Fine Grained Coordinated Parallelism in a Real World Application

Mohammad Rezaei, PhD

June 2012

Goldman Sachs

Outline

- Types of parallelism
- Algorithm description
- Why fine grained parallelism?
- “Concurrency is hard”… no, it’s different
- A new concurrent map
- Concurrent put benchmark
- JDK API used in the implementation
- Concurrency principles used in the implementation
- Future plans
- Conclusion

Types of Parallelism

- SIMD: Single Instruction, Multiple Data: not the subject of this talk
- Distributed Parallelism: not the subject of this talk
- MIMD: Multiple Instruction, Multiple Data (one process)
  - Coarse grained MIMD: each thread does something different and is not coordinated with the other threads. Already in wide use.
    - E.g. Tomcat's HTTP threads.
    - Not the subject of this talk
  - Fine grained MIMD: all threads are working together to compute the result
    - Typically the threads are running the same algorithm over a subset of the data.
    - This is very different from SIMD: threads running the same algorithm do not have to be in lock step at each instruction.
    - Works well with existing abstractions (e.g. OOP).
    - Works well with complex business logic.
    - Only necessary when a single user query time is too long, but some of the techniques involved here can improve multi-user performance as well
    - Lock free algorithms can be very effective: the subject of this talk ☺

Algorithm: Aggregation

```
Map<Object, MutableDouble> map = new HashMap();
for(Trade trade: tradeList)
{
    Object key = getKey(trade);
    MutableDouble val = map.get(key);
    if (val == null)
    {
        val = new MutableDouble();
        map.put(key, val);
    }
    val.increment(tran.getTradeQty());
}
```

- Above is an outline, not production code.
- getKey() can involve significant business logic. It also depends on user input.
- MutableDouble is also an example, not production code.
- We typically use a mutable key until we have to store the key in the map (not shown).

Goldman Sachs

Parallelization Strategies

- Two ways to parallelize this algorithm
  - Use a separate map for each thread and at the end combine the maps from all threads. Beware Amdahl's law.
  - Use a shared map for all threads. No extra work to do at the end.
    - Ensure correctness via locks or lock-free structures.

```
Map<Object, MutableDouble> map = new ConcurrentHashMap();
for(Trade trade: tradeList)
{
    Object key = getKey(trade);
    MutableDouble val = map.getIfAbsentPut(key, mutableDoubleFactory);
    val.increment(tran.getTradeQty()); // increment must be thread safe
}
```

- Which algorithm is best depends on getKey() (the input data)!
  - Small collapse factor (e.g. 10 million -> 4 million). Must use a shared map.
  - Large collapse factor (e.g. 10 million -> 100). Must use separate maps.
- Our business logic requires two separate aggregation steps. The first step always has a small collapse factor. The second step could have either!

