Thread-Safe and Thread-Neutral Bags

Richard T. Saunders – Rincon Research Corporation

Overview

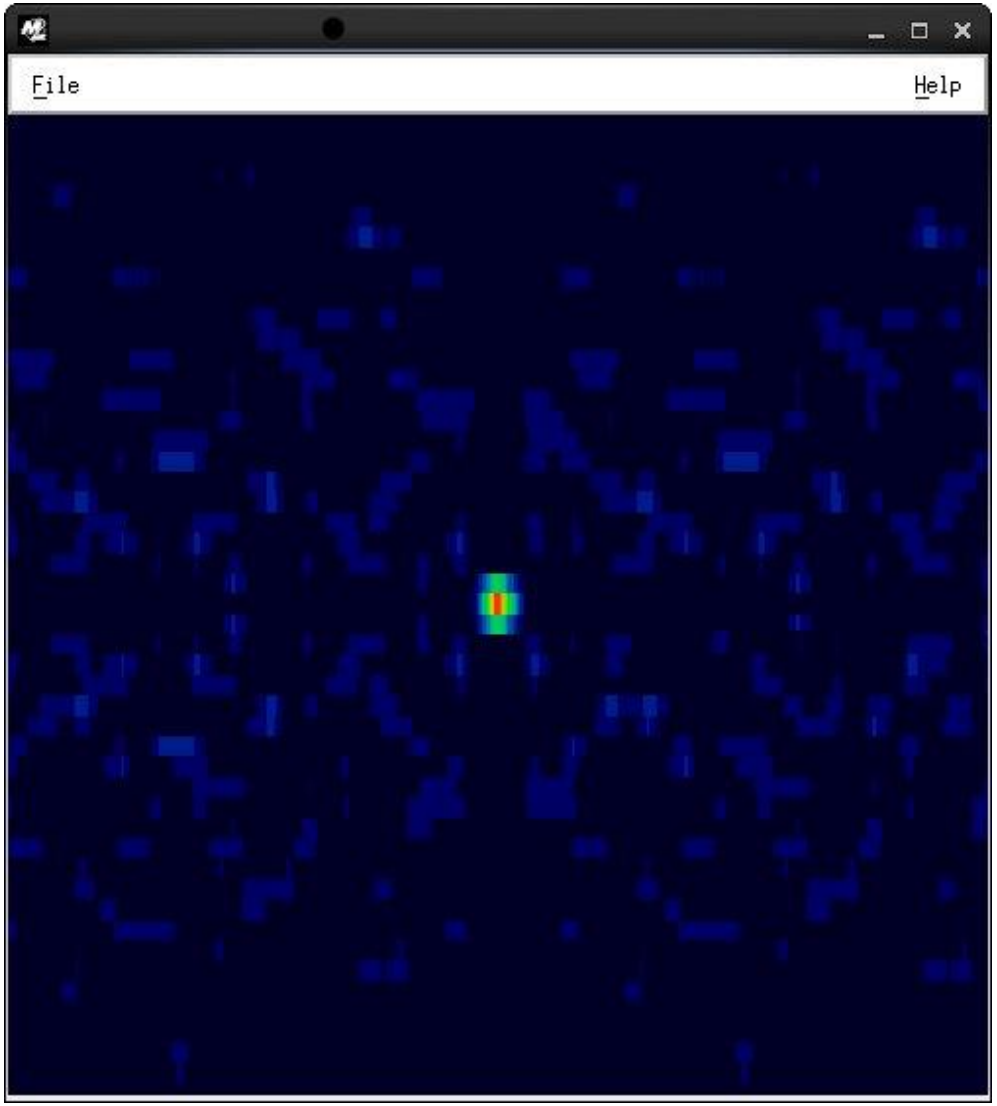
- Overview
- Work Crew
- Slowest Worker Problem
- The Bag
  - Implementation 1: The Drawer
  - Implementation 2: The Cupboard
- False Sharing
- Hyper-thread Problems
- Conclusion

It's Josh's Fault!

- Dealing with multiple threads in C++ to compute a heavy weight DSP object called a Cross Ambiguity Function (CAF)
- Too slow!
  - "Why do the threads all finish at different times?"
  - If they all finished at the same time, the CAF would be faster!

- This led to investigations of how to implement THE BAG, with all sorts of ramifications
  - Thread-Neutrality
  - Thread-Safety
  - Hyper-Threading
  - "Slowest Worker" Problem
  - False Sharing
  - C++ 11 Atomics

Cross Ambiguity Function: CAF



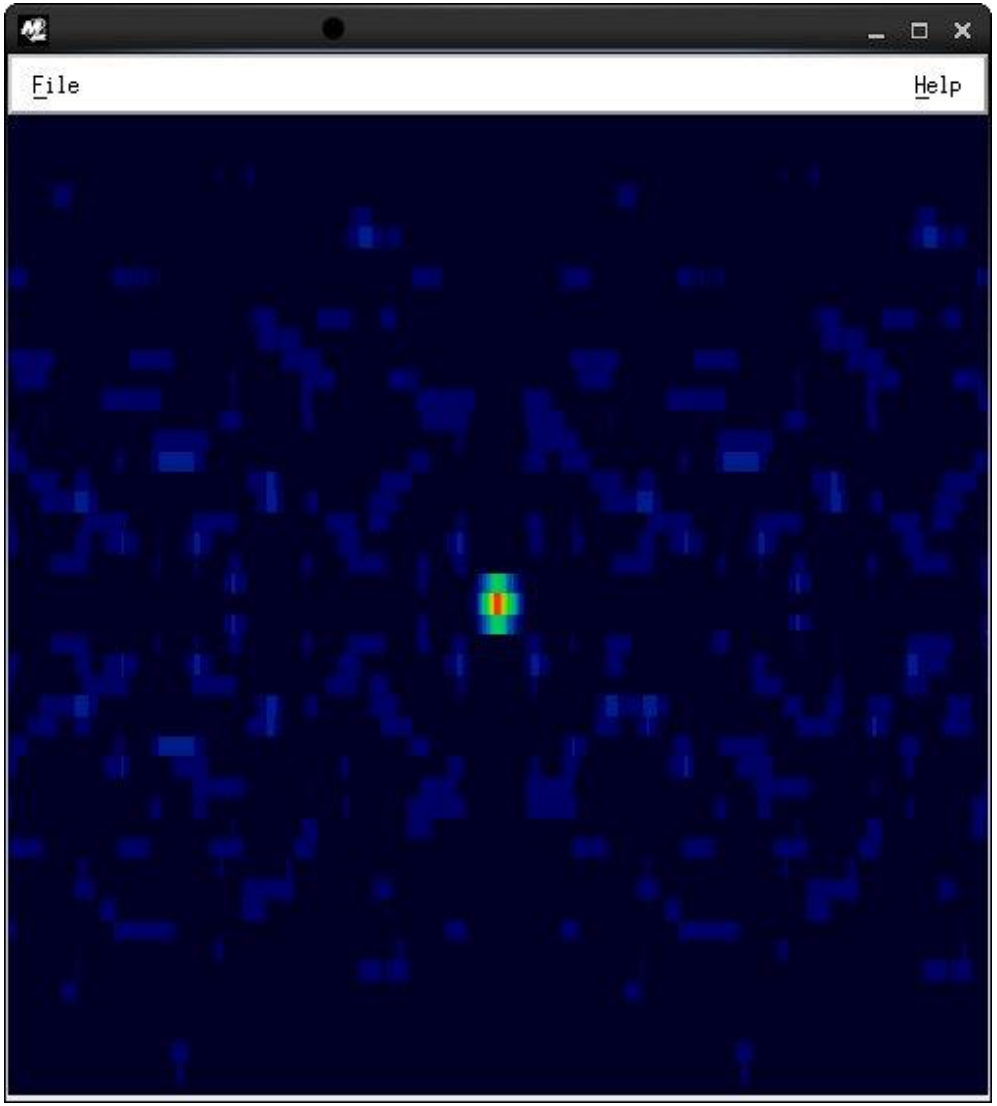
CAF: Embarrassingly Parallel

- CAF: Cross Ambiguity Function
  - Digital Signal Processing bread and butter function
  - Time Difference of Arrival (TDOA) vs Frequency Difference of Arrival (FDOA)
    - Signals have time and frequency shifts, trying to find true time and frequency
    - Try to find where energy is “maximized”

- Embarrassingly Parallel!
  - Every line is “essentially” an inverse Fast Fourier Transform
    - Read: each line is expensive to compute
  - Every line can be computed in parallel by separate thread

Original Strategy: a priori Division of Lines

After all threads finish, final CAF image is generated



Simple Way to Divide Work: a priori

- a priori
  - Evenly divide up work among all workers
  - Worker 1 gets X pieces of work, Worker 2 gets X pieces of work…
  - … seems fair …

- Ideally: If work is divided up a priori
  - each worker would get exactly the same amount of work
  - every worker would finish at exactly the same time
- Realistically:
  - Each worker takes a different amount of time
    - Scheduled by operating system differently
    - Other applications running simultaneously interfere with workers' consistency
    - Hyper-threading causes added scheduling irregularities

