Adaptive Data Partitioning Using Probability Distribution

Xipeng Shen, Yutao Zhong and Chen Ding

Computer Science Department, University of Rochester
{xshen,ytzhong,cding}@cs.rochester.edu

Abstract

Many computing problems benefit from dynamic data partitioning—dividing a large amount of data into smaller chunks with better locality. When data can be sorted, two methods are commonly used in partitioning. The first selects pivots, which enable balanced partitioning but cause a large overhead of up to half of the sorting time. The second method uses simple functions, which is fast but requires that the input data confirm to a uniform distribution. In this paper, we propose a new method, which partitions data using the cumulative distribution function. It partitions data of any distribution in linear time, independent to the number of sublists to be partitioned into. Experiments show 10-30% improvement in partitioning balance and 20-70% reduction in partitioning overhead. The new method is more scalable than existing methods. It yields greater benefit when the data set and the number of sub-lists grow larger. By applying this method, our sequential sorting beats Quick-sorting by 20% and parallel sorting exceeds the previous sorting algorithm by 33-50%.

1 Introduction

Many types of dynamic data have a total ordering and benefi t from partial sorting into sublists. Examples include N-body simulation, where particles are partitioned based on their coordinates, and discrete event simulation, where events are partially ordered by their arrival time. In large scale problems, the data are distributed across a large number of processors and may still be too large to be stored in the main memory, e.g. out-of-core computation [7, 23]. Effi cient and balanced data partitioning is critical to parallelism and locality in many important applications.

In this paper, we study the partitioning problem in the context of parallel sorting, where we can build on and compare against a large body of research work [2, 4, 6, 7, 9, 14, 17, 21]. Our goal is to partition data into a given number of buckets. While the range of the buckets is sorted, the data within each bucket are not. The buckets are then individually sorted with higher effi ciency on parallel machines. Even on a single-processor machine, the smaller buckets can take advantage of memory hierarchy and multi-threading techniques. We measure the quality of the partitioning by its speed and and the balance of its partitions.

On a parallel machine, the balance determines the load balance. On a sequential machine, the balance affects the locality. In both cases, the time of partitioning is part of the overall sorting time.

Two major classes of partitioning methods have been studied. The first uses pivots. It has two steps: the selection of pivots, and the assignment of data into buckets. These methods give balanced partitions but have a relatively high overhead. The first step needs partial sorting. *RegularSampling* [21] sorts its local data for the selection of pivots. *OverSampling* [4] and *OverPartitioning* [17] sort some candidates before pivots selection (see Section 2 for a more detailed discussion). The second step uses a binary search. Given $(b-1)$ pivots selected in the first step, $O(nlog\,b)$ time is needed to assign n data into b buckets. A previous study reported that about 33-55% of total sorting time was consumed by *OverSampling*, one of the fastest pivot-based methods [4].

Instead of partial sorting, the second class of methods uses direct calculation in a manner similar to the hashing function used in hash-tables, for example, selecting bucket by a simple function of the first several bits of a datum as in NOWSort by Arpaci-Dusseau et al [2]. While these methods partition data in linear time, they are so far based on the assumption that the input data have a uniform distribution. For other distributions, the partitions can be severely unbalanced. In most real uses, the distributions of data are skewed rather than uniform. Since neither class of methods gives balanced partitions in linear time, the quest for fast and balanced data partitioning continues.

In this paper we describe a new method that overcomes the previous limitations. It gives balanced partitions without using pivots. Instead, it first estimates the *CDF (Cumulative Distribution Function)* of the data based on sampling information and then assigns data into buckets through direct calculation. The cost the estimation step is linear to the number of samples, and the cost of the assignment is linear to the amount of data. Both are independent to the number of buckets. Our experiments show that CDF-based partitioning handles data of any distribution in linear time. Compared with other approaches, the new method improves partitioning balance by 10-30% and reduces overhead by 20-70%.

CDF-based partitioning may have broad uses. This paper demonstrates its effectiveness in both sequential and parallel sorting. Sorting is an important and well defined problem. It is said that about 25-50% of all the work performed by computers consists of sorting data [1]. Data partitioning is widely used by programmers in database, statistical analysis, discrete events simulation (in computer network modeling) [11]. It can also be part of a tool for automatic data distribution, widely used in program parallelization, database and web servers.

In the rest of this paper, we review related work in Section 2, describe CDF-based partitioning and analyze its properties in Section 3, present its evaluation in Section 4, and conclude with a summary at the end.

