Adaptive Data Partition for Sorting using Probability Distribution

Xipeng Shen and Chen Ding
Computer Science Department, University of Rochester
{xshen,cding}@cs.rochester.edu

Abstract

Many computing problems benefit from dynamic partition of data into smaller chunks with better parallelism and locality. However, it is difficult to partition all types of inputs with the same high efficiency. This paper presents a new partition method in sorting scenario based on probability distribution, an idea first studied by Janus and Lamagna in early 1980's on a mainframe computer. The new technique makes three improvements. The first is a rigorous sampling technique that ensures accurate estimate of the probability distribution. The second is an efficient implementation on modern, cache-based machines. The last is the use of probability distribution in parallel sorting. Experiments show 10-30% improvement in partition balance and 20-70% reduction in partition overhead, compared to two commonly used techniques. The new method reduces the parallel sorting time by 33-50% and outperforms the previous fastest sequential sorting technique by up to 30%.

1 Introduction

Many types of dynamic data have a total ordering and benefit from partial sorting into sublists. Examples include N-body simulation in physics and biology studies, where particles are partitioned based on their coordinates, and discrete-event simulation in computer networking and economics, where events are ordered by their arrival time. Partial sorting or partition allows these large scale problems to be solved by massively parallel computers. Data partition is also important on machines with a memory hierarchy because it dramatically improves cache performance for in-core data and memory performance for out-of-core data. Therefore, the efficient and balanced data partition is critical to good parallelism and locality.

Most previous methods either have a high overhead or only apply to uniformly distributed data (as discussed in Section 2.) For example, Blelloch et al. showed that one of the fastest pivot-based methods, *over-sampling*, consumed 33-55% of the total sorting time [2]. It remained an open question whether we could find balanced partitions for non-uniform data in linear time.

In early 80s, Janus and Lamagna published a method that first samples the data and estimates its cumulative distribution function (CDF); it then assigns data into equal-size (not necessarily equal-length) buckets through direct calculation [10]. This simple idea achieves balanced data partition in linear time, even for non-uniform distributions. We call the partition method *PD-partition* and its use in sorting *PD-sort* in short. Janus and Lamagna implemented their algorithm using the PL/1 language and measured the performance on an IBM 370 machine. Since then, however, this method seems forgotten and is rarely mentioned by later studies of sorting performance on modern cache-based machines.

In search for a better sorting method for unbalanced data sets, we independently discovered the idea of using probability distribution. Compared to the method of Janus and Lamagna, this work makes three improvements. The first is a rigorous sampling method that ensures accurate estimate of the probability distribution. Janus and Lamagna did not address the sampling problem for non-standard distributions. They used a fixed number of samples (465) in their experiments [10]. Using the sampling theory, our method guarantees the statistical accuracy for a large class of non-uniform distributions [21], with the number of samples independent of the size of data and number of categories.

The second contribution is an efficient implementation of probability calculations on modern machines. It uses temporary storage to avoid repeated calculations. It uses scalar expansion to improve instruction-level parallelism and hide memory latency. The latter two problems did not exist on machines used by Janus and Lamagna. Our implementation is 19-28% faster than the base implementation of *PD-partition*.

Finally, we measure the effect of *PD-partition* in sequential and parallel sorting and compare it with the fastest sorting methods in the recent literature. Our *PD-partition* outperforms other partition methods in both efficiency and balance. Our sorting implementation outperforms quick-sort

by over 10% and outperforms other cache-optimized algorithms by up to 30% [11, 22].

Furthermore, we designed a parallel sorting algorithm using *PD-partition*. It achieves 33-50% time saving compared to parallel sorting using *over-sampling* and *over-partitioning*, two techniques popular on modern systems (see Section 2).

In the rest of this paper, we review related work in Section 2 and describe PD-partition in Section 3, the statistical sampling in Section 4, and an optimized implementation in Section 5. We present an evaluation in Section 6, and conclude with a summary of our findings.

