6.895 Final Project: Serial and Parallel execution of Funnel Sort

Paul Youn

December 17, 2003

Abstract

The speed of a sorting algorithm is often measured based on the sheer number of calculations required to sort an n element array. However, in practice the sheer number of calcualtions is not the only factor in determining the running time of the sorting algorithm. For the purposes of the paper, we analyze the affect of cache/main memory memory transfers on the "Wall-clock running time" of a sorting algorithm. Analysis covers an implementation of the cache-oblivious, cache-efficient algorithm Funnel sort. Finally, this paper explores parallelizing the Funnel sort algorithm.

We find it difficult to determine the number of memory transfers that Funnel sort uses in execution, but is still outperformed by Quick sort by approximately a factor of 4 when the array to be sorted fits entirely in main memory. In addition, we find that Funnel sort shows moderate promise as a parallel algorithm.

1 Introduction

Many computationally efficient sorting algorithms are not very cache efficient. To analyze cache efficiency, we consider the number of memory transferes from main memory to cache required by a program. Consider 2-way Merge sort, or any other sorting algorithm that recurses in a binary fashion. Let the cache contain M lines, and let L be the number of words that each cache line can hold. At every level in the sort tree, all n elements must at some point enter the cache in order to be processed. Assuming that transfers from main memory to cache are done in blocks of L elements, at every level of the sort tree we incur $O(n/L)$ memory transfers. Note that the height of the sort tree for a 2-way Merge sort is $\log_2 n$. However, when all of the merge inputs are length $M * L/2$, then subsequent lower levels of the sort tree no longer require any memory transfers as the merge can take place entirely in cache. So, for the purposes of cache complexity, the sort tree has height $O(\log_2{(n/(M*L))})$. See figure on next page. Thus, the number of memory transfers, or cache complexity, is $O(\frac{n}{L} * \log_2{(n/(M*L))})$.

In order to better understand the goal of Funnel sort, we must ask what would be the memory transfer behavior of the idea cache-aware sorting algorithm? Consider an $(M-1)$-way Merge sort. In this case, at every level in the sort tree, there is one line available for every input array to be merged, and by assumption one line available for the output array. Again, notice that at every level of the sort tree, all n elements must at some point be transfered into the cache. So, the cache can be viewed as a moving window of the next L elements in each of the $M-1$ input buffers. When all

