Published in final edited form as:
J Parallel Distrib Comput. 2011 November 1; 71(1): 1427–1433. doi:10.1016/j.jpdc.2011.07.004.

Efficient Out of Core Sorting Algorithms for the Parallel Disks Model⋆

Vamsi Kundeti and **Sanguthevar Rajasekaran**
Department of Computer Science and Engineering, University of Connecticut Storrs, CT 06269, USA

Vamsi Kundeti: vamsik@engr.uconn.edu; Sanguthevar Rajasekaran: rajasek@engr.uconn.edu

Abstract

In this paper1 we present efficient algorithms for sorting on the *Parallel Disks Model (PDM)*. Numerous asymptotically optimal algorithms have been proposed in the literature. However many of these merge based algorithms have large underlying constants in the time bounds, because they suffer from the lack of *read parallelism* on PDM. The irregular consumption of the runs during the merge affects the read parallelism and contributes to the increased sorting time. In this paper we first introduce a novel idea called the *dirty sequence accumulation* that improves the read parallelism. Secondly, we show analytically that this idea can reduce the number of parallel I/O's required to sort the input close to the lower bound of $\Omega\left(\log_{\frac{M}{B}}(\frac{N}{M})\right)$. We experimentally verify our *dirty sequence* idea with the standard R-Way merge and show that our idea can reduce the number of parallel I/Os to sort on PDM significantly.

1 Introduction

Sorting is a fundamental problem that has numerous applications and hence has been studied extensively. When the amount of data to be processed is large, secondary storage devices such as disks have to be employed in which case the I/O becomes a bottleneck. To alleviate this bottleneck computing models with multiple disks have been proposed. One such model is the Parallel Disks Model (PDM). In a PDM, there is a (sequential or parallel) computer that has access to $D(\geq 1)$ disks. In one I/O operation, it is assumed that a block of size B can be fetched into the main memory from each disk. One typically assumes that the main memory is of size M where M is a (small) constant multiple of DB. Parameters that characterize a PDM are:

- D – the number of disks.
- B – block size of the disks.
- M – main memory size. Typically, $M \geq cDB$, where c is some constant.
- N – size of the input.

⋆This research has been supported in part by the NSF Grants ITR-0326155 and 0829916 and a UTC endowment.

[1]A preliminary version of this paper has been presented in HiPC 2008 [15]

The performance of any PDM sorting algorithm is measured in terms of the number of parallel I/O operations performed by the algorithm. The time taken for local computations is typically ignored since it tends to be much less than the time taken by I/O. A lower bound of $\Omega\left(\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)}\right)$ on the number of parallel I/O operations to sort N keys has been proven by Aggarwal and Vitter [2].

I/O parallelism plays a key role in the runtime of sorting algorithms on the PDM. Comparison based sorting algorithms (e.g. R-Way merge) on PDM suffer from from *read parallelism*. This problem arises especially due to the way the runs are arranged on the PDM (see Figure 1). Notice that the runs are arranged such that the leading block of each run resides on a different disk. This particular arrangement of the runs is called *striping*. Striping runs improves the read throughput, i.e. we can read B keys from each of the D runs in one parallel I/O. Merge based sorting algorithms such as the R-Way merge work really well as long as the runs are consumed uniformly and the leading block of each run resides on a different disk. However this is not possible in practice, irregular consumption of the runs during the R-Way merge can lead to degraded parallel I/O performance. Consider the situation in Figure 1 ($D = 4$) where the leading blocks of three runs reside on the same disk. In this case if the R-Way merge needs to access the leading blocks of the runs, it needs to spend 3 parallel I/Os. This is because in each parallel I/O we can read at most one block from each disk. In the worst case merge based sorting algorithms may have to spend D parallel I/Os to continue merging when all the leading blocks of the runs reside on the same disk. The goal of this work is to design algorithm which do not suffer from I/O parallelism on PDM.

The amount of resource (like time, space, etc.) used by a randomized algorithm is said to be $\tilde{O}(f(N))$ if the amount of resource used is no more than $c\alpha f(N)$ with probability $\geq (1 - N^{-\alpha})$ for any $N \geq n_0$, where c and n_0 are constants and α is a constant ≥ 1. We could also define the asymptotic functions $\tilde{\Theta}(.)$, $\tilde{o}(.)$, etc. in a similar manner. In this paper we use log to denote logarithms to the base 2 and ln to denote logarithms to the base e.

The organization of the paper is as follows. We present a summary of related work on PDM sorting in Section 2 and Section 3. We introduce our idea of *dirty sequence accumulation* in Section 4. Finally in Section 5 we describe our work on simulating the PDM model on a single disk.

2 Prior Algorithms and Our Results

Two kinds of algorithms can be found in the literature for PDM sorting. The first kind of algorithms (see e.g., [17, 16, 8]) distribute keys based on their values and hence are similar to bucket sort. The second kind of algorithms (see e.g., [1, 4, 5, 9, 10]) use R-way merging for a suitable value of R.

Ensuring full write parallelism is a challenge in distribution based algorithms and ensuring read parallelism is difficult in the case of merge based algorithms. The algorithm of Barve, Grove, and Vitter [4] ia based on merging and it uses a value of $R = M/B$. This algorithm called *Simple Randomized Mergesort (SRM)* stripes the runs across the disks. For each run the first block is stored in a random disk and the other blocks are stored in a cyclic fashion starting from the random disk. SRM has been shown to have an optimal expected performance only when the internal memory size M is $\Omega(BD \log D)$. No high probability bounds have been proven for SRM. The typical assumption made on M is that $M = O(DB)$.

