Parallel database sorting

David Taniar a,*, J. Wenny Rahayu b

a School of Business Systems, Monash University, P.O. Box 63B, Clayton, Vic. 3800, Australia
b Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, Vic. 3083, Australia

Received 9 January 2001; received in revised form 27 September 2001; accepted 1 November 2001

Abstract

Sorting in database processing is frequently required through the use of Order By and Distinct clauses in SQL. Sorting is also widely known in computer science community at large. Sorting in general covers internal and external sorting. Past published work has extensively focused on external sorting on uni-processors (serial external sorting), and internal sorting on multi-processors (parallel internal sorting). External sorting on multi-processors (parallel external sorting) has received surprisingly little attention; furthermore, the way current parallel database systems do sorting is far from optimal in many scenarios. In this paper, we present a taxonomy for parallel sorting in parallel database systems, which covers five sorting methods: namely parallel merge-all sort, parallel binary-merge sort, parallel redistribution binary-merge sort, parallel redistribution merge-all sort, and parallel partitioned sort. The first two methods are previously proposed approaches to parallel external sorting which have been adopted as status quo of parallel database sorting, whereas the latter three methods which are based on redistribution and repartitioning are new that have not been discussed in the literature of parallel external sorting. Performance of these five methods is investigated and the results are reported.

© 2002 Elsevier Science Inc. All rights reserved.

Keywords: External sorting; Internal sorting; Sorting in database queries; Parallel sorting; Parallel databases

* Corresponding author.
E-mail addresses: David.Taniar@infotech.monash.edu.au (D. Taniar), wenny@cs.latrobe.edu.au (J.W. Rahayu).

0020-0255/02/$ - see front matter © 2002 Elsevier Science Inc. All rights reserved.
PII: S0020-0255(02)00196-2
1. Introduction

Sorting is one of the most common operations in database processing [8]. Sorting may be requested explicitly by users through the use of Order By clause in SQL [15]. The Order By clause basically requires the query results to be ordered on the designated attributes in ascending or descending order. Another operation, which also requires sorting is duplicate removal through the use of Distinct keyword in SQL [15]. The Distinct operation basically removes all duplicates found in the query result. This can be achieved by first sorting the query results and then followed by removing duplicates through scanning [8].

Sorting may also be required in join operations through the use of sort-merge join algorithm [19]. This is less explicit than the use of Order By and Distinct clauses in SQL. However, some query optimizers allow users to specify any join algorithms to be used when invoking an SQL query [17].

The topic of sorting in traditional data structure and algorithm subjects is divided into two areas: namely internal and external sortings [13]. Internal sorting is where sorting takes place totally in main memory. The data to be sorted are assumed to be small and fits the main memory. Internal sorting has been a foundation of computer science. A number of internal sorting both serial and parallel have been explored [3]. External sorting on the other hand is where the data to be sorted are large and reside in secondary memory. Thus, external sorting is also known as file sorting. In databases, since data are stored in tables (or files) and are normally very large, database sorting is therefore an external sorting. External sorting is not really a new research topic. It has been explained in computer science textbooks [6]. However, external sorting has always been discussed in a uni-processor environment through the use of multiple disks or tapes [7]. Parallel external sorting has not been fully explored. The traditional approaches of parallel external sorting have been to perform local sort in each processor in the first stage and to carry out merging by the host or using a pipeline hierarchy of processors in the second stage [2,3,18].

It is the aim of this paper to fully explore parallel external sorting for high performance parallel database systems. We assume that the parallel database architecture used is a shared-nothing architecture, where each processor has its own processor and memory (main and secondary) [1]. In the taxonomy, besides the two traditional approaches, we add with three new approaches based on the redistribution/repartitioning methods. In these approaches, parallelism is better achieved because bottleneck problem and inefficient merging are solved through redistribution and repartitioning.

Before we present the taxonomy, it is necessary for the reader to be familiar with the concept of serial external sorting, since this is the foundation of parallel external sorting as local sort in each processor is actually a serial external sorting. A brief background on serial external sorting will be discussed in
Section 2. The taxonomy is then presented in Section 3. Section 4 describes the cost models. Section 5 presents some performance analyses. Finally, Section 6 gives the conclusions and explains future work.

2. Serial external sorting: a background

Serial external sorting is external sorting in a uni-processor environment. The most common serial external sorting algorithm is based on sort–merge. The underlying principle of sort–merge algorithm is to break the file up into unsorted subfiles, sort the subfiles, and then merge the sorted subfiles into larger and larger sorted subfiles until the entire file is sorted. Notice that the first stage is to sort the first lot of subfiles, whereas the second stage is actually the merging phase. In this scenario, it is important to determine the size of the first lot of subfiles, which are be sorted. Normally, each of these subfiles must be small enough to fit into main memory, so that sorting these subfiles can be done in main memory using any internal sorting technique. In other words, the size of these subfiles is usually determined by the buffer size in main-memory, which is to be used for sorting each subfile internally. A typical algorithm for external sorting using $B$ buffers is presented in Fig. 1.

The algorithm presented in Fig. 1 is divided into two phases: sort and merge phases. The merge phase consists of loops and each run in the outer loop is called a pass, and subsequently the merge phase contains $i$ passes, where $i = 1, 2, \ldots$. For consistency, the sort phase is named pass 0.

To explain the sort phase, consider the following example. Assume the size of the file to be sorted is 108 pages and we have 5 buffer pages available ($B = 5$ pages). First read 5 pages from the file, sort them, and write them as one subfile into the disk. Then read, sort, and write another 5 pages. In the last run, read, sort, and write 3 pages only. As a result of this sort phase, $\lceil 108/B \rceil = 22$

**Algorithm Serial External Sorting**

**Sort phase - Pass 0**
1. Read $B$ pages at a time into memory
2. Sort them, and Write out a subfile
3. Repeat steps 1-2 until all pages have been processed

**Merge phase - Pass $i = 1, 2, \ldots$**
4. While the number of subfiles at end of previous pass is $> 1$
5. While there are subfiles to be merged from previous pass
6. Choose $B-1$ sorted subfiles from the previous pass
7. Read each subfile into an input buffer page at a time
8. Merge these subfiles into one bigger subfile
9. Write to the output buffer one page at a time

Fig. 1. External sorting algorithm based on sort–merge.
subfiles, where the first 21 subfiles are of size 5 pages each, and the last subfile is only 3-page long.

Once the sorting of subfiles is completed, the merge phase starts. Continuing the example above, we will use \( B - 1 \) buffers (i.e. 4 buffers) for input and 1 buffer for output. The merging process is as follows. In pass 1, we first read 4 sorted subfiles which are produced in the sort phase. Then, we perform a 4-way merging (because only 4 buffers are used as input). This 4-way merging is actually a \( k \)-way merging, which in this case \( k = 4 \), since input buffers is 4 (i.e. \( B - 1 \) buffers = 4 buffers). An algorithm for a \( k \)-way merging is explained in Fig. 2.

The above 4-way merging is repeated until all subfiles (e.g. 22 subfiles from pass 0) are processed. This process is called pass 1, and produces \( \lceil 22/4 \rceil = 6 \) subfiles of 20 pages each, except for the last run which is only 8 pages long.

The next pass, pass 2, is to repeat the 4-way merging to merge the 6 subfiles produced in pass 1. We then first read 4 subfiles of 20 pages long and perform a 4-way merge. This results in a subfile of 80 pages long. Then we read the last 2 subfiles one of which is 20 pages long and the other is only 8 pages long, and merge them to become the second subfile in this pass. So, as a result, pass 2 produces \( \lceil 6/4 \rceil = 2 \) subfiles.

Finally, the final pass, pass 3, is to merge the 2 subfiles produced in pass 2, and to produce a sorted file. The process stops as there is no more subfiles.

In the above example, using 108 pages file and 5 buffer pages, we require to have 4 passes, where pass 0 is the sort phase, and passes 1–3 are the merge phases. Number of passes can be calculated as follows. The number of passes to sort a file with \( B \) buffers available is \( \lceil \log_{B-1} \left[ \text{file size} / B \right] \rceil + 1 \), where \( \lceil \text{file size} / B \rceil \) is actually the number of subfiles produced in pass 0, and \( \lceil \log_{B-1} \left[ \text{file size} / B \right] \rceil \) is number of passes in the merge phase.

In each pass, we read and write all the pages (e.g. 108 pages). Therefore the total I/O cost for the overall serial external sorting can be calculated as \( 2 \times \text{file size} \times \text{number of passes} = 2 \times 108 \times 4 = 864 \) pages. More comprehensive cost models for serial external sort are later explained in Section 4.