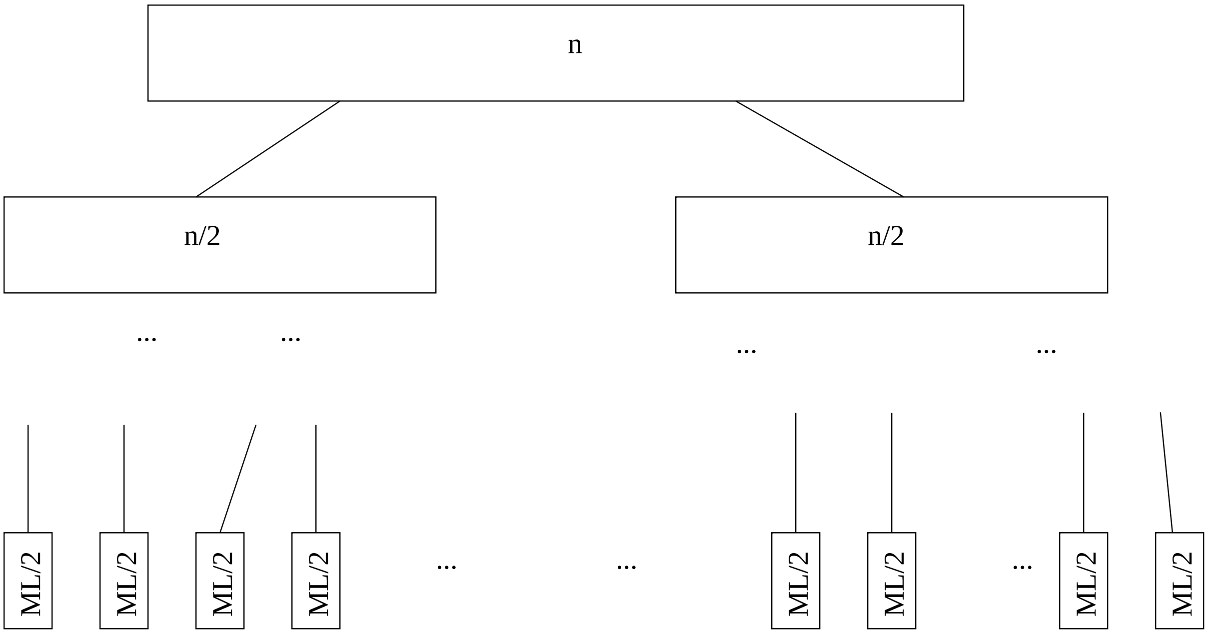

Figure 1: 2-way Merge sort tree of height $log_2(n)$

elements from a single cache line associated with input buffer i have successfully been merged, a cache miss occurs, and the next L elements from that input buffer i are read into cache. Note that if we had more than $M-1$ input buffers, occasionally we would need to read from a buffer that was not in cache, and have to overwrite a still useful cache line. But when we have $M-1$ input buffers, every block of L input elements is transfered into cache once per level of the sort tree, and we again incur n/L memory transfers per level in the sort tree. However, we also notice that our sort tree is now much shorter. By the same reasoning as above, we now have a sort tree height of $O(\log_M(n/L))$ and thus we have cache complexity: $O(n/L * \log_M(n/L))$.

The following section will describe how Funnel sort achieves the same bound on cache complexity as the above described cache-aware algorithm described above, but in a cache-oblivious manner. The paper then continues to report the performance observed from running both the serial and parallel implementation of Funnel sort. Along with results, the data will be analyzed and explained. Finally, potentials for future work on Funnel sort will be presented.

2 Funnel sort Algorithm

2.1 The Intuition behind the Funnel sort algorithm

Funnel sort approximates the M-way Merge sort described above. Funnel sort accomplishes this goal by recursively defining a k merge in terms of $\sqrt{k}$, $\sqrt{k}$-way merges. The key to the success of Funnel sort is that no matter where a k-way merge appears in the sort tree, the overhead associated with doing the k-way merge remains fixed. In other words, the amount of memory required to perform a k-way merge that is in addition to the k input buffers and the single output buffer is fixed. Note that this is not the case in a simple Merge sort. While a 4-way merge can be thought of as the composition of 2, 2-way merges, the entire intermediate output from the leaf 2-way merges must be considered part of the 4-way merge structure, as this intermediate output is neither one of the original 4 input buffers, nor the final output buffer. So, for example, near the bottom of the sort tree, a 4-way merge may fit entirely in cache if the intermediate output from the original 4 input buffers is small, but near the top of the sort tree, say the final 4-way merge, the intermediate output will be much larger and no longer fit in cache.

To solve this size problem, a k-funnel only keeps fixed size internal buffers to hold some (but not all) of the total eventual output from the $\sqrt{k}$-funnels. Then, those buffers are repeatedly filled as needed when they are emptied. Thus, regardless of location in the sort tree, the k-funnel maintains size. See figure 2 on the following page for a picture of a k funnel.

2.2 The Funnel sort algorithm

The following brief description comes from [6].
1. Split the input into $n^{1/3}$ continuous arrays of size $n^{2/3}$, and sort these recursively.
2. Merge the $n^{1/3}$ sorted sequences using a $n^{1/3}$-way merge.

A k-way merge is imlemented as a k-funnel. A funnel is recursively laid out as hinted at above in terms of $\sqrt{k}$ "children" funnels of size $\sqrt{k}$. These funnels are labeled L_i in the figure on the following page. The k-funnel also has $\sqrt{k}$ internal buffers feeding the output from each of the children funnels into a final $\sqrt{k}$ funnel, labeled R in the figure. Then when a k-funnel is asked to merge k inputs, it recursively asks it's children $\sqrt{k}$-funnels to merge $\sqrt{k}$ of the inputs each, and then the final R funnel outputs the merged result of the L_i funnels outputs. Every time a k funnel is asked to output some elements, or "invoked", it outputs k^3 elements if there are sufficient inputs. Thus, when a k-funnel is invoked, it must eventually invoke R a total of $k^{3/2}$ times. Before each invocation of R, we must check to make sure that each internal buffer i contains at least $k^{3/2}$ elements. If a buffer does not contain enough elements, the associated L_i funnel must be invoked. The internal buffers are each of size $2k^{3/2}$ in order to accommodate the $k^{3/2}$ elements output from the L_i buffer. Recall that these internal buffers are written to repeatedly and at various internal states of the buffer. Thus, these internal buffers are implemented as circular buffers so that as long as there are less than $k^{3/2}$ elements in the buffer, you can always write $k^{3/2}$ more elements to the buffer without moving any existing data.