2 Related work

Data partition is a basic step in program parallelization on machines with distributed memory. Many applications use irregular data, for example, particles moving inside a space or a mesh representing a surface. Parallelization is often done by an approach called inspector-executor, originally studied by Saltz and his colleagues [5]. For example, in N-body simulation, the inspector examines the coordinates of particles and partitions them into different machines. Much later work used this model, including the language-based support by Hanxleden et. al. [8], the compiler support by Han and Tseng [7], the use on DSM [15], and many others that are not enumerated here. When data can be sorted, the partition problem in N-body simulation can be viewed as a sorting problem. Instead of finding balanced sublists, we need to find subspaces with a similar number of data. Mellor-Crummey et al. [16] and Mitchell et al. [17] used bucket sort to significantly improve the cache locality of N-body simulation programs. Bucket sort produces balanced partitions for uniformly distributed data, but may produce severely unbalanced partitions for highly skewed distributions.

Parallel sorting [1, 2, 3, 4, 6, 9, 13, 20] is an important application of data partition. Recent work includes NOWSort by Arpaci-Dusseau et al. [1], $(l; m)$-merge sort by Rajasekaran [18], an implementation of column-sort by Chaudhry et al. [4], and parallel sorting on heterogeneous networks by Cerin [3]. Most of these methods use pivots to partition data. Three main ways to pick pivots are as follows. The term p represents the number of processors.

- *Regular-sampling (RS)* [20, 14]. Each processor sorts the initial local data. It picks p equally spaced candidates. All p^2 candidates are then sorted to pick $(p-1)$ equally spaced pivots.
- *Over-sampling (OS)* [2, 9]. The $p * r$ random candidates are picked from the initial data, where r is called the *over-sampling ratio*. The candidates are sorted to pick $(p-1)$ equally spaced ones as the pivots.
- *Over-partitioning (OP)* [13]. The $p * r * t$ candidates are picked randomly from the whole data set, where t is called *over-partitioning ratio*, and r is the same as in *Over-sampling*. The candidates are sorted, and $(pt-1)$ pivots are selected by taking $r^{th}, 2r^{th}, \cdots, (pt-1)r^{th}$ candidates. The whole data set is partitioned into $p * t$ sublists, which form a task queue for parallel processors.

Pivot-based methods have a significant time overhead. They sort candidates and use a binary search to find a suitable bucket for each data. To balance the partitions, they need a large number of candidates. As we will discuss later, *PD-partition* is more efficient for parallel sorting because it does not sort samples and calculates the bucket assignment with only a constant number of operations.

Instead of using pivots, Arpaci-Dusseau et al. partition data into even length buckets [1]. Xiao et al. used a similar algorithm—*inplaced flash quicksort* [22], which utilizes an additional array to reuse elements in a cache line. Both methods partition data in linear time, but both assume that the input data have a uniform distribution. For other distributions, the partitions may be severely unbalanced.

Our method provides a way to partition data efficiently and yields good balance even for non-uniform distributions. It extends the work by Janus and Lamagna [10] in three ways. *PD-partition* depends on the accurate estimation of the cumulative distribution function. Janus and Lamagna did not show how to ensure accurate estimation for non-standard input distribution. Our work solves this problem using the sampling theory [21]. Janus and Lamagna designed their algorithm for minimizing the total number of instructions in sequential sorting. We adapt the algorithm to maximize the parallelism and locality in dynamic data partition. We also give an optimized implementation that significantly reduces the partition overhead on modern machines. In addition, we evaluate dynamic data partition in the context of parallel sorting against popular parallel sorting methods. We also demonstrate fast sequential sorting of non-uniform inputs (faster than quick-sort and other cache-optimized sorting algorithms) on modern machines with a deep memory hierarchy.

3 Probability distribution-based partition algorithm

Probability Distribution-based partition (*PD-partition*), similar to the extended *distributive partitioned sorting* by Janus and Lamagna [10], has two steps: the selection of buckets (by estimating the *Cumulative Distribution Function or CDF*) and the assignment of data into buckets. The complexity of the two steps is linear to the size of the input.

