Patience is a Virtue: Revisiting Merge and Sort on Modern Processors

Badrish Chandramouli and Jonathan Goldstein
Microsoft Research
{badrishc, jongold}@microsoft.com

ABSTRACT

The vast quantities of log-based data appearing in data centers has generated an interest in sorting almost-sorted datasets. We revisit the problem of sorting and merging data in main memory, and show that a long-forgotten technique called *Patience Sort* can, with some key modifications, be made competitive with today's best comparison-based sorting techniques for both random and almost sorted data. Patience sort consists of two phases: the creation of sorted runs, and the merging of these runs. Through a combination of algorithmic and architectural innovations, we dramatically improve Patience sort for both random and almost-ordered data. Of particular interest is a new technique called *ping-pong merge* for merging sorted runs in main memory. Together, these innovations produce an extremely fast sorting technique that we call *P³ Sort* (for *Ping-Pong Patience+ Sort*), which is competitive with or better than the popular implementations of the fastest comparison-based sort techniques of today. For example, our implementation of P³ sort is around 20% faster than GNU Quicksort on random data, and 20% to 4x faster than Timsort for almost sorted data. Finally, we investigate replacement selection sort in the context of single-pass sorting of logs with bounded disorder, and leverage P³ sort to improve replacement selection. Experiments show that our proposal, P³ replacement selection, significantly improves performance, with speedups of 3x to 20x over classical replacement selection.

Categories and Subject Descriptors

E.0 [**Data**]: General; E.5 [**Data**]: Files – Sorting/searching.

Keywords

Sorting; Patience; Merging; Replacement Selection; Performance.

1. INTRODUCTION

In this paper, we investigate new and forgotten comparison based sorting techniques suitable for sorting both nearly sorted, and random data. While sorting randomly ordered data is a well-studied problem which has produced a plethora of useful results over the last five decades such as Quicksort, Merge Sort, and Heap Sort (see [9] for a summary), the importance of sorting almost sorted data quickly has just emerged over the last decade.

SIGMOD/PODS'14, June 22 - 27 2014, Salt Lake City, UT, USA

http://dx.doi.org/10.1145/2588555.2593662

In particular, the vast quantities of almost sorted log-based data appearing in data centers has generated this interest. In these scenarios, data is collected from many servers, and brought together either immediately, or periodically (e.g. every minute), and stored in a log. The log is then typically sorted, sometimes in multiple ways, according to the types of questions being asked. If those questions are temporal in nature [7][17][18], it is required that the log be sorted on time. A widely-used technique for sorting almost sorted data is *Timsort* [8], which works by finding contiguous *runs* of increasing or decreasing value in the dataset.

Our investigation has resulted in some surprising discoveries about a mostly-ignored 50-year-old sorting technique called *Patience Sort* [3]. Patience sort has an interesting history that we cover in Section 6. Briefly, Patience sort consists of two phases: the creation of natural sorted runs, and the merging of these runs. Patience sort can leverage the almost-sortedness of data, but the classical algorithm is not competitive with either Quicksort or Timsort. In this paper, through a combination of algorithmic innovations and architecture-sensitive, but not architecture-specific, implementation, we dramatically improve both phases of Patience sort for both random and almost-ordered data. Of particular interest is a novel technique for efficiently merging sorted runs in memory, called *Ping-Pong merge*. Together, these innovations produce an extremely fast sorting technique that we call *P³ Sort* (for *Ping-Pong Patience+ Sort*), which is competitive with or better than the popular implementations of the fastest comparison-based sort techniques on modern CPUs and main memory. For instance, our implementation of P³ sort is approximately 20% faster than GNU Quicksort on random data, and 20% to 4x faster than the popular Timsort implementation for almost-sorted data.

We then investigate methods for sorting almost-sorted datasets in a single pass, when the datasets are stored in external memory, and disorder is bounded by the amount of data which can fit in memory. We show how P³ sort may be combined with replacement selection sort to minimize the CPU cost associated with single pass sorting. We propose flat replacement selection, where a periodically sorted buffer is used instead of a heap, and P³ replacement selection, where the P³ sorting algorithm is deeply integrated into replacement. P³ replacement selection, in particular, is a dramatic practical improvement over classical replacement selection, achieving CPU speedups of between 3x and 20x.

We believe that these techniques form a foundation for the kind of tuning process that their brethren have already undergone, and given their current level of competitiveness, could become commonly used sorting techniques similar to Quicksort and Timsort. For instance, we do not, in this paper, explore methods of parallelizing or exploiting architecture-specific features (such as

SIMD [20]) for further performance improvement. Rather, we intend to establish a solid foundation for such future investigation.

We also expect this work to revive interest in replacement selection, which was once used to bring the number of external memory passes down to two, and which can now, for many logs, be used to bring the number of external memory passes down to one.

The paper is organized as outlined in Table 1. Sections 6 and 7 cover related work and conclude with directions for future work.

Basic Sort and Improvements (Section 2)	Patience sort	Sec. 2.1	Prior work
	Patience+ sort	Sec. 2.2	Re-architecture to make Patience sort competitive
Ping-Pong Merge (Section 3)	Balanced	Sec. 3.1	Basic merge approach
	Unbalanced	Sec. 3.2	Handles almost sorted data
P^3 Sort (Section 4)	Naïve P^3 sort	Sec. 4.1	First sorting version that combines our prior ideas
	CS P^3 sort	Sec. 4.2	Cache-sensitive version
	P^3 sort	Sec. 4.3	Final sorting version with all optimizations added
Replacement Selection (RS) (Section 5)	Flat RS	Sec. 5.2	Replace heap with a sort buffer in RS
	P^3 RS	Sec. 5.3	Integrates P^3 into the sort buffer in RS

Table 1: Paper Outline and Contributions

2. PATIENCE AND PATIENCE+ SORT

2.1 Background on Patience Sort

Patience Sort [3] derives its name from the British card game of Patience (called Solitaire in America), as a technique for sorting a deck of cards. The patience game (slightly modified for clarity) works as follows: Consider a shuffled deck of cards. We deal one card at a time from the deck into a sequence of *piles* on a table, according to the following rules:

1. We start with 0 piles. The first card forms a new pile by itself.
2. Each new card may be placed on either an existing pile whose top card has a value no greater than the new card, or, if no such pile exists, on a new pile below all existing piles.

The game ends when all cards have been dealt, and the goal of the game is to finish with as few piles as possible.

Patience sort is a comparison-based sorting technique based on the patience game, and sorts an array of elements as follows. Given an n-element array, we simulate the patience game played with the greedy strategy where we place each new card (or element) on the oldest (by pile creation time) legally allowed pile (or *sorted run*). This strategy guarantees that the top cards across all the piles are always in increasing order from the newest to oldest pile, which allows us to use binary search to quickly determine the sorted run that the next element needs to be added to.

After the *run generation phase* is over, we have a set of sorted runs that we merge into a single sorted array using an n-way merge (usually with a priority queue), during the *merge phase* [1].

Example 1 (Patience sort) Figure 1 shows a 10-element array that we use to create sorted runs.

3	5	4	2	1	7	6	8	9	10

Figure 1: Patience Sort Input

Patience sort scans the data from left to right. At the beginning, there are no sorted runs, so when the 3 is read, a new sorted run is created and 3 inserted at the end. Since 5 comes after three, it is added to the end of the first run. Since the 4 cannot be added to the end of the first sorted run, a new run is created with 4 as the only element. Since the 2 cannot be added to either the first or second sorted run, a third sorted run is created and 2 added. Similarly, a fourth sorted run is created with 1. At this point, we have the 4 sorted runs shown in Figure 2.

run 1	3	5
run 2	4	
run 3	2	
run 4	1	

Figure 2: Sorted Runs After 5 Inputs

Next, we read the 7. Since the first run is the run with the earliest creation time which 7 can be added to, we add the 7 to the first run. Next, we read the 6. In this case, the second run is the run with earliest creation time which 6 can be added to, so we add 6 to the second run. Similarly, we add the rest of the input to the first run, resulting in the sorted runs after phase 1 shown in Figure 3.

3	5	7	8	9	10
4	6				
2					
1					

Figure 3: Sorted Runs After Phase 1

The usual priority queue based remove and replace strategy is then used on the 4 final runs, resulting in the final sorted list.