2 Related Work

Li and Sevcik gave a summary of parallel sorting methods before 1994 [17]. Recent work includes NOWSort by Arpaci-Dusseau et al. [2], (l, m)-merge sort by Rajasekaran [20], an implementation of columnsort by Chaudhry et al. [7, 16], and parallel sorting on heteroge-

neous networks by Cerin [5].

Most methods are called single step because each element moves at most once between processors. They typically have three steps: they partition data into sub-lists, assign each sub-list to a parallel processor, and perform local sorting on each processor. The partitioning step often uses pivots. Previous work studied mainly three ways of pivoting as follows, where p is the number of processors.

- $RegularSampling(RS)$ [21, 18]. 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.

- $OverSampling(OS)$ [3, 4, 14]. 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.

- $OverPartitioning(OP)$ [17]. The $p * r * t$ candidates are picked randomly from the whole data set, where t is called *overpartitioning ratio*, and r is the same as in $OverSampling$. The candidates are sorted, and $(pt-1)$ pivots are selected by taking $s^{th}, 2s^{th}, \cdots, (pt-1)^{th}$ candidates. The whole data set is partitioned into pt sublists, which form a task queue for parallel processors.

Pivot-based methods has a higher time overhead than CDF-based partitioning. They need to sort candidates. To improve the partitioning balance, they need a large number of candidates. In addition, they need binary search to find a suitable bucket for each data. As we will discuss later, CDF-based partitioning is more efficient because it does not sort samples and the bucket assignment uses only a constant number of calculations.

Instead of using pivots, Arpaci-Dusseau et al. gave a method that directly calculates bucket assignment [2]. It partitions data in linear time. But it assumes that the input data have a uniform distribution. For other distributions, the partitions can be severely unbalanced.

Our work is built on *DPS (Distributive Partitioned Sorting)*. Dobosiewicz [10] first used it for sequential sorting. It uses n buckets of equal length to partition n data so that half buckets are on each side of the median. Then it repeats for each bucket with more than one item. The time complexity is $O(n)$ for uniformly distributed data and $O(nlogn)$ in the worst case. Janus and Lamagna [15] extended Dobosiewicz's method to any well-behaved distribution that can be transformed to a uniform distribution. We differ from these two studies in two ways. First, they have different purposes. They use partitioning as the kernel of sequential sorting. They keep partitioning data until one bucket contains only a single datum, when the sorting is completed. We partition data into equal-size buckets and then use other sorting algorithms such as quicksort or merge-sort for each bucket. Second, they partition data differently. They require that the range of each bucket be the same, while we require that the size of each bucket be the same. Since their goal is to sort all data, they partition data until reaching single-element buckets and do not need balanced sub-lists. Their methods yield balanced intermediate partitions only for uniformly distributed data. Our goal, in contrast, is to quickly find balanced sub-lists.

Data partition is a basic step in program parallelization on machines with distributed memory. Many applications have 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 [8]. For example, in N-body simulation, the inspector examines the coordinates of particles and partitions them onto different machines. Much later work has been based on this model, including run-time systems developed by Das et al. [8], language-based support by Hanxleden [13], compiler support by Han and Tseng [12], the use on DSM [19], and many other studies that we do not enumerate here. The partition problem in N-body simulation can be viewed as a problem of sorting. Instead of finding balanced sublists, we need to find subspaces that have an equal number of particles. We believe that the probability distribution-based partitioning can be used in data partition by an inspector in some irregular applications.

3 Probability Distribution-Based Partitioning Algorithm

The new method has two linear-time steps: the estimation of *Cumulative Distribution Function (CDF)*, and the assignment of data into buckets.

3.1 Estimation of Cumulative Distribution Function (CDF)

3.1.1 Estimation Method

We use the method of Janus and Lamagna [15]. It requires that the input distribution have a continuous and monotonic increasing CDF. Most distributions have this property. The estimation of CDF includes three steps as follows:

1) Divide the range between the maximum and the minimum of the data 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, as discussed in Section 3.3.

3.1.2 Statistical Measurement of CDF Estimation

Sampling theory provides the way to evaluate the estimated CDF given the number of samples [22]. The problem is the same as multinomial proportion estimation. The objective is to