2.3 Cache-complexity of Funnel sort

Claim 1 *Claim: Funnel sort uses $O(n/L * log_M(n/L))$ memory transfers, exactly the bound of a cache-aware ideal algorithm*

What follows is meant to be a sketch of a proof to provide the reader with better intuition of the algorithm and its cache-complexity. Ideas were taken from [6] and [7].

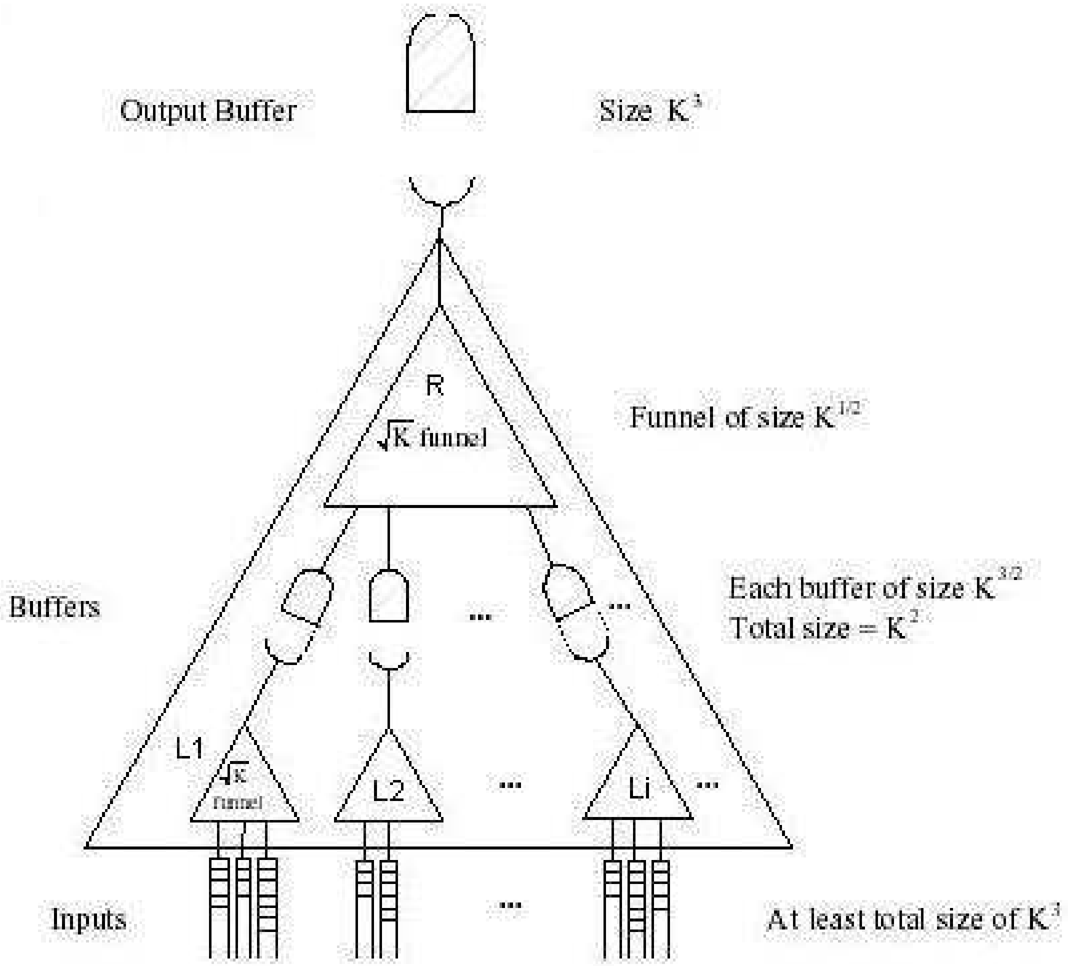

Figure 2: A k-funnel. Figure taken from [7]

Lemma 2 *Lemma 2: A k-funnel can be laid out in $O(k^2)$ continuous memory locations. Taken from [6].*