Runtime Complexity For uniform random data, on average, the number of sorted runs created by the run generation phase of Patience sort is $O(\sqrt{n})$ [2]. Since each element needs to perform a binary search across the runs, the expected runtime is $O(n \cdot \log n)$. The merge phase has to merge $O(\sqrt{n})$ sorted runs using a priority queue, which also takes $O(n \cdot \log n)$ time, for a total expected running time of $O(n \cdot \log n)$. It is easy to see that if the data is already sorted, only 1 run will be generated, and the algorithm time is $O(n)$. As the data becomes more disordered, the number of runs increases, and performance gracefully degrades into $O(n \cdot \log n)$. Observe that even if the number of lists is n, the execution time is still $O(n \cdot \log n)$, and the technique essentially becomes merge sort.

2.2 Patience+ Sort

To understand the historic lack of interest in Patience sort, we measured the time it takes to sort an array of uniform random 8-byte integers. We use Quicksort (GNU implementation [11]) and a standard implementation of Patience sort (from [12]). The Patience sort implementation is written using the C++ standard template library, and uses their priority queue and vector data structures. The GNU Quicksort implementation includes the well-known Sedgewick optimizations for efficiency [10]: there is no recursion, the split key is chosen from first, middle, and last, the smaller sub-partition is always processed first, and insertion sort is used on small lists. All versions of all sort techniques in this paper are written in C++ and compiled in Visual Studio 2012 with maximum time optimizations enabled. All the sorting techniques in this paper use the same key comparison API as the system `qsort`. All experiments in this paper were conducted on a Windows 2008 R2 64-bit machine with a 2.67GHz Intel Xeon W3520 CPU with 12GB of RAM. The array size is varied from 100000(~ 1MB) to around 50 million (~ 400MB). All experiments were performed 3 times on identical datasets, and the minimum of the times taken.

Figure 4 demonstrates the dismal performance of existing Patience sort implementations, which are 10x to 20x slower than Quicksort.

To some degree, these dismal results are a reflection of the lack of attention Patience sort has received. We first, therefore, re-implemented Patience sort using a collection of optimizations designed to eliminate most memory allocations, and mostly sequentialize memory access patterns, making better use of memory prefetching. We call our Patience sort implementation with this collection of optimizations *Patience+ Sort*.

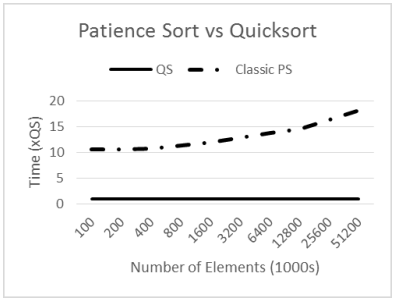

Figure 4: Patience Sort vs. Quicksort

More specifically, we pass in a pre-allocated buffer which is usually sufficient for storing data from sorted runs. In the event that this buffer is fully consumed, more large buffers are allocated and freed as necessary. All memory used in the algorithm comes from these large blocks, minimizing the number of memory allocations.

In order to best utilize memory bandwidth during the first phase, each sorted run is represented using a linked list of fixed size memory blocks. The final memory block of each linked list is pointed to in an array of memory block pointers (one for each run) to facilitate fast appends to the sorted runs. Copies of the individual tail elements of the sorted runs are stored and maintained separately in a dynamic array for fast binary searching.

The initial runs array and tails array sizes are set to the square root of the input size, and double when the number of runs exceeds this size. Since, for random data, the expected number of runs is the square root of input size, and since the number of runs decreases as the data becomes more sorted, array expansions are fairly rare.

While, for reverse ordered data, the number of runs is equal to the number of elements, this can be significantly mitigated by adding the capability of appending to either side of a sorted run. In this case, we would maintain both an array of tail elements and an array of head elements. After searching the tail elements, before adding another run, we would first binary search the head elements, which would naturally be in order, for a place to put the element. Sorting reverse ordered lists would then take linear time. The case where the number of runs is linear in the input is then quite obscure.

For the priority queue based merging in the second phase, we tried two approaches. The first is the classic *n*-way merge of *n* sorted runs using our own highly optimized heap implementation (e.g. no recursion). Note that there are no memory allocations since the result can be merged into the destination.

The second approach, which we call a tree-Q merge, is a less well known, more performant approach [4], which performs a balanced tree of *k*-way merges. Each *k*-way merge uses the same highly performant priority queue implementation as the classical approach. The idea of this approach is to choose *k* such that the priority queue fits into the processor's L2 cache. On our experimental machine, we tuned *k* to its optimal value of 1000. This approach is the best performing priority queue based merge technique known today. Note that each tree level processes the entire dataset exactly once. In order to improve performance, our implementation carefully packs all the destination runs for a particular level into a single array as large as the original dataset, and reuses unneeded memory from lower levels of the merge tree. As a result, there is no actual memory allocation during this phase.

Figure 5 shows how both versions of Patience+ sort fare against Quicksort. Note that Patience+ sort is already a dramatic improvement over Patience sort, bringing the execution time difference w.r.t. Quicksort down from 10x-20x to 1.5x-2x. Somewhat reassuring is that the tree-Q Patience+ sort seems to mostly eliminate the deterioration compared to Quicksort as dataset sizes increase. It is interesting to note, though, that re-designing Patience sort and merging around memory subsystem performance did significantly improve Patience sort.

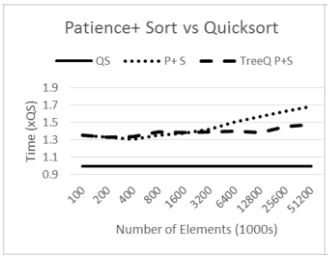

Figure 5: Patience+ Sort vs. Quicksort

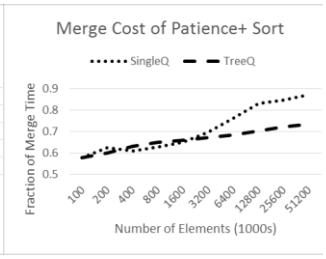

Figure 6: Patience+ Sort Merge Cost

The remaining bottleneck in Patience+ sort becomes clear from Figure 6, which shows the fraction of time spent in the merge phase of Patience+ sort. In both cases, the Patience+ sort cost is dominated by merging, reaching almost 90% for the single queue approach, and almost 75% for the tree-based merge. This led us to search for a faster merge technique, which we cover next.

3. PING-PONG MERGE

The previous section showed some of the benefits of sorting in a manner sensitive to memory subsystem performance, in particular by eliminating fine grained memory allocations, sequentializing memory access, and improving caching behavior. Recent work [20][22][23][24] has shown that using binary merges instead of a heap is effective when combined with architecture-specific features (such as SIMD) and parallelism. This section describes an algorithm for merging sorted runs, called *ping-pong merge*, which leverages binary merging in a single-core architecture-agnostic setting. Ping-pong merge is cache friendly, takes greater advantage of modern compilers and CPU prefetching to maximize memory bandwidth, and also significantly reduces the code path per merged element. Ping-pong merge demonstrates the superiority of binary-merge-based techniques over heap-based merging schemes, even in architecture independent settings. In this section, we introduce two variants, balanced and unbalanced. As we will see, the unbalanced version is important for sorting nearly ordered input.

3.1 Balanced Ping-Pong Merge

Ping-pong merge assumes the existence of two arrays, each of which is the size of the number of elements to be merged. Let *r* be the number of sorted runs. We begin by packing the *r* sorted runs into one of the arrays. For instance, Figure 7 shows a valid packing of the runs from Figure 3 packed into an array of 10 elements.

3	5	7	8	9	10	**4**	6	**2**	**1**

Figure 7: Packed Runs

Note the need for ancillary information storing the locations of the beginnings of each run, which are shown in bold. Adjacent runs are then combined, pairwise, into the target array. For instance, we can initially combine the first and second runs, and then combine the third and fourth runs, into the target array, resulting in the array with two runs shown in Figure 8.

3	4	5	6	7	8	9	10	**1**	2

Figure 8: Packed Runs After One Round of Merging

We now have just two runs. The first run begins at the first array position, while the second run begins at the ninth array position. We now merge these two runs back into the original array, resulting in the final sorted list. In general, one can go back and forth (i.e., ping-pong) in this manner between the two arrays until all runs have been merged. This style of merging has several benefits:

- The complexity of this algorithm is $O(n \cdot \log r)$, the same as the priority queue based approach, which is optimal.
- There are no memory allocations.
- The algorithm is cache friendly: we only need three cache lines for the data, one for each input and one for the output. These lines are fully read/written before they are invalidated.
- The number of instructions executed per merged element is (potentially) very small, consisting of one if-then block, one comparison, one index increment, and one element copy.
- Modern compilers and CPUs, due to the algorithm's simple, sequential nature, make excellent use of prefetching and memory bandwidth.