**Algorithm** \( k \)-way Merging (input files \( f_1, f_2, \ldots, f_n \); output file \( f_o \))

```plaintext
/* Sort files \( f_1, f_2, \ldots, f_n \), based on the attributes \( a_1 \) of all files */

1. Open files \( f_1, f_2, \ldots, f_n \).
2. Read a record from files \( f_1, f_2, \ldots, f_n \).
3. Find the smallest value among attributes \( a_1 \) of the records from step 2. Store this value to \( a_x \) and the file to \( f_x \) (\( f_1 \leq f_2 \leq f_n \)).
4. Write \( a_x \) to an output file \( f_o \).
5. Read a record from file \( f_x \).
6. Repeat steps 3-5, until no more record in all files \( f_1, f_2, \ldots, f_n \).
```

Fig. 2. \( k \)-Way merging algorithm.
As shown in the above example, an important aspect of serial external sorting is the buffer size, where each subfile comfortably fits into main-memory. The bigger the buffer (main memory) size, the lesser number of passes taken to sort a file, resulting performance gain. The following table (see Table 1) gives an illustration how performance is improved when the number of buffers increases.

In terms of total I/O cost, the number of passes is a determinant factor. For example, to sort 1 billion pages, performance with 129 buffers is six times smaller than that with 3 buffers (e.g. 30:5 = 6:1).

There are a number of variations to the above serial external sort–merge explained above, such as to use a double buffering technique, or a blocked I/O method. As our concern is not in the serial part of external sorting, our assumption of serial external sorting is based on the above sort–merge technique using \( B \) buffers.

As stated in the beginning that serial external sort is the basis for parallel external sort, because in a multi-processor system, particularly in a shared-nothing environment, each processor has its own data, and sorting this data locally in each processor is done as per serial external sort explained above. Therefore, the main concern in parallel external sort is not on the local sort, but when local sort is carried out (i.e. local sort is done first or later), and how merging is performed. The next section describes different ways of parallel external sort by basically considering the two factors mentioned above.

3. A taxonomy of parallel external sort

In this section, we present five parallel external sort methods for parallel database systems. They are parallel merge-all sort, parallel binary-merge sort, parallel redistribution binary-merge sort, parallel redistribution merge-all sort, and parallel partitioned sort. Each of them will be described in more details in the following.

<table>
<thead>
<tr>
<th>( R )</th>
<th>( B = 3 )</th>
<th>( B = 5 )</th>
<th>( B = 9 )</th>
<th>( B = 17 )</th>
<th>( B = 129 )</th>
<th>( B = 257 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1 million</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10 million</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100 million</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1 billion</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
3.1. Parallel merge-all sort

Parallel merge-all sort method is a traditional approach, which has been adopted as the basis for implementing sorting operations in several research prototypes (e.g. Gamma) [4] and some commercial Parallel DBMS. Parallel merge-all sort is composed of two phases: local sort and final merge. The local sort phase is carried out independently in each processor. Local sorting in each processor is performed as per normal serial external sorting mechanism. We emphasize that it is a serial external sorting as it is assumed that the data to be sorted in each processor are very large and they cannot be fitted into the main memory, and hence external sorting (as opposed to internal sorting) is required in each processor.

After local sort phase is completed, the second phase: final merge phase starts. In this final merge phase, the results from the local sort phase are transferred to the host for final merging. The final merge phase is carried out by one processor; namely the host. An algorithm for a k-way merging is previously explained in Fig. 2.

Fig. 3 gives an illustration of parallel merge-all sort process. For simplicity, we use a list of numbers, which are to be sorted. In the real world, the list of numbers is actually a list of records from very large tables.

Looking at the graphical illustration in Fig. 3, parallel merge-all sort is simple, as it is a one-level tree. Load balancing in each processor at the local sort phase is relatively easy to achieve, especially if a round-robin data placement technique is used in the initial data partitioning [5]. It is also easy to
predict the outcome of the process, as performance modeling of such process is relatively straightforward.

Despite its simplicity, parallel merge-all sort method incurs an obvious problem, particularly in the final merging phase, as merging in one processor is heavy. This is true especially if number of processors is large, and there is a limit of number of files to be merged (i.e. limitation in number of files to be opened). Another factor in merging is the buffer size as mentioned earlier in serial external sorting.

Another problem with parallel merge-all sort is network contention, as all temporary results from each processor in the local sort phase are passed to the host. The problem of merging by one host is to be tackled by the next sorting scheme whereby merging is not done by one processor but shared by multiple processors in a form of hierarchical merging.

3.2. Parallel binary-merge sort

*Parallel binary-merge sort* is first proposed by Bitton et al. [2,3]. The first phase of *parallel binary-merge sort* is a *local sort* as like in parallel merge-all.
sort. The second phase: the **merging phase** is pipelined, instead of concentrating on one processor. The way the merging phase works is by taking the results from two processors, and merging the two in one processor. As this merging technique uses only two processors, this merging is called “Binary Merging”. The result of the merging between two processors is passed on to the next level until one processor left; that is the host. Subsequently, the merging process forms a hierarchy. Fig. 4 gives an illustration of the process.

The main motivation to use parallel binary-merge sort is that the merging workload is spread to a pipeline of processors, instead of one processor. It is true however that final merging has still to be done by one processor.

Some of the benefits of parallel binary-merge sort are similar to those of parallel merge-all sort, such as balancing in local sort can be done if a round-robin data placement is initially used to the raw data to be sorted. Another benefit as stated before that merging workload is now shared among processors.

However, problems relating to the heavy merging workload in the host still exists, even though it is now the final merging only merges a pair of list of sorted data, not a $k$-way merging like that in parallel merge-all sort. Binary merging can still be time consuming, particularly if the two lists to be merged are very large. Fig. 5 gives an illustration of binary-merge vs $k$-way merge, which is carried out by the host.

The main difference between $k$-way merging and binary merging is that in $k$-way merging there is a searching process in the merging; that is to search the smallest value among all values compared at the same time. In binary merging, this searching is purely a comparison between two values compared at one time.

Regarding the system requirement, $k$-way merging requires sufficient number of files to be opened at the same time. This requirement is trivial in binary merging, as it only requires a maximum of two files to be opened, and this is easily satisfied by any operating systems.

![Parallel Merge-All Sort vs Parallel Binary-Merge Sort](image)

*Fig. 5. Binary-merge vs $k$-way merge in the merging phase.*
Pipeline system as in the binary merging will certainly produce extra work through the pipe itself. The pipeline mechanism also produces a higher tree, not a one-level tree as in the previous method. However, if the limitation of the number of opened files permitted in the $k$-way merging, parallel merge-all sort will incur merging overheads.

In parallel binary-merge sort, it is still no true parallelism in the merging as only a subset, not all, of available processors is used.

We proposed three possible alternatives using the concept of redistribution or repartitioning. The first approach we would like to introduce is a modification of parallel binary-merge sort by incorporating redistribution in the pipeline hierarchy of merging. The second approach is an alteration to parallel merge-all sort, also through the use of redistribution. The third approach we would like to include in this paper differs from the others, as local sorting is delayed after partitioning is done. The details of these three methods are described next.

3.3. Parallel redistribution binary-merge sort

Parallel redistribution binary-merge sort is motivated by parallelism at all levels in the pipeline hierarchy. Therefore, it is similar to parallel binary-merge sort where both methods use hierarchy pipeline for merging local sort results, but differ in the context of number of processors involved in the pipe. Using parallel redistribution binary-merge sort, all processors are used in each level in the hierarchy of merging.

The steps for parallel redistribution binary-merge sort can be described as follows. First, a local sort is carried out in each processor as like in the previous sorting methods. Second, redistribute the results of local sort to the same pool of processors. Third, do a merging using the same pool of processors. Finally, repeat the above two steps until final merging. The final result is the union of all temporary results obtained in each processor. Fig. 6 gives an illustration of parallel redistribution binary-merge sort method.

Notice from the illustration that in the final merge phase, some of the boxes are empty (e.g. gray boxes). They indicate that they do not receive any values from the designated processors. For example, the first gray box on the left is because there are no values ranging from 1 to 5 from processor 2. Practically, in this example, processor 1 performs final merging of two lists, because the other two lists are empty.

Also notice that the results produced by the intermediate merging in the above example are sorted within and among processors. It means that for example, processors 1 and 2 produce a sorted list each, and the union of these results is also sorted where the results from processor 2 are preceded by those from processor 1. This is applied to other pairs of processors. Each pair of processors in this case forms a pool of processor. In the next level of merging,
two pools of processors use the same strategy as in the previous level. Finally, in the final merging, all processors will form one pool, and therefore, results produced in each processor are sorted, and these results union altogether are sorted based on the processor order. In some systems, this is already a final result. If there is a need to place the results in one processor, results transfers are then carried out.

The apparent benefit of this method is that merging becomes lighter compared to those without redistribution, because merging is now shared by multiple processors, not monopolized by just one processor. Parallelism is therefore accomplished at all levels of merging, even though performance
beneficial of this mechanism is restricted (in the performance evaluation section, this matter will be further clarified and quantified).

The same problem as that in without redistribution method, which relates to the height of the tree remains outstanding. This is due to the fact that merging is done in a pipeline format. Another problem raised by the redistribution is skew [14]. Although initial placement in each disk is balanced through the use of round-robin data partitioning, redistribution in the merging process may likely produce skew, as shown in Fig. 6. Modeling skew is also known to be difficult, and hence simplified assumption is often used, such as using a Zipf distribution to model skew [20]. Like the merge-all sort method, final merging in the redistribution method is also dependent upon the maximum number of files opened.