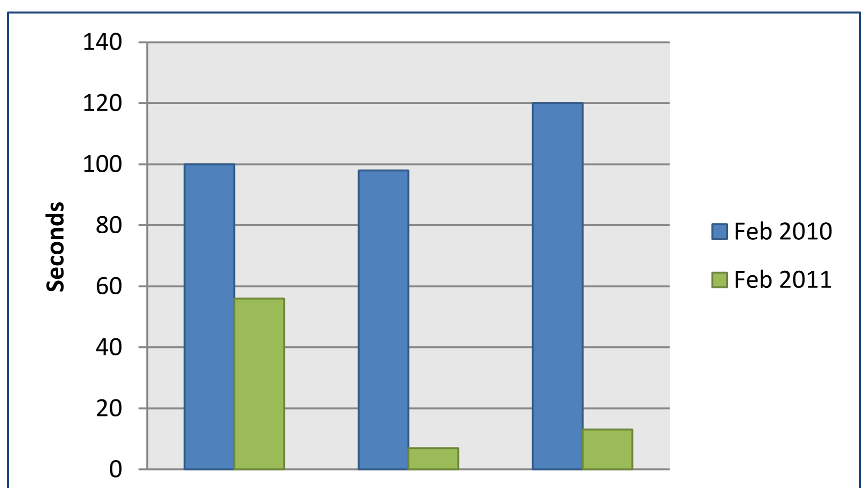
Why not …

- Why not distributed parallelism (e.g. Hadoop)?
  - A distributed algorithm would be restricted to just the first type of parallelization (separate maps with an extra combine step).
  - Without knowing the key ahead of time, we have to incur a lot of IO to distribute the data for each query.
  - Amdahl’s law seriously restricts the scaling because of the extra merge step.
  - The object domain is very far from flat. The domain consists of several hundred classes arranged in a complex object graph.
    - Distribution would have too much repeated reference data.
    - Data changes frequently, which makes keeping repeated data consistent difficult.
  - This algorithm, while important, is just one step in a larger computation. Distributing other parts is at best inconvenient and mostly useless.
- Why not an actor/immutable/functional approach?
  - Without shared memory, final merge step will dominate and limit scaling.
  - Copying memory over and over is usually too slow for fine grained MIMD.
    - “Latency from cache misses is quickly becoming the dominating factor in today's software performance“
    - "Too much object creation blows the cache.“
    - 64-bit CPU operation: 1ns latency, 20pJ energy.
    - Local memory: 100ns latency, 20nj energy (100x slower, 1000x more energy)

“Concurrency is Hard”… No it’s different!

- Shared state concurrency is harder, but it’s not too hard.
- A lot of people claim it’s too hard. They usually have something to sell ☹
- Moh’s perspective:
  - Fear is not the right approach. Spreading fear is even worse.
  - It’s a different mind set, so we have to take a step back and re-learn a few things.
  - Use different testing techniques for concurrent code.
  - Interesting observation: lock-free techniques are in many ways easier than ones involving locks!
  - It can make a big difference. We’ve parallelized many parts of our code to see up to 10x improvements.

Actual query times before and after parallelization



A New Concurrent Map

	GS CHM	JDK CHM	CHM V8	NB CHM
Concurrent Resize	✓	✓	✗	✓
Spread Function	Good	Good	Great	None
No Unsafe	✓	✗	✗	✗
Low garbage put	2	0	1	0
Parallel Iterate	✓	✗	✗	✗
No Knobs	✓	✗	✓	✓
Small initial footprint	✓	✗	✓	✓
Iterate while growing	✗	✗	?	✓

NB CHM: from Cliff Click's high_scale_lib.
CHM V8: from JSR 166e.