SRM algorithm has been modified by Hutchinson, Sanders and Vitter [7]. Their approach is based on an algorithm of Sanders, Egner and Korst [14] who use lazy writing at the expense

of an internal buffer. They employ Fully Randomized (FR) scheduling to allocate blocks of each stream to disks. An *expected* parallelism of $\Omega(D)$ is proven using asymptotic queuing theoretic analysis. Vitter and Hutchinson extended the above scheme to a scheduling scheme called Random Cycling (RC). The FR schedule is more complicated to implement and is not read-optimal for $M = o(BD \log D)$.

Many asymptotically optimal algorithms are known for sorting on the PDM (see e.g., [3], [9], [16], [10], etc.). Recently, Rajasekaran and Sen [11] have come up with practical and optimal algorithms for sorting on the PDM. In particular, they present an asymptotically optimal randomized algorithm for sorting. For the first time they have been able to obtain high probability bounds using basic principles.

In the next section we summarize the ideas in the algorithms of [11]. In this paper we present variants of the randomized algorithm presented in [11]. Our analysis indicates that the underlying constant in the run time of the new algorithm is less than that of [11]'s algorithm.

3 A Summary of Rajasekaran and Sen's Algorithm

The basic scheme of Rajasekaran and Sen's algorithm is to first randomly permute the given input sequence and then use a D-way merge on the permuted sequence. The random permutation is achieved using an integer sorting algorithm.

3.1 Integer Sorting

Consider the case when the keys to be sorted are integers in the range $[1, Q]$ for some integer Q. Integer sorting is a well studied problem in sequence and in parallel. For example, we can sort n integers in $O(n)$ time if the keys are in the range $[1, n^c]$ (for any constant c) (see e.g., [6]). There are two types of integer sorting algorithms, namely, forward radix sorting (or Most Significant Bit (MSB) first sorting) and backward radix sorting (or Least Significant Bit (LSB) first sorting). Rajasekaran and Sen [11] make use of backward radix sorting. The keys are sorted in phases where in each phase the keys are sorted only with respect to some number of bits.

The following Theorems are due to Rajasekaran and Sen [11]:

Theorem 1. *N random integers in the range $[1, R]$ (for any R) can be sorted in an expected $(1+\nu)\frac{\log(N/M)}{\log(M/B)}+1$ passes through the data, where ν is a constant < 1. In fact, this bound holds for a large fraction ($\geq 1 - N^{-\alpha}$ for any fixed $\alpha \geq 1$) of all possible inputs.*

The above integer sorting algorithm can be used to randomly permute N given keys such that each permutation is equally likely. The idea is to assign a random label to each key, where the labels are in the range $[1, N^{1+\beta}]$ (for any fixed $0 < \beta < 1$), and sort the keys with respect to their labels. When the key labels are in the range $[1, N^{1+\beta}]$, the labels may not be unique. The maximum number of times any label is repeated is $\tilde{O}(1)$. Keys with equal labels are permuted in one more pass through the data.

The following Theorem is proven in [11]:

Theorem 2. *We can permute N keys randomly in $O\left(\frac{\log(N/M)}{\log(M/B)}\right)$ passes through the data with probability $\geq 1 - N^{-\alpha}$ for any fixed $\alpha \geq 1$, where μ is a constant < 1, provided $B = \Phi(\log N)$.*

3.2 Randomized Sorting

An asymptotically optimal randomized algorithm can be developed using the random permutation algorithm and merge sort [11]. A simple version of the algorithm (called `RSort1`) works as follows: 1) Randomly permute the input N keys; 2) In one pass through the data form runs each of length $M = DB$; 3) Use a D-way merge sort to merge the N/M runs. Let an *iteration* refer to the task of merging D runs. At the beginning of any iteration 2 blocks are brought in from each disk. The D runs are merged to ship DB keys out to the disks. From thereon, maintain the invariant that for each run there are two leading blocks in the memory. This means that after shipping DB keys to the disks, bring in enough keys for each run so that there will be two leading blocks per run.

Even though `RSort1` makes an optimal number of scans through the input, each scan may take more than an optimal number of I/Os. This is because the runs get consumed at different rates. For instance, there could come a time when we need a block from each run and all of these blocks are in the same disk. At the beginning of any iteration, each run is striped across the disks one block per disk. Also, the leading blocks of successive runs will be in successive disks. For example, the leading block of run 1 could be in disk i, the leading block of run 2 could be in disk $i + 1$, etc. If this property could be maintained always, we can get perfect parallelism in reading the blocks. If the blocks are consumed at the same rate then this property will be maintained.

`RSort1` is modified in [11] to get `RSort2`. In `RSort2` leading blocks of successive runs are either in successive disks or very nearly so. When the leading blocks of successive runs deviate significantly in terms of their disk locations, a *rearrangement* operation is performed. This operation involves rearranging the leading DB keys of each run so that the above property is reinstated after the rearrangement. As time progresses, the leading blocks will deviate more and more from their expected disk locations. Call the step of bringing in enough keys to have 2 blocks per run, merging them and outputting DB keys as a *stage* of the algorithm. After every D stages `RSort2` performs a rearrangement of runs.

Theorem 3. `RSort2` *takes* $\tilde{O}\left(\frac{\log(N/M)}{\log(M/B)}\right)$ *read passes through the data provided* $B=\Omega(\sqrt{M \log N})$.