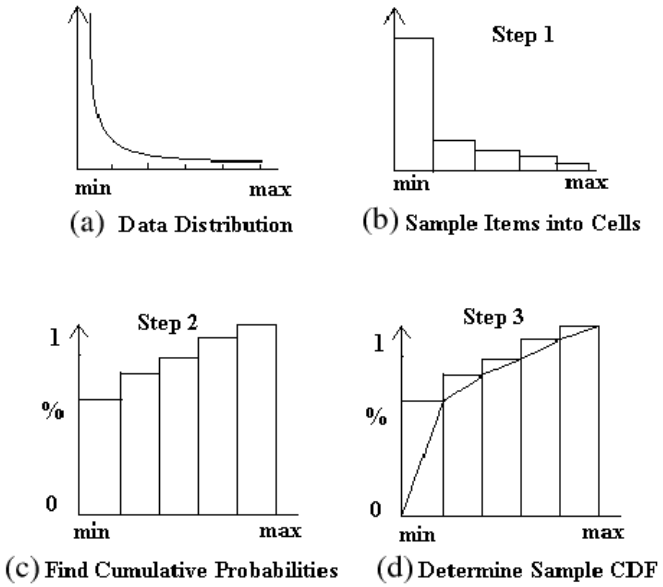


Figure 1: Estimation of CDF

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,

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

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 given error bounds. Thompson proposes the following formula for s [22]:

$$s = \max_{m} z^2(1/m)(1-1/m)/d^2 \tag{2}$$

where m is an integer from 0 to the total data size,(?) 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 $.8934 \leq \alpha < 1$ [22]. Note that for a given d and α, s is independent to the size of data and the number of categories k.

In our 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 minimum 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 minimum 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).

3.2 Assignment of Data into Buckets

In the second step, we assign data into buckets. Unlike previous distribution-based methods, our goal is to make each bucket contain the same number of data. We accomplish this in linear time using the algorithm shown in Figure 2. For each data element x, the algorithm finds the bucket assignment in three steps. First it finds the cell number of x. Recall that during the estimation of CDF, 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, which is calculated based on the slope of its cell.(?) The calculation assumes uniform distribution within each cell.

Using the cumulative probability p_x, we get the bucket number of x in one calculation. Let k be the number of buckets. Since we want balanced partitions, the buckets should have equal size. In other words, each bucket has the equal probability, $1/k$, 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 for x. Therefore, the time complexity of bucket assignment is $O(n)$, where n is the size of the input data.

It is important to make the bucket assignment as fast as possible since it contributes directly to the total running time. In the implementation, we make two optimizations. In the loop of Figure 2, there are four floating 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, as shown in Figure 3. The second optimization is scalar expansion inside the assignment loop to increase the parallelism and hide the calculation latency. Figure 4 shows the optimized algorithm. For the simplicity of presentation, the algorithm assumes that the total number of data is a multiplication of four, which is not required in the actual implementation.

3.3 Overhead Analysis

We analyze the overhead of CDF-based partitioning in sequential and parallel sorting. In the discussion, we use n to be the total number of data, s be the total number of samples, c be the number of cells, and k be the number of buckets. In parallel sorting, k is also the number of processors.

3.3.1 Sequential Sorting

Sequential sorting orders data using a single processor. We use the data partitioning to cut data into blocks smaller than the cache size and then sort each block. Since the block partitioning is linear time, it has better cache locality than the recursive partitioning schemes like quicksort, when n is much greater than the cache size. The exact overhead can be divided into four parts.

```
/*
   data is the array of data to be partitioned, whose size is N;
   min,max are the minimum and maximum of the data;
   range [min, max] is split into c cells of equal length;
  lc is the length of each cell;
  cdf[i] is the cdf of ith cell, cdf[0] = 0 and cdf[c] = 0.9999999;
  slope[i] is the slope of the fitting line in cell i;
  b is the number of buckets
*/
  ......
  for (int i=0;i<N;i++){
      /* find the cell number of data[i] */
       fbat dtmp = (data[i] - min)/lc;     int n = (int)dtmp;     fbat frac =
dtmp - n;

       /* find the cdf of data[i] */
           /* the distance of marray[i] from the starting position of cell n*/
           double l = marray[i]-min-n*lc;
           /* cdf of data[i] */
           double xcdf = cdf[n]+slop[n]*l;

       /* calculate bucket number of data[i] */
       buckets[i] = (int)(xcdf*b);}
```

Figure 2: Algorithm for the bucket assignment

```
      /* find the cell number of data[i] */
       fbat dtmp = (data[i] - min)/lc;     int n = (int)dtmp;     fbat frac =
dtmp - n;
       /* calculate bucket number of data[i] */
       buckets[itmp] = (int)(bucketNum[n0] + bucketNum1[n0+1]*frac0)
```