3.1 CDF estimation

The estimation of CDF includes three steps [10]:

1) Traverse data to find the range of data. Divide the range into c equal-length *cells*. Select s random samples from the data. Distribute the samples into the cells. Let s_i be the number of samples in cell i. To make $s_i > 0$ (well-behaved), always add 1 to s_i. Let $sc = s + c$. Figure 1(b) shows the first step. The height of each bar is the number of samples contained in each cell.

2) Take $\frac{s_i}{sc}$ as the probability of a randomly picked data value belonging to cell i. The cumulative probability $p_1, \cdots, p_c$ is therefore a cumulation of $\frac{s_i}{sc}$ for $i = 1, 2, \cdots, c$. Figure 1(c) shows the cumulation step.

3) To get CDF, the third step fits a line between each adjacent pair of cumulative probabilities. It saves the y-intersect of each line to get an estimate of the CDF of the total data. Figure 1(d) shows the final step.

The time cost of CDF estimation is linear to the number of samples. This estimation requires *well-behaved* distributions, i.e. the CDF is continuous and monotonic increasing.

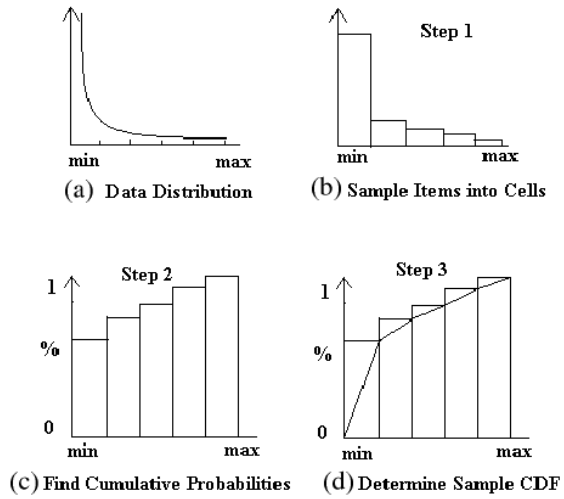


Figure 1. Estimation of CDF

3.2 Assignment of data into buckets

The second step assigns data into buckets. For each data element x, the algorithm finds the bucket assignment in three calculations. First it finds the cell number of x. Recall that during CDF estimation, the range between the maximum and minimum of the total data is divided into c equal-length cells. The cell number of x is therefore the floor of $(x - min)/lc$, where lc is the cell length. The second step finds p_x, the cumulative probability or CDF of x. It equals to the cumulative probability of its preceding cell plus the cumulative probability of elements smaller than x in this cell. The latter term is calculated based on the slope of the cell. The calculation assumes a uniform distribution within each cell.

Using the cumulative probability p_x, we can get the bucket number of x in one calculation. Let b be the number of buckets. Since we want balanced partitions, the buckets should have an equal size. In other words, each bucket has the equal probability, $1/b$, for x to fall into. Thus, the bucket number of x is $\lfloor p_x / \frac{1}{b} \rfloor$ or $\lfloor p_x * b \rfloor$. All three steps take a constant amount of time. Therefore, the time complexity of the bucket assignment is $O(n)$, where n is the size of the input data. Figure 2 shows the algorithm for assigning data into buckets.

4 Statistical measurement of CDF estimation

The accuracy of CDF estimation strongly effects the partition balance of *PD-partition*. An inaccurate CDF estimation may result in severely unbalanced partition. In the worst case, most data fall into a single partition and the performance degrades to that of quick-sort plus the partition overhead. The accuracy of CDF estimation is determined by the number of samples used for the CDF estimation. Janus and Lamagna used 465 samples in their sequential sorting algorithm without formal statistic analysis. In our experiments, we found it too few to generate accurate estimations. Based on sampling theory [21], we provide a way to determine the sample size with high accuracy guarantee.