Proof Sketch. It suffices to prove that each phase of the algorithm takes $\tilde{O}(N/DB)$ I/Os. In each stage of the algorithm the expected number of blocks consumed from each run is 1. If the starting block of a run is i then after q stages, the leading block of this run is expected to be in disk $(i + q - 1) \bmod D + 1$. Assume that the leading block of each run continues to be within one disk of its expected disk. Consider the task of bringing into main memory at most K leading blocks of each run. How many I/Os will be needed? It is easy to see that in the worst case, $3 \cdot K$ I/Os will suffice as each disk can have at most 3 of the leading blocks. Thus when $K = 1$, three I/O's suffice. See [11] for more detailed proof.

`RSort2` assumes that $B=\Omega(\sqrt{M \log N})$. This assumption is relaxed using a value of $R = D^{\varepsilon}$ for any constant $1 > \varepsilon > 0$. In the resultant algorithm `RSort`, rearrangements are done every D^{ε} stages.

For choice of $\varepsilon = 1/2$, the number of iterations made by the algorithm is no more than $2(1+\nu) \cdot \frac{\log(N/M)}{\log(M/B)}$ where each iteration involves $2(1 + \nu)$ read passes including rearrangement. The number of write passes is the same. This gives a total of $8(1+\nu)\frac{\log(N/M)}{\log(M/B)}$ passes with

probability $\geq (1 - N^{-\alpha})$ for any constant $\alpha \geq 1$. Here ν, μ are constants between 0 and 1. To this, we must add the time for generating the initial random permutation.

When B is large, we can decrease the number of read passes made by `RSort` [11]. For instance if $B = M^{\beta}$, the number of read passes made by `RSort` is

$$(4 - 4\beta)(1+\nu)\frac{\log(N/M)}{\log(M/B)} + (1+\mu)\frac{\log(N/M)}{\log(M/B)} + 2.$$

4 Key Idea behind Our Algorithm

In Algorithm `RSort2`, a rearrangement is done every $R = D$ stages. This rearrangement involves reading the leading DB keys of each run and writing them back so that the leading blocks of the runs are in successive disks. One of the key ideas of our proposed algorithm is not to do the rearrangements. Instead of rearranging the keys, we remove relevant leading keys from each run so that after removing these keys, the leading keys of each run start from successive disks. The removed keys are stored in the disks as a *dirty sequence*. After the algorithm completes execution in this fashion, there will be a long sorted sequence and a dirty sequence. The dirty sequence is sorted and merged with the long sorted sequence. We will show that the length of the dirty sequence is much smaller than the length of the main sorted sequence and hence the final merging step does not take much time. Call this algorithm `RSort3`.

We first consider the problem of merging a sorted sequence X of length n and an unsorted sequence Y (*dirty sequence*) of length m. Note that if $m < M$ we can solve this problem with exactly $\frac{n}{DB}$. However if $m > M$ we use the following result.

Theorem 4. *We can merge a sorted sequence X (of length n) with an unsorted sequence Y (of length m) in* $O\left(\frac{m}{DB}\frac{\log(m/M)}{\log(M/B)}\right) + \frac{2(m+n)}{DB}$ *parallel read I/O operations.*

Proof. Sort Y using any of the optimal algorithms, this takes $O\left(\frac{m}{DB}\frac{\log(m/M)}{\log(M/B)}\right)$ parallel I/O operations. Let the sorted order of Y be Z. We can merge X and Z as follows. Assume that X is striped starting from disk 1 and Z is striped starting from disk $(D/2) + 1$. We can use a simple merging algorithm where at the beginning we bring D blocks from each run. Get the DB smallest keys out of these and ship them out to the disks. Now bring in enough keys from each run so that there are DB keys from each run. Get the DB smallest keys from these and ship them out, and so on. For every DB keys output, we have to perform at most two parallel read I/O operations. Thus the total number of parallel read I/O's needed to merge X and Z is $\leq \frac{2(m+n)}{DB}$.

Length of the Dirty Sequence: Note that a rearrangement is done in `RSort2` every R stages, i.e., for every RDB keys output. The contribution to the length of the dirty sequence by every R stages is at most three blocks from every run (with high probability). In other words, the length of the dirty sequence increases by $3RB$ for every RDB keys output. This means that the length of the dirty sequence during the entire algorithm is no more than $(1+\nu)\frac{\log(N/DB)}{\log(D)} \times N \times \frac{3RB}{RDB} = (1+\nu)\frac{\log(N/DB)}{\log(D)} \times \frac{3N}{D}$ with high probability (where ν is any constant > 0 and < 1).

Theorem 5. `RSort3` *is asymptotically optimal and since it eliminates the rearrangement step completely, the underlying constant is smaller than that of* `RSort2`.

Proof of the asymptotic optimality of `RSort3` is exactly same as `RSort2` (Theorem 3), because the only change between `RSort3` and `RSort2` is the creation of dirty sequence with out making any re-arrangements.

The case of $R = D^{\varepsilon}$: The constraint on B is relaxed in [11] by choosing a value of $R = D^{\varepsilon}$ (for some constant $\varepsilon > 0$). For this version (called `RSort`) of the algorithm, a rearrangement is done every D^{ε} stages. The total number of keys output in these many stages is DBD^{ε}. The corresponding contribution to the length of the dirty sequence is $(DB/R)D^{\varepsilon}$. Thus the length of the dirty sequence in the entire algorithm does not exceed $\frac{(1+\nu)}{\varepsilon}\frac{\log(N/DB)}{\log(D)}\frac{N}{D^{\varepsilon}}$ with high probability.

