10.3 Paper Review

10.3.1 Cache oblivious algorithms

This paper builds on the I/O discussion we began last week. The motivation driving both of last week's papers was that I/O operations cost many orders of magnitude more than internal memory operations, a constraint that inspired the development of several algorithms that efficiently batch read/write operations in cases that require a lot of storage to external memory (e.g. graph operations on graphs with billions of edges).

Cache-oblivious algorithms by Frigo, Leiserson, Prokop, and Ramachandran from MIT's Laboratory for Computer Science (published 2012) propose a new approach toward designing algorithms that that efficiently modulate between cache and main memory. Rather than limiting themselves to cache-aware algorithms that are manually fine-tuned to the parameters of a given cache (M size of cache and B block size, or length of any line of cache that is written/read from main memory at the same time), the authors propose cache-oblivious algorithms that are provably efficient on any cache size. They assume that the cache is tall $M = \Omega(B^2)$.

The primary mechanism which drives many of the cache-oblivious algorithms is a dynamic partitioning of input matrices to fit in memory combined with divide-and-conquer solving. Typically, the same amount of arithmetic work is performed as in the naïve case (matrix multiplication still requires $O(n^3)$ work), but minimize the I/O operations by placing data contiguously into main memory in such a way that it can be re-used for multiple iterations of the algorithm (e.g. multiplications).

My favorite section of the paper describes funnelsort, a new type of sorting algorithm that is cache-oblivious (mergesort is criticised for being, in the classical case, oblivious to cache misses, and in the cache-optimized case introduced by Aggarwal and Vitter, cache-aware).

Beginning with regular mergesort to describe the difference, each of the two arrays being sorted is scanned once (n items fetched, so $O(n/B)$ cache misses in the worst case). Subsequent levels of merge also have $O(n/B)$ cache misses since the total elements merged at each level remains n. Over $\log n$ levels of merging, this gives a total $O(\frac{n}{B} \log n)$ cache misses. In the more optimal cache-aware multiway-merge sort, the cache complexity is reduced to $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$.

Cache-oblivious funnelsort works similarly but utilizes a k-merger to perform the merge operation. Rather than iteratively merging k sorted sequences, the k-merger begins with smaller $sqrt(k)$ sequences and recursively builds longer runs. To give an example over two levels, $sqrt(k)$ sequences are sorted and channeled into output buffers. Once an output buffer is sufficiently full ($2k^{3/2}$ elements), the k-merger suspends operations on that run, allowing data to remain in the cache and be reused. A final k-merge is performed over the $sqrt(k)$ buffer outputs, and this is the final result. This algorithm is work efficient ($O(n \log n)$), and after a lot of algebra, can be proved to be cache-efficient in accordance with Aggarwal and Vitter's result as well.

The paper ends with a number of rebuttals to anticipated challenges to its fundamental assumptions. This includes a justification for using LRU as a tactic for optimal replacement (it's competitive!) The authors also argue that the two-level cache structure is a reasonable model even for multi-level caches. I want to spend some time on this because, as I was reading the article, this appeared to be one of its major shortcomings in a world where real hardware *does* consist of multi-level caches. The authors argue that a multi-level cache using LRU replacement (a) has the property that each level of cache is a subset of the memory stored in the next higher level of cache (inclusion) and (b) will respond to cache misses in level i cache with a hit in higher levels of cache using a sequence of memory accesses comparable to a single-level cache. This is theoretically convincing but probably not how it practically works out. Although the authors show that LRU can be implemented, if a CPU uses pseudo-LRU, random, or tree-based replacement the assumption about inclusion breaks. Additionally, access costs in terms of latency and bandwidth to different levels of cache are not uniform and not discussed.

The empirical results are definitely compelling, but I would like to see a comparison to cache-aware algorithms as well. Comparing to the naïve implementation shows that the cache-oblivious algorithms work well, but it would also be great to show the comparison to tuned cache-aware algorithms (both on the hardware they were tuned for and that which they were not tuned for), which could be a robust argument in favor of the generality of the authors' method.

Last, it would be interesting to learn whether the algorithms work on SSDs, which experience write amplification and have a limited amount of write cycles before they wear.