3.4. Parallel redistribution merge-all sort

Parallel redistribution merge-all sort is motivated by two factors, particularly reducing the height of the tree while maintaining parallelism at the merging stage. This can be achieved by exploiting the feature of parallel merge-all and parallel redistribution binary-merge methods. In other words, parallel redistribution is a two-phase method (local sort and final merging) like parallel merge-all sort, but does a redistribution based on a range partitioning. Fig. 7 gives an illustration of parallel redistribution merge-all sort.

As shown in Fig. 7, parallel redistribution merge-all sort is a two-phase method, where in phase one, local sort is carried out like in other methods, and in phase two, results from local sort are redistributed to all processors based on a range partitioning, and merging is then performed by each processor.

Like in parallel redistribution binary-merge sort, empty boxes drawn by gray boxes are actually empty list as a result of data redistribution. In the above example, processor 4 has three empty lists coming from processors 2, 3, and 4, as they do not have values ranging from 16 to 20 specified by the range partitioning function.

Also notice that the final results produced in the final merging phase in each processor are sorted, and these are also sorted among all processors based on the order of the processors specified by the range partitioning function.

The advantage of this method is the same as that in parallel redistribution binary-merge sort, including true parallelism in the merging process. However, the tree of parallel redistribution merge-all sort is not a tall tree as in the parallel redistribution binary-merge sort. It is in fact a one-level tree like in parallel merge-all sort.

Not only the advantages of parallel redistribution merge-all sort are mirroring those in parallel merge-all sort and parallel redistribution binary-merge sort, but also the problems. Skew problems found in parallel redistribution binary-merge sort also exist in this method. Consequently, skew modeling
needs some simplified assumptions as well. Additionally, bottleneck problem in merging which is similar to that is parallel merge-all sort is also common here, especially if the number of processor is large and exceeds the limit of number of files can be opened at once.

3.5. Parallel partitioned sort

Parallel partitioned sort is influenced by the techniques used in parallel partitioned join, where the process is split into two stages: partitioning and independent local work [16,19]. In parallel partitioned sort, first we partition local data according to range partitioning used in the operation. Notice the difference between this method and others. In this method, the first phase is not a local sort. Local sort is not carried out here. Each local processor scans its records and redistribute or repartition according to some range partitioning. After partitioning is done, each processor will have an unsorted list in which the values come from various processors (places). It is then local sort is carried out. Thus, local sort is carried out after the partitioning, not before. It is also noticed that merging is not needed. The results produced by the local sort are...
already the final results. Each processor will have produced a sorted list, and all processors in the order of the range partitioning method used in this process are also sorted. Fig. 8 shows an illustration of this method.

The main benefit of parallel partitioned sort is that no merging is necessary, and hence the bottleneck in merging is avoided. It is also a true parallelism, as all processors are being used in the two phases. And most importantly, it is a one-level tree reducing unnecessary overhead in the pipeline hierarchy.

Despite these advantages, the problem remain outstanding is skew which is produced by the partitioning. This is a common problem even in the partitioned join. Load balancing in this situation is often carried out by producing more buckets than the available number processors, and workload arrangement of these buckets can then be carried out to evenly distribute buckets into processors [10–12]. For example, in Fig. 9, seven buckets are created for three processors. The size of each bucket is likely to be different, and after the buckets are created, bucket placement and arrangement are performed to make the workload of the three processors balanced. For example, buckets A, B, and

---

Fig. 8. Parallel partitioned sort.
G go to processor 1, buckets C and F to processor 2, and the rest to processor 3. In this way, the workload of these three processors will be balanced.

However, bucket tuning in the original form as shown in Fig. 9 is not relevant to parallel sort. This is because in parallel sort, the order of the processors is important. In the above example, buckets A will have values which are smaller than those in bucket B, and values in bucket B are smaller than those in bucket C, etc. Then buckets A–G are in order. The values in each bucket are to be sorted, and once they are sorted, the union of values from each bucket together with the bucket order produces a sorted list. Imagine that bucket tuning as shown in Fig. 9 is applied to parallel partitioned sort. Processor 1 will have three sorted lists; from buckets A, B, and G. Processors 2 and 3 will have two sorted lists each. However, since the buckets in the three processors are not in the original order (i.e. A–G), the union of sorted lists from processors 1, 2 and 3 will not produce a sorted list, unless further operation is carried out. We reserve the discussion on load balancing for future work, and thus it is out of scope of this paper.

4. Cost models

In order to study the behavior of the algorithms, we need to model them in terms of cost models. The notations used by the cost models are presented in Table 2. They are basically comprised of system and data parameters, time unit costs, and communication unit costs.

Before presenting the cost models for each of the five parallel external sorting discussed in the previous section, we will first start studying the cost models for serial external sort and skew model. These two cost models are the
foundation of cost models for the parallel versions, and understanding them is important in the context of parallel external sort. Some readers may find this detailed analysis tedious – we recommend that those readers skip over the cost models, and continue straight to the sensitivity analysis in the later section.

4.1. Cost models for serial external merge-sort

There are two main cost components for serial external sort, particularly the costs relating to I/O and those relating to CPU processing. The I/O costs are the disk costs, which consist of load cost and save cost. These I/O costs are as follows.

- **Load cost** is the cost for loading data from disk to main memory. Since data loading from disk is done page by page, the table size is divided by the page size to obtain number of pages. The load cost is the product of number of pages of the table, number of passes, and I/O unit cost:

  \[
  \text{Number of pages} \times \text{Number of passes} \times \text{Input Output unit cost},
  \]

  \[
  \text{where}
  \]

  \[
  \text{Number of pages} = \frac{R}{P}, \quad \text{and}
  \]

  \[
  \text{Number of passes} = (\lceil \log_{B^{-1}}(R/P/B) \rceil + 1). \tag{1}
  \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>Number of processors</td>
</tr>
<tr>
<td>( R )</td>
<td>Table size</td>
</tr>
<tr>
<td>( R_i )</td>
<td>Size of table fragment on node ( i )</td>
</tr>
<tr>
<td>(</td>
<td>R</td>
</tr>
<tr>
<td>(</td>
<td>R_i</td>
</tr>
<tr>
<td>( P )</td>
<td>Page size</td>
</tr>
<tr>
<td>( B )</td>
<td>Buffer size</td>
</tr>
<tr>
<td>( IO )</td>
<td>Effective time to read a page from disk</td>
</tr>
<tr>
<td>( t_r )</td>
<td>Time to read a record</td>
</tr>
<tr>
<td>( t_w )</td>
<td>Time to write a record</td>
</tr>
<tr>
<td>( t_m )</td>
<td>Time to merge</td>
</tr>
<tr>
<td>( t_s )</td>
<td>Time to compare and swap two keys</td>
</tr>
<tr>
<td>( t_v )</td>
<td>Time to move a record</td>
</tr>
<tr>
<td>( t_d )</td>
<td>Time to compute destination</td>
</tr>
<tr>
<td>( m_p )</td>
<td>Message protocol cost per page</td>
</tr>
<tr>
<td>( m_c )</td>
<td>Message latency for one page</td>
</tr>
</tbody>
</table>
Hence, the above load cost becomes

\[(R/P) \times ([\log_{B-1}(R/P/B)] + 1) \times IO.\]

- **Save cost** is the cost for writing data from main memory back to disk. The save cost is actually identical to the load cost, since number of pages loaded from disk are the same as number of pages written back to disk. No filtering to the input file has been done in sorting.

The CPU cost components are determined by the costs involved in getting record out of data page, sorting, merging, and generating results, which are as follows.

- **Select cost** is to get record out of data page, which is calculated as number of records loaded from the disk times reading and writing unit cost to the main-memory. Number of records loaded from the disk is influenced by number of passes, and therefore Eq. (1) above to calculate number of passes is being used here:

\[|R| \times \text{Number of passes} \times (t_r + t_w).\]

- **Sorting cost** is the internal sorting cost which has a \(O(N \times \log_2 N)\) complexity. Using our cost notation, the \(O(N \times \log_2 N)\) complexity has the following cost:

\[|R| \times [\log_2(|R|)] \times t_s.\]

The sorting cost is the cost to process record in pass 0 only.

- **Merging cost** is applied to pass 1 onward. It is calculated as on the number of records being processed, which is also influenced by number of passes in the algorithm, multiply with the merging unit cost. The merging unit cost is assumed to involve a \(k\)-way merging where searching for the lowest value in the merging is incorporated in the merging unit cost. Also bare in mind that number of passes must be subtracted with 1, as the first pass (i.e., pass 0) is used by sorting:

\[|R| \times (\text{Number of passes} - 1) \times t_m.\]

- **Generating result cost** is the number of records being generated or produced in each pass before they are being written to disk multiplies by the writing unit cost:

\[|R| \times \text{Number of passes} \times t_w.\]

### 4.2. Skew

Skew has been one of the major problems in parallel processing. Skew is defined as the non-uniformity of workload distribution among processing elements. In parallel external sorting, there are two different kinds of skew namely: data skew and processing skew.
Data skew is caused by the unevenness of data placement in a disk in each local processor, or caused by previous operator. Unevenness data placement is caused by the fact that data value distribution, which is used in the data partitioning function, may well be non-uniform due to the nature of data value distribution. If initial data placement is based on a round-robin data partitioning function, data skew will not occur. However, it is common that database processing does not involve a single operation only. It sometimes involves many operations, such as selection first, projection second, join third, and sort the last. In this case, although initial data placement is even, other operators may have rearranged the data – some data are eliminated, or joined, and consequently, data skew may occur when the sorting is about to start.