When $\varepsilon = 1/2$, the length of the dirty sequence is no more than $2(1+\nu)\frac{\log(N/DB)}{\log(D)}\frac{N}{\sqrt{D}}$ with high probability (for any constant $\nu > 0$). In this case the initial permutation takes $(1+\mu)\frac{\log(N/DB)}{\log(D)}$ passes through the data (for any constant $\mu > 0$). Initial runs can be formed in one pass. Merging of runs takes $2\frac{(1+\nu)}{\varepsilon}\frac{\log(N/DB)}{\log(D)} = 4(1+\nu)\frac{\log(N/DB)}{\log(D)}$ read passes through the data. Under the same settings, `RSort2` takes $8(1+\nu)\frac{\log(N/M)}{\log(M/B)} + 2 = 8(1+\nu)\frac{\log(N/DB)}{\log(D)} + 2$ passes. Since the rearrangement steps are avoided in `RSort3`, the underlying constant is better for `RSort3` than for `RSort2`.

5 Implementation and Experimental Details

We have simulated the PDM on a single disk computer with a focus on counting the number of parallel I/Os needed. Random data has been employed. We have compared the number of parallel I/Os with that of the standard R-Way merge – which refills the keys from the disk whenever it runs out of keys (in memory) from one of the runs being merged. Please note that this way of measuring the performance of a PDM algorithm is preferable since the results obtained could be easily translated onto any other implementation (hardware or otherwise) of a PDM.

A simple way of simulating a PDM with a single disk is to have a file corresponding to each disk. If we are only interested in counting the number of parallel I/O operations needed, the detail on how the disks are implemented is immaterial (See Section 5.1 on how we actually simulate the PDM).

The specific algorithm we have implemented works as follows: In one pass through the data we form initial runs of length DB each. We employ a D-way merge to merge the runs. This in particular means that there will be $\frac{\log(N/DB)}{\log D}$ *levels* of merging (such that in each level we reduce the number of runs by a factor of D). In each level there will be many *iterations*. In each iteration we merge D runs into one.

Consider any iteration of the algorithm where we merge the runs $R_1, R_2, \ldots, R_D$. At the beginning of the iteration, the runs will be stored in the disks as follows. Let the leading block of R_1 be in disk i (for some $0 \leq i \leq (D-1)$). The second block of R_1 will be in disk $(i + 1) \bmod D$, the third block of R_1 will be in disk $(i + 2) \bmod D$, and so on. Also, the leading block of R_2 will be in disk $(i + 1) \bmod D$, the second block of R_2 will be in disk $(i + 2) \bmod D$, etc.

At any time during an iteration of the algorithm, we keep two leading blocks per run in the main memory. In particular, we begin any iteration by bringing in two leading blocks per run. We identify the smallest *DB* keys from the *D* runs. It can be shown (using Chernoff bounds) that, with high probability, we will not have to bring in any more keys from the disks to produce these *DB* smallest keys. These *DB* keys are shipped to the disks in one parallel I/O operation. After this, we examine the runs in the memory and refill the runs so that we will have two leading blocks per run. This refill operation involves reading from the disks enough keys per run so that we will have the required two blocks per run.

The above process is repeated until the *D* runs are merged. An important question is how many I/O operations will be needed to do the refill operation. In the ideal case each run will be consumed at exactly the same rate. In this case the refill operation can be done in one parallel I/O operation. But this may not be the typical case. The runs may be consumed at different rates and hence we may have to read multiple blocks from each disk. Therefore, for each refill operation we figure out the maximum number of blocks that we may have to read in from any disk and charge the refill operation with these many I/Os. All the I/O counts reported in this paper are based on this scheme (See Algorithm 2 on how we count the number of parallel I/Os for each refill operation).

One can employ either the rearrangement scheme of [11] or the dirty sequence accumulation idea proposed in this paper to modify the algorithm. For example, when the maximum number of blocks to be fetched from any disk in a refill operation exceeds a threshold value (e.g., 3) we can either do a rearrangement or clean up with the creation of a dirty sequence. Note that is the major step in which our algorithm differs from the standard R-Way merge significantly; Standard R-Way merge performs the refill operation *on demand*, i.e. whenever it runs out of keys (in-memory) from a run.

Algorithm 1

Outline of our R-Way merge using dirty sequence idea

INPUT : A Binary file of integers

OUTPUT: A Sorted binary file of integers

$runs \leftarrow \frac{N}{DB}$

parallel_io ← 0;

`CreateRuns`(*runs*);

// Merge until $\frac{N}{DB}$ runs become one run

while *runs* > 1 **do**

  consumed_runs ← 0;

  while *consumed_runs* < *D* **do**

    //Fetch first 2B keys from each run *parallel_io* = *parallel_io* + 2;

    keys_filled ← 0;

    begin

      //D-Way merge

      (*min_key*, *runid*) = `GetMinKey(D)`;

      Output[*keys_filled*] = *min_key*;

      keys_filled++;

if *keys_filled* == *DB* **then**

//write and refill

parallel_io = *parallel_io* + `GetMaxParIO(readheads)`;

//Continue merging

if *run runid fully consumed* **then**

consumed_runs++;

//Continue merging

if *ran out of keys from runid* **then**

//Create a run from merged keys (good sequence)

//Write out the unmerged keys (dirty sequence)

//Stop Merging, start fetching 2B keys again

end

$$runs \leftarrow \frac{runs}{D}$$

return *parallel_io*;

5.1 Simulating the PDM Using a Single Disk

We use a single disk machine to simulate the PDM model. We start with a single binary input file (`key_file`) consisting of integers and create runs each of size *DB* and put them in a file called `run_file`. Every run in the `run_file` is preceded by a 4-byte `unsigned long` which is the size of the run following it. As described in the algorithm the runs are striped across the disks and a parallel read or write operation would read or write *DB* keys. In this section we give the details of how we simulated the parallel I/O operations. Every run *i* in the `run_file` has an offset pointer `run_offset[i]` which tells us at which position the next block of the run starts in the `run_file`. This section assumes that the readers are familiar with UNIX `lseek, read` and `write` system calls. Please see [18] for details of UNIX system calls.