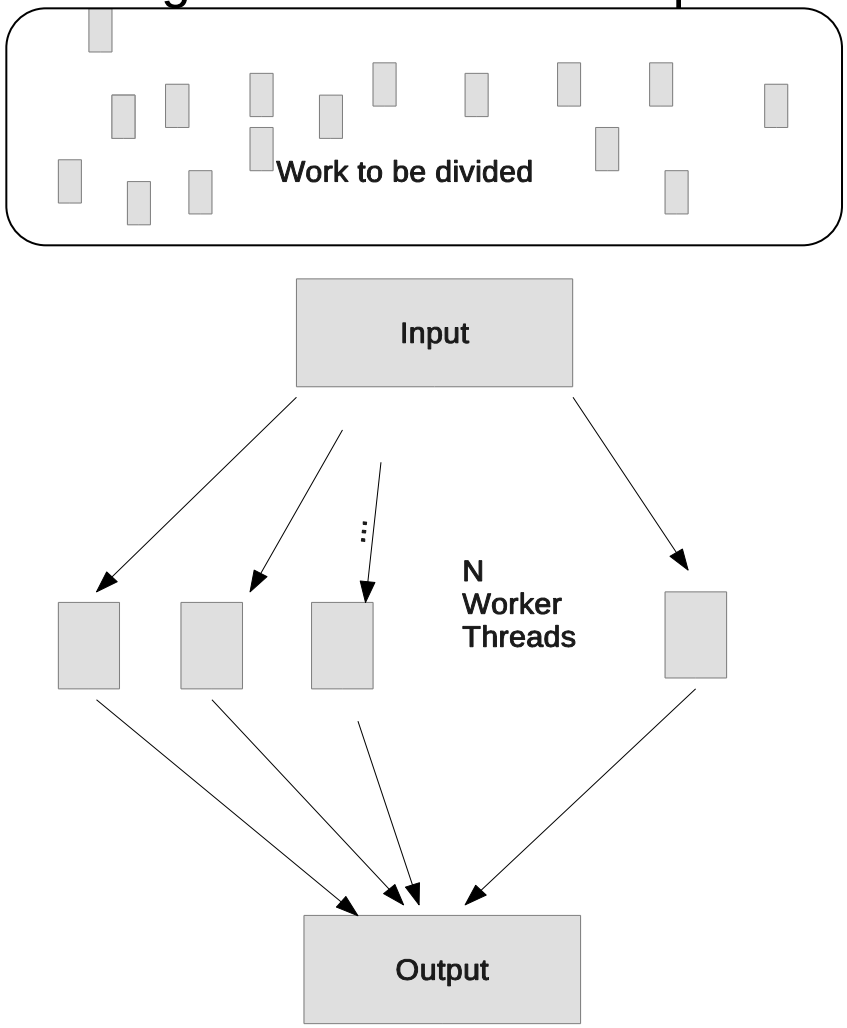
Definition: Slowest Worker Problem

- Under a priori division, work crew becomes throttled by the SLOWEST WORKER:
  - Each worker has done the “same amount of work”, but realistically, scheduling has caused the SLOWEST WORKER to limit how fast the work crew can finish
  - THIS IS THE SLOWEST WORKER PROBLEM
    - Scheduling inconsistencies throttle the work crew

Definitions: Bag

- A bag is a fundamental container holding future work for multiple threads
  - When a worker wants work, it **gets** a *single* piece of work from bag

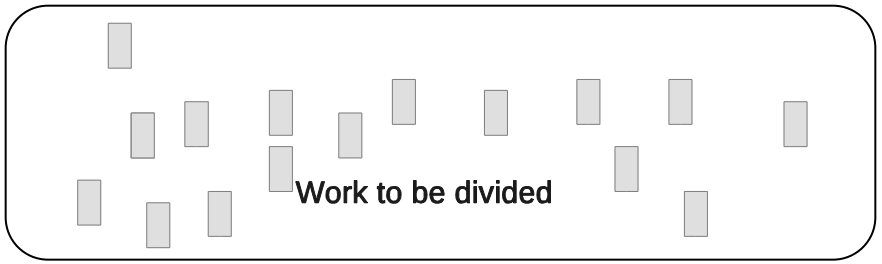
BAG:



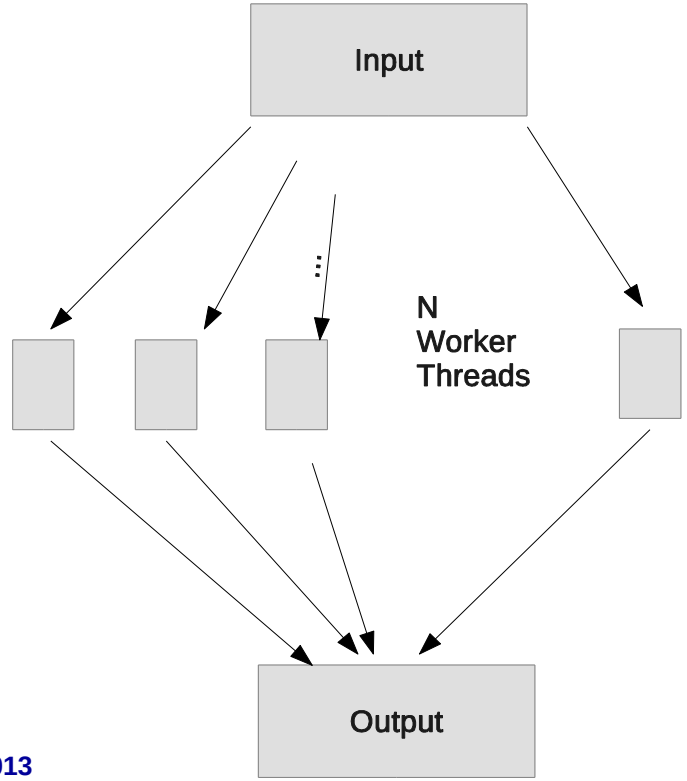
Definitions: Work Crew

- A work-crew is a group of related threads pulling work from the bag until the bag is empty:

BAG:



Work Crew
(aka, map-reduce model)
(aka, OpenMP model)



Definition: Thread-Safe and Thread-Neutral

- Thread-Safe Bag
  - The state of the bag is never inconsistent amidst multiple threads
    - i.e., each piece of work only served exactly once
    - (no accidental serves of same data or failing to serve)
  - BAG IS CORRECT

- Thread-Neutral Bag
  - Multiple threads do not impede each other's progress as they reach into the bag to grab work
    - Lack of "collateral damage" from other threads
  - BAG IS FAST

Bag Allows *Dynamic* Distribution of Work

- The bag allows threads to pull ***dynamically*** a piece of work whenever they need it (as opposed to *a priori* division of work)
  - Avoids SLOWEST WORKER problem
    - A poorly scheduled worker ("Fred") won't completely throttle a work-crew
      - "Come on!!! When is Fred going to finish???"
    - Instead, the X work of Fred will dynamically be redistributed among all threads
  - All threads will finish at "approximately" the same time
- Caveat: still have to divide up work into small enough quanta that work can be distributed
    - Poorly divided work can still suffer from similar problems … but that's the application's fault, not the scheduler
      - There is a sweet spot for work size
- Our purpose: *avoid slowdowns from scheduling inconsistencies*

Bag: Abstract Idea

- Fundamental Operation:
  - get: returns a single piece of work for worker thread
- Interface?
  - One Way: Microsoft `ConcurrentBag<T>`
    - Huge interface for gets, puts, complex interface
  - A Simpler Way:
    - A key simplification is that a bag should **only** return integers in range 0..n-1
    - Then “real work” can be kept in a well-understood container like STL vector, where the bag gives “indexes” into the vector.
- Leverage STL `vector<T>`
  - Everyone understands `vector`!
  - `Bag` is conceptually simple
    - Gives you simple mechanism to build work (but you can implement any policy you want for filling vector of work)

Bag of Ints

```
class BAG_OF_INTS {
 // Return false if the bag is empty.
 // Otherwise return true and return
 // an integer from the bag
 bool get (uint32_t& result);
};
```

Bag of Ints Usage

```
vector<string> work{“w1”,“w2”,“w3”};
BAG_OF_INTS bag(0, work.length());
            // Bag of {0,1,2}

// main loop: get work from bag
uint32_t index;
while (bag.get(index)) {
    string& single_work = work[index];
    do_work(single_work);
}
```

The Drawer: Bag Implementation #1

- The fundamental thread-safe bag is the *iDrawer*
  - Only to emphasize integer nature
  - We are only dealing with integers; why not use fundamentally fast atomic int operations?
    - Why atomic? (So bag is thread-safe)

- Implemented using a C++11 Atomic primitive
  - Using a C++11 std::atomic<uint32_t>
  - Usually implemented using a CPU's atomic instructions
    - On Intel, many C++11 ops correspond to a single instruction

- Get:
  - Fundamentally, increment integer
    - That's it! (Of course, a bit more to it …)

iDrawer Implementation

```
struct iDrawer {
    iDrawer(uint32_t start, uint32_t length) :
      current_(start),upperBound_(start+length) {}

    bool get(uint32_t& index) {
      if (current_>= upperBound_) return false;
      uint32_t t=current_.fetch_add(1);
      if (t >= upperBound_) return false;
            // double-checked lock pattern
      index = t;  return true;
    }
   protected:
    std::atomic<uint32_t> current_;
    uint32_t upperBound_;
};
```

Correctness

- Think of

```
uint32_t temp = current_.fetch_add(1);
```

  - As

```
uint32_t temp  = current_++;
```

- Except, in the face of multiple threads, each thread will increment `current_` exactly once, and the state will never be inconsistent:
  - It will increment 0,1,2,… in order without losses or skips

Double-Checked Lock

- Like Singleton "Double-Checked Lock" pattern (*Modern C++ Design*, 6.9.1), we have to check against the upper bound **twice**.
  - Metaphor: like buying tickets at a full theater with too many tickets
    - First check keeps threads out once upper bound is past.
      - "SOLD OUT"
    - Second check is to make sure there are enough seats
      - Made it into the theater, but too many tickets may have been sold. We have to see if there is a seat for us.
- The second check is for threads that "sneak in" just before the theater sells out of tickets (just before the upper bound is surpassed)

- Note that this technique ONLY works if the number of threads that sneak can't cause `current_` to wrap back around to 0
  - In reality, not a problem: can always throttle back and reset occasionally (and 4 billion is pretty big…)

Pre C++11 Compilers

- Not everyone is at C++11 yet.