At a high level, ping-pong merge, as described so far, is similar to merging using a tree-Q with a tree fanout of 2. The main difference is that instead of copying out the heads to a priority queue, and performing high instruction count heap adjustments, we hardcode the comparison between the two heads (without unnecessary copying), before writing out the result to the destination. In addition, we took care to minimize the codepath in the critical loop. For instance, we repeatedly merge l total elements, where l is the minimum of the unmerged number of elements from the two runs being merged. This allows us to avoid checking if one run has been fully merged in the inner loop. Finally, we manually unroll 4 loop iterations in the innermost loop. This much more explicit code path should be more easily understood by the compiler and CPU, and result in improved prefetching and memory throughput.

We now compare the performance of the binary tree-Q merge approach (we call it *B-TreeQ*), the *k*-way tree-Q approach (with *k* set to 1000) used in Section 2.2, and ping-pong merge. The sorted runs were generated by the first phase of Patience sort over uniform random data (i.e., identical to the random workload used in Section 2.2). The runs were packed into the array in order of creation. The results are shown in Figure 9. Note that ping-pong merge is consistently *3-4 times faster* than the k-way tree-Q merge.

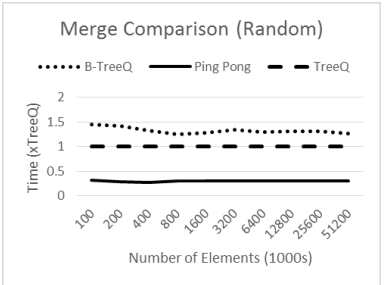

Figure 9: Merge Comparison (Random Data)

3.2 Unbalanced Ping-Pong Merge

Of course, one of the prime motivations for using Patience sort is its linear complexity on mostly sorted data. Perhaps the most prevalent real source of such data is event logs, where some data arrives late (for example, due to network delays), leading to a tardiness distribution. Therefore, we built a synthetic data generator to produce datasets that closely model such distributions. The generator takes two parameters: percentage of disorder (p) and amount of disorder (d). It starts with an in-order dataset with increasing timestamps, and makes p% of elements tardy by moving their timestamps backward, based on the absolute value of a sample from a normal distribution with mean 0 and standard deviation d.

For such a disorder model, the first phase of Patience sort typically produces a few very large runs. Figure 10 and Figure 11 illustrate this effect. Figure 10 shows the distribution of run size after the first phase of Patience sort on 100000 elements of random data. The distribution is shown in order of creation. Note that the total number of runs, and the maximum run size is roughly $\sqrt{n}$, which is expected given the theoretical properties of Patience sort over random data [1]. Figure 11 shows the run size distribution for a 100000 element disordered dataset with p and d set to 10. First, note that there are *only 5 runs*. In addition, the run size distribution is highly skewed, with the first run containing over 90% of the data.

Figure 10: Run Size – Random Data

Figure 11: Run Size - Almost Ordered Data

We know that when merging sorted runs two-at-a-time, it is more efficient to merge small runs together before merging the results with larger runs [13]. Put another way, rather than perform a balanced tree of merges, it is more efficient to merge larger runs higher in the tree. For instance, consider the packed sorted runs in Figure 7. Using the balanced approach described so far, we merge the first two runs, which involves copying 8 elements, and merge the last two elements into the second sorted run. We then do the final merge. Therefore, in total, we merge 8+2+10=20 elements.

Suppose, instead, we merge the last two runs into the second array, producing the state shown in Figure 12.

3	5	7	8	9	10	**4**	6		
								1	2

Figure 12: Unbalanced Merge (first merge)

We next merge the last run in the first array, and the last run in the second array. This is an interesting choice, because we are merging adjacent runs in different arrays. If we merge into the bottom array, we can actually *blindly merge* without worrying about overrunning the second run. The result is shown in Figure 13.

3	5	7	8	9	10				
						1	2	4	6

Figure 13: Unbalanced Merge (second merge)

Since we are down to two sorted runs, we can now do another blind merge into the bottom array. The result is shown in Figure 14.

1	2	3	4	5	6	**7**	8	9	10

Figure 14: Unbalanced Merge (third merge)

The total number of merges performed in the unbalanced merge is 2+4+10=16, compared to 20 for the balanced approach. As the distribution of run lengths becomes more skewed, the difference between the balanced and unbalanced approaches widens.

We now propose an improvement to ping-pong merge, which differs from the previous ping-pong merge algorithm in two ways:

1. Runs are initially packed into the first array in run size order, starting with the smallest run at the beginning.
2. Rather than merge all runs once before merging the result, merge in pairs, from smallest to largest. Reset the merge position back to the first two runs when either: we have merged the last two runs, or the next merge result will be larger than the result of merging the first two runs.

Together, the two changes above efficiently approximate always merging the two smallest runs.

```
Algorithm A1: Unbalanced Ping-Pong Merge
1  UPingPongMerge(Runs: Array of sorted runs,
                  Sizes: Array of sorted run sizes)
2    RunSizeRefs : Array of (RunIndex, RunSize) pairs
3    Elems1 : Array of sort elements
4    Elems2 : Array of sort elements
5    ElemsRuns : List of (ElemArr, ElemIndex, RunSize)
                 Triples

6    For each element i of Runs
7      RunSizeRefs[i] = (i, Sizes[i])
8    Sort RunSizeRefs by RunSize ascending
9    NextEmptyArrayLoc = 0;
10   for each element i of RunSizeRefs
11     copy Runs[i.RunIndex] into Elems1 starting at
           position NextEmptyArrayLoc
12     ElemsRuns.Insert (1, NextEmptyArrayLoc, i.RunSize)
13     NextEmpotyArrayLoc += i.RunSize

14   curRun = ElemsRuns.IterateFromFirst
15   while ElemsRuns has at least two runs
16     if (curRun has no next) or
          (size of merging curRun and its next >
           size of merging the first and second runs)
         CurRun = ElemsRuns.IterateFromFirst
17     if (curRun.ElemsArr == 1)
18       Blindly merge curRun and curRun's next into Elems2
             starting at element position curRun.ElemIndex
19       curRun.ElemArr = 2
20     else
21       Blindly merge curRun and curRun's next into Elems1
             starting at element position curRun.ElemIndex
22       curRun.ElemArr = 1
23     curRun.RunSize += curRun.Next.RunSize
24     remove curRun's next
25     curRun.MoveForward
26   if (ElemsRuns.First.RunIndex == 1) return Elems1
27   else return Elems2
```

Algorithm A1 shows unbalanced ping-pong merge. Lines 8-13 sort the runs and pack them in the array. Lines 14-25 perform the unbalanced merge. We finally return the array of sorted elements.

3.3 Evaluating Unbalanced Ping-Pong Merge

In order to better understand the potential impact of this optimization, we begin by examining the best and worst cases. Clearly, the best case occurs when we have one very large run, and many small runs. We therefore conducted an experiment where we have a single large run of 60 million 8 byte integers (about 500 MB), and a variable number of single element runs. We measured the time to merge using both the balanced and the optimized unbalanced approach. Figure 15 shows the results. The cost of the unoptimized approach increases exponentially with the number of runs (the x-axis is log scale). This is expected since every time we double the number of runs, the number of levels of the merge tree increases by one. Since the large run gets merged once at each level, the overall cost increases linearly.

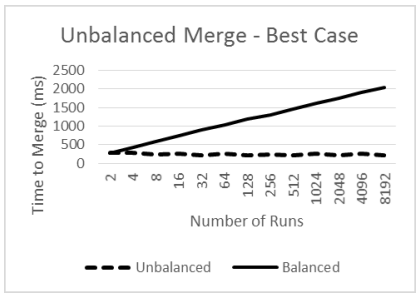

Figure 15: Unbalanced Merge - Best Case

On the other hand, the optimized approach is pretty insensitive to the number of runs, since the cost is dominated by the single merge that the large run participates in. These results generally hold in the situation where there are a few runs that contain almost all the data.