We model the problem as a multinomial proportion estimation [21]— a problem to find the smallest number, s, of random samples from a multinomial population (i.e. a population including multiple categories) such that with at least $1 - \alpha$ probability the estimated distribution is within a specified distance of the true population, that is,

$$Pr\{\bigcap_{i=1}^{k} \mid p_i - \pi_i \mid \leq d_i\} \geq 1 - \alpha \qquad (1)$$

where k is the number of categories in the population, p_i and π_i are the observed and the actual size of category i, d_i and α are the error bounds given. Thompson proposes the following formula for s [21]:

$$s = \max_m z^2(1/m)(1 - 1/m)/d^2 \qquad (2)$$

where m is an integer from 0 to k, z is the size of the upper $(\alpha/2m) * 100$th portion of the standard normal distribution, and d is the distance from the true distribution.

The m that gives the maximum in Formula (2) depends on α. Thompson shows, in the worst case, $m = 2$ for $0 \leq \alpha < .0344$; $m = 3$ for $.0344 \leq \alpha < .3466$; $m = 4$ for $.3466 \leq \alpha < .6311$; $m = 5$ for $.6311 \leq \alpha < .8934$; and $m = 6$ for $.8934 \leq \alpha < 1$ [21]. Note that for a given d and α, s is independent to the size of data and the number of categories k.

In the CDF estimation, each cell is a category. The CDF value in a cell is the size of this and all other cells of smaller values. Formula (2) gives the minimal size of samples for a given confidence $(1 - \alpha)$ and distance d. Suppose we want the probability to be at least 95% that CDF values are within 0.01 distance of the true distribution, Formula (2) gives the minimal sampling size 12736. In our experiments, we use 40000 samples, which guarantees with 95% confidence that the CDF is within 0.0056 distance (i.e. 99.4% accurate).

5 Optimizations in implementation

It is important to make the bucket assignment as fast as possible. In our implementation, we make two optimizations. In the loop of Figure 2, there are four floating-point computations. Let $bucketNum[i]$ be the bucket number covered by the range from the minimal data to the end of cell i. Let $bucketNum1[i]$ be the number of buckets covered by cell i, which is equal to $bucketNum[i] - bucketNum[i-1]$. We store these two numbers for each cell instead of recomputing them. Using the stored numbers, the assignment of each datum can be simplified to two instead of four floating-point computations. The second optimization is scalar expansion inside the assignment loop to increase the parallelism and hide the calculation latency. For lack of space, we leave the detail algorithm in a technical report [19]. The evaluation section 6.2 will show that the optimizations accelerate the PD-sort by 19-28%.

Notations: in the following discussion, n is the total number of data, s the total number of samples, c the number of cells, and b the number of buckets. In parallel sorting, b is the number of processors.

6 Evaluations

We first measure the efficiency and balance of *PD-partition* and compare them with those of other partition methods. We then use them as sorting methods by applying quick-sort within each bucket. We compare their speed in sequential and parallel sorting.

Our experiments were conducted on Linux machines with 2.0GHz Pentium 4 processors and 512MB main memory. The size of the second level cache was 512KB. All methods were compiled by $gcc\ -O3$. They sorted randomly generated integers of different distributions. For

```
/* data: the array of data, size is N;
   min,max: the minimum and maximum of data;
   C: number of cells in range [min, max];
   lc: the length of each cell;
   cdf[i]: the cdf of ith cell;
   cdf[0] = 0 and cdf[c] = 0.9999999;
   slope[i]: the slope of the fitting line in cell i;
   B: the number of buckets */
  .....
  lcR = 1/lc;
  for (int i=0;i<N;i++){
      /* find the cell number of data[i] */
       int n = (int)((data[i] - min)*lcR);
       /* find the cdf of data[i] */
       float l = data[i]-min-n*lc;
       float xcdf = cdf[n]+slop[n]*l;
      /* bucket number of data[i] */
       int bucketNum = (int)(xcdf*B);
       /* put data[i] into a new array corresponding to its bucket-
Num */
       .....}
```

Figure 2. Assigning data to buckets

PD-partition and sorting, unless otherwise noted, the cell number was 1000, sample size was 40000, bucket number was 128, and the input included 64 million integers. For each type of the distribution, the result is the average of 20 randomly generated data sets. The uniformly distributed data were generated by function $random()$ in the standard C library; the normally distributed data were generated by $problib$, a statistics library [12]. Our technical report includes the results of additional types of distributions [19].