- In a GNU world
  - GNU C++ supports the same kind of atomic primitives. Only need two changes to use GNU atomic primitives

```
// In get: atomic fetch and add for GNU
uint32_t temp=__sync_fetch_and_add(current_, 1);

// Declaration of current
volatile uint32_t current_;
```

  - Can still support bag idea in earlier C++ compilers using GNU intrinsics.

Thread-Neutrality

- Is the bag Thread-Safe?
  - Yes: correctness was argued earlier
- Is the bag Thread-Neutral?
  - Thread-Neutral:
    - Lack of collateral damage from other threads
    - Other threads don't affect the run-time of the current thread

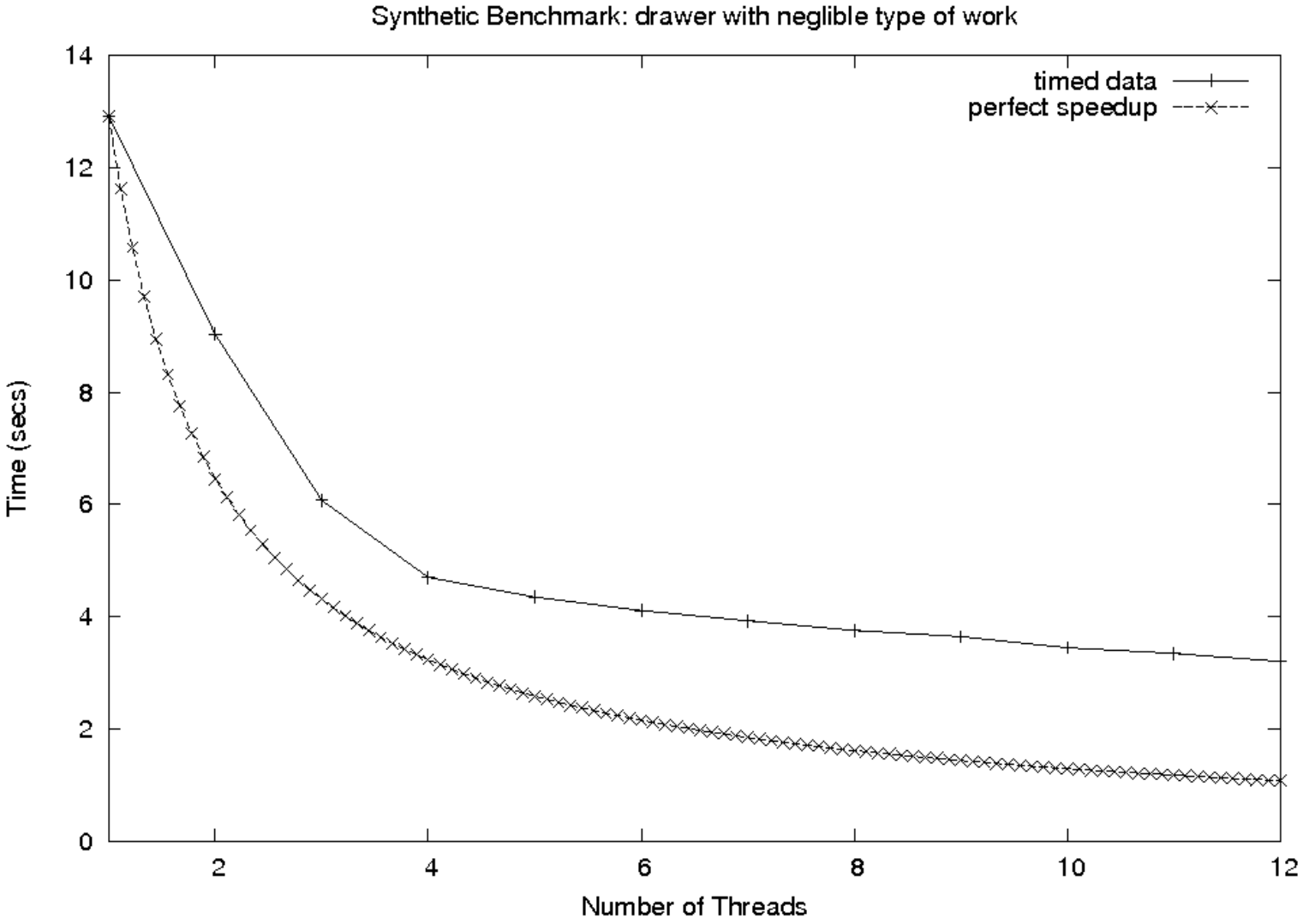
- Issues affecting Thread-Neutrality
  - How many worker threads are possibly getting in the way?
    - More threads == more likely that `current_` is hit hard
  - What kind of work is done by each thread?
    - *Negligible*: Each worker gets data out of bag as fast as possible
    - *Some*: Each worker gets data quickly, but some work is done before the worker calls get again
    - *Ample*: Each worker calls get infrequently

Synthetic Performance Tests

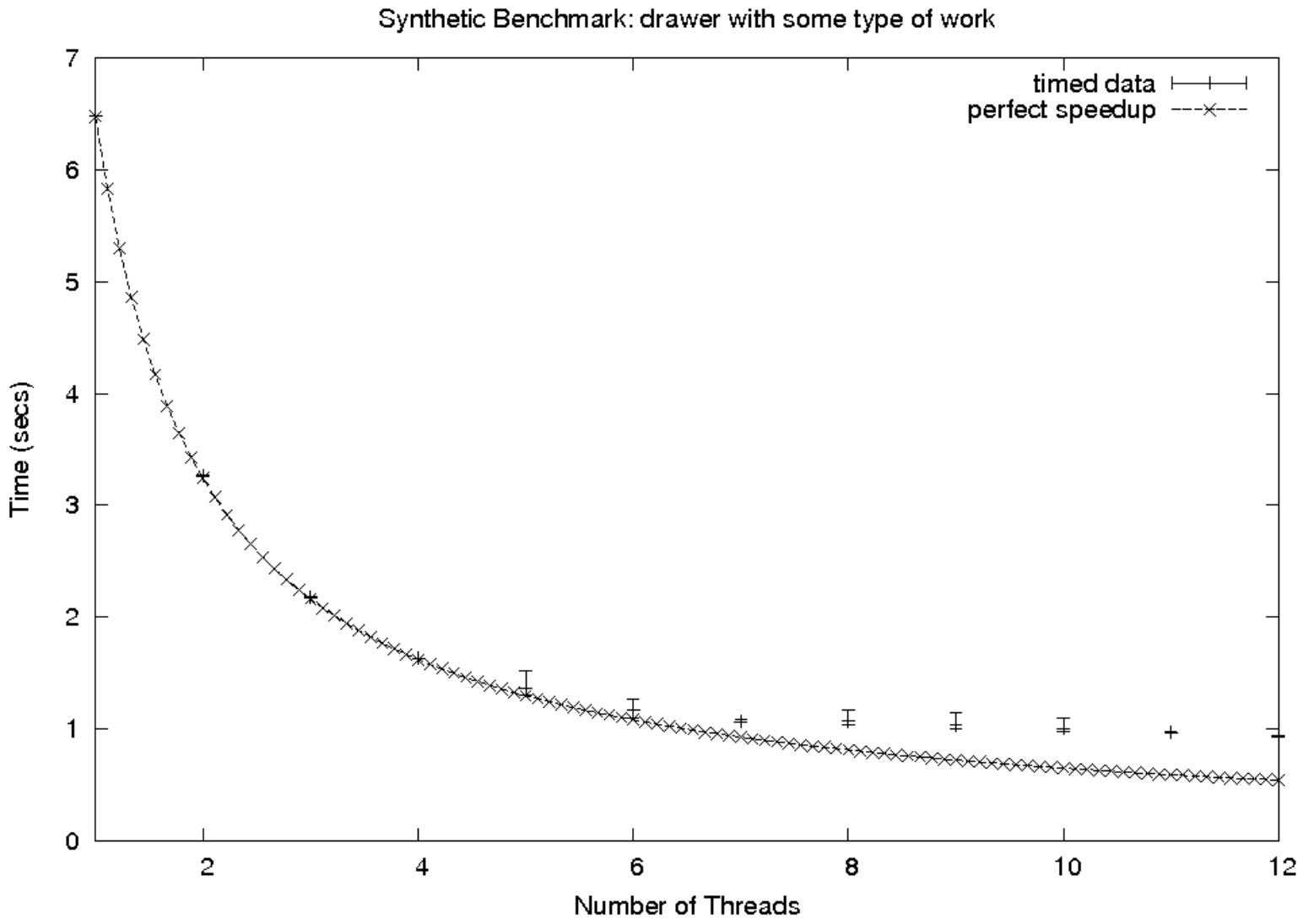
- Plot the number of threads vs. time: we plot two things
  - Time of the test
    - On a 6-CPU Intel Xeon machine (looks like 12 CPUS with hyper-threading)
  - Perfect Speedup
    - In a perfect world, n threads would be n TIMES faster than a single thread
    - Smaller is better (i.e., measuring runtimes)
- Work Types:
  - Negligible: exactly 1 add per get operation
  - Some: 100 adds before a get
  - Ample: 1e6 adds before a get