Proof The internal buffers of the k-funnel require $\sqrt{k} * k^{3/2}$ space, which is $O(k^2)$. Thus, the space required is solved by the recurrence:

$$S(k) \leq (\sqrt{k}+1)S(\sqrt{K}) + O(k^2) \tag{1}$$

Note that the k-funnel consists of $\sqrt{k}+1$ funnels of size $\sqrt{k}$ and the final term in the occurrence comes from the size of the internal buffers. This recurrence has solution has solution $S(k) = O(k^2)$ [6]. □

Now, say k is the smallest funnel that fits entirely in cache. Now, view the original problem as a tree of k merges. It is interesting but not important to note that the buffers between these k merge nodes are of different sizes (but constant at a given depth in the tree). The illustration of this model is in the figure on the following page.

Now, every time that a k node is invoked and has adequately filled input buffers, how many cache misses will occur? The invoke causes k^3 elements to be pulled into cache and output in blocks of L elements. Thus, we expect to make $O(\frac{k^3}{L})$ memory transfers. Also, at every level in the tree we need to process a total of n elements (eventually). This means that invoke is called a total of about $\frac{n}{k^3}$ times. Thus, the total cost per level is: $O(\frac{n}{L})$, exactly as in previous cases. But what is the height of the tree?

We must now solve for the largest k-funnel that fits in the cache. To fit in cache, we require a total of: $O(k^2 + (k+1) * L)$ memory locations, where k^2 refers to the size of the funnel, and $k+1$ refers to the number of cache lines to have one per input buffer and one for the output buffer. Here we apply what is called the tall cache assumption, which is simply that $M > L$, that the number of lines in the cache is greater than the number of words in a line of cache. So, setting $O(k) = O(L)$ implies that we need a total of: $O(L^2)$ memory in cache to fit the k-funnel in cache, and we have $M * L > L^2$ memory locations in cache. So, the size of funnel that fits entirely in cache is $O(L)$.

Thus, our sort tree has branch factor $O(L)$, and has height:

$$O(\log_L{(n/L)}) = O(\frac{\log_L{(n/L)}}{\log_L M}) = O(\log_M n/L) \tag{2}$$

Here we have used the weaker assumption that $M < L^k$ for some constant factor k. For a more detailed proof, see [6]. Thus, we conclude that the total number of memory transfers is:

$$O(n/L * \log_{M*L}{(n/L)}) \tag{3}$$

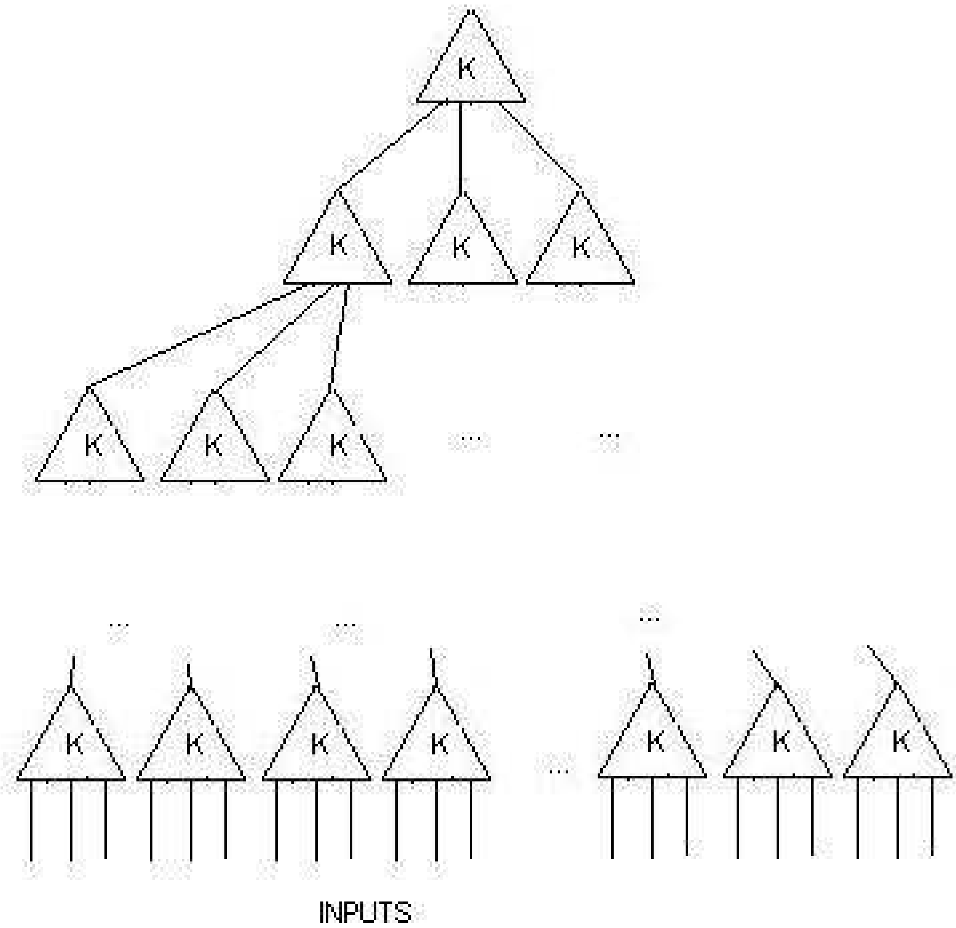

Figure 3: Decomposition of Funnel sort into k-merges

3 Serial Implementation of Funnel sort

3.1 Relevant difficulties in implementing Funnel sort

There are several issues that must be addressed when dealing with an implementation of Funnel sort. Somewhat surprisingly, efficient implementation of the circular buffers within a funnel is crucial. The the merger of circular buffers was optimized by pre-calculating when a head or tail pointer needed to be wrapped from the end of a buffer to the beginning of that buffer. This and related optimizations led to a 50 percent speed up in execution. This fact is only mentioned because it illustrates the computational complexity of Funnel sort and its associated data structures.

Not surprisingly, the issue of rounding also needed to be carefully considered. Because funnels are laid out recursively out of $\sqrt{k}$-funnels, we have to deal with values of k that are not square. For example, how do you recursively lay out a 10-funnel? For inconvenient k, it becomes impossible to have a balanced k-funnel consisting of $\sqrt{k}$-funnels. In this implementation, a 10-funnel, is laid out with the R funnel being a 3-way merge with inputs coming from L_1, L_2, and L_3 where L_1 and L_2 are 4-funnels and L_3 is a 2-funnel. Thus, in solving this problem, we end up with a slightly inballanced k-funnel, and we need to increase the size of internal buffers slightly to accommodate the inability to invoke a funnel exactly $k^{3/2}$ times.

A special thanks to Matteo Frigo is due here, as I compared runtime against his code, and also integrated his code for Quick sort into my implementation of Funnel sort.

3.2 Performance

The performance of the final implementation performed between 4 and 5 times as slow as Quick sort, and just under 4 times as slow as Matteo's Cilksort. When compared to a straight 2-way Merge sort, Funnel sort consistently runs just under 3 times as slowly. The below figure 4 shows the results of the comparison.

3.3 Approximate number of memory transfers

To approximate the number of memory transfered used by Funnel sort, the assumption is made that sorting elements that are half the size should incur half as many memory transfers. This is not strictly true as changing the size of elements could potentially change the execution of the program in other subtle ways. The change in running time from halving the size of the elements is displayed in the below figure. For comparison purposes, the same experiment was done with Cilksort. Experiments were done by having both Cilksort and Funnel sort break into their base cases (of resorting to Quick sort) when the array to be sorted had less than 100 elements in it. By this metric, Cilksort appears to use outperform Funnel sort, which perhaps implies that Cilksort does do fewer memory transfers than Funnel sort does. See figure 5 a couple of pages in advance for the results.