Figure 3: Optimized bucket assignment using two floating-point operations per datum

```
. . . . . .
/* Get the number of buckets covered by each cell*/
bucketNum[0] =cdf[0]*b; bucketNum1[0]=bucketNum[0];
for (int i=1;i<N;i++){
      bucketNum[i] = cdf[i]*b;
      bucketNum1[i] = bucketNum[i] - bucketNum[i-1];}
/* Assign each datum into a bucket*/
int N4 = N>> 2;// get N/4
for (int i=0;i<N4;i++){
      int itmp = i*4;
     /* find the cell number of data[4*i], data[4*i+1], data[4*i+2], data[4*i+3]
*/
      fbat dtmp0 = (data[i] - min)/lc;        fbat dtmp1 = (data[i+1] - min)/lc;
      fbat dtmp2 = (data[i+2] - min)/lc;      fbat dtmp3 = (data[i+3] - min)/lc;
      int n0 = (int)dtmp0;                    int n1 = (int)dtmp1;
      int n2 = (int)dtmp2;                    int n3 = (int)dtmp3;
      fbat frac0 = dtmp0 - n0;                fbat frac1 = dtmp1 - n1;
      fbat frac2 = dtmp2 - n2;                fbat frac3 = dtmp3 - n3;
      /* calculate bucket number of data[4*i], data[4*i+1], data[4*i+2], data[4*i+3]
*/
      buckets[itmp] = (int)(bucketNum[n0] + bucketNum1[n0+1]*frac0)
      buckets[itmp+1] = (int)(bucketNum[n1] + bucketNum1[n1+1]*frac1)
      buckets[itmp+2] = (int)(bucketNum[n2] + bucketNum1[n2+1]*frac2)
      buckets[itmp+3] = (int)(bucketNum[n3] + bucketNum1[n3+1]*frac3)}
```

Figure 4: Optimized algorithm using scalar expansion

1. **Finding Min and Max:** It traverses all data to find the maximum and minimum in $O(n)$ operations.

2. **Sampling:** It generates s random numbers in $O(s)$ operations.

3. **CDF Estimation:** It assigns samples into cells in $O(s)$ operations.

4. **Data Partitioning:** It assigns data into buckets in $O(n)$ operations.

The data partitioning part has the highest overhead because of the floating-point operations. The breakdown is showed in Figure 5. The first three parts are showed as "others" in the figure. The whole partition cost is about 20% of all cost. The total cost of distribution-based sorting is 20% less than Quick-sorting for large data set. The larger is the data set, the better is our method (see Table 2.) Section 4.2 shows the experimental details.

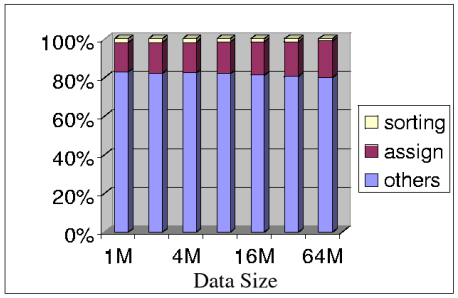


Figure 5: for break down of adaptive sorting.

3.3.2 Parallel Sorting

In parallel sorting, we minimize the communication cost as well as the computation cost. We discuss the cost in four parts as in the sequential sorting case. We assume the unsorted data are evenly distributed on all k processors initially.

1. **Finding Min and Max:** Each processor traverses n/k data to find the local maximum and minimum. The local maximums and minimums are sent to a processor to find the global maximum and minimum. The computation cost is $O(max(n/k, k))$. The global maximum and minimum are broadcast to all processors in $2k$ messages, each requiring one round-trip time. Each message contains 2 integers.

2. **Sampling:** Each processor generates s/k random numbers. It requires $O(s/k)$ operations and no communication.

3. **CDF Estimation:** Each processor finds the cell numbers of the local samples and sends them to a processor, which combines the results of all processors and computes the CDF estimation. The cumulative distribution is then broadcast back to all processors. This step needs $O(max(s/k, c))$ operations and $2k$ messages. The maximal message size is $O(c)$.

4. **Data Partitioning:** each processor assigns n/k samples into k buckets and sends them to the corresponding processors. The required computations are $O(n/k)$. In the worst case, every data element is moved in n messages, each requiring a one-way trip.

Both the number of messages and operations are linear to the number of data. The forth part accounts for the majority of communications. The communication cost of the other three parts are only 2 round trips, which is negligible. Since any parallel sorting algorithm requires the forth part, our distribution-based partition requires the lower bound communications of parallel sorting.

4 Evaluation

We first measure the efficiency and balance of CDF-based partitioning and compare them with those of other partitioning 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 sorted randomly generated integers of different distributions. For the CDF-based method, the cell number was 1000, sample size was 40000.

