Supplement: Empirical Shotgun Scalability Analysis

Updated April 22, 2014

Abstract

In this supplement, we analyze the system-level bottlenecks of `Shotgun`'s scalability on multicore. Currently, we are able to get maximum of about $3-4\times$ speedup on 8 cores.

To analyze multicore performance, we run a series of tests which try to isolate effects of various factors. Our analysis indicates that `Shotgun` is held back by memory access, but we give suggestions for future improvements to help approach linear speedups.

Keywords: Lasso, sparse logistic regression, parallel optimization, coordinate descent, multicore

1. Introduction

We first explain the simplified Lasso problem, implementation, and timing methodology we used for our tests.

1.1 Computation

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

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

The function `pseudo_cd_update()` simulates one coordinate update. It does a sparse-dense dot-product of vectors and updates sparsely elements of the dense global vector (length 30,000):

```
def pseudo_cd_update(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
```

The global vector corresponds to the residual vector shared by all compute cores. Mathematically (z = `globalvec`, $A_{:,j}$ = column j of A):

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

In the dataset we used ("finance1000" from Kogan et al. (2009)), the columns are mostly very sparse. The sparsity (percent of non-zero elements) of the matrix A is approximately 0.05%. Each column has an average of 17 non-zeros. Therefore, the number of operations in the coordinate update 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 a higher level of temporal locality and fewer challenges such as false sharing (described below).

1.2 Implementation

We used C++ and Cilk++ for parallelism. Tests were run on two machines: the first was an AMD Opteron 2348 with 2×4 cores, and the second was an 8-core Nehalem computer (Amazon EC2, HPC instance).

1.3 About timings

We present "speedup," which is computed as the ratio of the running time on a sequential computer versus the running time on a parallel computer. The sequential runs took 30–500 seconds, which should be long enough to hide most of the overhead of starting threads. The time for loading data was excluded; we only include the time used in the `parallel for`.

2. Analysis by Factors

We present our results, organized according to the hypothesis or problem factor being isolated and tested. **Note**: All tests except for the atomicity tests (Section 2.1) have padding enabled.

2.1 Atomicity

The residual vector `globalvec` is shared between processors. Updates to it should technically be atomic since there is the possibility of write-races. After testing several methods for protecting the vector, we chose *compare-and-swap* (CAS) atomic instructions. In a CAS instruction, the program attempts to write a new value to a memory address, but that value is written *only if the current value matches* the value which the program originally used to compute the new value. The program will loop until the CAS instruction succeeds.

Architecture	Test	Speedup		
		2 cpus	4 cpus	8 cpus
AMD	Atomic	0.7	0.9	1.3
	Unsafe	1.4	1.8	2.0
Nehalem	Atomic	1.1	1.8	2.8
	Unsafe	1.8	2.8	3.5

Figure 1: **Atomicity.** Speedups, comparing atomic and unsafe updates for the shared residual vector. Atomic operations cause considerable slowdown, but speedups are far from linear even without atomic operations.

```
while(not success)
   newvalue  = prev + delta
   orig_value = compare_and_swap(&address, prev, newvalue)
   if prev = orig_value then
       success = true
   else prev = orig_value
 end
```

Above, the `compare_and_swap()` operation returns the original value in order to notify the program if the CAS instruction succeeded.

To measure the cost of these atomic instructions, we ran the test program with atomic instructions ("Atomic") and without any protection ("Unsafe") for the vector values. Note that the baseline 1-core version did not use atomic instructions. Fig. 1 summarizes the results.

Verdict

Atomic instructions have tremendous cost to the scalability. On 2 cpus, we experience significant *slow-down* on the AMD machine. The Nehalem machine scales much better. However, the speedups for even the unsafe version are far from linear.

2.2 False sharing

The 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* elements 0 to 7 of the vector from their cache (assuming we use doubles). To prevent this, we can pad the vector with 7 unused words between each element. (This means that $v[3] \mapsto v[24]$.) The vector x can be padded similarly.