- Synthetic Performance Tests:
  - no real work was done in the running of these tests

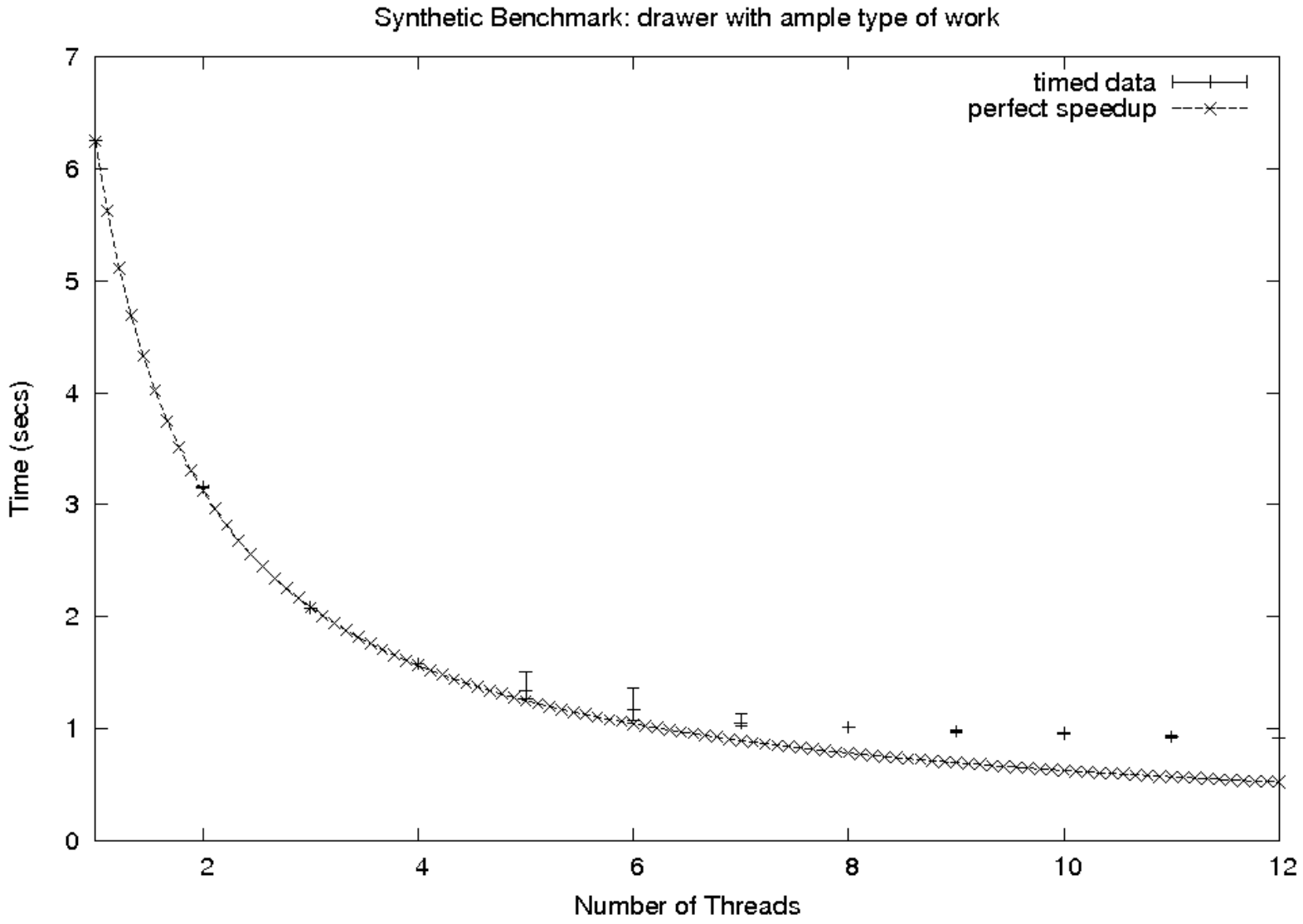
Drawer Timings: Negligible Work Per Get



Drawer Timings: Some Work per Get



Drawer Timings: Ample Work Per Get



Drawer Conclusions

- For *ample*, and *some* work types
  - Drawer is essentially “Thread-Neutral”
  - Reaches “perfect speedup” line
    - Modulo it doesn’t scale well after 6 processors … we’ll discuss this more in Hyper-Threading section

- For *negligible* work type:
  - Drawer is NOT “Thread-Neutral”!
  - Other workers slow down
    - Makes sense, all worker threads are incrementing `current_` as fast as they possibly can; at some point they get into each other’s way

- For most work types, a Drawer is probably “good enough”

Gnomes

Handling the Negligible Work Type: The Cupboard

- How can we keep workers ("gnomes") out of each other's way?
  - Solution: Give each gnome his own drawer
- In C++ speak:
  - A cupboard is a Vector of Drawers
  - Work is divided evenly between drawers
    - Drawer 0 gets integers 0..x
    - Drawer 1 gets integers x+1..2x
      - Incidentally, this is why Drawer constructor specifies start, len
  - NOTE: we are building the Cupboard atop an atomic primitive that already works:
    - we are not adding any more "synchronization" code!
    - Defer to drawer when possible
- Starts like a priori, but then rifle through others' drawers

Cupboard: get Usage Changes Slightly

```
vector<string> work = { … };
// Each thread gets its own number: 0..workers-1
// Main loop for each thread
void thread_main_loop (int thread_number) {

    int starting_drawer = thread_number;
    int ending_drawer = -1;  // returned by get
    int work_index = -1;     // returned by get
    while (bag.get(starting_drawer,
                   work_index, ending_drawer)) {
        string& single_work = work[work_index];
        doSomething(single_work);
        starting_drawer = ending_drawer;
    }
}
```

Cupboard get

- Once each worker is looking in a particular drawer, they tend to stay in that drawer
  - Multiple workers do NOT interfere with each other, as they each have their own drawer (most of the time)

Cupboard Get Implementation

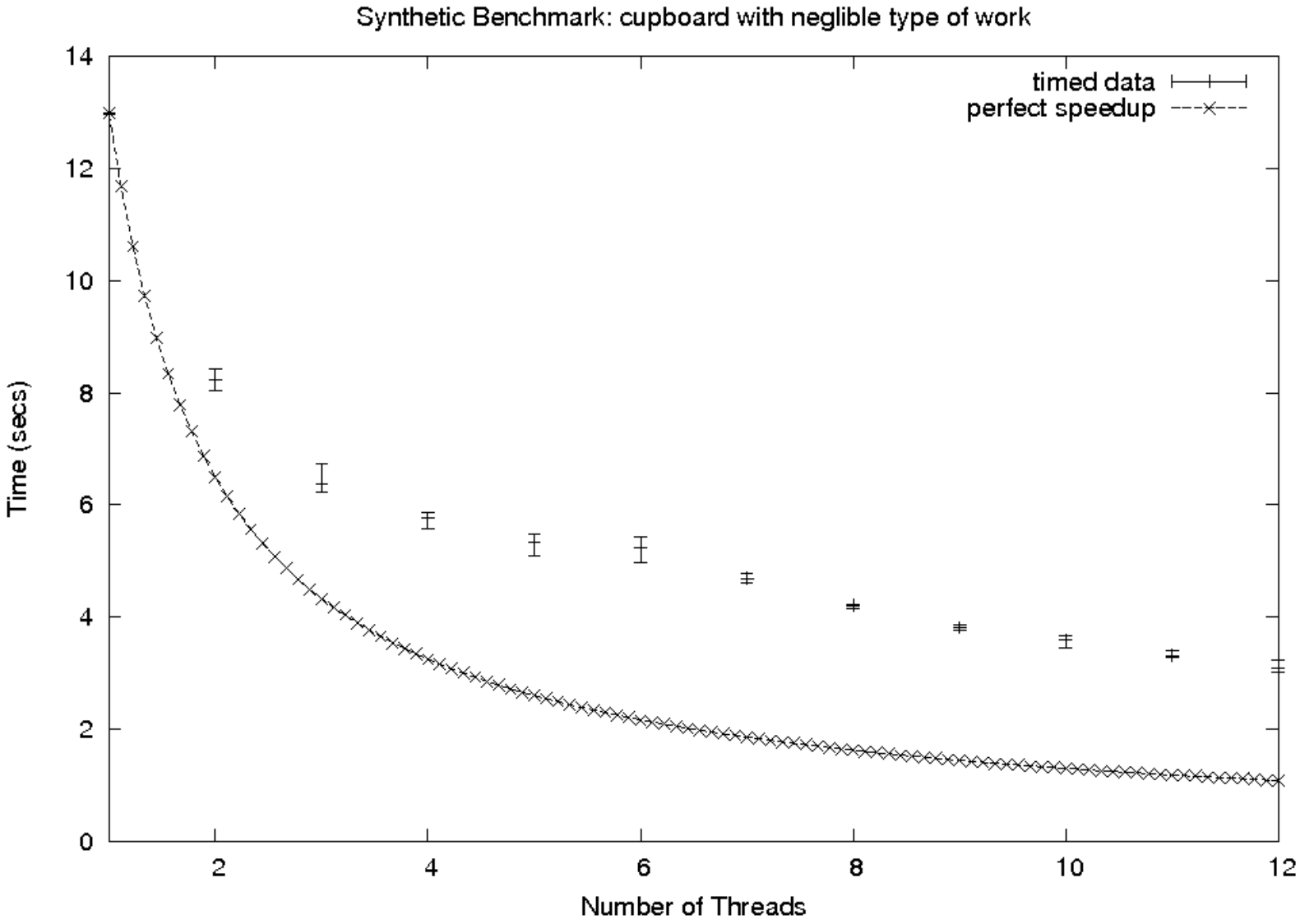
```
// Get from SOME drawer: start looking from the
// given drawer.  If something found, return true
// with the index found as well as ending drawer.
bool get(int starting_drawer,
         uint32_t& index, int& ending_drawer) {
      int drawer = starting_drawer;
      for (int ii=0; ii<drawers_.length(); ii++) {
         if (drawers_[drawer].get(index)) {
            // FOUND an item! Done!
            ending_drawer=drawer; return true;
         }
         drawer = (drawer+1) % drawers_.length();
      }
      return false;
}
```

Cupboard get: No Extra Synchronization Needed!