In contrast, we now consider the worst case for unbalanced merging. This occurs when there is little to no benefit of merging the small runs early, but where the cost of sorting the run sizes is significant compared to merging. Initially, we tried a large number of single element lists. The sorting overhead turned out to be negligible since sorting already sorted data is quite fast. We then tried, with more success, a run size pattern of 1-2-1-2-1-2… This run pattern ensured that the sort had to push half the ones into the first half, and half the twos into the second half. The sorting overhead in this case was significant enough to be noticeable. We then tried 1-2-3-1-2-3…, which was even worse. We continued to increase the maximum run size up to 40, and varied the total number of runs between 800k and 3.2M. Figure 16 shows the results.

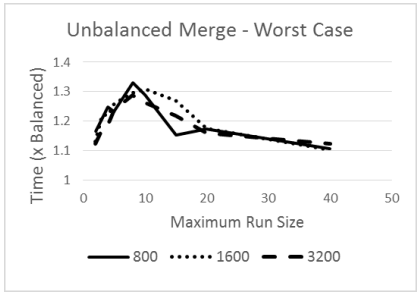

Figure 16: Unbalanced Merge - Worst Case

The worst case peaks at ~30% overall penalty, with a maximum run size of about 9. As the maximum run size increases, so does the average run size. As a result, more time is spent merging instead of sorting run sizes, and the overhead of sorting eventually decreases.

Given the data dependent effect of unbalanced merging, we now investigate the effect of unbalanced merging on sorted runs generated by the first phase of Patience sort. In particular, consider our disordered data generator from Section 3.2. In Figure 11, we fixed both the percentage and the standard deviation of disorder. Figure 17 shows the results of an experiment where we varied both the disorder percentage, and the amount of disorder, and measured the effect of the unbalanced merge improvement on merge time.

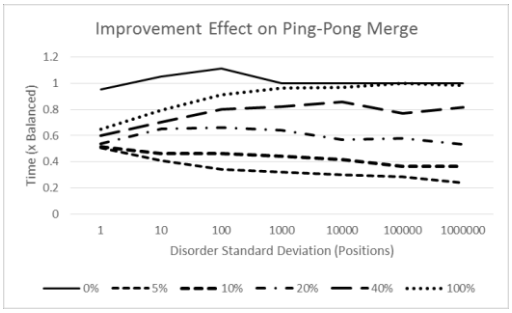

Figure 17: Effect of Unbalanced Merging on Ping-Pong Merge

First, notice that when there is no disorder (0%) unbalanced merging has no effect. This makes sense when one considers that there is only one sorted run. Looking at the other extreme, where 100% of the data is disordered, as the disorder amount grows, the optimization becomes less effective. This makes sense when one considers that for purely disordered data, the run sizes are more uniform than for disordered data, as is illustrated in Figure 10 and Figure 11. When disorder is rare, increasing the amount of disorder actually causes the optimization to become more effective, because the likelihood that each disordered element causes a new run to form is much higher, than if there is a small amount of disorder.

There are two important takeaways from this experiment: First, unbalanced tree merging is never detrimental to performance for the types of Patience sort workloads we are targeting. Second, we saw improvements by as much as a factor of 5, with possibly even higher levels of improvement for other Patience sort cases. For the best case measured here, unbalanced ping-pong merge was more than a factor of 10 faster than state-of-the-art heap based merge techniques such as cache-aware k-way tree-Q merge.

4. PING-PONG PATIENCE SORT

4.1 Naïve P³ Sort

We now combine our efficient implementation of the first phase of Patience+ Sort, with our optimized ping-pong merge. We call the result naïve Ping-Pong Patience+ Sort (i.e., naïve P³ Sort). We call this version "naïve" because we introduce further important optimizations later in this section. In particular, we introduce a cache-sensitive version, called Cache Sensitive Ping-Pong Patience Sort, which includes optimizations in the first phase to improve cache related performance. In the final variant, simply called Ping-Pong Patience Sort, additional optimizations are made in the first phase to improve performance for almost ordered input.

To begin, we re-run our experiment comparing Quicksort with Patience sort, but this time use naïve P³ sort. The result is shown in Figure 18. First, note that naïve P³ sort is *faster than Quicksort* in all measured cases, ranging between 73% and 83% of the time taken by Quicksort.

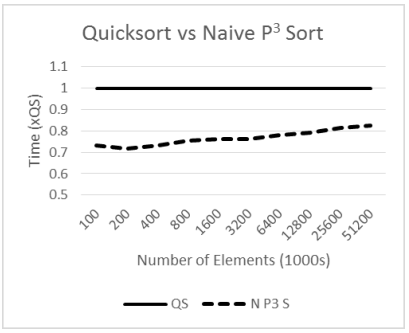

Figure 18: Quicksort vs Naïve P³ Sort

4.2 Cache-Sensitive P³ Sort

Note that naïve P³ sort's improvement over Quicksort diminishes as the dataset size increases. To understand this phenomenon more clearly, we first examine the percentage of time spent in the first phase of naïve P³ sort as dataset size increases. The results are shown in Figure 19.

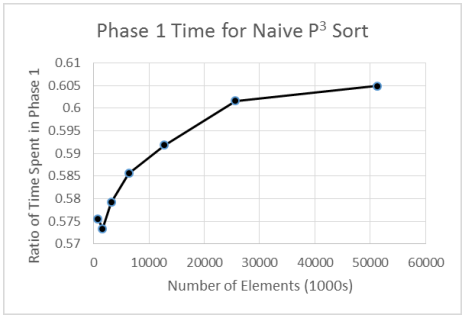

Figure 19: Phase 1 Time for Naïve P³ Sort

Note that as dataset size increases, so does the percentage of time spent in the first phase. Suspecting this might be a caching effect, we decided to limit the binary search of tail values in phase 1 to a fixed number of the most recently created runs. This idea is to fit all the searched tail values in the cache. We tried a size of 1000, which is the optimal size for the array in tree-Q merge (see Section 2.2). The results are shown in Figure 20. Limiting the number of runs which can be actively appended to, achieves the intended effect. Cache-Sensitive (CS) P³ sort on random data takes 73% to 75% of the time it takes Quicksort to sort the same data, with no observable change in relative performance as dataset size increases.

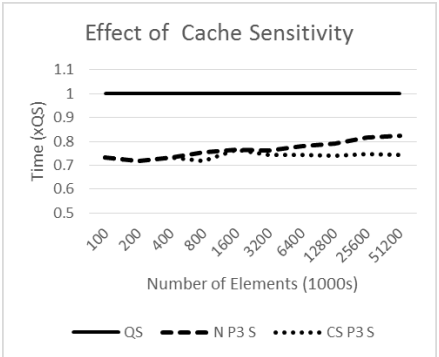

Figure 20: Effect of Cache Sensitivity

So far, our focus for P³ sort has been on random data, where Quicksort excels, and P³ sort is unable to leverage order. We now examine the performance of cache sensitive P³ sort for the synthetic

workloads described in Section 3.2. In these workloads we effectively "push forward" a fixed percentage of in-order data by a number of positions which follows the absolute value of a normal distribution with mean 0. We vary the percentage of data which is pushed, and also vary the standard deviation of amount of push. We compare, for these workloads, the time taken to sort using cache sensitive P^3 sort and Quicksort. The results are shown in Figure 21.

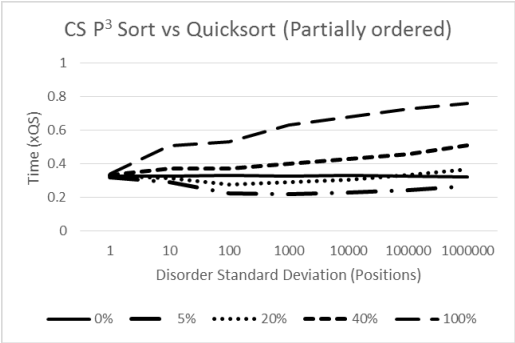

Figure 21: CS P^3 Sort vs. Quicksort (partially ordered)

First, note that when the data is already sorted, CS P^3 sort is approximately 3x faster than Quicksort. It is worth noting that while Quicksort benefits significantly from the data being sorted, performance falls off faster for Quicksort than CS P^3 sort as the data becomes mildly disordered. This explains the up to 5x improvement of CS P^3 sort over Quicksort with mild disorder.

4.3 Final P^3 Sort

While this is an excellent showing for CS P^3 sort, it is worth noting that the total cost of CS P^3 sort is still significantly higher (about 7 times) than performing two memory copies of the dataset for perfectly ordered data. In fact, when considering the cost this way, the overall costs of mildly disordered data seems too high. To better understand this, we measured the ratio of time spent in phase 1 for the above experiment. The results are shown in Figure 22.

