# Experimental Shotgun Scalability Analysis

August 1, 2013

## 1 Introduction

In this document, we analyze the bottlenecks of Shotgun's scalability on multicore. Currently we are able to get maximum of about 3-4x speedup on 8 cores. As good tools for analyzing multicore performance do not exist, one has to practically try different hypothesis in order to isolate effects of various factors. This analysis is based on creating an application which does similar computations as Shotgun, but which does not actually compute the "Lasso". For example, in order to study the effect of temporal locality of memory access, we created a version which did same "shoot" 10 or 50 times.

### 1.1 Computation

For our analysis, we studied the scalability of following shooting-like parallel computation:

```
FOR 500 times DO:
 PARALLEL FOR i=1 to n (=220,000)
     pseudo_shoot(i)
```

The function `pseudo_shoot()` does a sparse-dense dot-product of vectors and updates sparsely elements of the dense global vector (length 30,000):

```
def pseudo_shoot(i):
      sparsecolumn = column i of sparse matrix A
      S_i = dotproduct( sparsecolumn, globalvec)
      x_i = softthreshold(S_i, \lambda)
      if (x_i changed)
          globalvec = globalvec + sparsecolumn *  \delta x_i
```

Mathematically (z = `globalvec`, $A_{:,j}$ = column $j$ of A):

$$
\begin{aligned}
S_j &= A_{:,j}^T z + \text{[a constant element for variable j]} \\
x_j^* &= (|S_j| - \lambda)_+ \\
z &= z + (x_j^* - x_j) A_{:,j}
\end{aligned}
$$

In the dataset we used ("finance1000"), the columns are mostly very sparse. Sparsity of matrix A is approximately 0.05%. Each column has average 17 non-zeros. Therefore, the number of operations in the shoot are:

- average of 17 multiplications and additions (first dot product)
- simple comparison
- if changed: average of 17 multiplications and additions (change in the globalvec)

The high sparsity is an important feature of the problem: parallelizing dense operations is typically much easier due to higher level of temporal locality and less challenges such as false sharing (see below).

### 1.2 Implementation

With C++ and Cilk++ for parallelism. Tests were run on machine1 (AMD Opteron, 2x4 cores) and a 8-core Nehalem computer (Amazon EC2, HPC instance).

### 1.3 About timings

we present only "speedups". Speedup is computed as time on sequential computer / time on parallel computer. The sequential runs took 30 - 500 seconds, which should be long enough to hide most of the overhead of starting threads. Data loading etc. has not been computed, only the time used in the `parallel for`.

### 1.4 Acknowledgements

*anonymized*

## 2 Analysis by Factors

### 2.1 Atomicity

Vector `globalvec` is shared between processors. Updates to it should be atomic, since there is possibility of write-races. After trying out with several methods for protecting the vector, we ended up using *compare-and-swap* (CAS) atomic instructions. The idea of CAS instructions is that program offers a new value to a memory address, and it is written *if the previous value matches* the value program used for computing the new value. Program will loop until the instruction succeeds:

```
while(not success)
   newvalue  = prev + delta
   valuewas = compare_and_swap(&address, prev, newvalue)
   if prev = valuewas then
```

```
      success = true
   else prev = valuewas
 end
```

To measure the cost of these atomic instructions, we ran the test program with atomic instructions and without any protection (labelled as "unsafe") to the vector values. Note that the baseline 1-cpu version did not use atomic instructions. Scalability was as follows:

| *Architecture: AMD* | 2 cpus | 4 cpus | 8 cpus |
|---|---|---|---|
| Atomic | 0.7 | 0.9 | 1.3 |
| Unsafe | 1.4 | 1.8 | 2.0 |

| *Architecture: Nehalem* | 2 cpus | 4 cpus | 8 cpus |
|---|---|---|---|
| Atomic | 1.1 | 1.8 | 2.8 |
| Unsafe | 1.8 | 2.8 | 3.5 |

**Verdict**

Atomic instructions have tremendous cost to the scalability. On 2 cpus, we experience significant *slow-down* on AMD. Nehalem scales much better. However, the scalability even of the unsafe version is very disappointing.

## 2.2 False sharing

Cache line of a typical modern 64-bit processor is 64 bytes = 8 words. This can easily result in *false sharing*: if element 3 of `globalvec` is updated, all other cores have to *flush* from its cache elements 0 to 7 of the vector (assuming we use doubles). To prevent this, we can pad the vector with 7 unused words between each element. That is $v[3] \mapsto v[24]$. In addition, vector $x$ can be padded similarly.