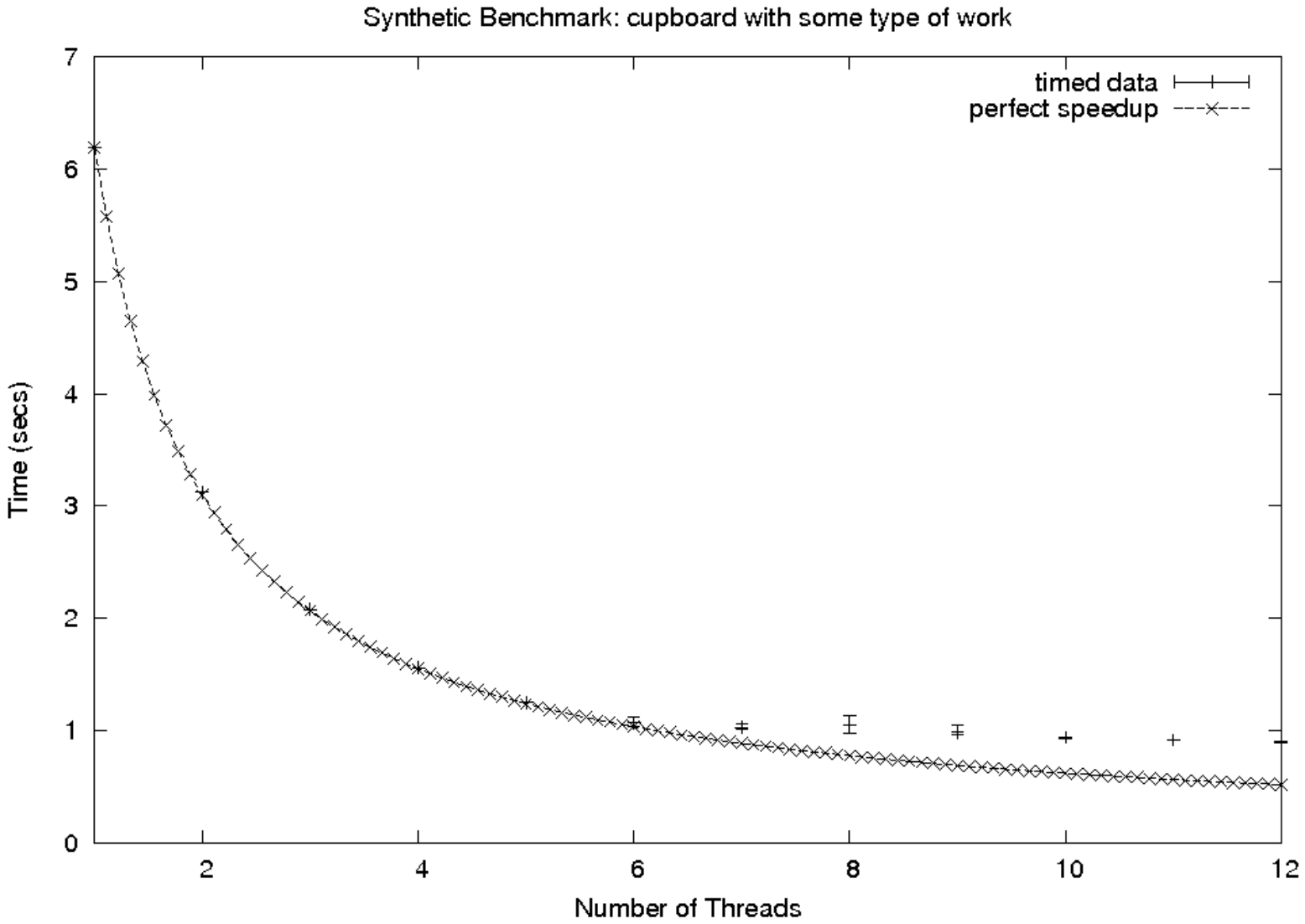
- Cupboard relies on Drawer being correct (Thread-Safe)

- Once a drawer is empty, it's empty
  - It can't get "refilled" except by creating a new cupboard
    - Seems like a limitation but …
    - Model of OpenMP:
      - A program is a series of map-reduce points
        - » Work created
        - » Work ends
        - » Move to next worker crew

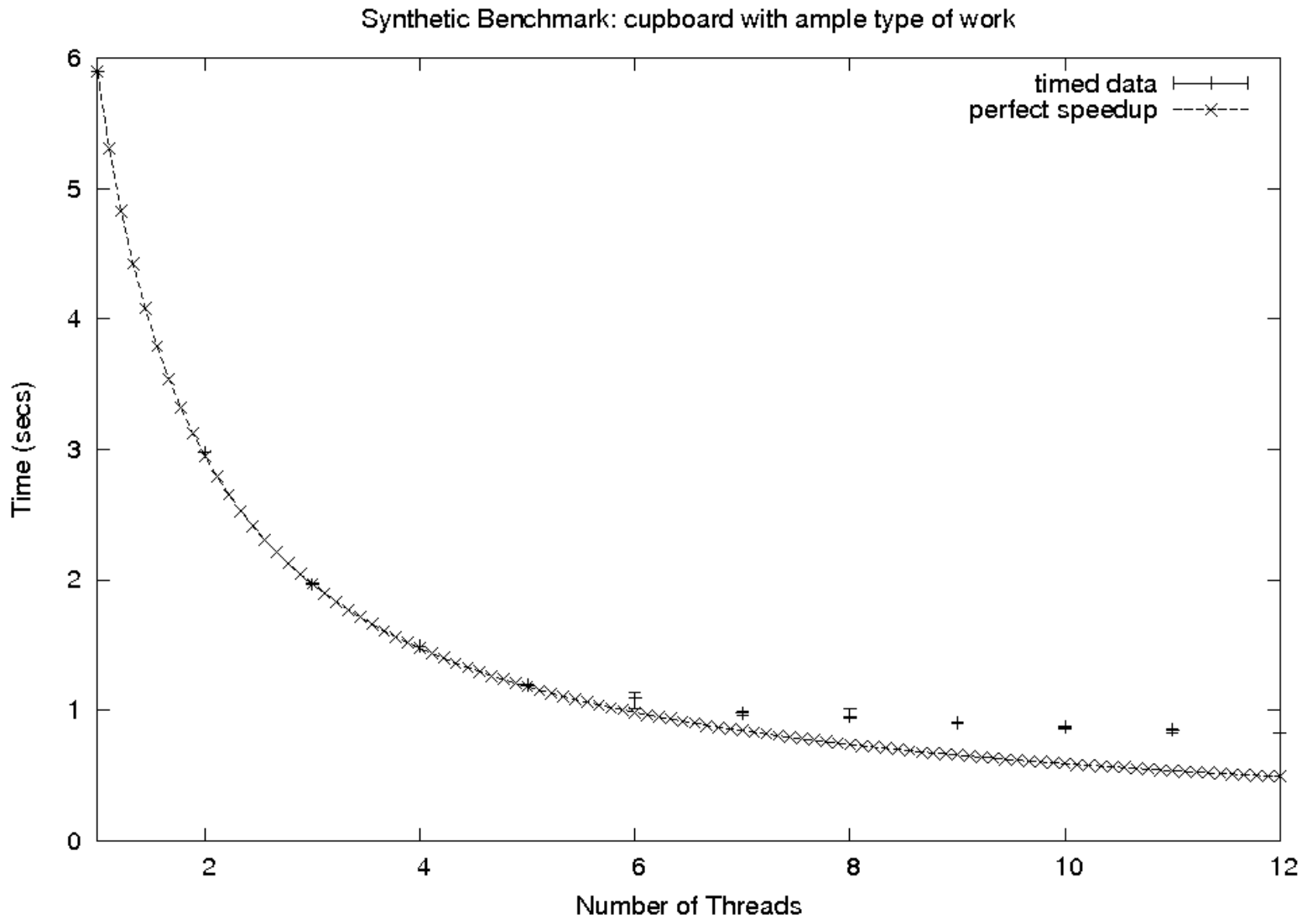
Cupboard Timings: Negligible



Cupboard Timings: Some



Cupboard Timings: Ample



Cupboard Performance: Why??