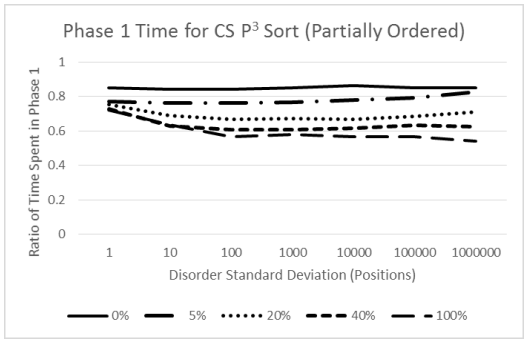

Figure 22: Phase 1 Time for CS P^3 Sort (Partially Ordered)

Note that while the time is about evenly split between phase 1 and phase 2 for mostly random data, as the data becomes more ordered, more of the time is spent in the first phase. At the most extreme, more than 80% of the time is spent in the first phase. Looking to the literature on sorting almost-sorted data, the best-known current technique is Timsort, which is the system sort in Python, and is used to sort arrays of non-primitive type in Java SE 7, on the Android platform, and in GNU Octave [8]. Timsort has a heavily optimized Java implementation (which we translated to C++). While we defer a comparison until Section 4.4, we note that this implementation has many optimizations around trying to, as quickly as possible, copy consecutive sorted elements in the input into sorted runs.

Learning from this approach, we optimize our CS P^3 sort implementation to handle this case as follows. After we add a new element to the tail of a sorted run during phase 1, we introduce a small loop where we try to insert as many subsequent elements from the input as possible to the same sorted run. This is achieved by comparing each new element in the input with the current tail as well as the tail of the previous sorted run (if one exists, i.e., this is not the first sorted run). If the new element lies between the current tail and previous tail, we can add it to the current sorted run and resume the loop. This allows us to quickly process a sequence of increasing elements in the input, which lie between the current and previous tail. This loop is terminated when we encounter an element that does not belong to the current sorted run. In our current implementation, we apply this optimization only to the first sorted run – this is usually the largest run for data such as logs, where elements are tardy. This allows us to avoid the second comparison with the tail of the previous run. Incorporating these optimizations produces our final P^3 sort variant, which we simply call P^3 sort.

Algorithm A2 shows P^3 sort. Line 8 shows the optimization that makes the algorithm cache-sensitive. Lines 14-17 depict (at a high level) the optimization to continue adding elements to the chosen tail. Finally, Line 18 invokes unbalanced ping-pong merge to complete the second phase of the algorithm. For clarity, we have excluded from this algorithm the optimization described in Section 2.2 for efficiently handling reverse sorted lists.

Algorithm A2: P^3 Sort

```
1  P3Sort(ElemsToSort: Array of comparable elements)
2    Runs: Array of sorted runs
3    Tails: Array of sorted run tails
4    Sizes: Array of sorted run sizes

5    CurElemIndex: Index of the element being processed
6    CurElemIndex = 0
7    while CurElemIndex < ElemsToSort.Size
8      Binary search the k highest indexed tails for the
           earliest which is <= ElemsToSort[CurElemIndex]
9      If there isn't such a tail
10       Add a new sorted run, with highest index,
         containing just ElemsToSort[CurElemIndex]
11       Update Tails and Sizes
12       Increment CurElemIndex
13     else
14       do
15         Add ElemsToSort[CurElemIndex] to found run
16         Increment CurElemIndex
17       while ElemsToSort[CurElemIndex] should be added to
             the chosen tail
18   UPingPongMerge(Runs, Sizes)
```

4.4 Evaluating P^3 Sort

Figure 23 shows the results of re-running our comparison with Quicksort, using P^3 sort instead of the cache-sensitive CS P^3 sort. First, note the dramatic overall improvement. P^3 sort is approximately 10 times faster than Quicksort when 5% or less of the data is disordered, regardless of the degree of disorder. As the dataset size increases, P^3 sort improves further against Quicksort due to its linear complexity for ordered data. In addition, observe that the random case has not degraded as a result of these in-order data optimizations. Finally, the total time taken to sort sorted data is now approximately the cost of 2 memory copies of the entire dataset. This indicates that there are no further opportunities to improve this case.

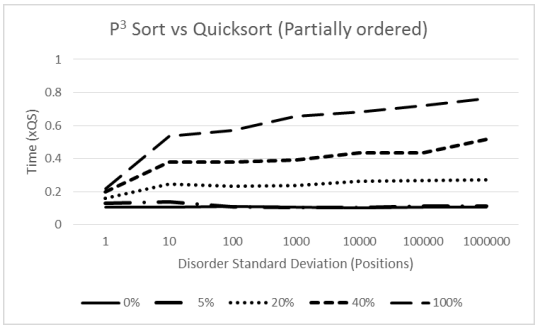

Figure 23: P³ Sort vs. Quicksort (partially ordered)

As mentioned earlier, Timsort [8] becomes linear as the data becomes ordered. It is currently recognized as the best in-memory technique for sorting almost sorted data. Timsort is conceptually related to Patience sort, with a run generation phase and a run merge phase. The run generation phase, however, only recognizes runs that are already contiguous in the input data. As it scans the data, there is only one active run which can be appended to, and there is, therefore, no binary search or tails array. In the popular implementation, run generation and run merging are commingled to create an approximately in place algorithm.

We therefore ported the Java implementation of Timsort to C++ in the most careful and straightforward possible way, preserving the optimizations in the existing implementation. There is no dynamic memory allocation, and the implementation is entirely array based, producing excellent sequential memory access patterns. The results of comparing Timsort to P³ sort are shown in Figure 24.

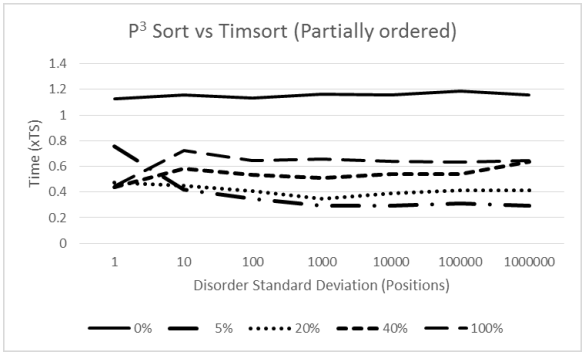

Figure 24: P³ Sort vs. Timsort (partially ordered)

First, note that in all cases except sorted data, P³ sort is faster than Timsort. For the case where 5% of the data is disordered by a large amount, P³ sort is between 3 and 4 times faster than Timsort. For in-order data, Timsort is approximately 10% faster than P³ sort, due to the fact that the Timsort implementation we used is in-place, and does substantially less memory copying, although it performs the same number of comparisons.

5. Improving Replacement Selection Sort

In this section, we combine replacement selection sort and P³ sort to efficiently sort almost-sorted datasets too large to fit in main memory. Such datasets have become commonplace in cloud applications, where timestamped application, user, and system telemetry data are usually dumped into large logs for subsequent processing. Since these logs combine information from distributed sources, network delays, intermittent machine failures, and race conditions introduce delays and jitter, which ultimately create time disorder in the stored log.

Temporal analytics are then typically performed over these logs. For instance, one may want to roll up historical behavior over time, or correlate events across time [7][17][18]. Such query processing typically requires that the log first be sorted on time.

In the past, replacement selection was used to reduce the number of external memory passes when more than two passes were required, with the hope of reducing the number of passes to a minimum of two. Here, we discuss and introduce methods for sorting almost sorted datasets in a single external memory pass. We show how P³ sort may be combined with replacement selection sort to minimize the CPU cost associated with single pass external sorting.

5.1 Replacement Selection Sort

Replacement selection [13] is the most well-known method of reducing the number of runs in an external sort by exploiting bounded disorder in the input data. This strategy scans the data from beginning to end, and stores the scanned data in a heap. When reading the next element from the input will overrun memory, the smallest element in the heap is removed and written to external memory, making space for the new element. If the disorder is bounded by the size of the heap, only a single run is written to disk, and no further passes over the data are needed.

We first evaluate the CPU cost of sorting in this manner, as compared to the cost of sorting the entire dataset with P³ sort. In this experiment, we sort 50 million 8 byte integers (400MB), all of which are pushed forward. The standard deviation of the number of positions pushed was varied from 1 to 1 million. For replacement selection, we also varied the size of the heap from 1MB to 256MB.