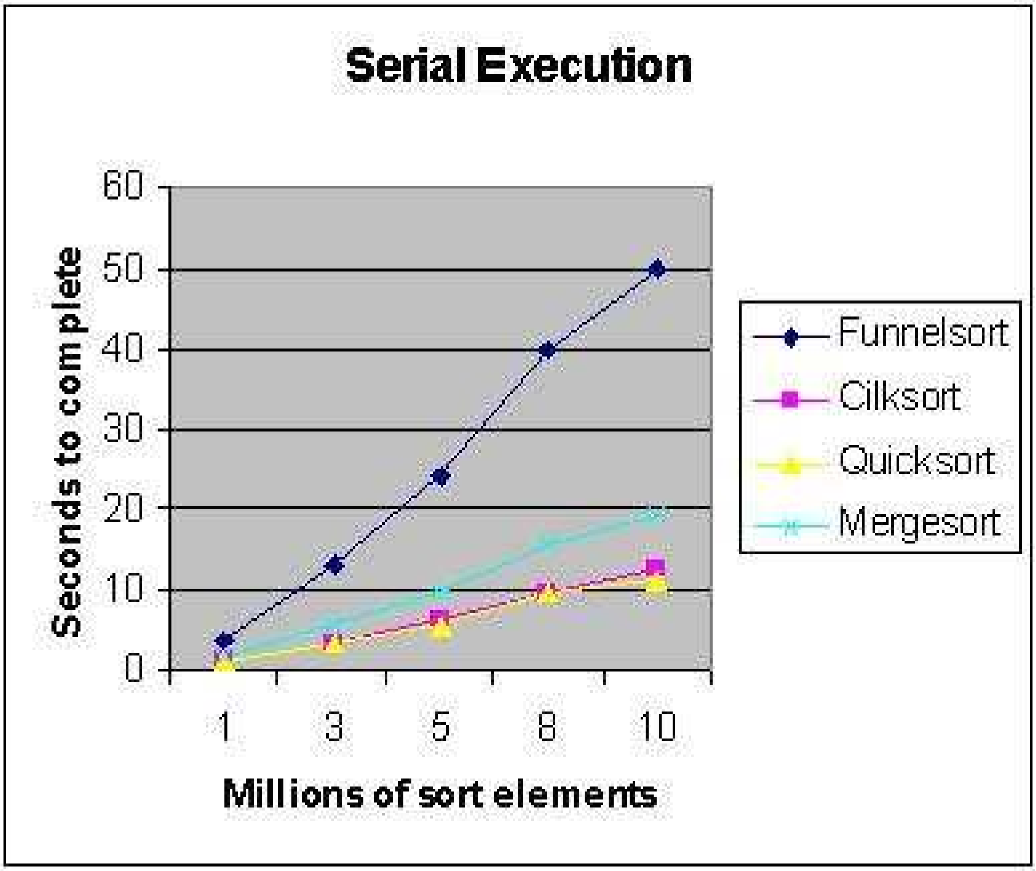

Figure 4: Serial comparison

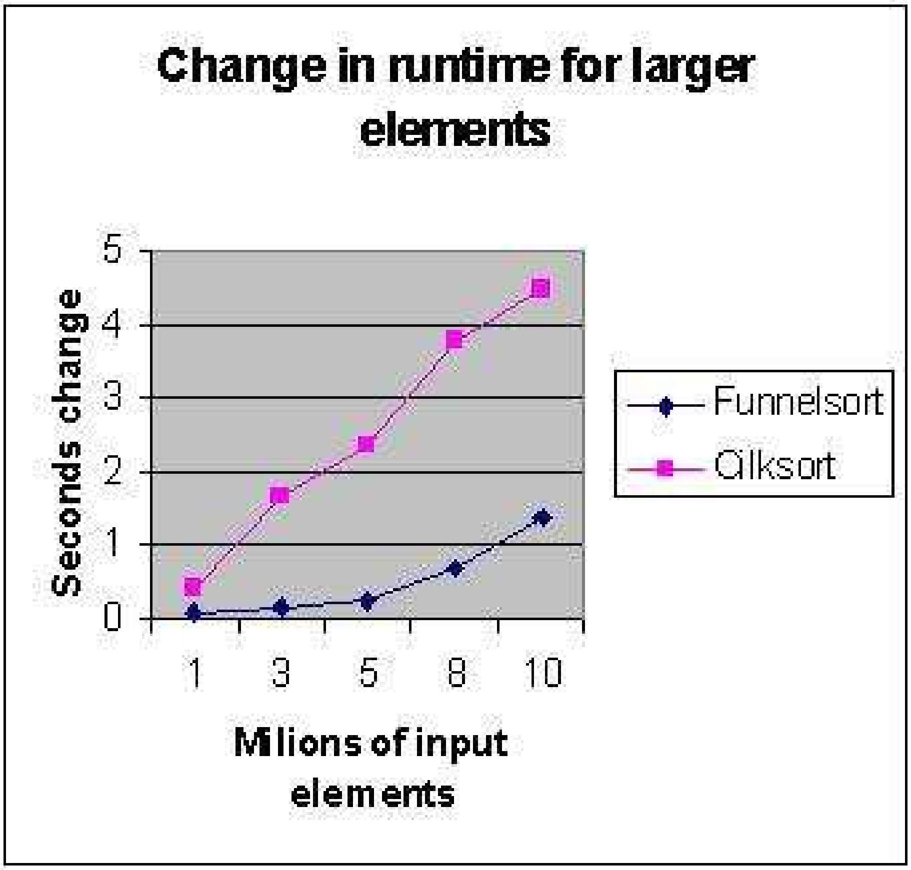

Figure 5: Change in time for larger elements. Longs vs Shorts.