- The entire purpose of the cupboard is to make “negligible” work type scenario Thread-Neutral
  - Obviously, it didn’t work …

- Why?
  - Look to *High-Speed Producer/Consumer* work from IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2006:
    - Saunders, Jeffery, Jones
  - Same kind of problem: multiple threads accessing resources as fast as possible

- Problem:
  - False Sharing

Problem: False Sharing

- False Sharing:
  - When two threads accidentally access the same cache line
    - Causes cache-misses and forces caches to be refilled
      - EXPENSIVE operation!
  - Cache lines are typically 32-64 bytes: if two threads access data from same cache line … cache data "ping pongs" between caches

… ……..Inside the cupboard …….

```
vector<iDrawer> cupboard_data; // cupboard impl.
```

Drawers are stored contiguously in memory, inside the vector:

```
current_ = 4 bytes
upperBound _= 4 bytes     (drawer 0)
current = 4 bytes
upperBound_ = 4 bytes      (drawer 1)
```

- Drawer 0 and 1 in the same cache line!!!

Gnomes With No Elbow Room

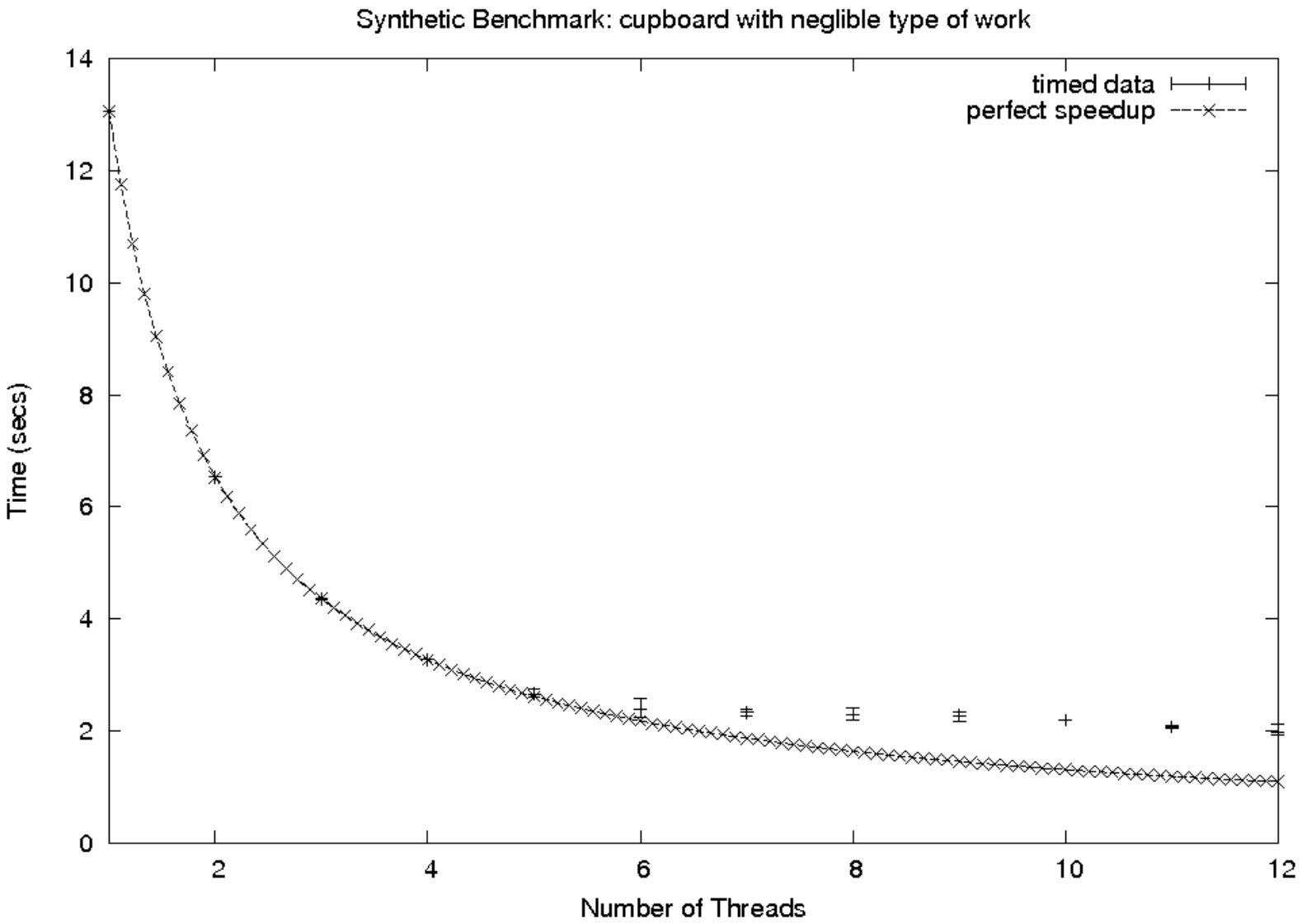
False Sharing Solution

- Add Padding
  - Fill out each drawer so it is the size of a cache line

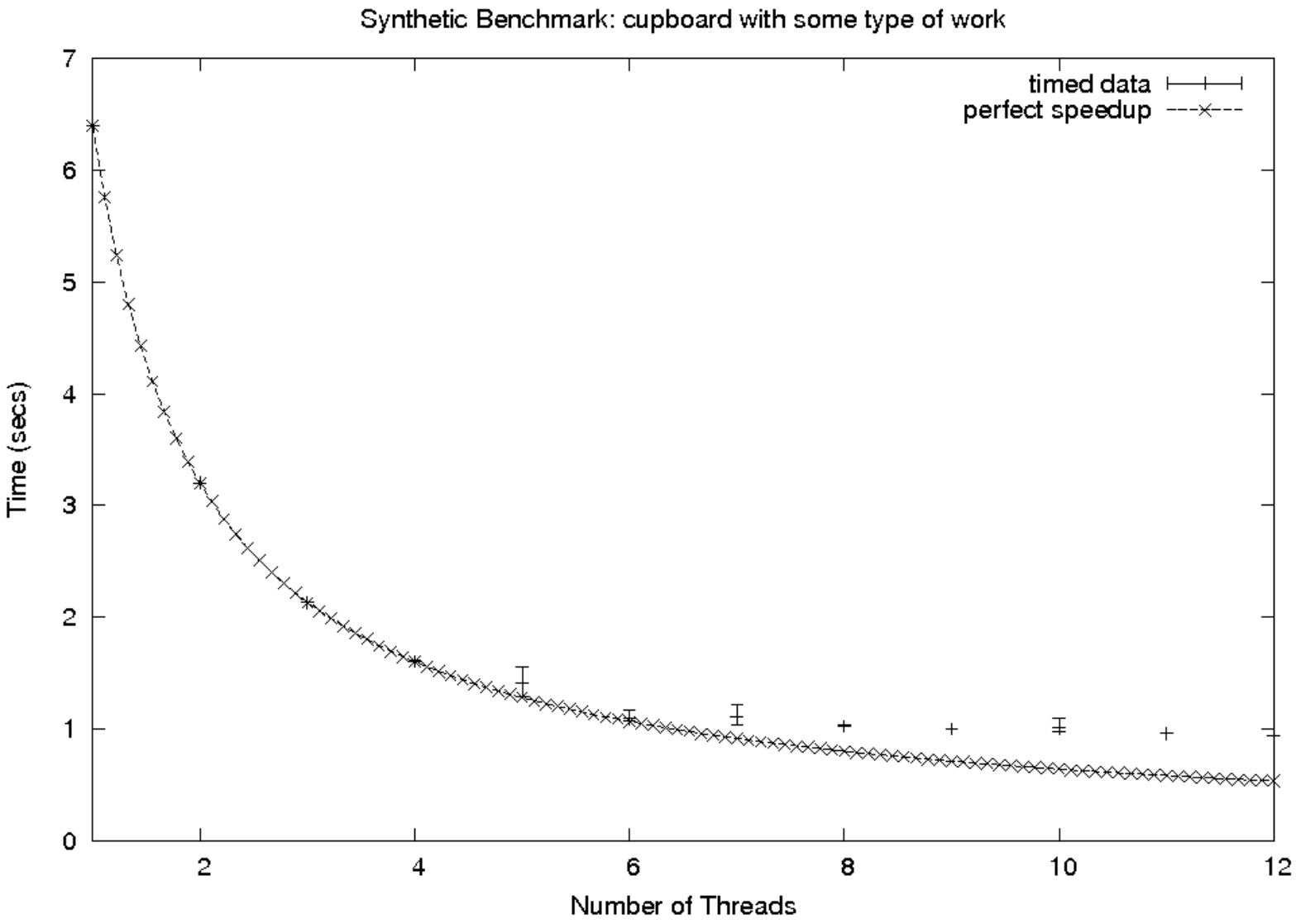
iDrawer Implementation (False Sharing Fixed)

```
struct iDrawer {
   …
  protected:
    std::atomic<uint32_t> current_;

    // Eliminate false sharing between current and
    // upper bound: processors can now cache
    // upperBound at construction
    char padding_between_frequent[64];

    uint32_t upperBound_;

    // Eliminate false sharing between drawers
    char padding_between_drawers[64];
};
```