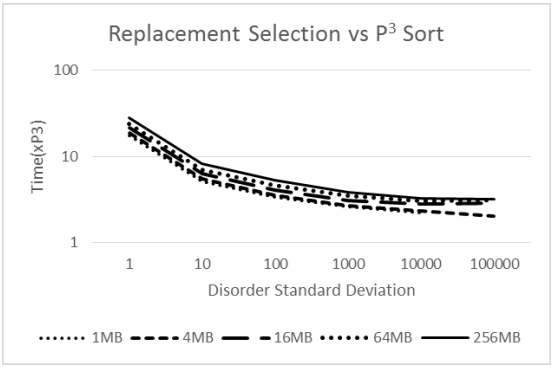

Figure 25: Replacement Selection vs. P³ Sort

The results are shown in Figure 25. The performance of replacement selection is pretty dismal, taking anywhere from 2x to 28x longer than P³ sort. This is due to a combination of two factors:

1) To the left, when disorder is small, P³ sort approaches linear complexity, while heaps are $O(n \cdot \log n)$ in all cases.
2) As observed earlier in the paper, heaps are expensive to maintain. Even for the most disordered case (stddev = 100k), when using a 256MB sized heap, replacement selection takes more than 3 times as long as P³ sort.

Unlike P³ sort, in replacement sort, the time taken is sensitive to the size of the heap, which determines the disorder tolerance. Higher tolerance when using replacement selection has higher CPU cost, even if the disorder level of the data is the same.

While this experiment doesn't involve reading and writing from disk, it provides a comparative upper bound on sorting throughput. All further experiments with replacement selection will be similarly focused on CPU costs.

5.2 Flat Replacement Selection (FRS) Sort

When replacement selection is used for run formation in external memory sorting, data is typically flushed, read, and enqueued in batches. This is done to optimize the bandwidth of external memory, which is generally block oriented. The resulting tolerance to disorder is the memory footprint minus the batch size. For instance, if the batch size is one quarter the memory footprint, the disorder tolerance is three quarters of the memory footprint.

We now introduce a variant of batched replacement selection, called flat replacement selection, which overcomes the two replacement selection deficiencies identified in the previous section. In particular, instead of maintaining a heap, we maintain a sorted list in a buffer. Initially, we fill the buffer with the first portion of the dataset, and sort it. We then flush the initial portion (determined by the batch size) of the sorted list, fill the empty portion of the list with the next portion of the input data, re-sort, and repeat until the entire dataset is processed. Algorithm A3 depicts this technique. Line 7 sorts the elements in the buffer, while Lines 8-12 write out BatchSize elements to the output. Lines 13-14 read more data in, and the process is repeated until the end of input.

```
Algorithm A3: Flat Replacement Selection Sort
1  FlatRSSort(InputElems: Input sequence,
2             OutputElems: Sorted output sequence,
              BatchSize: The # of elems in a batch)
3    Buffer: Array of k elements

4    ElemsToRead = min(InputElems.#Unread, k)
5    Read the first ElemsToRead elements of InputElems
         into Buffer
6    do
7      sort the elements in Buffer
8      if there are no more input elements
9        write the elements in Buffer to OutputElems
10     else
11       write the first Batchsize elements in Buffer to
              OutputElems
12       delete the first Batchsize elements in Buffer
13       ElemsToRead = min(InputElems.#Unread, k-Batchsize)
14       append the next ElemsToRead elements of InputElems
             into Buffer
15   while OutputElems hasn't had all elements written
```

This seems like a very straightforward idea, and we were surprised we didn't find any reference to something like it in the literature. In order to better understand this gap, we first tried this technique using Quicksort when we re-sort. We then reran the previous experiment, using replacement selection as the baseline, with a batch size of half the buffer. The results are shown in Figure 26.

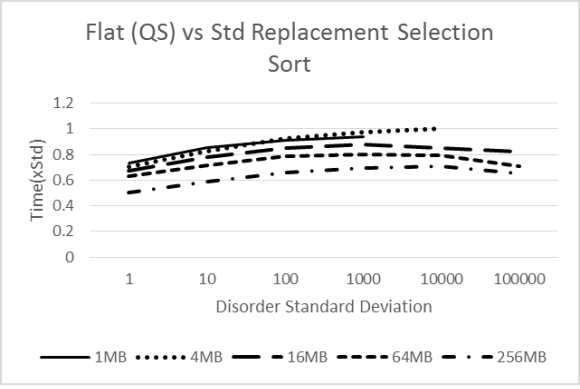

Figure 26: Flat (QS) vs. Standard Replacement Selection Sort

This is a significant improvement over classical replacement selection, particularly on modern hardware. Modern caches and memory hierarchies have been far kinder to Quicksort than replacement selection, which is based on heaps. Two decades ago, when there was far more interest in replacement selection, flat replacement selection was probably not an improvement over standard replacement selection. The fact that the two techniques are quite close for small buffer sizes is evidence of this.

If, on the other hand, we use a sorting technique which is linear on sorted data, like P^3 sort, when we re-sort, the already sorted portion of the data is simply copied into the correct final location. This should significantly improve upon standard replacement selection.

Note that both replacement selection deficiencies identified in the previous section are addressed: Because we are using P^3 sort, the cost of sorting is now nearly linear for nearly sorted data. Also, since there is no heap, the constant time inefficiencies associated with maintaining heaps are no longer relevant. We reran the previous experiment, using replacement selection as the baseline, with a batch size of half the buffer. The results are in Figure 27.

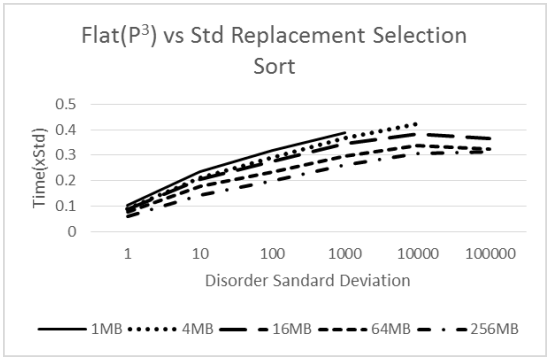

Figure 27: Flat (P^3) vs. Standard Replacement Selection Sort

The improvement in performance is now far more dramatic, ranging between a 3x and 10x speedup. Additionally, the performance gap narrows as disorder increased, and widens as disorder tolerance (buffer size) increases. As disorder increases, flat selection sort, which ultimately relies on P^3 sort, loses its linear advantage over heap sort on mostly sorted data.

On the other hand, as the buffer size increases, the extra memory copying associated with moving data around in the buffer for piecewise sorting decreases.

5.3 P^3 Replacement Selection Sort

We now introduce the final variant of replacement selection, which we call P^3 replacement selection. This sorting variant deeply integrates the batch replacement strategy into the P^3 sorting algorithm itself. Algorithm A4 shows P^3 replacement selection.

In particular, we begin by performing phase 1 of P^3 sort, until our memory budget is half used. By building a histogram over a sample of the data as we process it in phase 1, we determine the approximate median of all the data stored in the sorted runs (Lines 11-14). We then perform phase 2 of the P3 sort on the smallest half of the data, as determined by the median, and output the result (Lines 15-19). Furthermore, we remove the outputted data from the sorted runs held in memory. Note that this might result in the removal of some runs.

We then continue phase 1, processing new input until the memory taken by the sorted runs is once again half the memory footprint. We then repeat merging and flushing the smallest half of the data.

```
Algorithm A4: P3 Replacement Selection Sort
1  P3RSSort(InputElems: Input sequence,
2           OutputElems: Sorted output sequence,
            BatchSize: The # of elems in a batch)
3    RunSizeRefs : Array of (RunIndex, MergeSize) pairs
4    Runs: Array of sorted runs
5    Tails: Array of sorted run tails
6    Sizes: Array of sorted run sizes
7    k: The target maximum memory footprint in elements
8    SampleFreq: The number of elements between samples
9    Samples: Array of k/SampleFreq elements

10   While not all output has been written
11     perform phase 1 of P3 sort, correctly sampling,
           and stopping when either all input is consumed,
           or Samples is full
12     Sort Samples using P3 Sort
13     MergeVal = Samples[BatchSize/SampleFreq]
14     delete the first BatchSize/SampleFreq
           values from Samples
15     For each sorted run index i
          RunSizeRefs[i] = (i, # of elements <= MergeVal)
16     Sort ElemsToMerge by MergeSize
17     Pack the first ElemsToMerge.MergeSize elements of
           each run, in ElemsToMerge order, into the
           first ping pong array
18     Delete the first ElemsToMerge.MergeSize elements of
           each run, maintaining Tails, and Sizes
19     Use unbalanced ping pong merge to merge, writing the
           result to OutputElems
```

We continue alternating between phases 1 and 2 in this manner until all the data is processed.