Processing skew is caused by the processing itself, and may be propagated by the data skew initially. For example, a parallel external sorting processing consists of several stages. Somewhere along the process, the workload of each processing element may not be balanced, and this is called processing skew. Notice that even when data skew may not exist in the start of the processing, skew may exist at a later stage of processing. If data skew exists in the first place, it is very likely that processing skew will also occur.

Modeling skew is known to be a difficult task, and often a simplified assumption is used. A number of attempts to model skewness in parallel databases have been reported [14]. Most of them use the Zipf distribution model [20].

Skew is measured in terms of different sizes of fragments that are allocated to the processors for the parallel processing of the operation. Given the total number of records \( |R| \), the number of processors \( N \), and a skew factor \( \theta \); the size of the \( i \)th fragment \( |R_i| \) can be represented by

\[
|R_i| = \frac{|R|}{\theta^i \times \sum_{j=1}^{N} \frac{1}{\theta^j}}, \quad \text{where } 0 \leq \theta \leq 1.
\]

The symbol \( \theta \) denotes the degree of skewness, where \( \theta = 0 \) indicates no skew and \( \theta = 1 \) highly skew. Clearly, when \( \theta = 0 \), the fragment sizes follow a discrete uniform distribution shown in \( |R_i| = |R|/N \). This is an ideal distribution, as there is no skew. In contrast, when \( \theta = 1 \) indicating a high degree of skewness, the fragment sizes follow a pure Zipf distribution. Therefore, the above equation becomes

\[
|R_i| = \frac{|R|}{i \times \sum_{j=1}^{N} \frac{1}{j}} = \frac{|R|}{i \times H_N} \approx \frac{|R|}{i \times (\gamma + \ln N)},
\]

where \( \gamma = 0.57721 \) (Euler’s constant) and \( H_N \) is the Harmonic number [13] which may be approximated by \( (\gamma + \ln N) \). In the case of \( \theta > 0 \), the first fragment \( |R_1| \) is always the largest in size whereas the last one \( |R_N| \) is always the smallest. (Note that fragment \( i \) is not necessarily allocated at processor \( i \).) Thus, the load skew can be given by
\[ |R_{\text{max}}| = \frac{|R|}{\sum_{j=1}^{N} \frac{1}{f_j}}. \]

For simplicity and generality of notation, we use \(|R_i|\) instead of \(|R_{\text{max}}|\). When there is no skew \(|R_i| = |R|/N\), and when it is highly skew

\[ R_i = \frac{|R|}{\sum_{j=1}^{N} \frac{1}{f_j}}. \]

Comparing these two \(|R_i|\), the difference is determined by the divisor of the division. Table 3 gives comparative figures between the two extreme cases. Notice that the divisor with highly skew remains quite steady compared to that of without skew. This indicates that skew can hurt the performance very badly. For example, the divisor without skew for total number of processors 256 is 256, whereas that of with highly skew is only 6.12. Assuming that the total number of records is 100,000; the workload of each processor where distribution is uniform (i.e. \(\theta = 0\)) is around 390 records. In contrast, the most overloaded processor in the case of highly skew (i.e. \(\theta = 1\)) holds more than 16,000 records. Our data skew and processing skew adopt the above Zipf skew model.

### 4.3. Cost models for parallel merge-all sort

We divide the cost models for parallel merge-all sort into two categories: *local merge-sort costs*, and *final merging costs*. Local merge-sort costs are the costs for local sorting in each processor using a merge-sort technique, whereas the final merging costs are the costs for consolidating temporary results from all processing elements at the host.

The *local merge-sort costs* are similar to the serial external merge-sort cost models explained in the previous section, except for two major differences. One difference is that for the local merge-sort costs in parallel merge-all sort, the fragment size to be sorted in each processor is determined by the values of \(R_i\) and \(|R_i|\), instead of just \(R\) and \(|R|\). This is because in parallel merge-all sort, the data have been partitioned to all processors, whereas in the serial external merge-sort, only one processor is used. Since we now use \(R_i\) and \(|R_i|\), these two cost elements may involve *data skew*. As have been explained in the previous section, when skew is involved the values of \(R_i\) and \(|R_i|\) are calculated not by a straight division with \(N\), but with much less value than \(N\) due to skewness.

Table 3

<table>
<thead>
<tr>
<th>Divisors (with vs without skew)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
</tr>
<tr>
<td>Divisor without skew</td>
</tr>
<tr>
<td>Divisor with skew</td>
</tr>
</tbody>
</table>
The second difference is that the local merge-sort costs for parallel merge-all sort involve communication costs, which do not appear in the original serial external sort cost models. The communication costs are the costs associated with the data transfer costs from each processor to the host at the end of the local sorting phase.

The local merge-sort costs, consisting of I/O costs, CPU costs, and communication costs, are summarized as follows:

- **I/O costs**, which consist of load and save costs, are as follows:

  \[
  \text{Save cost} = \text{Load cost} = \left(\frac{R_i}{P}\right) \times \text{Number of passes} \times \text{IO},
  \]

  where \(\text{Number of passes} = ([\log_{B-1}(\frac{R_i}{P}/B)] + 1)\).

  The differences between cost models for parallel merge-all sort and serial external sort are printed in **bold** throughout this section.

- **CPU costs**, which consist of select cost, sorting cost, merging cost, and generating result cost, are as follows:

  \[
  \text{Select cost} = |R_i| \times \text{Number of passes} \times (t_r + t_w),
  \]

  \[
  \text{Sorting cost} = |R_i| \times \log_2(|R_i|) \times t_s,
  \]

  \[
  \text{Merging cost} = |R_i| \times (\text{Number of passes} - 1) \times t_m,
  \]

  \[
  \text{Generating result cost} = |R_i| \times \text{Number of passes} \times t_w,
  \]

  where \(\text{Number of passes}\) is as shown in Eq. (2).

- **Communication costs** for sending local sorted results to the host are given by the number of pages to be transferred multiply by the message unit cost, which is given as follows:

  \[
  \text{Communication cost} = \left(\frac{R_i}{P}\right) \times (m_p + m_e).
  \]

  The final merging costs involve communication costs, I/O costs, and CPU costs. The communication costs are the costs involved when the host receives data from all other processors. The I/O and CPU costs are the costs associated directly with the merging process at the host. The three cost components for the final merging costs are given as follows:

  - **Communication cost**, which is the receiving record cost from local sorting operators is calculated by the number of records being received (in this case the total number of records from all processors) multiply by the message unit cost.

  \[
  \text{Communication cost} = \left(\frac{R}{P}\right) \times m_p.
  \]

  - **I/O costs**, which consist of load and save costs, are influenced by two factors, particularly the total number of records being received and processed, and number of passes in the merging of \(N\) subfiles. When the data are first received from the local sorting operator, the data have to be written out to the disk in the host. Following this, the host starts the \(k\)-way merging
process by first loading the data from local host disk, processing them, and saving the results back to the local host disk.

As the $k$-way merging process may be done at a number of passes, data loading and saving are done as many times as the number of passes in the merging process. Moreover, the total number of data saving is one more than the total number of data loading, as the first data saving must be done when the data is first received by the host:

$$\text{Save cost} = \frac{R}{P} \times (\text{Number of merging passes} + 1) \times \text{IO},$$
$$\text{Load cost} = \frac{R}{P} \times \text{Number of merging passes} \times \text{IO},$$

where $\text{Number of merging passes} = \lceil \log_{B-1}(N) \rceil$.  \hspace{1cm} (3)

Notice that $\text{Number of merging passes}$ is determined by number of processors $N$, and number of buffers. The number of processors $N$ is served as the number of streams in the $k$-way merging, and each stream contains a sorted list of data, which is obtained from the local sorting phase. Since all processors participate in the local sorting phase, the value of $N$ is not influenced by skew. Whether there is data skew or not in the local sorting phase, all processors will have at least one record to work with, and subsequently when these data are transferred to the host, none of the stream is empty.

- $\text{CPU costs}$ consist of the select costs, merging costs, and generating results costs only. Sorting costs are not included since the host does not sort data, but merge only. CPU costs are determined by the total number of records being merged, number of merging passes, and the unit cost:

$$\text{Select cost} = |R| \times \text{Number of merging passes} \times (t_s + t_w),$$
$$\text{Merging cost} = |R| \times \text{Number of merging passes} \times t_m,$$
$$\text{Generating result cost} = |R| \times \text{Number of merging passes} \times t_w,$$

where $\text{Number of merging passes}$ is as shown in Eq. (3).