3.4 Analysis of results

The results we have seen imply that, at least when all data fits in main memory, memory transfers do not dominate the run time. Instead, the overhead associated with maintaining Funnelsort's data structures outweighs the benefits of cache-efficiency. In addition, it appears that the given implementation of Funnel sort performs more memory transfers than the implementation of Cilksort. Does this mean that we should abandon Funnel sort?

First we consider the issue of overhead computation required to run Funnel sort. Essentially, we are paying a relatively large amount of computation time to save on memory transfers. In this case, where the memory transfer is from main memory to L2 cache, the time it takes to transfer a block of memory from main memory to cache may be comparable to the time it takes to do 100 computations. However, if the comparison is instead from main memory to hard disc, then each block transfer costs upwards of thousands of computations. Thus, the overhead computation will become a much smaller percentage of the total Wall-clock running time.

In addition, it is very possible to decrease the amount of overhead in the given implemtation. A variant on Funnel sort that uses Lazy funnels is a much more memory compact and simpler version of Funnel sort that deals with only binary merges. An implementation of Lazy Funnel sort may prove to take much less overhead. Also, it is very likely that the given implementation can be further optimized to decrease the amount of overhead.

Now we consider the issue of memory transfers. The given benchmark is perhaps flawed in that much of the computation (not directly related to memory transfers) could also be affected by the size of the elements being sorted. For example, comparisons could take longer. In this case, because Funnel sort does a much larger number of computations than Cilksort, Funnelsort's performance increase would be magnified when sorting smaller elements. The previous arguments largely apply here as well. A more efficient implementation of Funnel sort is very possible to exist.

We can even see an example where these principles are at work in the Masters Thesis of Kristoffer Vinther. His thesis focused solely on implementing Funnel sort and he produced a very efficient serial implementation. In fact, he was able to beat the built in C++ optimized sort std::sort. In his case, the material to be sorted did in fact come from disc instead of from main memory. Thus, the cost of a memory transfer becomes enormous, and this no doubt helped reduce the percentage of runtime spent on the computational overhead associated with Funnel sort.

4 Parallel Implementation of Funnel sort

4.1 Motivation

Funnel sort is tempting to parallelize because it breaks down into large independent subproblems very easily. Before the R funnel of a k-funnel can invoked, it's $\sqrt{k}$ input buffers must have sufficient elements in them. In order to fill those intermediate buffers, multiple processors can all work concurrently.