P^3 replacement selection sort introduces the following additional work over P^3 sort:

- Maintains a sample of the input which resides in a sorted run
- Must sort the sample once per batch
- After phase 1, before we ping-pong merge, we don't know how much of each run is smaller than the median across the sample, we must therefore make an extra pass over the blocks of memory which hold the run, so that we can pack the partial runs into the merge buffer by size before ping-pong merge.
- Because some runs may become empty, we must compact the run pointer arrays after phase 2.

Observe, however, that compared to flat selection sort, we are significantly reducing sorted data movement (i.e. re-sorting sorted data). Also, we eliminate the overhead of repeatedly initializing and cleaning up the resources associated with calls to P^3 sort. We reran the previous experiment, using flat replacement selection sort as a baseline. The results are shown in Figure 28.

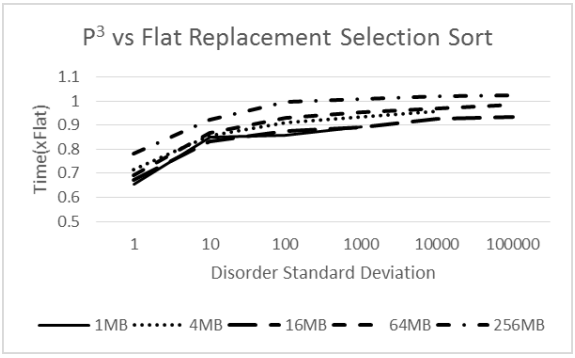

Figure 28: P^3 vs. Flat Replacement Selection Sort

Note that while not as dramatic as the introduction of flat replacement sort, there are, nevertheless, significant gains, especially when the level of disorder is low.

There are two trends worth discussing. The first trend is the closing of the performance gap as the memory footprint increases. As the memory footprint approaches the size of the dataset, the two algorithms behave very similarly, although there is some extra overhead in the P^3 replacement selection version. The second trend is that as disorder increases, the gap again closes. As disorder increases, both algorithms become $O(n \cdot \log n)$, and the linear time extra work associated with both techniques becomes irrelevant.

In our final comparison over synthetic data, we compare P^3 selection sort with P^3 sort over the entire dataset. The results are shown in Figure 29.

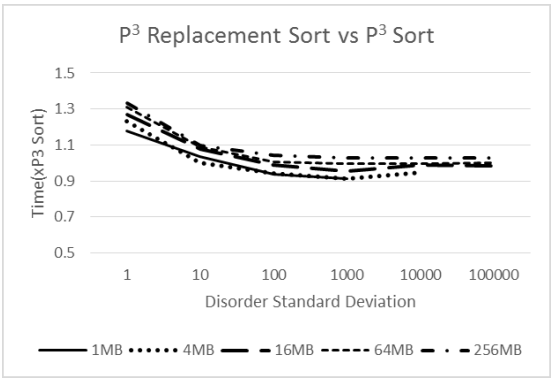

Figure 29: P^3 Replacement Sort vs. P^3 Sort

First, note that smaller memory footprints improve the performance of P^3 replacement sort. This is due to improved caching behavior.

In addition, as disorder increases, the two techniques converge to identical performance for the same reason as in the previous experiment: The two algorithms become identically dominated by their $O(n \cdot \log n)$ components. The additional constant time work performed by P^3 replacement sort therefore becomes insignificant.

It is interesting to note that these two effects combine, in some cases, making P^3 replacement sort *faster* than P^3 sort, despite the extra work. Even though both algorithms, in these cases, produce the same output, P^3 replacement sort is fundamentally not able to sort random datasets using a buffer smaller than the dataset size.

5.4 Replacement Selection, Batch Size, Memory, and Disorder Tolerance

For classical replacement selection, the disorder tolerance is the size of memory, the maximum of any technique presented here. On the other hand, there is an extreme sacrifice in efficiency which is made to achieve this robustness to disorder.

On the other hand, our flat replacement selection and P^3 replacement selection experiments chose a batch size of one half the buffer size. For flat replacement selection, this resulted in a disorder tolerance of half the buffer size, or half of available memory. By choosing a smaller batch size, for instance ¼ the buffer size, we could have increased the disorder tolerance to ¾ of main memory, but we would also have doubled the number of re-sorts, where each re-sort would move ¾ the buffer size of already sorted data instead of ½. This is clearly an unfortunate situation for flat replacement selection, where the cost of improving the disorder tolerance is very high.

P^3 replacement selection, in contrast, needs twice the batch size extra memory in order to perform ping-pong merge, and also needs to sort the sample every batch. If the memory buffer is large, say 1 GB, and 50 MB batches are merged at a time, the total memory needed is only 1.1 GB, and the disorder tolerance is ~95% of the buffer size. Note that the number of samples needed is dependent on the batch size as a percentage of the buffer size.

In order to better understand the effect of smaller batch sizes on P^3 replacement selection, we re-ran the previous experiment with a buffer size of 128MB, and varied the batch size. We used a sampling frequency of 1024. This resulted in 16K samples, which is more than enough for our smallest batch size of 1%. The results are shown in Figure 30.

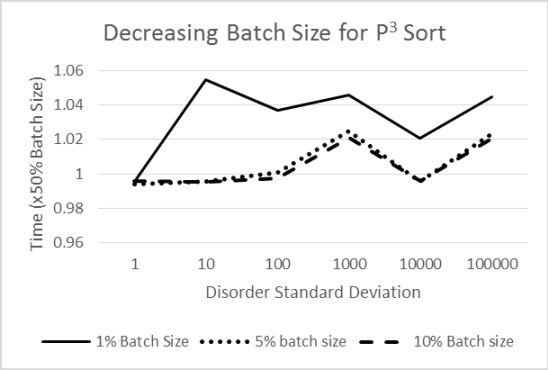

Figure 30: Decreasing Batch Size for P^3 Replacement Selection

For batch sizes of 12MB and 6MB, there is no measureable impact of using a smaller batch size (compared to 50MB). At a batch size of 1.2 MB, we are just beginning to see the effect of reducing batch size. As a result, one can use P^3 replacement selection as a much more performant alternative to classical replacement selection, without significant adverse effects on either memory footprint or disorder tolerance.

6. RELATED WORK

Sorting has a long history, even predating computer science [9][13]. Over the years, there have been many algorithms, each with their own unique requirements and characteristics. In this paper, we focus on two cases of high practical value: (1) In-memory, single node comparison based sorting of randomly ordered data; and (2) Single node comparison based sorting of almost ordered data (in-memory and external).

For in-memory single node comparison based sorting of randomly ordered data, Quicksort [14] remains the most commonly implemented technique, due to both its high efficiency and ease of implementation. Quicksort also has excellent cache performance, focusing for long periods of time on small subsets of the data. Where appropriate, we compare our proposed sorting techniques to the GNU C++ implementation of Quicksort [11], which includes the four popular Sedgewick optimizations [10] described earlier. For in-memory single-node comparison-based sorting of almost ordered data, Timsort [8] has emerged as the clear winner in prior work. Like Patience sort, Timsort is $O(n \cdot \log n)$ in the worst case, and is linear on sorted data. Timsort is the system sort in Python and is used to sort arrays of non-primitive type in Java SE 7, on the Android platform, and in GNU Octave [8]. Timsort has a very popular and heavily optimized implementation in Java.

The basic idea of Timsort is to recognize and merge existing sorted runs in the input. Unlike Patience sort, only one run can be added to at any point in the algorithm. When a data element is processed, it either extends the current run, or starts a new one. The run recognition and merge phases are commingled cleverly in order to sort the data efficiently, and approximately in-place. Where appropriate, we compare our proposed sorting techniques in this paper to our careful port of Timsort's Java implementation to C++.