5.2 Counting Parallel I/O's

A parallel I/O (for *D* disks) read operation can be simulated as follows. The following operations show how to initialize the `run_offset[i]` values of the consecutive runs. Assume that the file descriptor for the `run_file` is *runfd*.

- `lseek(runfd,(off_t)0,SEEK_SET).`
- `read(runfd,&run_size,sizeof(unsigned long).`
- `run_offset[i] = lseek(runfd, (off_t)0, SEEK_CUR)`

The above steps will read the length of the run and keep track of its offset for the next read of the block. Once we read the current run its length is stored in `run_size`. We can use this to read the next run as follows.

- `lseek(runfd,(off_t)run_size,SEEK_CUR)`
- `read(runfd,&run size,sizeof(unsigned long))`
- `run_offset[i + 1] = lseek(runfd, (off_t)0, SEEK_CUR).`

The above steps indicate how to fill `run_offset[i]` for each run *i*. Now we show how we can do a parallel read. The following steps basically position the disk head corresponding to

each run at `run_offset[i]` and reads a block of size B and updates the `run_offset[i]` so that the next read can get the next block. The block size (page size) on a 32-bit UNIX machine is $B = 4096$.

- `lseek(runfd,(off_t)run_offset[i],SEEK_SET)`
- `read(runfd,(void *)buf,4096)`
- `run_offset[i] = lseek(runfd, (off_t)0, SEEK_CUR)`

With this background on how to simulate the PDM model on a single disk we describe the algorithm based on this framework in the next section.

5.3 Algorithm Implementation

Algorithm 1 gives an outline of the ideas. We start off with $\frac{N}{DB}$ runs and merge D runs each time and during the merge we handle two situations. In the first situation during the D-way merge we may be able to fill up to DB keys without running out of keys from the merge buffers of the D runs. This will successfully complete a *refill* step. The *refill* steps contribute to the bulk of the parallel I/O's because the disk heads may not move uniformly. To compute the parallel I/O's required to *refill* we keep track of how much of a run is consumed (or read into main memory) in a variable called `runhead`. So `runhead[i]` at any stage of the D-way merge indicates how much of run i has been consumed. This `runhead` information is passed to the parallel I/O computation algorithm in Algorithm 2. This algorithm finds for each run in which disk the leading block of the run is found and then sorts these numbers and finds the maximum number of times a disk is repeated. This gives the maximum number of parallel I/O's needed for the refill operation.

Algorithm 2

`GetMaxParIO` computes parallel I/O's to refill

INPUT : *runheads* of the D runs

OUTPUT: Parallel I/O's to refill

//Find the disk number of leading block

//of each run

for $i = 0$ to D do

$$disk\_number[k] = (\lceil \frac{runhead[i]}{B} \rceil + k) \bmod (D);$$

`Sort`(*disk_number*);

temp_par_io ← 1;

max_par_io ← 1;

for $i = 1$ *to* D **do**

  if *disk_number*[$i - 1$] == *disk_number*[i] **then**

    temp par_io = *temp_par_io* + 1;

  else

    if *temp_par_io* > *max_par_io* **then**

      max_par_io = *temp_par_io*;

    temp_par_io = 1;

return *max par io*;

5.4 Discussion On The Results

Our goal in this section is to demonstrate the parallel I/O efficiency of our algorithm when compared with the standard R-Way merge algorithm which suffers from the *read parallelism* while it refills the keys. We present our results on two random data sets for $D = 16$ and $D = 32$. Our first data set contains keys of size 24 bytes and our second set contains keys of size 8 bytes. In all our experiments we assume $M = 2 \times D \times B$, the block size B in our experiments is 4096. Table 1 shows the results on the data set with key size 24 bytes and Table 2 shows the results on the data set with key size 8 bytes. In Tables 1 and Table 2 column **P2-I/Os** indicates the parallel I/Os required to sort the final dirty sequence. Note that when the length of the dirty sequence is smaller than $M = 2DB$ we can sort the dirty sequence in memory and would not need extra I/Os. The column **P1-I/Os** is equal to the total number of I/Os (**Tot. I/Os**) minus the I/Os needed to sort the dirty sequence. Note that **P1-I/Os** also includes the I/Os to merge the sorted dirty sequence with the original sequence. The last column in both the tables shows the improvement we get by using the idea of dirty sequence during refilling of the keys. Also note that when the runs are consumed uniformly the dirty sequence is not generated and the number of parallel I/Os for both the algorithms match. On the data set with key size 24 bytes we get an average improvement of $1.3X$ and $1.9X$ on the data set with 8 byte keys.