Cupboard Timings (False Sharing Fixed): Negligible Work Type

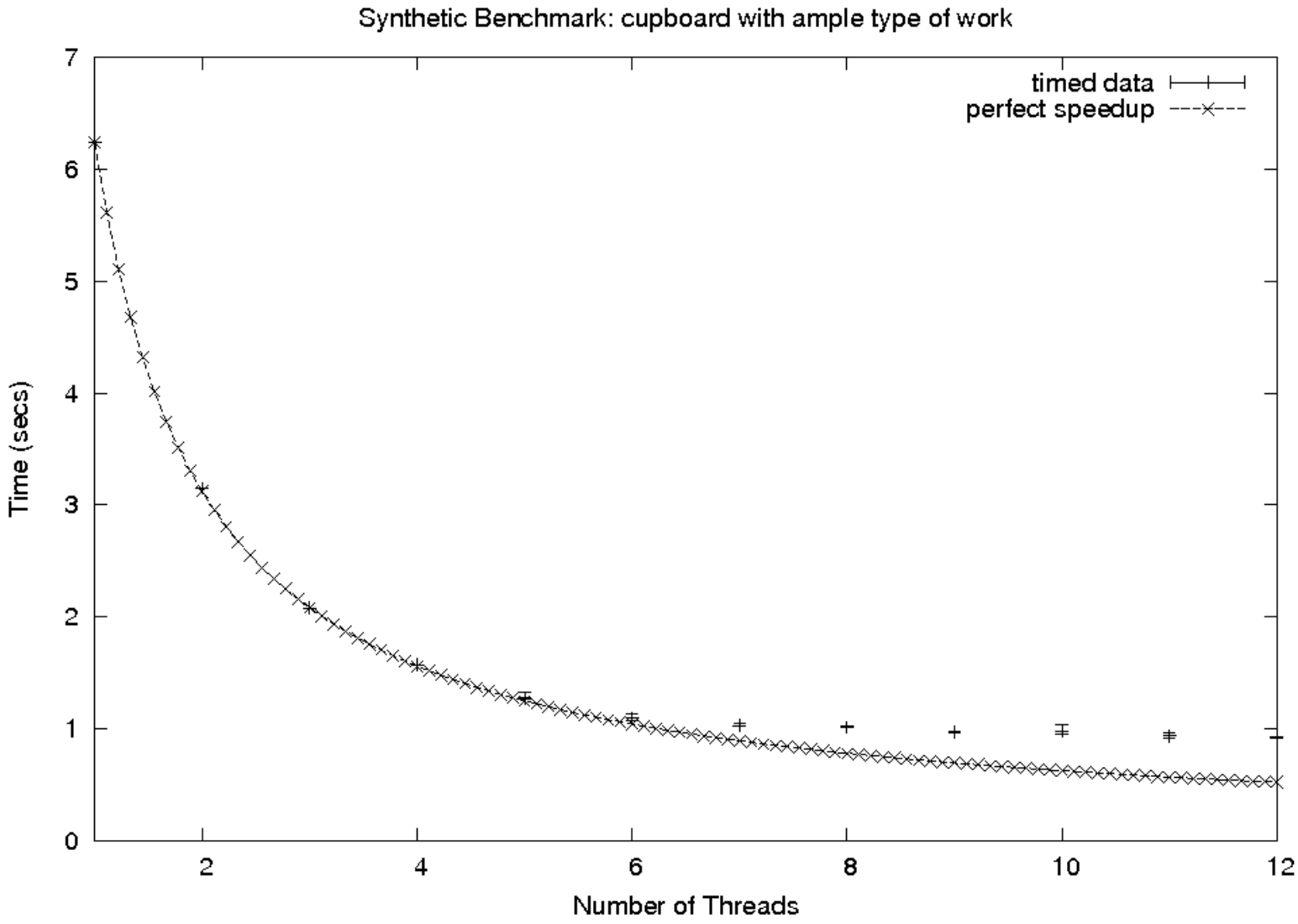


Cupboard Timings (False Sharing Fixed): Some Work Type



Rincon Research Corporation

Cupboard Timings (False Sharing Fixed): Ample Work Type



Hyper-Threading (1)

- AKA Simultaneous Multithreading

- Ideally, Hyper-Threading (simplified) allows you to reuse functional units in the CPU
  - When doing “floating point” work, integer ALU is not being used
  - When doing “Integer” work, floating point ALU is not being used
    - Hyper-Threading is a hardware technique (supported by the processor) for allowing two threads to proceed simultaneously
      - 1 hyper-thread does floating point work
      - 1 hyper-thread does integer work
        - » Net gain! 2x!

- That’s why a 6-CPU machine looks like a 12-CPU machine
  - Operating System exposes hyper-threads as “real threads”

Hyper-Threading (2)

- Unfortunately, Hyper-Threading can be problematic
  - For much scientific computing, all work is floating point
    - Integer unit sits idle
  - CAF: all floating point work once we throw a bunch of threads at it

- WORST Case for Scientific Computation:
  - Scheduler schedules 2 threads per CPU
    - expecting one thread to use the floating point unit, and the other thread to use the integer unit
    - BUT ALL Floating Point work!
      - So, one thread sits idle waiting for FP unit to become available
  - Naïve scheduler may cut performance by 2x

Hyper-Threading (3)

- In real code, the scheduler decides where to run threads, and sometimes things work out, sometimes they don't
  - Scheduler irregularities
- Whole purpose of Bag is to help mitigate scheduler irregularities
- If we consider the 6 CPU machine with Hyper-Threading to be a 6-FPU machine ONLY
  - For the purpose of our Scientific Computing, only HAS 6 CPUs
    - We do achieve perfect speedup
- Another sanity check on Hyper-Threading:
  - *Evaluating the Impact of Simultaneous Multithreading on Network Servers using Real Hardware* from ACM SIGMETRICS, 2005
    - Y. Ruan, V.S. Pai, E. Nahum, J.M. Tracey

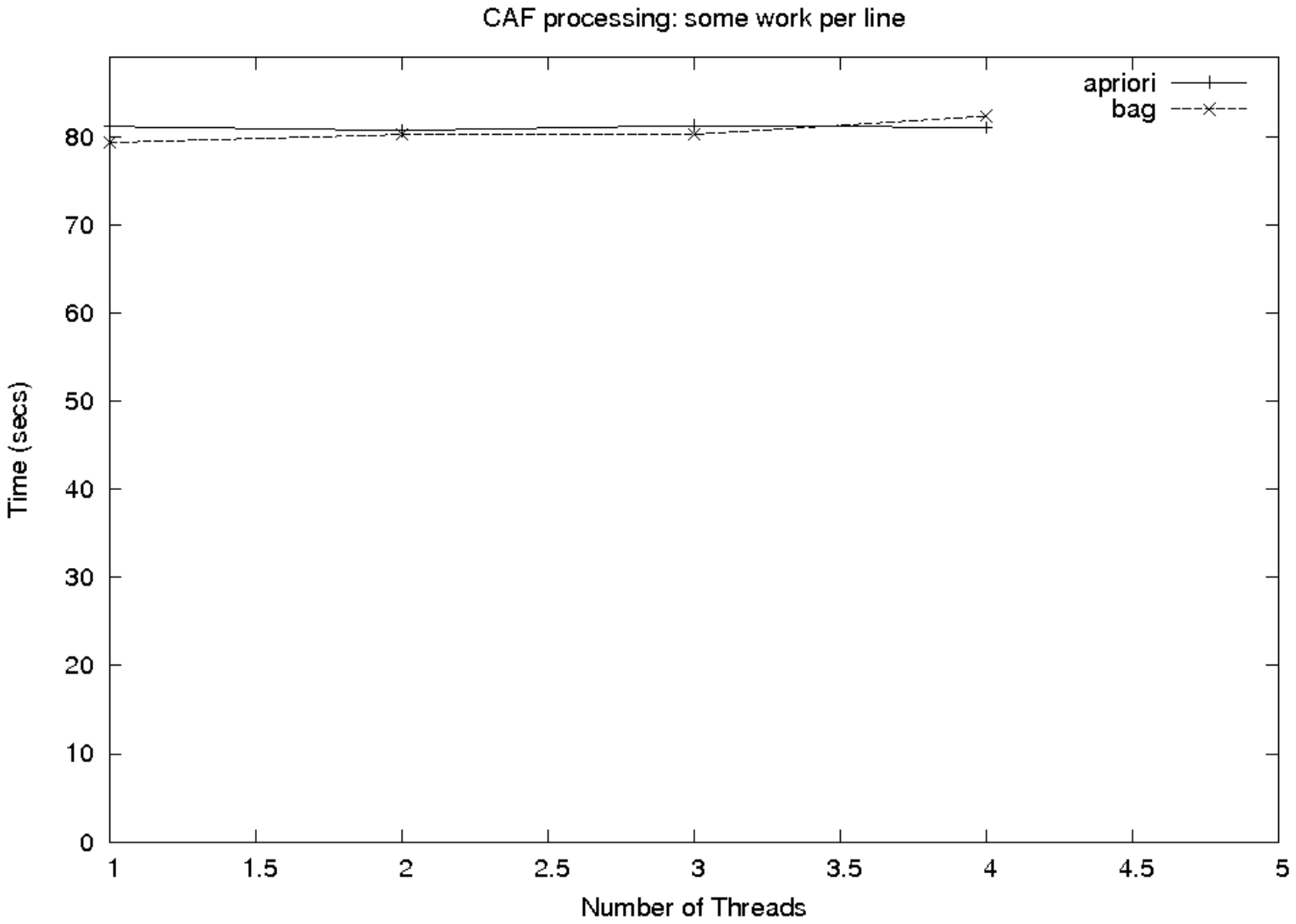
Back to the CAF

- The original purpose of the bag was to make CAF processing “faster”
  - Mitigate scheduler irregularities
  - Dole out work dynamically so “over-zealous” worker threads can pick up the slack of “slacker” worker threads