Bitonic sorting [19] has emerged as a popular sorting technique, but its benefits are mostly limited to massively parallel GPU architectures. Chhugani et al. [20] show how to exploit SIMD and modern processor architectures to speed up merge in the context of merge sort and bitonic sorting. Efficient merging techniques have also been investigated in the context of merge joins with modern hardware [24][23][22]. For instance, Balkesen et al. [22] argue that merging more than two runs at once is beneficial, while using a tree of binary merges to perform the merge. Like ping-pong merge, all these techniques use binary merges instead of heaps, but focus on taking advantage of multiple cores and processor-specific features such as SIMD. Further, they do not target or optimize for almost-sorted datasets. We focus on general single-core processor-agnostic techniques in this paper, and believe that multiple cores and processor-specific techniques can be adapted to make P^3 sort even faster; this is a rich area for future work (see Section 7).

The most related previous work is Patience sort itself [3]. The name Patience sorting (Patience is the British name for solitaire) comes from Mallows [15], who in [3] credits A.S.C. Ross for its discovery. Mallows' analysis was done in 1960, but was not published until much later. Aldous et al. also point out in [1] that Patience sorting was discovered independently by Bob Floyd in 1964 and developed briefly in letters between Floyd and Knuth, but their work has apparently not been published. Hammersley [16] independently recognized its use as an algorithm for computing the length of the longest increasing subsequence. More recently, Gopalan et al. [5] showed how Patience sort can be used to estimate the sortedness of a sequence of elements.

P^3 sort uses ping-pong merge for merging sorted runs in memory. Ping-pong merge and its run ordering for the unbalanced case draw motivation from the early tape-based merging techniques described by Knuth [13]. A key difference is that main memory buffers allow simultaneous reads and writes, which allows us to perform the merge with just two ping-pong buffers and execute "blind merges" when merging runs. Moreover, our run-ordering targets a different need – that of making merge extremely lightweight for highly skewed runs generated from almost sorted data by phase 1 of Patience sort. Other merging approaches proposed in the past include the classic heap-based approach such as the selection tree algorithms from Knuth [13]. Wickremesinghe et al. [4] introduced a variant of these algorithms, which uses a tree of priority-queue based merges, limiting the size of the heap in order to improve cache behavior. An extensive comparison to this technique is presented in Section 3.

Further contributions of this paper include two new variants of replacement selection [13][21], a technique for reducing the number of sorted runs when performing external-memory-based sort-merge. The potential importance of replacement selection and its variants has become especially acute due to the plethora of almost sorted telemetry logs generated by Big Data and Cloud applications, where network delivery of data introduces jitter and delay [7][17][18]. In many of these cases, the number of runs can

be reduced to 1, eliminating the second pass of sort-merge entirely. Unfortunately, the CPU costs associated with such techniques, which are heap, or tree based [13][4], are an order of magnitude higher than conventional high performance sorting techniques, significantly limiting the achievable throughput.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have reexamined and significantly improved upon Patience sort, a 50+ year old sorting technique mostly overlooked by the sorting literature. In particular, we have introduced both algorithmic, and architecture-sensitive, but not architecture-specific, improvements to both the run generation phase and the run merging phase. For the run merging phase, we have introduced a new technique for merging sorted runs called Ping-Pong merge. The result is a new sorting technique, called Ping-Pong Patience Sort (P^3 Sort), which is ~20% faster than GNU Quicksort on random data, and 20%-4x faster than our careful C++ port of the popular Java implementation of Timsort on almost ordered data.

This paper also investigates new opportunities for replacement selection sort, which can be used to sort many external memory resident datasets in a single pass. In particular, we introduce two new variants of replacement selection sort, which integrate P^3 sort into replacement selection sort in two different ways. The faster approach, which more deeply integrates P^3 sort into selection sort, improves CPU performance/throughput by 3x-20x over classical replacement selection sort, with little effect on either memory footprint or disorder tolerance.

This work is the beginning of several research threads. The observation that Patience sort, a mostly overlooked sorting technique, can be the basis of a highly competitive sort algorithm (P^3 sort) is new. Undoubtedly there will be other interesting innovations to come, further improving on the bar for Patience sort variants established in this paper. For instance, a related sort technique, Timsort, was able to be algorithmically restructured in a way which made it almost in place. A similar optimization may be possible with P^3 sort, improving P^3 sort's utility when main memory is limited.

Inside a DBMS, P^3 sort is also applicable as a sorting technique that can automatically exploit the potential of efficiently sorting a dataset by a new sort order that is closely related to an existing sort order (for example, when a dataset sorted on columns {A, B} needs to be sorted on column {B}, and A has low cardinality).

Also, Patience sort's explicit decomposition of sorting into run generation and run merging forms an intriguing basis for an investigation into multicore, SIMD, and distributed parallel execution with Patience sort. In fact, it is immediately clear that some of the architecture specific innovations, like the use of SSE instructions, described in [20] could be applied to ping-pong merge.

Finally, there has been very little interest in replacement selection sort and its variants over the last 15 years. This is easy to understand when one considers that the previous goal of replacement selection sort was to reduce the number of external memory passes to 2. Since, the size of 2 pass sortable (without replacement selection sort) datasets increases quadratically with the size of main memory, as main memories have grown, the value of replacement selection sort has drastically diminished.

Replacement selection sort, however, now has the opportunity, for many logs, which typically have bounded disorder, to reduce the number of passes from 2 to 1. This paper represents the first work in that direction, which again, is likely to be improved upon.

ACKNOWLEDGEMENTS

We would like to thank Isaac Kunen, Paul Larson, Yinan Li, Burton Smith, and the anonymous reviewers for their comments, advice, and support.

8. REFERENCES

[1] David Aldous and Persi Diaconis. Longest increasing subsequences: from Patience sorting to the Baik-Deift-Johansson theorem. Bull. of the Amer. Math. Society, Vol. 36, No. 4, pages 413–432.

[2] Sergei Bespamyatnikh and Michael Segal. Enumerating Longest Increasing Subsequences and Patience Sorting. Pacific Inst. for the Math. Sci. Preprints, PIMS-99-3., pp.7–8.

[3] C. L. Mallows. "Problem 62-2, Patience Sorting". SIAM Review 4 (1962), 148–149.

[4] Rajiv Wickremesinghe, Lars Arge, Jeffrey S. Chase, Jeffrey Scott Vitter: Efficient Sorting Using Registers and Caches. ACM Journal of Experimental Algorithmics 7: 9 (2002).

[5] P. Gopalan et al. Estimating the Sortedness of a Data Stream. In SODA 2007.

[6] A. LaMarca and R.E. Ladner. The influence of caches on the performance of sorting. Volume 7 (1997), pp. 370-379.

[7] B. Chandramouli, J. Goldstein, and S. Duan. Temporal Analytics on Big Data for Web Advertising. In ICDE 2012.

[8] TimSort. http://en.wikipedia.org/wiki/Timsort.

[9] Sorting Algorithms. http://en.wikipedia.org/wiki/Sort_algorithms.

[10] R. Sedgewick. Implementing Quicksort programs. Comm. ACM 21 (10): 847–857.

[11] GNU Quicksort Implementation. http://aka.ms/X5ho47.

[12] Patience Sorting. http://en.wikipedia.org/wiki/Patience_sorting.

[13] Donald Knuth. The Art of Computer Programming, Sorting and Searching, Volume 3, 1998.

[14] C.A.R. Hoare. Quicksort. Computer J. 5, 4, April 1962.

[15] C.L. Mallows. Patience sorting. Bull. Inst. Math. Appl., 9:216-224, 1973.

[16] J.M. Hammersley. A few seedlings of research. In Proc. Sixth Berkeley Symp. Math. Statist. and Probability, Volume 1, pages 345-394. University of California Press, 1972.

[17] M. Kaufmann et al. Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA. In SIGMOD, 2013.

[18] Splunk. http://www.splunk.com/.

[19] K. E. Batcher. Sorting networks and their applications. In Spring Joint Computer Conference, pages 307–314, 1968.

[20] J. Chhugani et al. Efficient Implementation of Sorting on MultiCore SIMD CPU Architecture. In VLDB, 2008.

[21] P. Larson. External Sorting: Run Formation Revisited. IEEE Trans. Knowl. Data Eng. 15(4): 961-972 (2003).

[22] C. Balkesen et al. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. In VLDB, 2014.

[23] M.-C. Albutiu et al. Massively parallel sort-merge joins in main memory multi-core database systems. In VLDB, 2012.

[24] C. Kim et al. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. In VLDB, 2009.