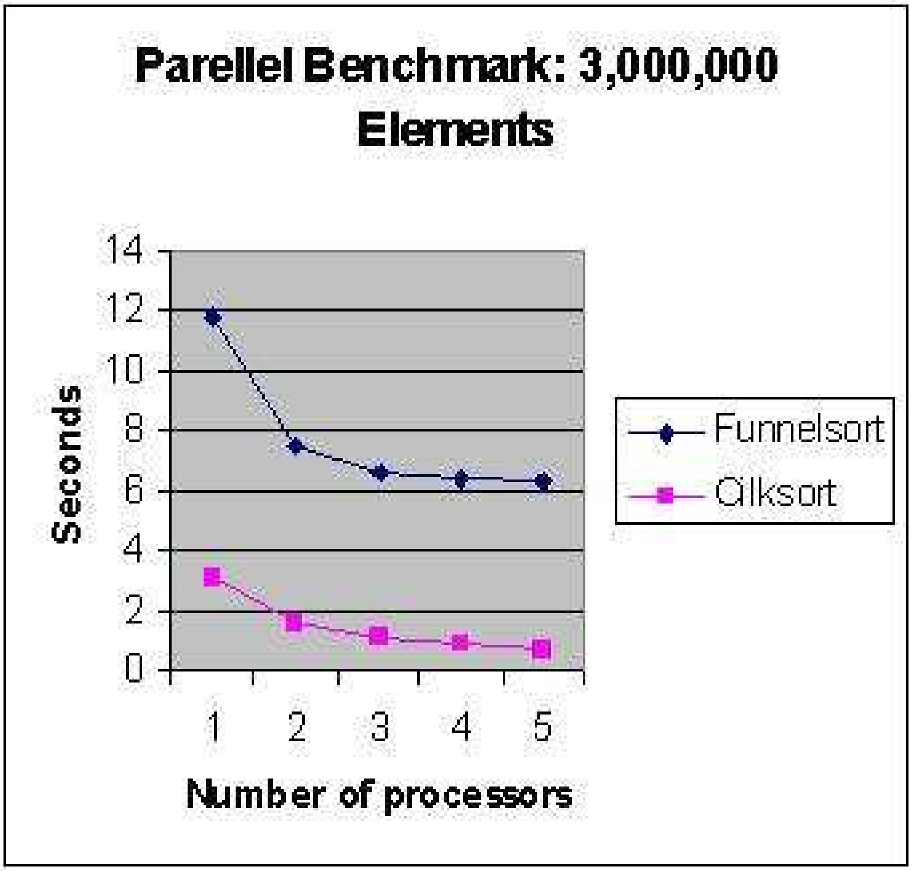

Figure 6: A parallel comparison

4.2 Performance

As shown in the below graph, the parallelization of Funnel sort is fairly good, but outperformed by Cilksort. The reason is almost certainly an absence of a parallel merge in Funnel sort. In the end the final merge must be done on a single processor, a merge that can be done in parallel by Cilksort. A parallel merge would be difficult to code in Funnel sort because the size of the final merge is dependent on the size of the input.

4.3 Potential optimizations

To improve parallel performance, a parallel merge must be implemented. Again, if Funnel sort is implemented in the Lazy funnel manner mentioned above, all merges will be 2-way, and implementing a parallel merge is trivial. Also, it seems possible that by using memory locks and some scheduling manipulation, performance could be sped up. In the current parallel implementation of Funnel sort, we must wait until all input buffers have been fully written before we can proceed with a merger. However, the empty portion of the circular buffer could be written to at the same time that the filled portion of the circular buffer is being read. In this manner, we could gain a slight performance benefit. In theory, parallelizing Funnel sort is exciting because Funnel sort can begin the final merge much sooner than Cilksort. Since the final merge is definitely on the critical path of the problem, it is likely that Funnel sort has a shorter critical path than Cilksort. Unfortunately, until a parallel merge is implemented in Funnel sort, this will not be true.

5 Conclusions

In theory, Funnel sort is order as cache-efficient as a cache-aware idealized algorithm. However, in practice Funnel sort suffers from a large amount of overhead calculations neccessary to maintain the complex funnel data structures, and these extra calcualtions can overshadow the benefits of a decrease in memory transfers. However, as verified by Vinther, once the memory transfers become expensive enough, the extra computation can become justified.

In addition, Funnel sort is potentially very parallelizable, and the given implementation has several visible areas of improvement. It is quite possible that Funnel sort is more parallelizable than Cilksort, because the final merger can begin much earlier, potentially before all of the inputs have even been processed once.

References

[1] Jimenez-Gonzales, et. al. Exploiting the Memory Hiearchy in Parallel Sorting Algorithms. 2000.

[2] Daniel Jimenez-Gonzales, et. al. The Effect Of Local Sort on Parallel Sorting Algorithms. 2002.

[3] Richard P. Brent and Andrew Tridgell. A Fast, Storage-Efficient Parallel Sorting Algorithm. http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pd/rpb140.pdf. 1993.

[4] Scandal Project. Review of Sorting Algorithms. http://www-2.cs.cmu.edu/ scandal/nesl/algorithms.html

[5] 6.895 Lecture Notes. 2003.

[6] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. 1986.

[7] Matteo Frigo, et al. Cache-Oblivious Algorithms. 1999.

[8] Erik Demaine, 6.897 Spring '03, Lecture 17 notes, scribed by Jonathan Burnsman, Glenn Eguchi.

[9] Matteo Frigo. cilksort.cilk, from Cilk example programs.

[10] Kristoffer Vinther, "Engineering Cache-Oblivious Sorting Algorithms", Masters Thesis, currently available at: web.mit.edu/ youn/www/thesis.pdf