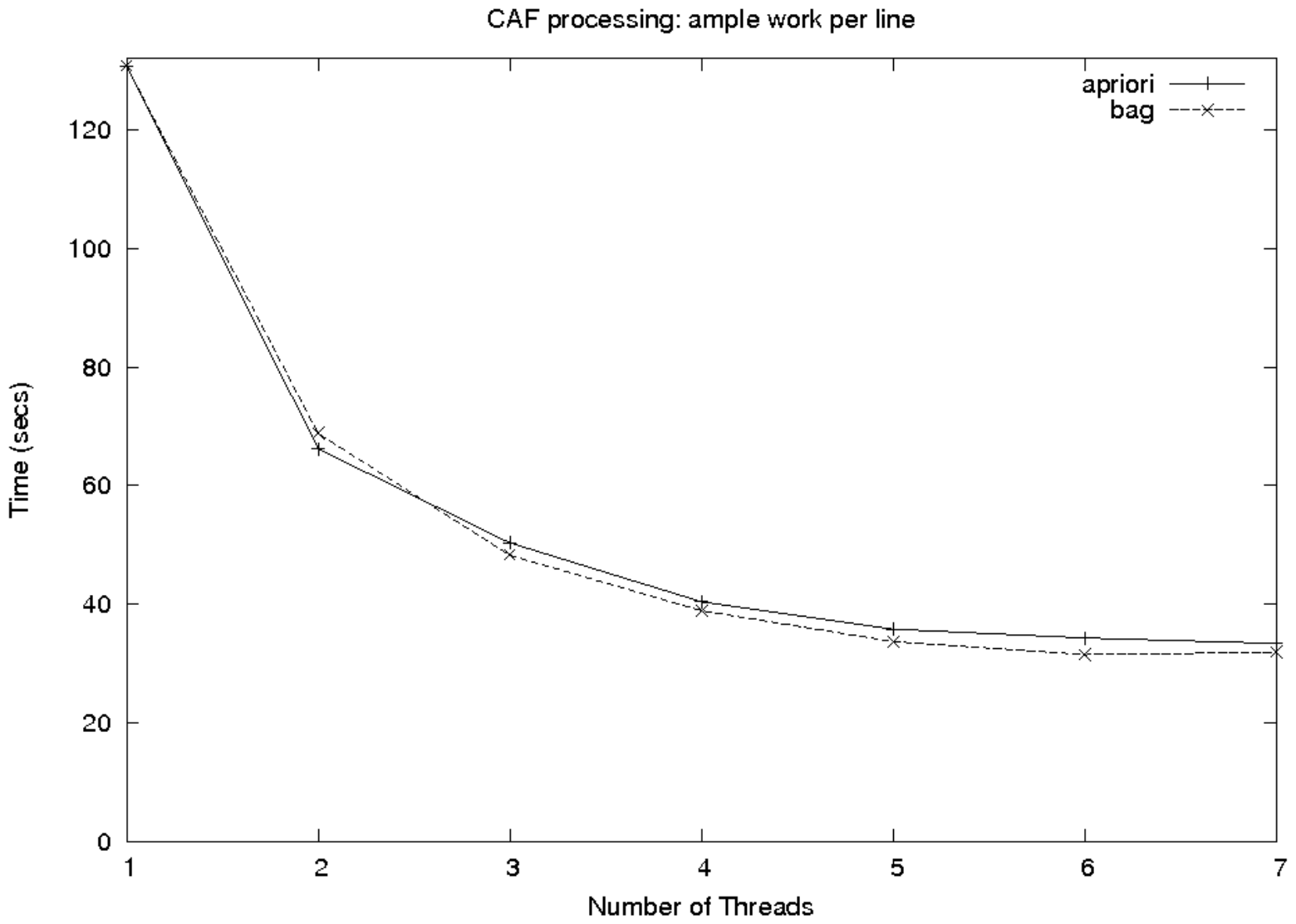
- In real CAF code
  - “negligible” doesn’t make too much sense (FFT per CAF line)
  - “some” work types were not historically a problem
  - “ample” work type was the only work type that seemed to suffer from irregular completion times
    - longer running work is more likely to suffer the effects of poor scheduling due to hyper-threading, other applications, etc.

GOAL: (1) Speed up CAF in “ample” work case
(2) and NOT hinder other work types

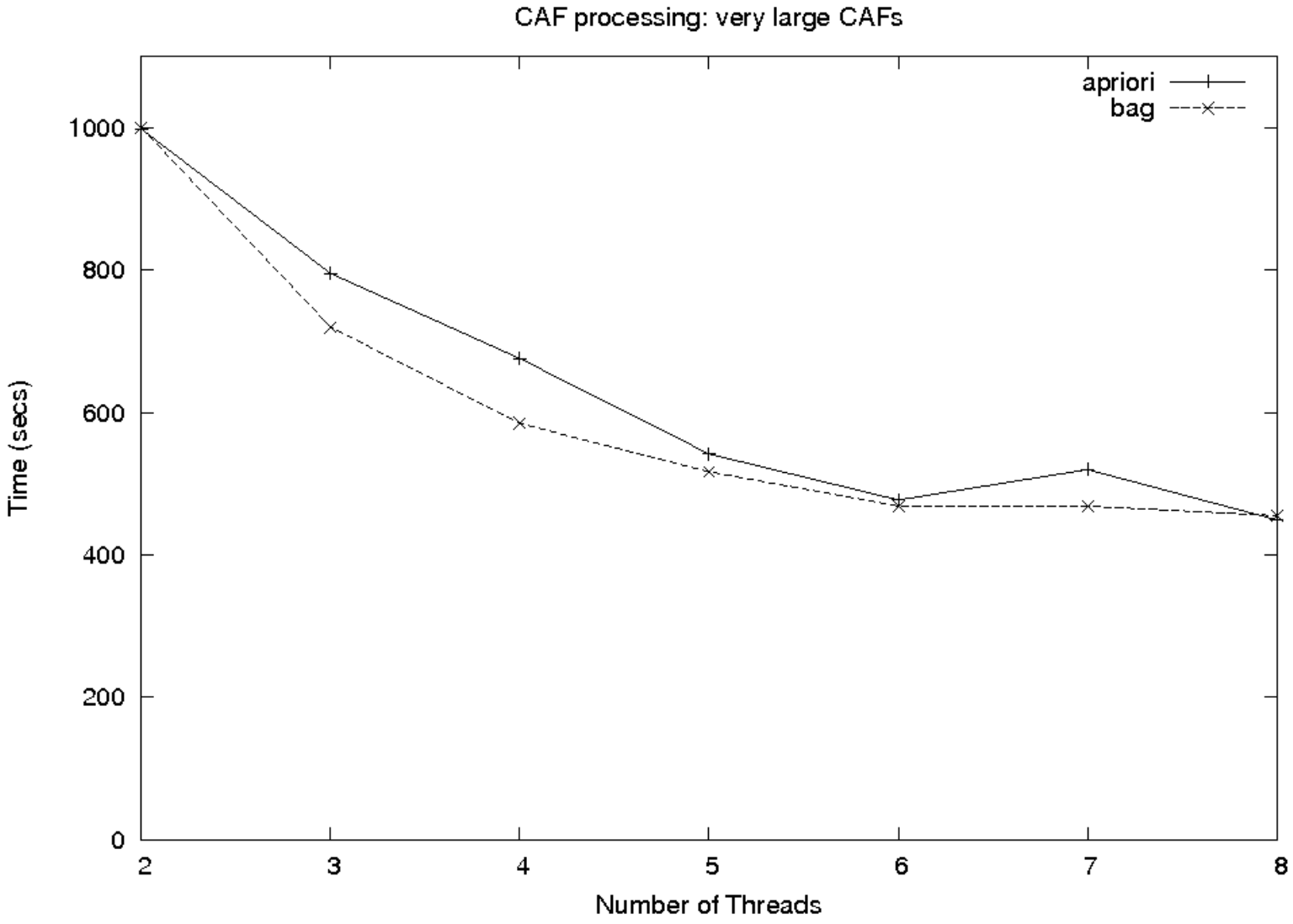
CAF Timing: Some



CAF Timings: Ample Time



CAF Timings: Ample (extra) Work



CAF Timings Results

- Bag did not hinder performance
  - "some" work type
- Bag helped performance
  - "ample"
    - maybe 3%-5% faster
  - bigger "ample" work types
    - 15-20% faster

- All threads "finished at the same time" when computing the CAF
  - Achieves part of the original purpose, use whole machine

No perfect speedup for CAFs?

- CAFs inherently break cache
  - Very large FFTs don't sit in cache
  - Stride across memory, taxing memory subsystem to its limits
    - Direct correlation to front-side bus speedup and CAF speedup
- Perfect speedup is hard to achieve for CAFs
  - Memory system being hit hard by all threads
    - "collateral damage" of other threads is clogging the memory subsystem of the machine

Conclusion

- There are different types of bags for different applications
  - Drawer: useful in most situations
    - Simple and fast
    - Easy interface
    - Built on single C++11 atomic primitive (also can use GNU...)
  - Cupboard : better for high-speed extractions
    - Built on Drawer
    - Slightly more complex interface

- Real World Numbers (CAF):
  - The bag is at least as good as a priori division of work (some)
  - The bag is better than a priori division of work (ample work type)

- The bag helps mitigate scheduling irregularities (Hyper-Threads, etc).