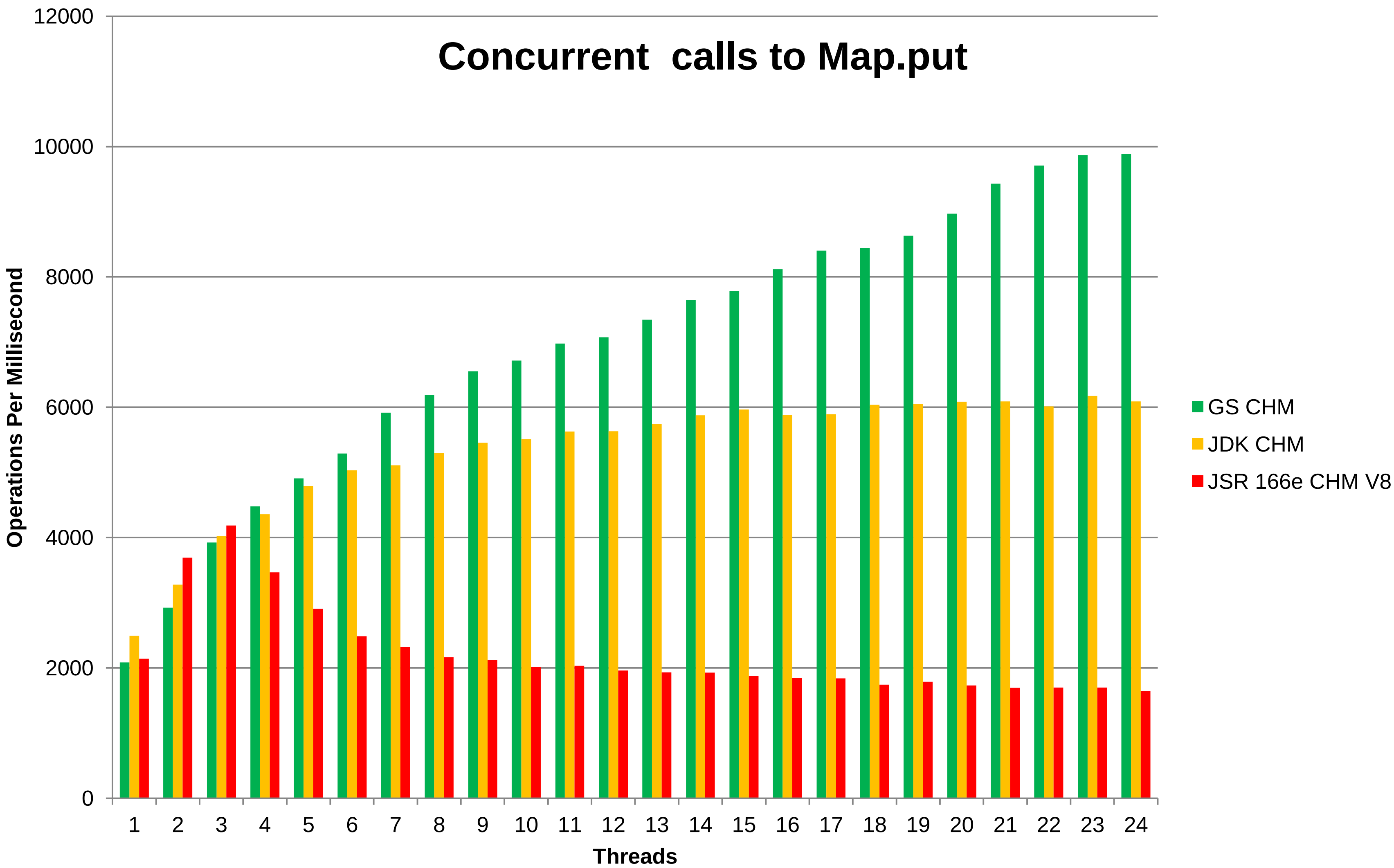
Custom Methods

```
public interface ConcurrentMapEx<K,V> extends ConcurrentMap<K,V>
{
    V getIfAbsentPut(K key, Factory<K, V> factory);

    <P1, P2> V putIfAbsentGetIfPresent(Object key,
        TwoArgumentBlock<K,V,K> keyTransformer,
        ThreeArgumentBlock<P1, P2, K, V> factory, P1 param1,
        P2 param2);

    void putAllInParallel(Map<K, V> map, int chunks,
        Executor executor);

    void parallelForEachEntry(List<TwoArgumentVoidBlock<K, V>>
        blocks, Executor executor);

    void parallelForEachValue(List<VoidBlock<V>> blocks,
        Executor executor);
}
```

Benchmark Caveats

- Benchmarking Java code is hard.
  - GC
  - JIT warm-up
  - Runtime dead code elimination
  - Non-production like data allocation (memory locality)
  - Non-production like megamorphic call sites.
- Benchmarking maps is even harder.
  - There are many methods on a map.
  - In what order/frequency/concurrency should they be called?
- There is a natural bias for an author of a map to have a benchmark that performs best for his/her implementation.
  - This is not the result of cheating or fraud!
    - Often code is written with a particular use case in mind.
    - Often code is tuned to a particular benchmark.
- View all benchmarks with a healthy dose of scepticism.

Concurrent Put Benchmark



Hardware: Westmere 2 processor, 12 core, 24 thread

JDK Low Level Atomic API

- java.util.concurrent.atomic package:
  - AtomicInteger
  - AtomicReferenceArray
  - AtomicIntegerFieldUpdater
  - …
- All have compareAndSet (CAS) method: it either succeeds atomically, or not at all.
- Based on “compare and swap” CPU instructions.
  - Old idea (early 80’s)
  - On Intel processors (486+) cmpxchg instructions
- Atomic*FieldUpdaters are particularly interesting: you can modify a volatile field of an object using CAS. This allows for much better memory utilization. E.g. an int field is just 4 bytes. An AtomicInteger is 16 bytes + another 4 bytes for the reference to it.