6.1 Data partition

We show the partition results from *PD-partition*, *over-sampling* [2, 9] and *over-partitioning* [13]. *Regular sampling* [20, 14] has more overhead and is not included in the evaluation. In *over-sampling*, the over-sampling rate is 32 (as in [2]). In *over-partitioning*, the over-sampling rate is 3 and the over-partitioning rate is $\log b$ (as in [13]), where b is the number of buckets. Section 2 describes the three algorithms in more detail.

We measure their speed by the partition time, and measure the partition balance using a concept called *bucket expansion* (BE), which is the ratio of the largest size to the average size of all buckets. It measures the worst-case (im)balance. The ratio is equal to or greater than 1. A ratio of 1 means perfect balance because all buckets have the same size.

We use the relative balance and speed in the evaluation to compare all three methods in a single figure in Figure 3. The following factors affect the partition balance and cost.

Effects of data distribution We use uniform and three normal distributions to study the effects of data distribution, shown in Figure 3(a). Our method takes less than 80% and 60% time of *over-sampling* and *over-partitioning* and achieves better balance. The uniform distribution has the greatest gain because a uniform distribution is estimated better by samples than heavily skewed distributions are. A problem shown is that the bucket ratio of our method becomes worse when the distribution becomes less uniform. The problem can be significantly alleviated by using more samples as explained in Section 4.

Effects of data size and bucket number Figure 3(b,c) show the effect of the data size and bucket number on the uniform distribution. Our method reduces partition overhead from $O(Nlog\,b)$ to $O(N)$. Thus, the time ratio is $\frac{O(N)}{O(Nlog\,b)}$, which is independent of N but proportional to $\frac{1}{log\,b}$, consistent with Figure 3(b,c). Figure 3(c) also shows that PD-partition is more scalable and produces larger number of buckets faster with better balance than *over-sampling* and *over-partitioning*.

Effects of the number of samples and cells More cells allow the CDF fitting at a finer granularity (see Figure 1.) More samples yields better estimation for each cell. The number of samples depend on the number of cells because more cells require more samples, as shown in Formula 2. In Figure 3(d), we use 1000 cells but vary the number of samples from 5000 to 80000. We show results on uniform distribution and other distributions have similar results. The figure shows that PD-based partition obtains greater improvement in balance from using more samples than the other two methods do. The time of PD-based sorting increases slightly faster than the other two methods as the number of samples increases. But the overhead is small even for a larger number of samples. The time ratios are still less than 0.6 and 0.4.

Figure 3(e) shows the changes of the number of cells. For a normal distribution (mean=3000,deviation=300.), the partition balance is improved when the number of cells is increased from 100 to 1600. It is worsened when the number of cells is increased from 1600 to 12800, because the samples in a cell become too few to estimate the probability. Figure 3(f) shows the effects using the uniform distribution. The balance ratio decreases monotonically as the number of samples increases, because a small number of samples is enough for estimating CDF in this case.

6.2 Comparison with sequential quick-sort

Quick-sort is believed by many to be the fastest internal (in-memory) sorting method. A recent study by Xiao et al. shows that for uniformly distributed input data, simple partition is faster. But for non-uniform data distributions,

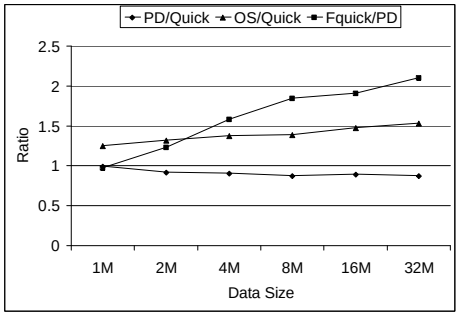


Figure 4. Comparison of the sequential sorting time on normal distribution (mean=3000, std=1000). PD: PD-sort; Quick: quick-sort; OS: *over-sampling*. **Fquick: inplaced flash quick-sort.**