4.1 Data Partitioning

We show the partitioning results from CDF-based partitioning, *over-sampling* [4, 14] and *over-partitioning* [17]. *Regular sampling* [21, 18] has more overhead and is not included in the evaluation. In $oversampling$, the over-sampling rate is 32 (as in [4]). In $overpartitioning$, the over-sampling rate is 3 and the over partitioning rate is $log\, k$ (as in [17]), where k is the number of buckets. Section 2 describe the three algorithms in more detail. Since the focus of this paper is data partitioning, most experiments use a relatively small data input of 64 million random integers. We measure the partitioning cost and balance. Each result is the average result of 20 data groups randomly generated for each distribution. We use uniform and different normal distributions as inputs.

We measure the partitioning balance using a concept called *bucket expansion*, 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. In other cases, the lower the ratio is (closer to 1), the better the worst-case balance. We compare two methods by their balance ratio, in particular, the ratio of the bucket expansion of the CDF-based partitioning to the bucket expansion of over sampling, i.e. BE(d)/BE(os), and the ratio of CDF-based partitioning to over partitioning, i.e. BE(d)/BE(op). A ratio of less than one means that CDF-based method has a better balance. The lower the more balanced.

After partitioning, we use quick-sort in each bucket to produce the effect of complete sorting. We compare the speed of these sorting methods by the *time ratio*, the ratio of the running times of two sorting methods using the same input. We will measure the ratio of CDF-based sorting to the sorting time using over sampling, i.e. Time(d)/Time(os), and the

ratio of CDF-based sorting to over partitioning, i.e. Time(d)/Time(op). A ratio of less than one means that CDF-based sorting is faster. The lower the faster.

We use the relative balance and speed in the evaluation because we can compare the balance and speed of all three methods in a single fi gure by displaying their balance and time ratios. The lower the ratios are, the better the CDF-based partitioning and sorting. If the time ratio is less than 1, our approach has less overhead than the other methods. If the balance ratio is less than 1, our approach yields better partition balance than the other methods.

Five factors may affect the partition balance and cost: the distribution of data, and the number of data elements, buckets, cells, and samples. The fi rst two factors are the properties of the data. The third one is the requirement of partition, i.e. how many sublists the data should be partitioned into. The last two factors determine the quality of CDF estimation in our approach. We show the effect of those factors respectively. More factors have a similar effect on all four distributions, we show only the uniform distribution results. Otherwise we make an explicit note and show results for additional distributions.

4.1.1 Effect of Data Distribution

This distribution-based approach applies to any well-behaved distribution, however complicated it is. We use four input distributions: uniform and three normal distributions. The three normal distributions have the same mean value of 3000, but the standard deviations are 3000, 1000, and 300. They represent three cases: most data are distributed in a large range, a medium range, and a small range. We generate 20 random inputs for each distribution and present their average. In all cases, the size of data is 64 million, the number of buckets is 128, the number of cells is 1000, and the number of samples is 40 thousand.

The three algorithms are compared through the time and balance ratios in Figure 6. The ratios are linked by lines to better show their differences. The actual running time, bucket expansion, and values of these ratios are listed in Table 1.

For all four distributions, the time ratios are less than 0.8 and 0.6, so the overhead of our approach is less than 80% and 60% to over sampling and over partitioning. At the same time, the distribution-based approach yields better partition balance than the other two methods. The uniform distribution has the greatest gain because a uniform distribution is estimated better by samples than heavily skewed distributions are.

4.1.2 Effect of Data Size

Figure 6(b) shows the effect of the data size. We use the uniformly distribution. The relative effects in normal distributions are similar (??).

Different data sizes do not change the time or the balance ratios. Regardless of the data size, the time ratios are around 0.5 and 0.4, and the balance ratios are around 0.8 and 0.9. All are less than 1, showing consistent improvement of PD-based sorting over the other two methods in all data sizes.

The constant time ratios are expected. The PD-based method reduces partition overhead from $O(Nlog\,b)$ to $O(N)$ (where b is the number of buckets). Thus, the time ratio is $\frac{O(N)}{O(Nlog\,b)}$, which is independent of N but proportional to $\frac{1}{log\,b}$. We now show the effect from the number of buckets.