Concurrent Map Implementation Details

- Use an AtomicReferenceArray for backing the map.
  - The references in the array are set via CAS.
  - The references are strictly immutable, which simplifies the logic.
  - If a CAS fails, the entire operation is tried again.
- Each bucket has 4 possible states
  - null: empty
  - An Entry: an existing map entry
  - ResizeContainer: this bucket is currently being moved to the next, larger array.
  - RESIZED: this bucket has been moved to the next, larger array.
- Collisions are handled by chaining Entry objects, like HashMap.
- Resize is the most interesting part.
  - There is an extra slot in the array that points to the resized array.
  - Multiple resizes can even happen simultaneously!
  - The thread that gets to allocate the next array does the allocation with a lock.
    - During this lock, other threads typically don't wait.
  - For each bucket
    - Mark it immutable. Move it to the next array. Mark it moved.
    - If a get/put encounters an immutable or moved bucket, it starts helping with the resize.

There is More to Lock Free

- For this algorithm, also need a way of summing doubles without locks.
  - There is no such thing as an AtomicDouble or AtomicDoubleFieldUpdater!
  - However, you can “cast” a double to a long and back via
    - `Double.doubleToRawLongBits`
    - `Double.longBitsToDouble`
  - We have a subclass of AtomicLongArray with appropriate methods for doubles.
- We’ve found uses for atomics in many places. We have two dozen classes that use these to provide lock free operation (sets, pools, caches, etc).
- There is a lot more to the topic that can be covered in an hour.
  - False sharing.
  - Equitable work splitting.
  - Testing concurrent classes.
- Our concurrent map implementation will become open source.
  - GS Collections: https://github.com/goldmansachs/gs-collections
  - Has other interesting implementations: fast/low memory list/map/set.
- Intel’s Haswell instructions are an exciting extension to the existing one. These can make lock free algorithms simpler and more wide spread.

Conclusion

- The future is lock free ☺
- Owning the core data structure in your algorithm has advantages beyond performance. Tuning the API can be just as important as the speed and memory footprint.
- Multi core processors are here to stay. Efficiently coding for these will be more and more necessary. Lock free algorithms are an attractive avenue of exploiting the increasing CPU power.
- Not every algorithm can be well parallelized via distributed computing.

Appendix: Testing Concurrent Classes

- Idea: spawn a large number of threads that will call the class's API in such a way as to cause race conditions.
- Run the test with varying number of threads, especially much more than the number of available cores.
- Arrange the test so that the result will be deterministic, even though the number of threads or the order of operations might not be.
- Always test on a large core machine.
- Problems in lock free code show up quite readily in actual usage: the result will vary from run to run.

Appendix: False Sharing

- Even though AtomicInteger has CAS, the CPU doesn’t set just those 4 bytes.
- CPU’s implement cache coherency via a cache line, which is typically much wider than 4 bytes. Currently it’s 64 bytes.
- If multiple integers are allocated next to each other, they are not independent! This is called false sharing.
- Solution: pad the objects so it’s at least 64 bytes wide.
- Careful: this should only be done for highly critical pieces, as the memory overhead is not justified when allocating large numbers of these objects.
- We’ve chosen <u>not</u> to do this for the size variable in ConcurrentHashMap due to memory overhead.

Appendix: Improving Aggregation With Stochastic Sampling

- How can we handle aggregation that could either be small or large collapse factor?
- Idea: write both algorithms, and chose between them by peeking into the input data.
- Important: the work performed here must be small!
- Use a random sampling and just compute the number of unique keys.
- If the number of keys are much bigger than the number of threads, use a concurrent map. Otherwise, use separate maps and combine.

	Before	After
10 Million -> 100	6.571s	0.693s
4 Million -> 1	4.748s	0.288s

Appendix: AtomicFieldUpdater Example

```
public class MutableConcurrentDouble
{
    private static final AtomicLongFieldUpdater VALUE_UPDATER =
        AtomicLongFieldUpdater.newUpdater(MutableConcurrentDouble.class, "value");

    private volatile long value;

    public MutableConcurrentDouble(double val)
    {
        this.value = Double.doubleToLongBits(val);
    }

    public double getValue()
    {
        return Double.longBitsToDouble(value);
    }

    public void increment(double val)
    {
        while(true)
        {
            long cur = value;
            double v = Double.longBitsToDouble(cur);
            double newVal = v + val;
            if (VALUE_UPDATER.compareAndSet(this, cur, Double.doubleToLongBits(newVal)))
            {
                return;
            }
        }
    }
}
```