**Note: All following tests have padding enabled.**

**Verdict**

In our experiments, this trick did not have significant effect on the performance. we believe that the total memory traffic is so high, that the values of `globalvec` may not fit into cache anyway for any long period of time. Indeed, as the vector contains 30K elements, it will take 234kb of memory unpadded. As padded, the size grows to 1.8mb. Level 2 cache of *AMD* Opteron 2348 - which was used - is 512 kb per core. Level 3 cache is 6 megabytes. Now, vector $x$ is 220K long, and thus takes about 7 times more memory. we would expect these vectors to be washed out of cache frequently, since the access patterns are sparse. Therefore, optimizing false sharing has little effect. [However, as all following experiments had padding of arrays enabled, we did not systematically its interactions with the other factors.]

## 2.3 Locality - partitioning

If we could partition the variables so, that cores would only seldom touch each other's parts of vector `globalvec`, we could hope to get much better temporal and spatial cache locality.

This factor was simulated as follows: each core was assigned an id. Cores updated and read only such elements of `globalvec`with index modulo number of threads = worker id. That is, worker id 5 on 8 cpus only read and wrote elements with index $5, 13, 21, ....$ It is good to note that this is a sparse access pattern (which is realistic). This partitioning is perfect, i.e no core will write to same addresses during one parallel subiteration. Subiteration consists of "shooting" 1000 variables by each core. (See last subsection for the effect of the length of a subiteration).

| *Architecture: AMD* | 2 cpus | 4 cpus | 8 cpus |
|---|---|---|---|
| No partitioning - unsafe | 1.4 | 1.8 | 2.0 |
| Partitioning - unsafe | 1.5 | 2.8 | 4.9 |
| No partitioning - atomic | 0.7 | 0.9 | 1.3 |
| Partitioning - atomic | 1.1 | 2.1 | 3.8 |

| *Architecture: Nehalem* | 2 cpus | 4 cpus | 8 cpus |
|---|---|---|---|
| No partitioning - unsafe | 1.8 | 2.8 | 3.5 |
| Partitioning - unsafe | 1.6 | 2.9 | 4.8 |
| No partitioning - atomic | 1.1 | 1.8 | 2.8 |
| Partitioning - atomic | 1.2 | 2.2 | 3.8 |

**Verdict**

Partitioning has a very large effect on both architectures (although on Nehalem the effect can be seen only at 8 cpus) . It is important to note, that in practice we could not get a perfect partitioning. Also, partitioning will have a cost involved. The plan is to try use MANIS-algorithm (Maximal Approximate Nearly Independent Sets) by Kanat and Guy to create a approximate graph coloring. According to these results, there is hope of achieving significant performance benefits on sparse problems. With partitioning, even the atomic algorithm can obtain speedups on only 2 cores (albeit not great).

## 2.4 Temporal locality

To study whether we actually saturate the memory bus by loading the columns of matrix A, we studied how improving temporal locality would affect speedup. This was simply done by running the `pseudoshoot()` function in a loop for 10 or 50 times. We can expect the data to remain in L1 or L2 cache during the loop.

| *Architecture: AMD* | 2 cpus | 4 cpus | 8 cpus |
|---|---|---|---|
| 1 shoot (normal) - unsafe | 1.4 | 1.8 | 2.0 |
| 10 shoots - unsafe | 1.7 | 2.4 | 3.2 |
| 50 shoots -unsafe | 1.7 | 2.5 | 3.4 |

| *Architecture: Nehalem* | 2 cpus | 4 cpus | 8 cpus |
|---|---|---|---|
| 1 shoot (normal) - unsafe | 1.8 | 2.8 | 3.5 |
| 10 shoots - unsafe | 1.8 | 2.6 | 3.8 |

In addition, we studied the combined effect of temporal locality and partitioning.