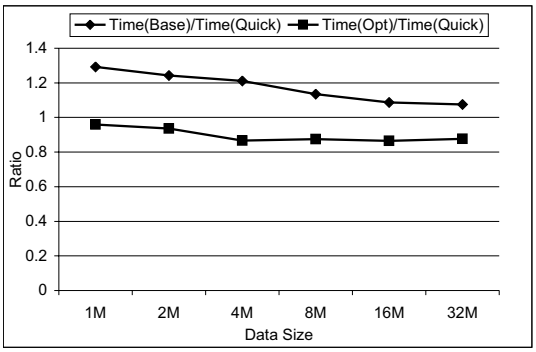
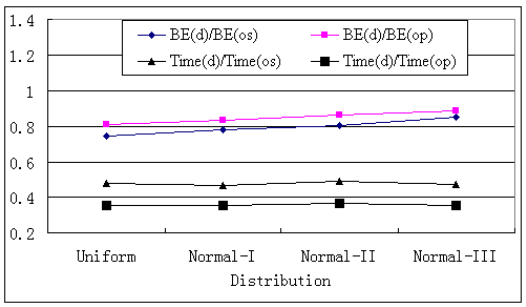


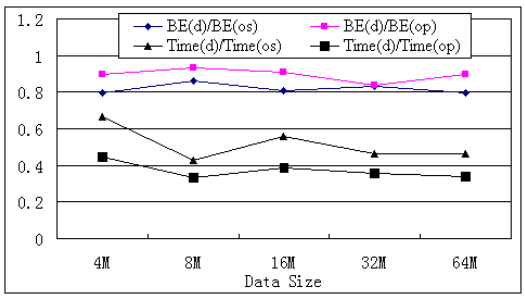
Figure 5. The effect of algorithm and implementation improvement to *PD-partition*. $Base$ **is our basic data assignment algorithm.** $Opt.$ **is the optimized version.** $Quick$ **is quick-sort. The input data are in normal distribution (mean=3000, std=1000).**

Table 1. Time for Sequential Sorting (sec.)

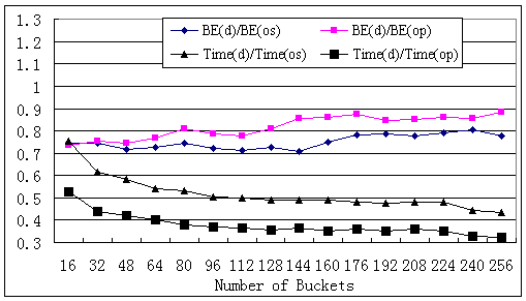
Data Size	PD-sort	Quick-sort	Over-sampling sort
2M	0.334	0.365	0.480
4M	0.647	0.719	0.985
8M	1.298	1.491	2.064
16M	2.624	2.951	4.341
32M	5.227	6.008	9.197



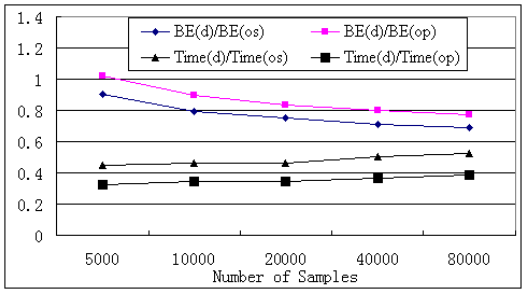
(a) Comparison on the uniform distribution, and three normal distributions (m=3000, d=3000,1000,300)



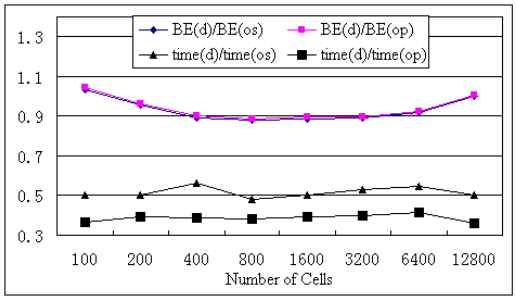
(b) Effect of data size on the uniform distribution