There are two things that we need to mention regarding the above final merging costs, one is that the host processes all records, and hence we use $R$ and $|R|$ in the cost equations, not $R_i$ and $|R_i|$, and two is that since only one processor, namely the host, is working, the notion of skew does not exist in the cost equation. In other words, data skew may occur in the local sorting phase, but in the final merging phase, only the host performs its work.

4.4. Cost models for parallel binary-merge sort

The cost models for parallel binary-merge sort are divided into two parts: $\text{local merge-sort costs}$, and $\text{pipeline merging costs}$. The $\text{local merge-sort costs}$ are exactly the same as those of parallel merge-all sort, since the local sorting phase in both parallel sorting methods is the same. Therefore, we focus on the cost models for pipeline merging only.
In the pipeline merging, we first need to determine the number of levels in the pipeline. Since we use binary-merge in which each merging takes the results from two processors, the number of levels in the pipeline is $\lceil \log_2(N) \rceil$. Level numbers start from 1, which is the immediate level after local sort, to the last level $\lceil \log_2(N) \rceil$, which is basically a final merging done by one processor, namely the host.

In level 1 in the pipeline, number of processors used is basically up to half, and we use a notation of $N_0$, where $N_0 = \frac{N}{2}$. The implication to the skew equation is that

$$|R'|_i = \frac{|R|}{\sum_{j=1}^{N'} \frac{1}{j}}.$$

Notice that we use the notations $|R'|_i$ and $N'$, where $|R'|_i$ indicates the number of records being processed at a node in a level of pipeline merging, and $N'$ is the number of processors involved. If no skew is involved, $|R'|_i = |R|/N'$.

The process in level 1 basically follows the following order. First, receive records from the local sort operator. Second, save and load these records on local disks. These I/O process is particularly needed especially when the data being transferred is very large, and hence storing them on local disk upon arrival is necessary. The actual merging process starts with data loading from local disk. Third, merge the data, which involves select, merge, and generating results costs. And fourth, transfer the merging results to the next level of pipeline possibly to a different processor. The cost models for these processes are as follows:

- **Receiving cost** = $(R'/P) \times m_p$,
- **Save cost** = $(R'/P) \times IO$,
- **Load cost** = $(R'/P) \times IO$,
- **Select cost** = $|R'|_i \times (t_r + t_w)$,
- **Merging cost** = $|R'|_i \times t_m$,
- **Generating result cost** = $|R'|_i \times t_w$,
- **Data Transfer cost** = $(R'/P) \times (m_p + m_t)$.

In the subsequent levels, the number of processors involved is even reduced by half, due to binary merging. Using the $N'$ notation, the new $N'$ values become $N' = \lceil N'/2 \rceil$. This also gives impact to the skew equation where $N'$ is used. Apart from the number of processors involved in the next level of pipeline merging, the process is the same, and therefore the above cost equations can be used.

In the last level of pipeline merging in which the host performs a final binary merging, $N' = 1$. Another main difference between the last level and previous
levels, in the last level of pipeline merging, data transfer cost is substituted with another save cost, since the final results are not transferred but saved in the host disks.

To summarize, the total pipeline binary merging costs are as follows:

- **Receiving cost**
  \[ R_0^i / P \times \lfloor \log_2(N) \rfloor \times m_p, \]

- **Save cost**
  \[ (R_0^i / P) \times ([\log_2(N)] + 1) \times IO, \]

- **Load cost**
  \[ (R_0^i / P) \times [\log_2(N)] \times IO, \]

- **Select cost**
  \[ |R_0^i| \times [\log_2(N)] \times (t_r + t_w), \]

- **Merging cost**
  \[ |R_0^i| \times [\log_2(N)] \times t_m, \]

- **Generating result cost**
  \[ |R_0^i| \times [\log_2(N)] \times t_w, \]

- **Data Transfer cost**
  \[ (R_0^i / P) \times ([\log_2(N)] - 1) \times (m_p + m_t). \]

It must be stressed that the values of \( R_0^i \) and \( |R_0^i| \) are not constant throughout the pipeline, but increases from level to level as the number of processors \( N' \) used is reduced by half when progressing from one level to another. Another point is that \( R_0^i \) and \( |R_0^i| \) may be affected by processing skew.

### 4.5. Cost models for parallel redistribution binary-merge sort

Like those for parallel binary-merge sort, parallel redistribution binary-merge sort costs have two main components: **local merge-sort costs** and **pipeline merging costs**.

Local sort operation in parallel redistribution binary-merge sort is similar to parallel merge-all sort and parallel binary-merge sort. The main difference is that in parallel redistribution binary-merge sort, temporary results are being redistributed to processors in the next level of operations. This redistribution operation incurs additional overhead, particularly for each record being redistributed, it needs to be determined to where this record is going to be sent based on the partitioning method used. We call this overhead **compute destination cost**:

\[ \text{Compute destination cost} = |R_0^i| \times t_d. \]

Like in parallel merge-all sort and parallel binary-merge sort, \( R_0^i \) in the above equation may involve data skew. Other than that compute destination cost, the local merge-sort costs in parallel redistribution binary-merge sort are the same as those in parallel merge-all sort.

The **pipeline merging costs** in parallel redistribution binary-merge sort are similar to those in parallel “without redistribution” binary-merge sort. We first mention several similarities. One is the number of levels of the pipeline is \([\log_2(N)]\), where level 1 is the first level after the local sorting phase. Two is the
order of the process is similar, starting from data receiving from the network to data transfer to the next level of pipeline.

However, there are a number of principle differences. One is relating to number of processors participating in each level. In parallel redistribution binary-merge sort, all processors participate. Hence, in the cost equations, we should use \( R_i \) and \(|R_i|\), not \( R' \) and \(|R'|\). Another main difference is relating to the compute destination costs, which are absence in the parallel “without redistribution” binary-merge sort costs. Compute destination costs are applicable here in all level of pipeline, except the last one where the results are written back to disk, not redistributed over the network.

In summary, the pipeline merging costs for parallel redistribution binary-merge sort are as follows:

- Receiving cost \( = \left( \frac{R_i}{P} \right) \times \left[ \log_2(N) \right] \times m_p \)
- Save cost \( = \left( \frac{R_i}{P} \right) \times \left( \left[ \log_2(N) \right] + 1 \right) \times IO \)
- Load cost \( = \left( \frac{R_i}{P} \right) \times \left[ \log_2(N) \right] \times IO \)
- Select cost \( = |R_i| \times \left[ \log_2(N) \right] \times (t_r + t_w) \)
- Merging cost \( = |R_i| \times \left[ \log_2(N) \right] \times t_m \)
- Generating result cost \( = |R_i| \times \left[ \log_2(N) \right] \times t_w \)
- Compute destination cost \( = |R_i| \times \left( \left[ \log_2(N) \right] - 1 \right) \times t_d \)
- Data Transfer cost \( = \left( \frac{R_i}{P} \right) \times \left( \left[ \log_2(N) \right] - 1 \right) \times (m_p + m_i) \)

4.6. Cost models for parallel redistribution merge-all sort

Like the other parallel sort methods, parallel redistribution merge-all sort has two main cost components: local merge-sort costs and merging costs.

The local merge-sort costs are the same as those of parallel redistribution binary-merge sort. Both have the compute destination costs, as both redistribute data from local sort phase to the merging phase.

The merging costs are somewhat similar to those of parallel merge-all sort, except one main difference, that is in here we use \( R_i \) and \(|R_i|\), not \( R \) and \(|R|\) in parallel merge-all sort. The reason is simple – in parallel redistribution merge-all sort, all processors are being used in the merging phase, whereas in parallel “without redistribution” merge-all sort, only the host is used in the merging phase. As now, \( R_i \) and \(|R_i|\) are used in the merging costs, both may be affected by processing skew, and hence the previously explained skew model is applied.

The merging costs for parallel redistribution merge-all sort are given as follows:

- Communication cost \( = \left( \frac{R_i}{P} \right) \times m_p \)
- Save cost \( = \left( \frac{R_i}{P} \right) \times (Number \ of \ merging \ passes + 1) \times IO \)
Load cost = \( \frac{R_i}{P} \times \text{Number of merging passes} \times IO \),
Select cost = \( |R_i| \times \text{Number of merging passes} \times (t_r + t_w) \),
Merging cost = \( |R_i| \times \text{Number of merging passes} \times t_m \),
Generating result cost = \( |R_i| \times (\text{Number of merging passes}) \times t_w \),

where

\[ \text{Number of merging passes} = \left\lceil \log_{B-1}(N) \right\rceil. \]

Despite the similarity between the above merging costs for parallel redistribution merge-all sort and those for parallel redistribution binary-merge sort, there are major differences. One is regarding number of levels in the pipeline, which is \( \left\lceil \log_2(N) \right\rceil \) for parallel redistribution binary-merge sort, and 1 for parallel redistribution merge-all sort. Two is regarding number of merging passes as involved in the \( k \)-way merging. In parallel redistribution binary-merge sort, the merging is binary, and hence number of merging passes is 1. In contrast, merging in parallel redistribution merge-all sort is multiple depending on the number of processors \( N \) and number of buffers \( B \), and hence number of merging passes is calculated as \( \left\lceil \log_{B-1}(N) \right\rceil \).