Table 1: A comparison of the partitioning balance and speed for four distributions

Distributions	Metrics	PD-based	over sampling	over partitioning
Uniform	Time (ms)	82.5	174.0	232.5
	Time Ratio	1.0	0.474	0.355
	Bucket Expansion	1.152	1.538	1.416
	BE Ratio	1.0	0.749	0.814
Normal-I	Time (ms)	82.5	176.5	235.0
	Time Ratio	1.0	0.467	0.351
	Bucket Expansion	1.209	1.542	1.447
	BE Ratio	1.0	0.784	0.836
Normal-II	Time (ms)	85.0	173.5	234.0
	Time Ration	1.0	0.490	0.363
	Bucket Expansion	1.241	1.542	1.438
	BE Ratio	1.0	0.805	0.863
Normal-III	Time (ms)	83.0	175.5	235.5
	Time Ration	1.0	0.473	0.352
	Bucket Expansion	1.329	1.555	1.492
	BE Ratio	1.0	0.855	0.891

4.1.3 Effect of The Number of Buckets

The number of buckets has similar effects in all distributions (??). We show only the results of the uniform distribution in Figure 6(c). The time ratio decreases as the number of buckets increases, as predicted by our analysis. The balance ratio increases slightly but is still less than 0.9 for over partitioning and less than 0.8 for over sampling. The time saving is measured for the whole sorting time. The saving in partitioning time should be signifi cantly higher.

Large-scale parallel sorting needs data partitioned into a large number of buckets or sublists. Figure 6(c) shows that PD-based partitioning is more scalable and produces larger number of buckets faster and with better balance than over sampling and over partitioning, the two method used most frequently in parallel sorting.

4.1.4 Effect of The Number of Samples and Cells

More samples and cells increase the estimation accuracy of the cumulative distribution function or CDF. More cells allow the CDF fi tting at fi ner 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. Their effects are shown in Figure 6(d, e, f).

In Figure 6(d), we use ?? cells and vary the number of samples from 5 thousand to 80 thousand. We use uniform distribution, since the results in other distributions are similar. Using more samples improves the CDF estimation. The fi gure shows that PD-based partitioning obtains greater improvement 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 6(e) shows the changes from the number of cells. We use 40 thousand samples. For a normal distribution (mean=3000,deviation=300.), the partitioning balance is improved when the number of cells is increased from 100 to 1600. It then worsens 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 6(f) shows the effects using the uniform distribution. The balance ratios in that fi gure decrease all the way to ??, because a small amount of samples is enough for estimating CDF in this case.

4.2 Comparison with Quick-sort

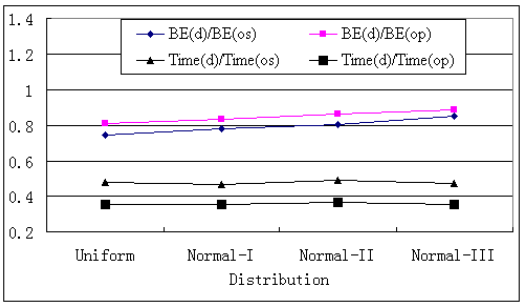
Quick-sort is believed by many to be the fastest in-memory sorting method. A recent study by Xiao et al. shows that for uniform distributed input data, simple partitioning is faster. But for non-uniform data distributions, quick-sort still gives the best performance because simple methods could not give balanced partitions [24]. We now show that for non-uniform but well-behaved distributions, PD-based sorting outperforms quick-sort because of its better partitioning balance and speed.

Table 2 shows the time to sort integers of a normal distribution (m=3000, d=1000). We take 64K as the cache size for partitioning. It is less than the real cache size (512K) to leave room for other data and to avoid cache interference. Figure 7 compares the speed of all four distributions: uniform distribution, normal distribution with mean of 3000 and variance of 3000,1000,300. The two curves in each graph are the ratios of PD-sort to quick-sort and the ratios of over sampling (??) to quick-sort. For sequential sorting, over partitioning is similar to over sampling. The difference is that the former uses more buckets for the over-partitioning factor. Therefore, we do not need to show over sampling (??) results.

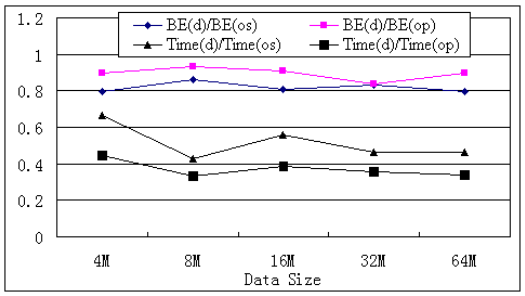
PD-sort is consistently faster than quick-sort. The improvement becomes greater as the data size gets larger. When sorting more than ? million data, PD-sort outperforms quick-sort by more than 20%. In comparison, over sampling is slower than quick-sort because of the high partitioning overhead. The speed gap grows wider on larger data inputs.

