem was never a real problem for the application, so the best to be hoped for is that the bag has no effect. And that's what Figures 11 and 12 demonstrate.

When the CAF being computed is large is when the bag makes a difference: See Figure 13. The cupboard basically trims the ragged edge of the ending off so that slowest worker(s) problem is mitigated and the runtime is more predictable: for the very large CAF it seems to make the CAF between 5 and 10% faster.

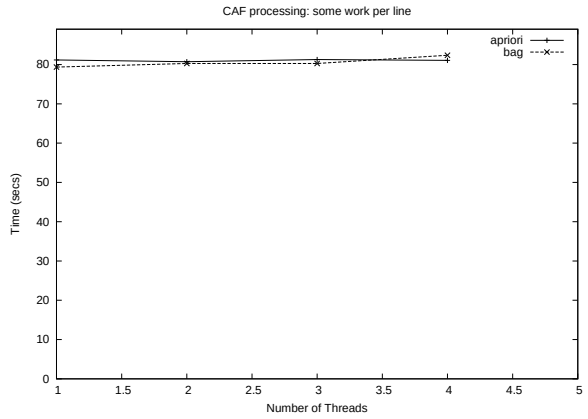

Figure 11. Computing a small CAF within a full application where each CAF line is *negligible* amount of work: Comparing apriori division of work vs. pulling from a bag dynamically. For this workload, the change in running time is within the error of the experiment, so changing to a bag does not deliteriously change the running time of the application. Note that the work type was so negligible that dividing the work among multiple threads had no impact.

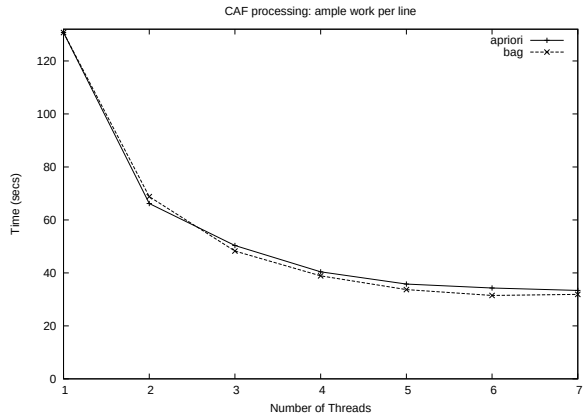

Figure 12. Computing a medium-sized CAF within a full application where each CAF line is a *some* amount of work: Comparing apriori division of work vs. pulling from a bag dynamically. For this workload, the change is within the error of the experiment, so changing to a bag does not deliteriously change the running time of the application.

The very large FFTs illustrate an important feature of CAF processing: at some point, multiple threads do not give much speedup (some of this effect was seen in Figures 11 and 12). There are certainly many possible explanations (false sharing, serialization of heap), but the massive memory of the CAF is key. All the threads have their buffers pre-allocated and each thread is essentially doing only floating point operations across memory. An important insight is that the very large CAF computation breaks cache; when FFTs become very large, the caches aren't really helpful as the computation strides all across memory. *The main memory subsystem is being stressed to the breaking point*. The threads can process as fast as they can get data, but the bus delivering the data to the CPUS is being stressed by every thread, as data tends to not be in any caches; CPU caches are megabytes large, but not $32 * 8$ megabytes

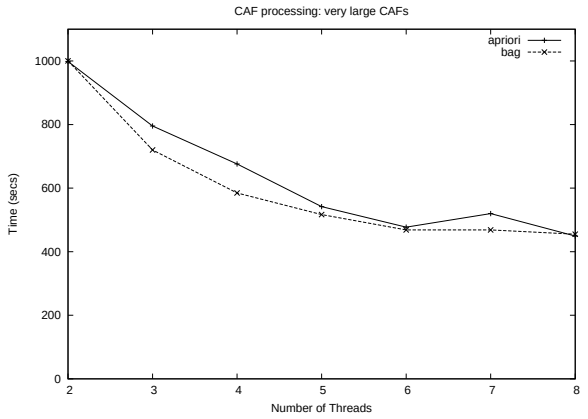

Figure 13. Computing a large-sized CAF within a full application where each CAF line is an *ample* amount of work: Comparing apriori division of work vs. pulling from a bag dynamically. For this workload, the bag offers some real speedup. Seing more threads would have been useful to see the trends further out, but swapping would occur because the CAFs were just too big.

large. Some correlation of this fact (that the memory subsystem is to blame for the lack of CAF speedup) can be seen by moving to a machine with the same configuration, but a faster front-side bus. The speedup seen correlates directly with the speedup of the front-side bus.

The conclusion of the real-world application data is that the bag does not adversely change the running time at all for small CAFS, but can make improvements on the running time when the CAF is large–this is done by avoiding the *slowest worker* problem. These conclusions would hold for many CAF-style workloads, where the work done is primarily floating point work.

7. Conclusion

Two new abstractions have been introduced for the multi-threaded bag. These thread-safe bag abstractions are simple and can be built in C++11 or pre-C++11 compilers using simple atomic primitives. The drawer is the fundamental thread-safe container that works well in most situations, but it has no notion of the number of workers attached so it scales poorly when the number of workers gets large and the work is negligible. The cupboard (which is built of drawers) scales much better when the number of workers is larger and the workload is negligible.

Although it appears the drawer and cupboard don't scale well using the extra threads provided by hyper-threading, the work of [9] and [10] suggest that hyperthreading doesn't really offer any benefits for these types of experiments. The fact that the drawer and cupboard do not degrade performance as hyperthreads are used, coupled with the perfect scaling (up to the number of full processors on the machine), suggest that the drawer and bag are thread-neutral.

Author

- Richard T. Saunders has worked with C++ for 20+ years at Rincon Research Corporation doing soft real-time Digital Signal Processing; He has built and fielded many real systems using both C++ and Python. He also occasionally teaches Computer Organization, Software Engineering, Python and C at the University of Arizona in Tucson for the Computer Science and SISTA departments.

References

[1] A. Alexandescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[2] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[3] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley, Boston, MA, 1997.

[4] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[6] R. Gerber and A. Binstock. *Programming with Hyper-Threading Technology*. Intel Press, 2004.

[7] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[8] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[9] Y. Ruan, V. S. Pai, E. Nahum, and J. M. Tracey. Evaluating the impact of simultaneous multithreading on network servers using real hardware. In *ACM SIGMETRICS Conference on Measurement and modeling of computer systems*, pages 315–326, 2005.

[10] R. T. Saunders. A portable framework for high-speed parallel producer/consumers on real cmp, smt and smp architectures. In *Parallel and Distributed Processing Symposium, IPDPS 2007, IEEE International*, pages 1–8. ACM, 2007.

[11] R. T. Saunders. Opencontainers: A study of portable techniques for thread-heavy applications. *www.picklingtools.com/openconnew.pdf*, 2011.

[12] D. C. Smith and D. J. Nelson. A generalized cross-ambiguity function for geolocation. In *Proceedings of the Fifth IASTED International Conference on Circuits, Signals and Systems*, CSS '07, pages 239–244, Anaheim, CA, USA, 2007. ACTA Press.

[13] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *37th Symposium on Microarchitecture*, pages 183–194. IEEE, 2004.

[14] A. Williams. *C++ concurrency in action: practical multithreading*. Manning Publ., Shelter Island, NY, 2012. The book can be consulted by contacting: PH-ADT: Sicoe, Alexandru Dan.