4.7. Cost models for parallel partitioned sort

Parallel partitioned sort costs have two components as well, but not local merge-sort costs and merging costs, but scanning and partitioning costs and local merge-sort costs. As explained previously, in parallel partitioned sort, local sorting is done after the partitioning.

The scanning and partitioning costs involve I/O costs, CPU costs, and communication costs. The I/O cost is basically a load cost during the scanning of all records. The CPU costs mainly involve the select costs and compute destination costs. The communication cost is a data transfer cost from each processor in the scanning/partitioning phase to processors in the sorting phase.

- **I/O costs**, which consist of load costs, are as follows:
  \( \frac{(R_i/P)}{IO} \).

- **CPU costs** consist of select cost which is the cost associated with getting record out of data page, and computing destination:
  \( |R_i| \times (t_r + t_w + t_d) \).

- **Communication costs** consist of data transfer costs, which are given as follows:
  \( \frac{(R_i/P)}{m_p + m_t} \).

The first phase costs, as like others, may be affected by data skew. The local merge-sort costs are to some degrees similar to other local merge-sort costs,
except the communication costs are associated with data receiving from the
first phase of processing, not with data transfer as in other local sort–merge
costs.

- **Communication costs** consist of data receiving costs, which are given as follows:

  \[
  \text{Data receiving costs} = \left( \frac{R_i}{P} \right) \times m_p.
  \]

- **I/O costs** consist of load and save costs. The save costs are doubled than the
  load costs as data saving is done twice: one after the data arriving from the
  network, and two when final results are produced and saved to disk.

  \[
  \begin{align*}
  \text{Save cost} & = \left( \frac{R_i}{P} \right) \times (\text{Number of passes} + 1) \times \text{IO}, \\
  \text{Load cost} & = \left( \frac{R_i}{P} \right) \times \text{Number of passes} \times \text{IO},
  \end{align*}
  \]

  where \(\text{Number of passes} = \left[ \log_{B-1} \left( \frac{R_i}{P} / B \right) \right] + 1\).

- **CPU costs**, which consist of select cost, sorting cost, merging cost, and generating result cost, are as follows:

  \[
  \begin{align*}
  \text{Select cost} & = |R_i| \times \text{Number of passes} \times (t_r + t_w), \\
  \text{Sorting cost} & = |R_i| \times \left[ \log_2(|R_i|) \right] \times t_s, \\
  \text{Merging cost} & = |R_i| \times (\text{Number of passes} - 1) \times t_m, \\
  \text{Generating result cost} & = |R_i| \times \text{Number of passes} \times t_w,
  \end{align*}
  \]

  where \(\text{Number of passes}\) is as shown in Eq. (4).

The above CPU costs are identical to CPU costs of local merge-sort in
parallel merge-all sort.

### 5. Sensitivity analysis

In order to study the behavior of each parallel external sorting methods
described earlier and to compare performance of these parallel external sorting
methods, we carried out a sensitivity analysis. Sensitivity analysis is done by
varying the performance parameters. The parameters that we used are pre-
sented earlier in Table 2, and the value of each parameter is shown in Table 4.
The parameters used reflect the current technology. For example, processor
speed is 450 Mips, which is the speed for current Pentium II. Disk speed is
estimated to 3.5 ms, and message latency per page is 1.3 ms. Table size is ex-
tremely large, that is between 1 GB and 1 TB (i.e. around 10 million up to 10
billion records), so that the use of external sort is enforced.

In the experimentation, a simulation package called Transim [9] was used.
Transim is a transputer-based simulator, which adopts an Occam-like lan-
guage. Using Transim, the number of processors and the architecture topology
can be configured, and communication is done through channels. The
parameters shown in Table 4 were incorporated in the simulation. Skew is modeled using a Zipf distribution model [20].

The processing method adopted by Transim is to distribute the process over the available interconnected processing devices in the environment. The user initiates the process by requesting a sorting process through the host and the process is then distributed to all processors. Depending on whether the sorting is a follow up of another process, for simplicity and clarity we assume the data are already distributed over the processing devices’ disks, as previously shown in Figs. 3, 4, 6–8.

After finishing the sorting process, depending on which method is used, the results are kept on the local processors, which perform the final sorting consolidation. Whether these final processor(s) would like to redistribute the sorted data or not, it will depend on the process which follows up the sorting operation, and hence data redistribution will become part of the next process following the sorting process. Therefore, in some cases, the data are spooled into one processor (as in parallel merge-all sort and parallel binary-merge sort), while in others; the data are naturally distributed to all processors (as in the other three parallel sort methods).

We have done numerous experimentations, and for clarity of results presentation, we divide them into several sections. First, we present a comparative performance between parallel merge-all sort and parallel binary-merge sort, which are the first two parallel external sorting methods discussed in this paper.

Table 4
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System and data parameters</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Number of processors</td>
<td>4–256 processors</td>
</tr>
<tr>
<td>Mips</td>
<td>MIPS of the processor</td>
<td>450 Mips</td>
</tr>
<tr>
<td>R</td>
<td>Table size</td>
<td>1 GB–1 TB</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>Page size</td>
<td>4 KB</td>
</tr>
<tr>
<td>B</td>
<td>Buffer size</td>
<td>16–256 kB</td>
</tr>
<tr>
<td><strong>Time unit costs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IO</td>
<td>Effective time to read a page from disk</td>
<td>3.5 ms</td>
</tr>
<tr>
<td>t_r</td>
<td>Time to read a record</td>
<td>300/Mips</td>
</tr>
<tr>
<td>t_w</td>
<td>Time to write a record</td>
<td>100/Mips</td>
</tr>
<tr>
<td>t_m</td>
<td>Time to merge</td>
<td>150/Mips</td>
</tr>
<tr>
<td>t_s</td>
<td>Time to compare and swap two keys</td>
<td>500/Mips</td>
</tr>
<tr>
<td>t_v</td>
<td>Time to move a record</td>
<td>400/Mips</td>
</tr>
<tr>
<td>t_d</td>
<td>Time to compute destination</td>
<td>10/Mips</td>
</tr>
<tr>
<td><strong>Communication unit costs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m_p</td>
<td>Message protocol cost per page</td>
<td>1000/Mips</td>
</tr>
<tr>
<td>m_t</td>
<td>Message latency for one page</td>
<td>1.3 ms</td>
</tr>
</tbody>
</table>
In this comparison, we would like to find out whether prolonging the merging phase into a pipeline offered better performance, and whether the merging phase in the parallel merge-all sort was more expensive than that in the pipeline.

Second, we present a comparative performance between the “without” and “with” redistribution methods. This actually comprises two comparisons: one is between parallel merge-all sort and parallel redistributionmerge-all sort, and two is between parallel binary-merge sort and parallel redistributionbinary-merge sort. The purpose of these comparisons was to find out whether the “with” redistribution methods performed better than the “without” redistribution methods. As stated before, in our experimentations, the final results will not have to be spooled on one processor or redistributed to all processors. We assume that this requirement will be enforced by the next process following up the sorting. Using our experimental architecture in Transim, the final sorted data are simply displayed by the host console.

Third, we compared the two “with” redistribution methods, namely parallel redistribution merge-all sort and parallel redistribution binary-merge sort. From the experimentation in the second stage above, we found that the “with” redistribution methods offer better performance, as promised. In this stage of performance evaluation, we would like to analyze which one of the two redistribution methods was better, and to what degree.

Fourth, we included parallel partitioned sort in the comparison. In the previous three stages, parallel partitioned sort was not included, as they focus on the “with” and “without” redistribution of parallel merge-all sort and parallel binary-merge sort. By including parallel partitioned sort, we would like to compare the best of the previous four with parallel partitioned sort. We would also like to analyze the impact of skew in the performance, as in the previous experimentation, skew was not assumed.

Finally, we performed an overall comparison by including all methods into one graph. We also analyzed the speed up of each parallel external sorting method. Fig. 10 gives an illustration of the stages in the experimentation.

5.1. Parallel merge-all sort vs parallel binary-merge sort

In this section, we present the results of comparing the first two parallel external sorting methods, particularly between parallel merge-all sort and parallel binary-merge sort. The graphs in Fig. 11 show some comparative results.

Fig. 11(a) shows performance of the two methods by varying processor numbers. The graph shows that performance of parallel merge-all sort is always better that of parallel binary-merge sort, even when the number of processors increases. It can be noticed from the graph, that the reduction in cost of parallel merge-all sort is much more than that of parallel binary-merge sort.
sort when the number of processors increases. Also notice that parallel binary-merge sort with 256 processors is still more expensive than parallel merge-all sort with only four processors. Overall, parallel merge-all sort is around 20–50% faster than parallel binary-merge sort, depending on the number of processors used.

In more details, we can analyze the costs involved in each processing phase, particularly the local sort phase costs and the merging phase costs.