5.5 Comparison with STXXL

Although our goal is to count the parallel I/Os we compare our implementation of R-Way merge(http://lib-ex-sort.sourceforge.net) with algorithm `stxxl::sort` (http://stxxl.sourceforge.net). We compare both the algorithms when $D = 1$ and $M = 700MB$. When $D = 1$ our algorithm does not create any dirty sequence. However we compare both the algorithms for practical interest. We have used a $32 - bit$ machine with $1024MB$ of RAM and an external hard-drive for our experiments. We used randomly generated phone call log data (which was part of `stxxl::sort` tutorial) for comparison. The size of each key is 24 bytes. The `stxxl::sort` operated on a `stxxl::vector` container. We measure real time, system time and number of page faults for both the algorithms. All the reported data is the average of two runs of both the algorithms. We notice that `stxxl::sort` consistently has more page faults than our implementation. Also the system time of `stxxl::sort` is consistently higher than our implementation. Finally our implementation of R-Way merge is on an average $1.2X$ faster than `stxxl::sort`. Note that a number of engineering tricks have been employed in `stxxl::sort`. However this is not the case for the implementation of our algorithm.

6 Conclusions

In this paper we have presented an idea which improves the read parallelism of the R-way merge in the PDM model. Our idea is based on a novel concept called the *dirty sequence accumulation* which collects the keys from partially merged runs whenever one of the runs in the merge runs out of keys. We analytically show that the size of the this dirty sequence is small, thus improving the read parallelism. We verify experimentally that this idea performs much better than the standard R-Way merge on the PDM model.

All the data sets used for comparison and implementation of our algorithms can be obtained from the following URL http://trinity.engr.uconn.edu/~vamsik/PDMSorting/index.html.

Highlights

1. Improved I/O parallelism during R-Way merging with a novel concept of a 'Dirty Sequence'.

2. About 1.94X reduction in the parallel I/O's when compared to standard R-Way merge.
3. The simplicity of our idea makes it easy to realize an efficient practical implementation on PDM.

References

1. Aggarwal A, Plaxton G. Optimal parallel sorting in multi-level storage. Proc of the ACM-SIAM SODA. 1994:659–668.
2. Aggarwal A, Vitter JS. The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM. 1988; 31(9):1116–1127.
3. Arge L. The Buffer Tree: A New Technique for Optimal I/O-Algorithms. Proc. 4th International Workshop on Algorithms and Data Structures (WADS). 1995:334–345.
4. Barve R, Grove EF, Vitter JS. Simple Randomized Mergesort on Parallel Disks. Parallel Computing. 1997; 23(4–5):601–631.
5. Dementiev R, Sanders P. Asynchronous Parallel Disk Sorting. Proc. ACM Symposium on Parallel Algorithms and Architectures. 2003:138–148.
6. Horowitz, E.; Sahni, S.; Rajasekaran, S. Computer Algorithms. W. H. Freeman Press; 1998.
7. Hutchinson, D.; Sanders, P.; Vitter, J. Duality between prefetching and queued writing with parallel disks; 9th European Symposium on Algorithms 2001; p. 62-73.LNCS 2161
8. Nodine, M.; Vitter, J. Deterministic distribution sort in shared and distributed memory multirocessors; Proc. of the ACM SPAA 1993. p. 120-129.Full version available online from http://www.cs.duke.edu/jsv/Papers/catalog/node16.html
9. Nodine MH, Vitter JS. Greed Sort: Optimal Deterministic Sorting on Parallel Disks. Journal of the ACM. 1995; 42(4):919–933.
10. Rajasekaran S. A Framework for Simple Sorting Algorithms on Parallel Disk Systems. Theory of Computing Systems. 2001; 34(2):101–114.
11. Rajasekaran S, Sen S. Optimal and Practical Algorithms for Sorting on the PDM. IEEE Transactions on Computers. 2008; 57(4)
12. Rajasekaran, S.; Sen, S. PDM Sorting Algorithms That Take A Small Number Of Passes; Proc. International Parallel and Distributed Processing Symposium (IPDPS); 2005.
13. Rajasekaran, S.; Sen, S. A Simple Optimal Randomized Sorting Algorithm for the PDM; Proc. International Symposium on Algorithms and Computation (ISAAC); 2005. p. 543-552.Springer-Verlag Lecture Notes in Computer Science 3827, 2005
14. Sanders P, Enger S, Korst J. Fast concurrent access to parallel disks. Proc of the ACM-SIAM SODA 2000. :849–858.
15. Kundeti V, Rajasekaran S. Efficient PDM Sorting Algorithms. Proc of the High Performance Computing. 2008:97–107.
16. Vitter, JS.; Hutchinson, DA. Distribution Sort with Randomized Cycling; Proc. 12th Annual SIAM/ACM Symposium on Discrete Algorithms; 2001.
17. Vitter JS, Shriver EAM. Algorithms for Parallel Memory I: Two-Level Memories. Algorithmica. 1994; 12(2–3):110–147.
18. Stevens, WR.; Rago, SA. Advanced Programming in the UNIX Environment. 2Ed. Addison-Wesley: Professional Computing Series; 2007.

Appendix A

Chernoff bounds

If a random variable X is the sum of n independent and identically distributed Bernoulli trials with a success probability of p in each trial, the following equations give us concentration bounds of deviation of X from the expected value of np. The first equation is more useful for

large deviations whereas the other two are useful for small deviations from a large expected value.

$$Prob(X \geq m) \leq \left(\frac{np}{m}\right)^m e^{m-np} \tag{1}$$

$$Prob(X \leq (1-\varepsilon)pn) \leq exp(-\varepsilon^2 np/2) \tag{2}$$

$$Prob(X \geq (1+\varepsilon)np) \leq exp(-\varepsilon^2 np/3) \tag{3}$$

for all $0 < \varepsilon < 1$.