PD-sort is faster because it incurs fewer data traversals than quick-sort. PD-sort partitions data into buckets less than the case size in two passes. The fi rst fi nds the maximum and minimum,and the second assigns data into buckets. The estimation of CDF causes extra overhead when taking samples and counting them in each cell. But the overhead is negligible compared to the traversal of the larger input data. Quick-sort, however, needs $log(D/C)$ passes, where D is the data size and C is the average bucket size. This assumes that quick-sort uses the right pivots to fi nd even partitions at each step. The improvement is greater for larger data sizes.

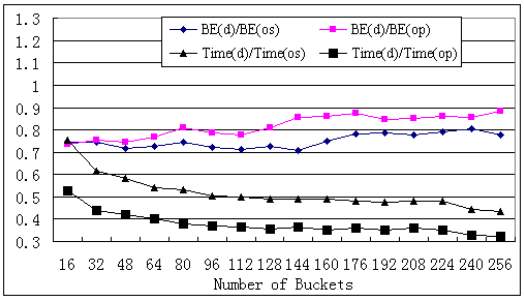
We use the basic quick-sort, which recurs until a single data element. Previous studies show that after the sub-section is smaller than some k, using insertion sort is faster [**?**]. Such improvement would not signifi cantly diminish the advantage of PD-sort, which uses quick-sort in each bucket and benefi ts equally from a faster sort in small data sections. In addition, the constant k depends on many factors. It may alter quick-sort effi ciency completely if not carefully chosen. We therefore choose to compare with the basic quick-sort.



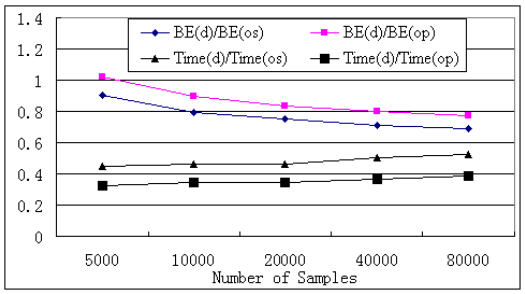
(a) Partition results for uniform distribution, and three normal distributions (m=3000, d=3000,1000,300)



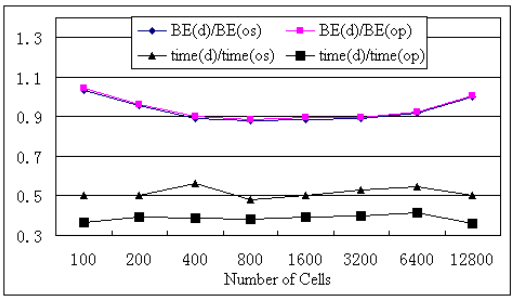
(b) Effects of data size on uniform distribution



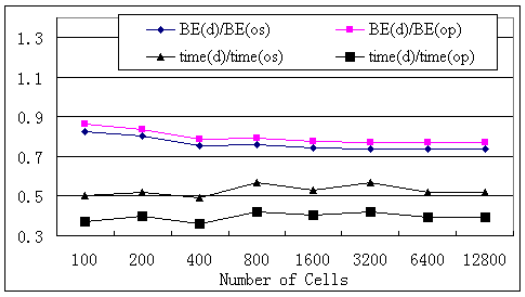
(c) Effects of bucket number on uniform distribution



(d) Effects of sample number on uniform distribution



(e) Effects of cell number on normal distribution (m=3000, d=300)

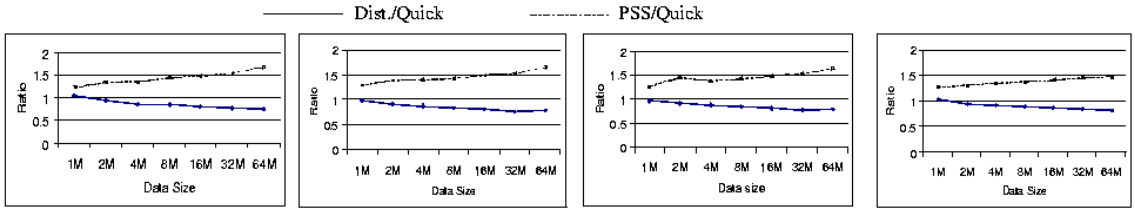


(f) Effects of cell number on uniform distribution

Figure 6: Partition Evaluation. BE: Bucket Expansion; d: distribution-based approach; os: oversampling; op: overpartitioning.

Table 2: Time for Sequential Sorting (sec.)