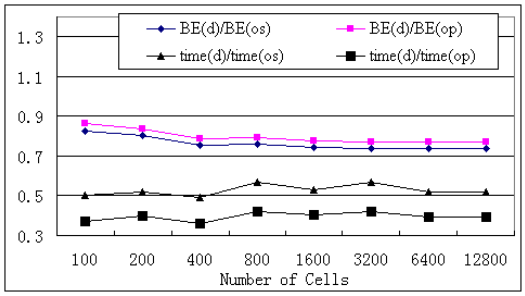
(c) Effect of the number of buckets on the uniform distribution



(d) Effect of the number of samples on the uniform distribution



(e) Effect of the number of cells on the normal distribution (m=3000, d=300)



(f) Effect of the number of cells on the uniform distribution

Figure 3. Evaluating the major factors in data partition. BE: Bucket Expansion; d: *PD-partition*; **os:** *over-sampling*; **op:** *over-partitioning*. **The balance and speed ratios are lower than 1, showing that** *PD-partition* **is more balanced and efficient than the** *over-sampling* **and** *over-partitioning*.

quick-sort still gives the best performance because simple methods could not give balanced partitions [22]. We now show that PD-sort outperforms quick-sort for uniform and non-uniform distributions.

We use PD-partition to cut data into blocks smaller than the cache size and then sort each block. Since the block partition takes linear time, it has better cache locality than the recursive partition schemes like quicksort, when data size is much greater than the cache size.

We target buckets of the size 256KB in the partition methods. It is less than the real cache size (512K) to leave room for other data and to reduce cache interference. Figure 4 compares the speed on normal distribution with mean of 3000 and variance of 1000. Other distributions have similar results [19]. Table 1 shows the sorting time. We also show *over-sampling* for comparison. For sequential sorting, *over-partitioning* is similar to *over-sampling* except using more buckets.

When sorting more than four million data, PD-sort outperforms quicksort by more than 10%. In comparison, *over-sampling* is slower than quick-sort because of the high partition overhead. The speed gap widens on larger data inputs.

The algorithm and implementation improvement described in Section 5 are critical: without them the PD-sort is no faster than quicksort. Figure 5 shows 19-28% time reduction in the overall sorting time. The optimization is designed to speed up the probability calculation and therefore not applicable to quicksort.

We also compared our method with *Inplaced flash quicksort*, proposed by Xiao et. al [22]. It partitions data assuming a uniform distribution and then sorts each bucket using quick-sort. It takes an additional array to reuse elements in a cache line. It was shown to be faster than many other sequential sorting methods [22]. Our method is slightly faster than the inplaced flash quick-sort for data sizes smaller than 4 million and about 30% faster when sorting larger data sets.

6.3 Parallel sorting

We compare the parallel performance of PD-sort with sorting methods using *over-sampling* and *over-partitioning*. In the absence of a large scale parallel computer, we analyze the communication costs and implement a simulator to measure the computation costs. All three algorithms perform data partition on a single processor and then sort each sub-list on a parallel processor. The communication cost has two parts: the cost to obtain pivots or CDF, and the cost to move data to their assigned processors. The second part is similar in all methods. The first part is negligible as shown in [19]. Assuming the same communication cost, we use the computation costs to measure the performance of parallel sorting.

Figure 6 compares the sorting speed on a normal distribution (mean=3000, std=1000.) The other distributions show similar results [19]. In the experiment, we assume that the data are perfectly partitioned by all three algorithms, so the computation time is the sorting time of D/P integers. Section 6.1 shows better partition balance by PD-sort than by *over-sampling* and *over-partitioning*. Since the actual computation time is determined by the size of the largest bucket, the perfect balance assumption grants higher benefits to *over-sampling*. For *over-partitioning*, the use of the task queue may lead to better load balance, but it also may increase the run-time overhead. Assuming the perfect data partition, Figure 6 shows over 33%-50% speed improvement by PD-sort for large data sets ($>$ 4M) because of faster partitioning.