In our experiments, this trick did not have a significant effect on the performance. We believe that the total memory traffic is so high that the elements of `globalvec` do not remain in the cache for long; they are quickly replaced as other elements of `globalvec` are requested. Indeed, as the vector contains 30K elements, it will take 240KB of memory unpadded. With padding, the size grows to 1.9MB. The vector x is 220K long, and thus takes about 7 times more memory than `globalvec`. For the *AMD* Opteron 2348, the

Architecture	Test	Speedup		
		2 cpus	4 cpus	8 cpus
AMD	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
Nehalem	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

Figure 2: **Locality: Partitioning Coordinates** Improving speedups by partitioning coordinates to reduce contention on the shared residual vector `globalvec`, as described in Section 2.3.

Level 2 cache has 512KB per core, and the Level 3 cache is 6MB. We would thus expect the vectors to be flushed out of the cache frequently, especially since the access patterns are sparse. Therefore, optimizing false sharing has little effect. However, we note that, since all following experiments had padding of arrays enabled, we did not systematically its interactions with the other factors.

VERDICT

False sharing does not seem to affect performance significantly, likely because the algorithm has a sparse access pattern to the large vectors `globalvec` and x.

2.3 Locality: Partitioning Coordinates

Our atomicity and false sharing discussions hinted at a major issue with the algorithm: Cores need to access many parts of the data in a sparse and nearly random pattern, so they risk interfering with each other via overlapping memory access. If we could partition the coordinates so that cores seldom touch each other's parts of the residual vector `globalvec`, we could potentially 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. E.g., worker ID 5 on 8 cpus only read and wrote elements with indices $5, 13, 21, ...$

This access pattern is sparse (which is realistic); however, this partitioning is perfect, i.e., no core will write to same address during one parallel sub-iteration. In practice, we could not expect to find a perfect partitioning, but clever pre-processing of coordinates (features) might reduce collisions in accessing the shared residual vector. The computational cost of partitioning will be a major restriction on any partitioning method used. One efficient method which might be helpful is the Maximal Approximate Nearly Independent Sets (MANIS) algorithm (Blelloch et al., 2011) to create an approximate graph coloring.

Fig. 2 summarizes the speedup results. Partitioning significantly improves the speedups. With partitioning, even the atomic algorithm can obtain speedups on only 2 cores.

Verdict

Partitioning has a very large effect on both architectures (especially on AMD). There is hope of achieving significant performance benefits on sparse problems by careful partitioning. However, more work is required to identify or develop partitioning algorithms which are fast and find good parititions for sparse datasets.

2.4 Temporal Locality

Very little computation is done for each datum loaded, so we posited that memory access could be a bottleneck. To study whether we saturate the memory bus by loading the columns of matrix A, we studied how improving temporal locality could affect speedup. Improved temporal locality would mean reusing data more often for future computations, increasing the total amount of computation per datum loaded. We tested this factor by running the `pseudo_cd_update()` function in a loop for more times on the same data. We could expect the data to remain in L1 or L2 cache during the loop. Fig. 3 summarizes the results, including the complication of partitioning (Section 2.3); all updates are unsafe (not atomic).

Verdict

Without partitioning, temporal locality improves scalability; this result 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 in practice.

Interestingly, with partitioning, increasing temporal locality by repeating the update 10 or 50 times leads to worse speedups than updating the coordinate a single time. It is surprising that these two effects—spatial and temporal locality—are not 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 at its disposal.

2.5 Ratio of Computation and Memory Access

Section 2.4 studied the high memory bandwidth requirements. In this section, we study this issue further by doing more computation per update, rather than repeating identical updates. We modified the algorithm to compute a $\sin(x)$ function for each element in the first loop of `pseudo_cd_update()`. This is an arbitrary modification, but it makes the algorithm execute much more slowly because of the expense of computing the $\sin(x)$ function. As expected, scalability improves dramatically, as summarized in Fig. 4. All tests used partitioning and used unsafe (not atomic) updates.

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 speedups. Also, the speedups on AMD and Nehalem are similar, as the Nehalem architecture does not benefit from the faster memory system (in a relative speedup sense).