The local sort phase costs for both methods are identical. Both local sort phase costs are reduced when more processors are involved. The data transfer cost, which is the cost for transferring the results of local sort operator to the merging phase, are quite comparable with the actual sorting cost. This indicates that the $O(N \log N)$ sorting method is as expensive as the communication cost to transfer the sorted results to the next phase of processing. The merging and generating results costs, which are part of the local sort phase costs, are the cheapest cost components, as these costs are the overheads associated with the processor. The major cost components are still the I/O costs, which involve disk accesses.

The merging phase costs for parallel merge-all sort are not affected by the number of processors, since final merging is done by one processor only, namely the host. In contrast, the merging phase costs for parallel binary-merge sort surprisingly grow as more processors are used. This is primarily because of the increase in the height of the pipeline, and this does not reduce the number of records involved during the pipeline, but in fact process more records. Not surprisingly however, we also notice that the disk accesses are still dominant. Other costs such as data receiving, select data page, merging, and generating
results costs are quite small. Data transfer costs within the pipeline are significant, although not as much as the disk costs.

Overall we can deduce that the difference in term of performance between the two parallel external sorting methods is determined by the costs of the merging phase. As shown in Fig. 11(a), the merging phase costs of parallel binary-merge sort are much more expensive than those of parallel merge-all sort.

Fig. 11(b) shows a comparative performance when buffer size was varied from 16 to 256 KB. Number of processors used was 64, and the table size is 1 TB (around 10 billion records). The result shows that parallel merge-all sort is always faster than parallel binary-merge sort, and the level of differences depends on the buffer size. The performance of parallel binary-merge sort is quite constant regardless the buffer size, whereas the performance of parallel
merge-all sort improves when more buffers are available. The former indicates that performance of parallel binary-merge sort is dependent on the overheads incurred during the pipeline merging, whereas the latter is true as final merging is done by one processor and with more buffers uni-processor merging can be fasten.

Also notice from the graph that there is a sudden fall of parallel merge-all sort costs when the buffer is increased from 64 to 128 KB. This is because the total number of processors processing the local sort is 64, and when the buffer size is larger than the number of processors used in the local sorting phase, performance improves greatly. Remember that each processor in the local sorting phase forms an input stream in the final merging phase, and when all input streams can be fitted in the main-memory during the final merging phase, the merging process can be fasten. Without being able to fit all input streams in the main-memory during the final merging phase, the uni-processor merging process must go through several stages and this subsequently raises additional processing costs. The graph also shows that even when the buffer size is small, parallel merge-all sort is still around 8% faster than parallel binary-merge sort, and when the buffer size is adequately large, parallel merge-all sort can be up to 50% better than the other method.

Looking into more details into the graph shown in Fig. 11(b), the local sorting phase costs for both methods are identical as also indicated previously in Fig. 11(a). Both local sorting phase costs reduce up to 50% from buffer size 16 to 256 KB.

In the merging phase, the costs for parallel binary-merge sort are constant regardless number of buffers used. This is because parallel binary-merge sort utilizes input buffers at a time, and the buffers available are more than what is needed. On the other hand, parallel merge-all sort is dependent on the buffer size. When the buffer size is smaller than number of processors the overall merging phase cost can be doubled than when the buffer size is larger than the number of processors. We also notice that when there is enough buffers, the merging phase cost for parallel merge-all sort becomes constant, since the final merging phase needs buffers to cater input streams, and additional buffers are not used. On the other hand, if the available buffers are not enough, multiple levels of merging must be done. In our case, for buffers of 16–64 KB, the final merging at the host must be done all in 2 levels, and hence, the final merging costs for buffers of 16–64 KB are the same. We expected when the buffers are even smaller than the minimum level of 16 KB, the merging phase cost will increase especially when the number of merging level increases.

Overall Fig. 11(b) shows that the difference in performance between the two parallel external sorting methods depends on the available buffers, in which parallel merge-all sort can take an advantage of it.

Parallel merge-all sort and parallel binary-merge sort represent the two extreme of the merging phase, in which the final merging in parallel merge-all
sort is actually a \( k \)-way merging where \( k \) is equal to number of processors (we then call this all-way merging), and the pipeline merging in parallel binary-merge sort is a 2-way merging. In Fig. 11(c) we show the results by varying the degree of \( k \) in the \( k \)-way merging. The results show that performance gets better when \( k \) increases from binary to all. In this experimentation, we used 64 processors. Hence, when \( k = 64 \), performance of both parallel external sorting methods is the same.

The difference in performance when \( k \) is varied is very much determined by the height of the pipeline. When \( k = 2 \), the height of the pipeline is 6 levels, and this is reduced to 3 levels when \( k \) is doubled to 4. Further reduction in height to 2 levels is achieved when \( k \) is increased to 8 and up to 32. Finally, when \( k = 64 \) (is equal to number of processors used), the height of the pipeline is equal to 1, and this is the same as that of parallel merge-all sort.

Regardless the value of \( k \), the local sorting phase costs for both methods are identical, as previously mentioned, and are constant as well, regardless the height of pipeline. The overall cost is very much determined by the costs involved in the merging phase. The merging phase costs have indicated that most of the cost components are reduced to almost half when \( k \) is increased from 2 to 4. Further reduction of around 25% is gained when \( k \) is from 8 to 32. At the end, the merging phase cost of \( k = 64 \) is around half of that of \( k = 2 \). This indicates that reducing the height of pipeline merging is essential if performance improvement is to be obtained.

One reason why the total cost grows as the height of pipeline increases is that the total number of records processed from one level of pipeline to another level is not reduced, and in fact it is constant. This is because we are doing sorting, and no record elimination is done. Other operations, such as aggregation may take an advantage of pipeline, as in some operations; number of records being processed is reduced during and within the pipeline, but not the case with sorting.

Based on these three results presented in Figs. 11(a)–(c), we can conclude that parallel merge-all sort outperforms parallel binary-merge sort. The degree of superiority depends on many factors, one of which is the buffer size available in the system. It is also worth to mention that the time taken to process a 1 TB table is very time consuming, as indicated by the minutes on the \( y \)-axis of the graphs. Time reduction is critically sought.

5.2. With vs without redistribution

In this section, we present a comparative performance results between the two “without” redistribution methods (parallel merge-all sort and parallel binary-merge sort) and the two “with” redistribution methods (parallel redistribution merge-all sort and parallel redistribution binary-merge sort). In the experimentation, we varied a number of factors, such as number of processors,
number of records, buffer size, communication speed, CPU speed, and disk speed. The results are shown in Figs. 12(a)–(f).

Fig. 12(a) shows the results of four parallel external sorting methods when number of processors is varied from 4 to 256 processors. It clearly shows that the “with” redistribution methods outperform the “without” redistribution methods. This indicates that workload balancing through a redistribution method is advantages. The improvement offered by the redistribution methods can be enormous and up to 5 folds, depending on number of processors used and which the “without” redistribution method to compared with. For example, comparing parallel redistribution merge-all sort and parallel merge-all sort (without redistribution), the former outperforms the latter by 2–9 folds from 4 to 256 processors. Comparing parallel redistribution binary-merge sort and parallel binary-merge sort (without redistribution), the former outperforms the latter by 2–8 folds from 4 to 256 processors. The difference between the two redistribution method performances is not clearly seen in the graph, and we will magnify this in the next section.

Fig. 12. (a) Varying number of processors; (b) varying number of records; (c) varying buffer size; (d) varying network speed; (e) varying CPU speed; (f) varying disk speed.
Looking into more details the four parallel external sorting methods, here is what we have found. The local sorting phase costs for all of the four methods are very similar, except one minor difference, that is a compute destination cost in the redistribution methods. As this is minor, the overall impression is that the local sorting costs for the four parallel external sorting methods are the same.

The merging phase costs for the four parallel external sorting methods vary. For parallel merge-all sort, as described in the previous section, the merge phase costs are constant regardless number of processors used. Therefore, looking at the graph in Fig. 12(a) the reduction in cost of parallel merge-all sort is very much determined by the reduction of cost in the local sorting phase. For parallel redistribution merge-all sort, the costs in the merging phase decline as number of processors increase. The reduction of merging costs can be significant, especially when more processors are used, as workload is shared among these processors. This reduction is also felt by parallel redistribution binary-merge sort. For parallel binary-merge sort, the merging phase costs increase when more processors are being used.
Fig. 12(b) shows the performance when number of records is varied from 10 million to 10 billion records. These represent a table size of 10 GB–10 TB. The number of processors is 64 and the buffer size is 257.

The graph shows that when number of records to be sorted expands the time taken to sort them also increases. It also clearly shows that the two “without” redistribution methods perform poorly compared with their counterparts: the “with” redistribution methods. The graph does not clearly show the performance impact when number of records is small, as the graph is shown in an algorithmic mode. Nevertheless, as also shown in the previous graph, the performance of the two redistribution methods is superior to those without redistribution. The difference can be up to 20 folds. For example, the difference between parallel with and without redistribution merge-all sort is approximately around 23 folds regardless number of records processed. For the parallel binary-merge sort with and without redistribution, the difference is around 16 folds consistently throughout.