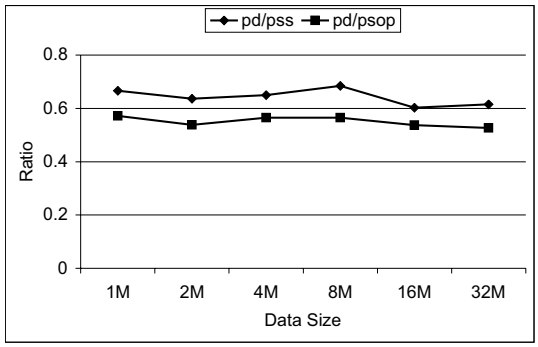


Figure 6. Computation cost ratios of *over-sampling* (PSS) and *over-partitioning* (PSOP) to PD-sort on normal distribution (mean=3000, std=1000). PD-sort is represented by pd.

7 Conclusions

We have presented a new partition method, PD-partition, for sequential and parallel sorting. Through extensive comparisons with previous methods, we found that

- *PD-partition* consistently improves the partition efficiency for all types of distributions tested, while maintaining better partition balance than other methods do.
- The performance improvement is independent of the data size.
- Unlike pivot-based methods, the overhead of this approach does not increase with the number of buckets. In fact, the improvement is greater for more buckets, showing that it is suitable for use in large scale parallel sorting.
- Using more samples improves the partition balance

with a slight increase in the overhead. The effect from the number of cells depends on the number of samples.

Overall, *PD-partition* shows 10-30% improvement in the partition balance and 20-70% reduction in the partition speed. Our cache-optimized PD-sort method is over 10% faster than quick-sort, commonly believed to be the fastest sequential sorting method for unbalanced data inputs. It slightly outperforms other cache-optimized algorithms [11, 22] for data size smaller than 4 million and about 30% for large data sets. The corresponding parallel sorting method is 33% to 50% faster than two popular approaches in the recent literature.

Adaptive data partition has important uses in many application domains such as scientific simulation, sparse matrix solvers, computer network simulation, and distributed database and Web servers. We expect that *PD-partition* will significantly improve the partition balance and speed in problems with unbalanced data inputs.

References

[1] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of ACM SIGMOD'97*, pages 243–254, 1997.

[2] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998.

[3] C. Cerin. An out-of-core sorting algorithm for clusters with processors at different speed. In *16th International Parallel and Distributed Processing Symposium (IPDPS)*, Ft Lauderdale, Florida, USA, 2002.

[4] G. Chaudhry, T. H. Cormen, and L. F. Wisniewski. Columnsort lives! an efficient out-of-core sorting program. In *Proceedings of the Thirteenth Annual Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 2001.

[5] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, Sept. 1994.

[6] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 280–291, 1991.

[7] H. Han and C.-W. Tseng. Improving compiler and runtime support for adaptive irregular codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1998.

[8] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, Aug. 1992.

[9] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *Proceedings of the IEEE Computer Society's 7th International Computer Software and Applications Conference*, pages 627–631, 1983.

[10] P. J. Janus and E. A. Lamagna. An adaptive method for unknown distributions in distributive partitioned sorting. *IEEE Transactions on Computers*, c-34(4):367–372, April 1985.

[11] A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. In *Proceedings of 8th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA97)*, pages 370–379, 1997.

[12] A. Larrosa. http://developer.kde.org/ larrosa/otherapps.html.

[13] H. Li and K. C. Sevcik. Parallel sorting by overpartitioning. In *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures*, pages 46–56, New York, NY, USA, June 1994.

[14] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):543–550, October 1993.

[15] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of 1997 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1997.

[16] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. *International Journal of Parallel Programming*, 29(3), June 2001.

[17] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, California, October 1999.

[18] S. Rajasekaran. A framework for simple sorting algorithms on parallel disk systems. In *Proceedings of the Tenth Annual Symposium on Parallel Algorithms and Architectures*, 1998.

[19] X. P. Shen, Y. Z. Zhong, and C. Ding. Adaptive data partitioning using probability distribution. Technical report 823, Computer Science, University of Rochester, Rochester, NY, 2003.

[20] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[21] S. K. Thompson. Sample size for estimating multinomial proportions. *The American Statistician*, 1987.

[22] L. Xiao, X. Zhang, and S. A. Kubricht. Improving memory performance of sorting algorithms. *ACM Journal on Experimental Algorithmics*, 5:1–23, 2000.