Biographies

VAMSI KUNDETI: is a graduate student in Computer Science at University of Connecticut, Storrs, CT. He works in the area of algorithm engineering, combinatorial optimization and applied algorithms for bioinformatics and VLSI Design automation.

SANGUTHEVAR RAJASEKARAN: is a UTC chair professor at the department of Computer Science at University of Connecticut. He conducts research in the area of Applied Algorithms. Some of the areas he had worked are: out-of-core computing, computational biology, parallel sorting and selection, packet routing, web-caching, learning theory, model checking, random variate generation (computer simulations), optimization, cryptography (authorization, authentication, secret sharing), particle simulations, coastal wave simulations, mobile computing, and image processing.

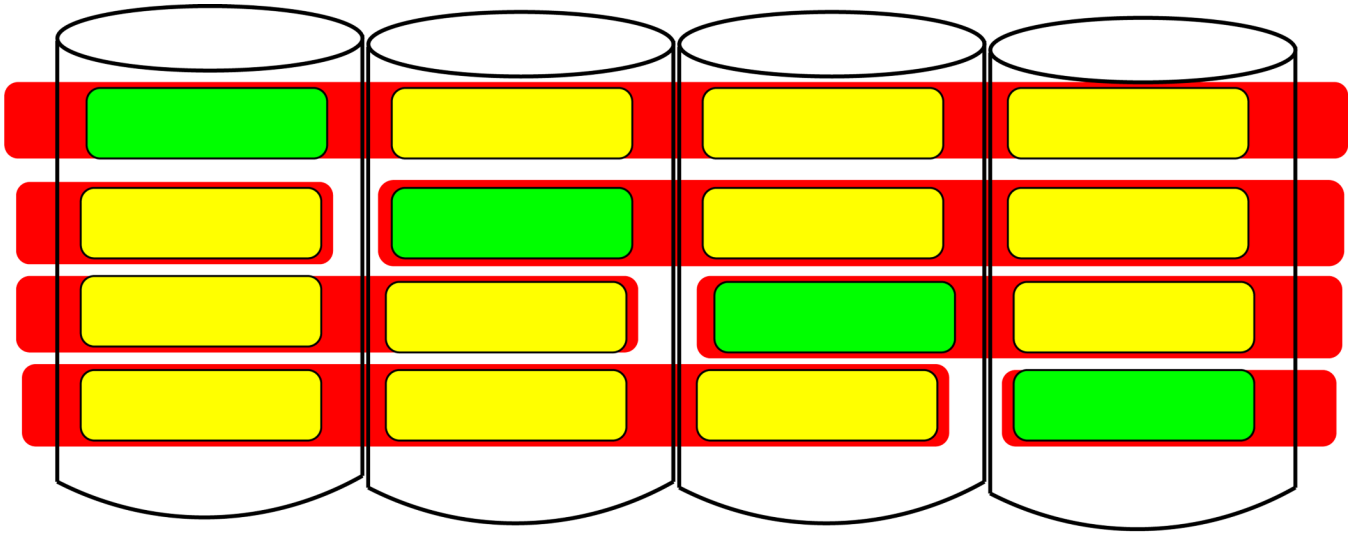
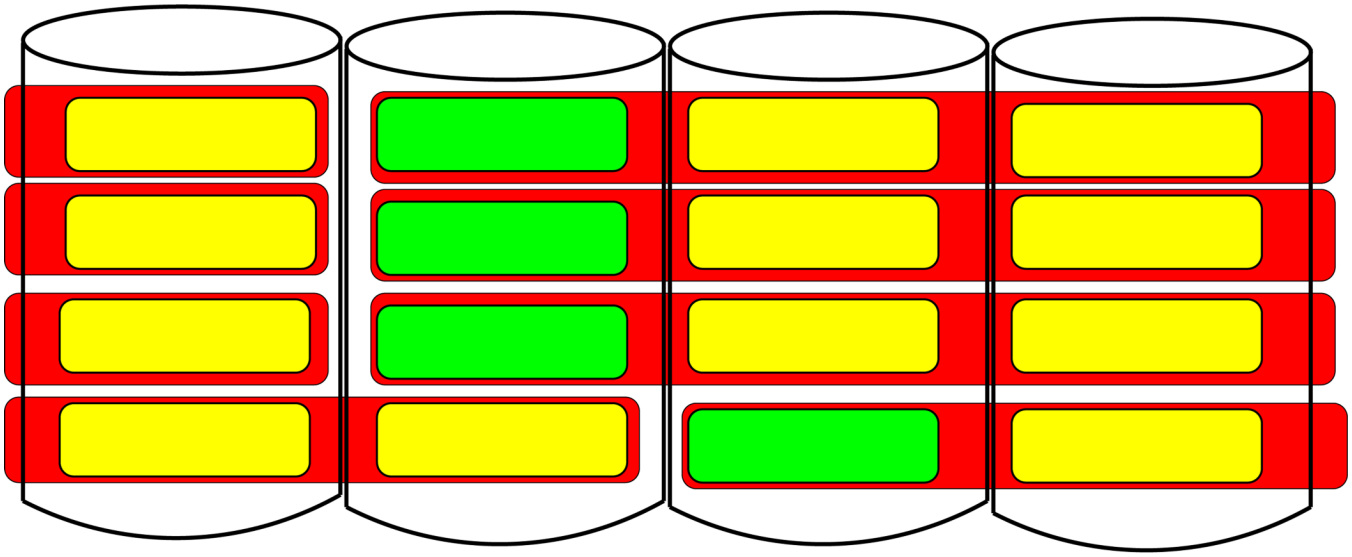


Fig. 1.
Initial arrangement of runs on the PDM. The leading block of each run resides on a different disk. The leading blocks of each of these runs can be read in one parallel I/O during the R-Way merge.