| *Architecture: AMD* | 2 cpus | 4 cpus | 8 cpus |
|---|---|---|---|
| 1 shoot (normal) - unsafe | 1.4 | 1.8 | 2.0 |
| 1 shoot (normal) + partitioned | 1.5 | 2.8 | 4.9 |
| 10 shoots - unsafe | 1.7 | 2.4 | 3.2 |
| 10 shoots + partitioned | 1.2 | 2.4 | 4.1 |
| 50 shoots -unsafe | 1.7 | 2.5 | 3.4 |
| 50 shoots + partitioned | 1.2 | 2.3 | 4.0 |

| *Architecture: Nehalem* | 2 cpus | 4 cpus | 8 cpus |
|---|---|---|---|
| 1 shoot (normal) - unsafe | 1.8 | 2.8 | 3.5 |
| 1 shoot (normal) + partitioned | 1.6 | 2.9 | 4.8 |
| 10 shoots - unsafe | 1.8 | 2.6 | 3.8 |
| 10 shoots + partitioned | 1.4 | 2.6 | x |
| 50 shoots -unsafe | x | x | x |
| 50 shoots + partitioned | x | x | x |

**Verdict**

Temporal locality alone clearly improves scalability, and suggests that our problem might be *saturating the memory bus*. Our ratio of computation to memory access is very low, and there is almost no temporal locality.

However, it is interesting that relatively speaking an algorithm which repeats the shoot 10 or 50 times and uses partitioning scales worse than algorithm which just has good partitioning. This is surprising, as one would assume the effects be independent of each other. One hypothesis is that the sequential algorithm benefits relatively more from temporal locality than parallel version since it has the whole L3 cache and memory bus in its disposal.

## 2.5 Ratio of computation and memory access

To study more the effect of our high use of memory bandwidth, we modified the algorithm to compute a $\sin(x)$ - function for each element in the first loop of `pseudoshoot()`. This is of course just an arbitrary algorithm, but make the algorithm execute much slower. And indeed, scalability improves dramatically.

| *Architecture: AMD* | 2 cpus | 4 cpus | 8 cpus |
|---|---|---|---|
| 1 shoot (normal) - partitioned - unsafe | 1.5 | 2.8 | 4.9 |
| sin() - partitioned - unsafe | 1.8 | 3.5 | 6.1 |
| 10 shoots with sin() - partitioned -unsafe | 2.0 | 3.8 | 6.6 |

| *Architecture: Nehalem* | 2 cpus | 4 cpus | 8 cpus |
|---|---|---|---|
| 1 shoot (normal) + partitioned | 1.6 | 2.9 | 4.8 |
| sin() - partitioned - unsafe | 2.2 | 3.5 | 6.0 |
| 10 shoots with sin() - partitioned -unsafe | 2.0 | 3.8 | 6.5 |

**Verdict**

This experiment shows very clearly that our algorithm is memory-bound. By replacing the main computation with a much heavier mathematical computation, we achieve almost linear scalability. Also, the scalability on AMD and Nehalem is equivalent, as Nehalem does not benefit from the faster memory system in relative sense.

### 2.6 Workset size = granularity of parallelism

As a sidenote, we also studied the performance of CILK++ as a parallel scheduler. The algorithm works by allocating each cpu (thread) a set of variables (workset) to optimize a time. There is a tradeoff: smaller workset size improves work balance, but bigger worksets have less synchronization between threads. There is clearly a sweet spot around of 1000-3000 variables per thread a time:

| Workset | Shoots per second on 8 cpus (unsafe) |
|---|---|
| 50 | 3.9 mil |
| 200 | 4.5 mil |
| 500 | 4.4 mil |
| 1000 | 4.7 mil |
| 2000 | 4.85 mil |
| 2700 | 5.4 mil |
| 5000 | 5.4 mil |
| 10000 | 5.3 mil (on some tasks, much slower) |

## 3 Summary and Conclusions

This report presented results of a numerous experiments which tried to isolate factors affecting the scalability of Shotgun. Main priority was to prove the hypothesis that Shotgun is *memory bound*, since its ratio of computation to memory access is low.

And indeed, by increasing artificially the amount of computation per memory access, we can achieve almost linear scalability. This suggests that Shotgun is able to saturate the memory system of the computer with only two parallel cores, and can enjoy only limited benefits from added parallel computing capability. However, as good partitioning of the variables improved scalability dramatically, this suggests that large amount of the memory bandwidth is used for cache coherence protocol. By eliminating or reducing communication between cores, more memory bandwidth is available for accessing the data.

Already known was the large penalty of using atomic instructions. However, good partitioning can help with this as well, and allows us to employ coarser locking for the shared data.

As a disclaimer, this kind of study is challenging since there are many variables to tune. It is not possible to try all combinations in a short time, and important interactions of performance factors might not be discovered.