Increasing Temporal Locality

Architecture	Test	Speedup		
		2 cpus	4 cpus	8 cpus
AMD	1 update (normal)	1.4	1.8	2.0
	10 updates	1.7	2.4	3.2
	50 updates	1.7	2.5	3.4
Nehalem	1 update (normal)	1.8	2.8	3.5
	10 updates	1.8	2.6	3.8

Increasing Temporal Locality + Partitioning

Architecture	Test	Speedup		
		2 cpus	4 cpus	8 cpus
AMD	1 update (normal)	1.4	1.8	2.0
	1 update (normal) + partitioned	1.5	2.8	4.9
	10 updates	1.7	2.4	3.2
	10 updates + partitioned	1.2	2.4	4.1
	50 updates	1.7	2.5	3.4
	50 updates + partitioned	1.2	2.3	4.0
Nehalem	1 update (normal)	1.8	2.8	3.5
	1 update (normal) + partitioned	1.6	2.9	4.8
	10 updates	1.8	2.6	3.8
	10 updates + partitioned	1.4	2.6	x

Figure 3: **Temporal Locality.** We study the effect of improving temporal locality (reusing data for more consecutive computations); in these tables, more updates means more temporal locality. The first table summarizes speedup results, and the second table includes additional results from partitioning to improve locality (Section 2.3). All updates are unsafe (not atomic).

Architecture	Test	Speedup		
		2 cpus	4 cpus	8 cpus
AMD	1 update (normal)	1.5	2.8	4.9
	sin()	1.8	3.5	6.1
	10 updates with sin()	2.0	3.8	6.6
Nehalem	1 update (normal)	1.6	2.9	4.8
	sin()	2.2	3.5	6.0
	10 updates with sin()	2.0	3.8	6.5

Figure 4: **Increasing the ratio of computation to memory access.** The "normal" updates are as in Lasso, while the "sin" updates compute the sine of each element of x to increase computation per datum loaded. Extra computation improves speedups dramatically. All tests used partitioning and used unsafe (not atomic) updates.

Workset	Updates 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)

Figure 5: **Workset Size (Granularity of Parallelism).**

While it is not reasonable to increase computation in this way for Lasso, these results help to explain the somewhat improved speedups for logistic regression (in the main paper), for logistic regression does more computation per datum.

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 coordinates (workset) to optimize on each iteration. There is a tradeoff: a smaller workset size improves work balance, but bigger worksets have less synchronization between threads. The results, summarized in Fig. 5, show a sweet spot around 1000–3000 coordinates per thread.

3. Summary and Conclusions

This report presented results of several experiments which tried to isolate factors affecting the scalability of `Shotgun`. Our main goal was to prove the hypothesis that Shotgun is *memory bound*, since its ratio of computation to memory access is low.

Indeed, by artificially increasing the amount of computation per memory access, we can achieve almost linear speedups. This result suggests that Shotgun is able to saturate the memory system of the computer with only two parallel cores, and can therefore enjoy only limited benefits from added parallel computing capability.

However, our results suggest directions for improving this scaling. Careful partitioning of the coordinates to reduce contention on the shared residual vector improved scalability dramatically, which suggests that a large amount of the memory bandwidth is used for the cache coherence protocol. By eliminating or reducing communication between cores, more memory bandwidth is available for accessing the data. Also, avoiding atomic instructions might improve speedups, although one must take care to limit corruption in the shared residual vector (discussed more in the main paper). Note that good partitioning can help reduce the impact of atomic instructions as well, and paritioning allows us to employ coarser locking for the shared data.

As a disclaimer, this kind of study is challenging since there are many coordinates to tune. It is difficult to try all combinations of factors, so important interactions between performance factors might not be discovered.

References

Guy E. Blelloch, Richard Peng, and Kanat Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *Proc. of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

S. Kogan, D. Levin, B.R. Routledge, J.S. Sagi, and N.A. Smith. Predicting risk from financial reports with regression. In *Human Language Tech.-NAACL*, 2009.