One possible situation which needs 3 Parallel I/Os to fetch the leading blocks of the four runs.

Fig. 2.
One possible situation during R-Way merge; Leading blocks of three runs reside on the same disk.

Table 1

Comparison between the number or parallel I/Os between our algorithm and standard R-Way merge. The main memory size has been fixed at $M = 2 \times D \times B$. The size of each keys is 24 bytes.

		DIRTY R-Way				R-Way	Improvement
N	**D**	**P1-I/Os**	**P2-I/Os**	**D.Len(%)**	**Tot. I/Os**	**Tot. I/Os**	
1048576	16	114	0	0 (0.0000)	114	114	1.0000
2097152	16	315	0	41578 (1.9826)	315	484	1.5365
4194304	16	627	0	87086 (2.0763)	627	966	1.5407
8388608	16	1384	21	174450 (2.0796)	1405	1930	1.3737
16777216	16	2768	37	334189 (1.9919)	2805	3858	1.3754
33554432	16	9549	75	683845 (2.0380)	9624	11812	1.2273
67108864	16	19095	322	1388368 (2.0688)	19417	23622	1.2166
1048576	32	122	0	0 (0.0000)	122	122	1.0000
2097152	32	242	0	0 (0.0000)	242	242	1.0000
4194304	32	482	0	0 (0.0000)	482	482	1.0000
8388608	32	1174	42	345854 (4.1229)	1216	1988	1.6349
16777216	32	2354	75	636348 (3.7929)	2429	3974	1.6361
33554432	32	4711	144	1242484 (3.7029)	4855	7946	1.6367
67108864	32	9430	277	2407181 (3.5870)	9707	15890	1.6370

P2-I/Os number of I/Os to sort and merge the dirty seq.

P1-I/Os number of I/Os excluding I/Os to sort and merge the dirty seq.

D.Len(%) Length of the dirty sequence (percentage)

AVG. IMPROVEMENT IN PARALLEL I/Os: 1.3439

Table 2

Comparison between the number of parallel I/Os between our algorithm and standard R-Way merge. The main memory size has been fixed at $M = 2 \times D \times B$. The size of each key is 8 bytes.

		DIRTY R-Way				R-Way	Improvement
N	**D**	**P1-I/Os**	**P2-I/Os**	**D.Len(%)**	**Tot. I/Os**	**Tot. I/Os**	
1048576	16	31	0	22313 (2.1279)	31	114	3.6774
2097152	16	315	0	41945 (2.0001)	315	484	1.5365
4194304	16	628	0	83793 (1.9978)	628	966	1.5382
8388608	16	1384	21	174590 (2.0813)	1405	1930	1.3737
16777216	16	2766	39	349866 (2.0854)	2805	3858	1.3754
33554432	16	9549	75	681025 (2.0296)	9624	11812	1.2273
67108864	16	19100	313	1351065 (2.0132)	19413	23622	1.2168
134217728	16	38196	625	2709078 (2.0184)	38821	47242	1.2169
1048576	32	122	0	0 (0.0000)	122	122	1.0000
2097152	32	242	0	0 (0.0000)	242	242	1.0000
4194304	32	63	0	118692 (2.8298)	63	482	7.6508
8388608	32	1120	0	257013 (3.0638)	1120	1988	1.7750
16777216	32	2363	66	561324 (3.3458)	2429	3974	1.6361
33554432	32	4729	126	1087345 (3.2405)	4855	7946	1.6367
67108864	32	9445	261	2266083 (3.3767)	9706	15890	1.6371
134217728	32	18886	1081	4546775 (3.3876)	19967	31778	1.5915

P2-I/Os number of I/Os to sort and merge the dirty seq.

P1-I/Os number of I/Os excluding I/Os to sort and merge the dirty seq.

D.Len(%) Length of the dirty sequence (percentage)

AVG. IMPROVEMENT IN PARALLEL I/Os: 1.9431

Table 3

Comparison between our R-Way merge algorithm and `stxxl:sort`

	OUR ALGO			STXXL SORT			SPEED-UP
N	real	sys.	pg.faults	real	sys.	pg.faults	
15728640	55.215	2.995	2	58.355	24.5	6.2	1.057
20971520	94.79	6.925	3	113.23	28	8.305	1.195
26214400	120.105	8.94	1.5	157.465	28	10.63	1.311
31457280	153.435	10.04	2	207.29	30	17.345	1.351
36700160	185.865	11.52	4	242.335	44	20.46	1.304
41943040	214.615	13.605	31	270.8	53.5	23.725	1.262
47185920	245.98	14.77	7.5	306.95	38.5	26.51	1.248
52428800	272.19	16.14	31.5	336.58	44	29.23	1.237
57671680	300.66	17.815	31	367.715	150	32.36	1.223
62914560	331.82	19.54	33	399.625	45.5	35.55	1.204
68157440	359.04	20.76	8	434.735	46	38.02	1.211
73400320	386.49	22.835	12	460.735	45	41.23	1.192
78643200	422.54	23.69	9.5	503.075	142	43.675	1.191
83886080	453.745	25.385	47	525.84	128	47.49	1.159

AVG. REAL TIME SPEEDUP: 1.225