Data Size	Dist.	Quick	OverSampling
4M	1.357	1.552	2.195
8M	2.68	3.207	4.607
16M	5.343	6.541	9.804
32M	10.425	13.492	2.0685
64M	21.455	27.183	45.033



(a) Uniform (b) Normal (d=3000) (c) Normal (d=1000) (d) Normal (d=300)

Figure 7: Sequential Sorting Results.

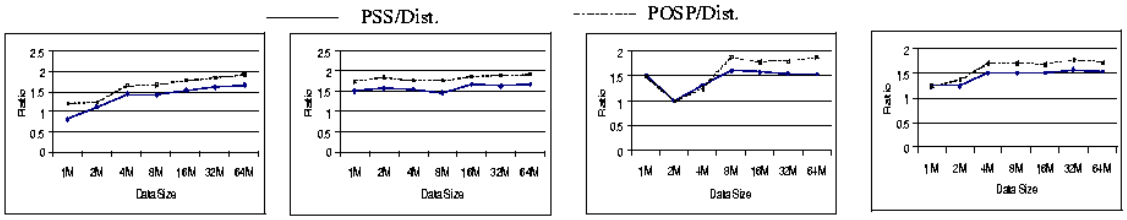
4.3 Parallel Sorting

We compare the parallel performance of PD-sort and sorting 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 partitioning on a single processor and then sort each sub-list on a parallel processor. The communication cost have 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 different as follows.

- PD-sort: it needs two round trips (see Section 3.3.2). The maximal message kernel size is C integers, where C is ??.
- over sampling: it needs one round trip. Each processor sends s samples to the central processor. The central processor finds $(P-1)$ pivots and broadcasts them to all other processors. Here s is the oversampling rate, and P is the number of processors. The maximal message kernel size is $max(s,(P-1))$ integers.
- over partitioning: The same as over sampling, except that the maximal message kernel size is $max(s*k,(P*k-1))$.

Usually, C and P are less than several thousands, and s and k are less than one hundred. In terms of data volume, the communication in the first part is negligible compared to that of the second part. Assuming the same communication cost, we use the computation costs to measure the performance of parallel sorting.

Figure 8 compares the sorting speed on all four data distributions. 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 4.1 shows better partition balance by PD-sort



(a) Uniform (b) Normal (d=3000) (c) Normal (d=1000) (d) Normal (d=300)

Figure 8: Computation costs in parallel sorting

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 benefi ts to over sampling and over partitioning. Under such favorable conditions, Figure 8 still shows over 33%-50% speed improvement by PD-sort for large data sets.

5 Conclusions

In this paper we have presented a new data partitioning method based on probability distribution. By comparing with previous methods and varying important parameters, we found that

- The new method consistently improves the partitioning time for all types of distributions tested, while maintaining better partitioning balance than other methods.
- 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 mre samples causes a slight increase in overhead but a signifi cant improvement in partitioning balance.
- In general, the effect of the number of cells depends on the number of samples.

Overall, the new method shows 10-30% improvement in partitioning balance and 20-70% reduction in partitioning speed. It is scalable and yields more benefi ts for use in larger data inputs or for more data buckets.

In internal sequential sorting, the new method is 20% faster than quick-sort, commonly believed to be the fastest sequential sorting method for unbalanced data inputs. The new method is 33% to 50% faster in parallel sorting than two popular approaches in the past.

Sorting problem is an important and well-defi ned problem. It is widely used in numerous tasks, such as database, statistics, and discrete events simulator [11] in computer network. We should note that this distribution-based partition approach can also be used in other tasks requiring data partitions, like program parallelization, database servers and web servers.

6 Acknowledgment

This work is supported by the National Science Foundation (Contract No. CCR-0238176, CCR-0219848, and EIA-0080124) and the Department of Energy (Contract No. DE-FG02-02ER25525).

References

[1] S. Akl. *Parallel Sorting Algorithms*, chapter 1, page 2. Academic Press, 1985.

[2] 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.

[3] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, 1991.

[4] 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.

[5] 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.

[6] C. Cerin, H. Fkaier, and M. Jemni. A synthesis of parallel out-of-core sorting programs on heterogeneous clusters. In *3st International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, May 2003.

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

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

[9] 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.

[10] W. Dobosiewicz. Sorting by distributive partitioning. *Information Processing Letters*, 7(1):1–6, 1978.

[11] K. Fall and K. Varadhan. *ns notes and documentation*. The VINT Project, UC Berkeley, LBL, USC/ISI, and Xerox PARC, November 1997.

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

[13] 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, August 1992.

[14] 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.

[15] 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.

[16] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, 34(4):344–354, April 1985.

[17] 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.

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

[19] 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.

[20] 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.

[21] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

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

[23] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, Providence, RI, 1999.

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