Investigating the four parallel external sorting methods in more details, we learn a few lessons. First is that the local sorting phase costs are similar as indicated earlier. Second is that the increase in costs in both phases (local sorting phase and merging phase) are consistent as the number of records are increased. Third, since the local sorting phase costs are the same for the four parallel external sorting methods, the overall performance is very much determined by the merging phase costs. Since the difference between with and without redistribution is ranging between 16 and 23 folds, the difference in their merging phase costs is even more as their local sorting phase costs are the same.

Fig. 12(c) shows the results of the four parallel external sorting methods with buffer size varied from 4 to 256 KB. Number of processors used was 16 and number of records processed was 1 TB. The graph shows the superiority of the redistribution methods over the “without” redistribution methods. In most cases, the benefits of the redistribution methods are around 14 folds. Only when parallel merge-all sort is equipped with enough buffers, the benefits of the redistribution method “drops” to around 8 folds.

As described in the previous section, particularly concerning the performance of parallel merge-all sort, the processing cost declines sharply when the buffer size is larger than the number of available processors. In regard to performance of the redistribution version of merge-all sort, the improvement is not clearly seen in the graph, although the improvement due to the increase of buffer size is quite significant, that is doubled, from the smallest buffer size to the largest buffer size used in the experimentation. The trend of both version of merge sort (with and without redistribution) is actually quite similar. The local sorting phase costs are very similar except for the compute destination overhead imposed by the redistribution version of merge-all sort. The merging phase costs are similar in pattern as well, but the actual costing figures are very
much different, since in the redistribution method, the merging costs are reduced by the number of available processors.

For both versions of binary-merge sort, the effect of buffer size is trivial to the overall performance, as pipeline merging is binary and does not require many buffers. Overall, the improvement for both methods is less than 10% from buffers of 4 to 256 KB.

The difference in performance of the two redistribution methods is not clearly seen in the graph, and we will discuss in more details in the next section when we compare parallel redistribution merge-all sort and parallel redistribution binary-merge sort.

Fig. 12(d) shows the results when network speed is increased up to 4 times. In the experimentation, we used 64 processors, 257 KB buffers, and 1 TB records. By increasing the communication speed, we would like to evaluate the impact to the overall performance. From the graph, we can see that performance improvement is only very slightly (e.g. less than 5%), despite the improvement of communication speed, which was up to 4 times. This indicates that the communication costs form a small cost component of the overall processing costs.

As also shown previously, in this graph performance of the redistribution methods is far better than those without the redistribution. The improvement can be up to 18–20 folds. Again here, the degree of differences between the two redistribution methods is not clearly seen in the graph, but it seems that parallel redistribution merge-all sort performs slightly better than parallel redistribution binary-merge sort. It is clear that the difference is not as great compared with parallel merge-all sort vs parallel binary-merge sort. However, the degree of differences between the two redistribution methods is the same as those without redistribution methods; that is around double.

Fig. 12(e) shows the results when the CPU speed is increased up to 4 folds. A general impression is that the overall performance is not as greatly affected. Nevertheless, the redistribution methods outperform the “without” redistribution methods. The improvement offered by the redistribution methods is up to 20 folds, as also indicated in the previous graph when communication speed is increased.

Finally, Fig. 12(f) shows the results when disk speed is improved up to four times. The impact of disk speed greatly affects overall performance, as the processing costs decline sharply. The performance is increased up to 3 folds when the disk speed is improved four times than the initial disk unit costs.

The graph also proves that the redistribution methods outperform the “without” redistribution methods. The difference between the “with” and “without” redistribution can be up to 20 folds, where parallel redistribution merge-all sort performs the best.

From the graphs in Figs. 12(d)–(f), it clearly indicates that sorting 1 TB records, disk costs are very dominant compared to network and CPU speed, and reducing processing costs can be achieved by reducing disk access time.
We can conclude this section by saying that the redistribution methods offer far better performance deal than the “without” redistribution methods. In many cases, we can expect that the redistribution methods perform up to 10–20 folds better than those without redistribution methods. The graphs shown previously do not clearly exhibit the difference between the two redistribution methods, although they are both show far better performance than those without redistribution. In the next section, we will investigate more closely the difference between the two redistribution methods: parallel redistribution merge-all sort and parallel redistribution binary-merge sort.

5.3. Parallel redistribution merge-all sort vs parallel redistribution binary-merge sort

In this section, we present comparison results between the redistribution methods of merge-all sort and binary-merge sort. We need to highlight the differences between these two redistribution methods for two reasons. One is that the redistribution methods are shown to be better compared with the “without” redistribution methods as indicated in the previous section and can be clearly seen in the performance graphs. The other reason is that the difference in term of performance between the two redistribution methods in the previous section cannot be clearly seen in the performance graphs. Therefore, we need to highlight the difference in separate graphs, which are to be shown in this section. Figs. 13(a)–(c) present some of the results that we obtained from our experimentation.

Fig. 13(a) shows a comparative performance between parallel redistribution merge-all sort and parallel redistribution binary-merge sort by varying number of processors. The graph shows that parallel redistribution merge-all sort is approximately 15–25% better than parallel redistribution binary-merge sort. The more processors we use the larger the gaps between the two redistribution methods. This is because parallel redistribution binary-merge sort does not reduce much of the processing costs due to the overhead produced by pipeline merging. On the other hand, much performance improvement is applied to parallel redistribution merge-all sort due to the use of all processors in the final merging phase. Since local sorting phase costs for both methods are identical, performance differences are determined by the costs in the merging phase.

Fig. 13(b) shows a comparison between the two redistribution methods when buffer size is varied. The graph shows that parallel redistribution merge-all sort is around 75–100% better than parallel redistribution binary-merge sort. Performance gained by parallel redistribution binary-merge sort is not as much as that of parallel redistribution merge-all sort. Parallel redistribution binary-merge sort improves around 20% when buffer size is increased to 256 KB, whereas parallel redistribution merge-all sort gains almost double.
We recall that the "without" redistribution merge-all sort, there is a decline is cost when the buffer size is larger than the available number of processors. In

Fig. 13. (a) Varying number of processors; (b) varying buffer size; (c) k-way merge.
the experimentation we used 64 processors and when the buffer exceeds this number, performance greatly improves. In Fig. 13(b), we do not see such significant improvement in parallel redistribution merge-all sort. The reason is as follows. The processing costs are divided into local sorting phase costs and final merging phase costs. The local sorting phase costs for both “with” and “without” redistribution of merge-all sort are quite similar, and this is quite a major bulk in the overall processing cost. The final merging costs of the redistribution method are greatly reduced when more processors are used due to workload sharing. The reduction exhibited by parallel merge-all sort comes from the reduction in the merging phase costs, where enough buffers are available during the final merging. The same reduction is also applied to the redistribution method. However, the impact of the reduction is not as much as that “without” redistribution, because in “without” redistribution method, the merging phase costs are quite large, whereas in “with” redistribution method, the merging phase costs are small. As a result, the performance graph of parallel redistribution merge-all sort, there is no great decline in processing cost when the buffer size exceeds number of available processors.

Finally, we would like to show the impact of \( k \)-way merging between the two redistribution methods, which is shown in Fig. 13(c). In this graph, \( k \) is varied from 2 (i.e. the original parallel redistribution binary-merge sort) to 64 (i.e. parallel redistribution merge-all sort). In this experimentation, we use 64 processors, and consequently, parallel redistribution “binary”-merge sort with \( k = 64 \) is actually the same as parallel redistribution merge-all sort.

The graph shows a reduction in the processing costs as the value of \( k \) is increased up to the maximum value that is the total number of processors used in the local sorting phase. The reduction is caused by the reduction in height of the pipeline merging. The height of pipeline is shorten from 6 levels for \( k = 2 \) to 3 levels for \( k = 4 \), and further to only 2 levels for \( k = 8–32 \), and finally to 1 level for \( k = 64 \). Overall, the cost reduction is doubled, particularly when we compare the original parallel redistribution binary-merge sort (i.e. \( k = 2 \)) and parallel redistribution merge-all sort (i.e. \( k = 64 \)). The difference in term of processing costs exhibited in Fig. 13(c) actually comes from the costs incurred during the final merging phase, since the costs in local sorting phase remain constant, despite the increase of the value of \( k \), as \( k \) is associated with the merging process only.

So far, parallel merge-all sort is better than parallel binary-merge sort, and parallel “with” redistribution methods are better than parallel “without” redistribution. And in this section, we show that parallel redistribution merge-all sort outperforms parallel redistribution binary-merge sort. It is now left parallel partitioned sort. In the next section, we not only compare parallel partitioned sort with the current best, that is parallel redistribution merge-all sort, but also consider the skew factor to general performance. Two skew problems are to be used: that is data skew and processing skew.
5.4. Comparison with parallel partitioned sort

In this section, we would particularly like to compare parallel redistribution merge-all sort and parallel partitioned sort. We mainly use two parameters in the experimentation, particularly number of processors and buffer size. Figs. 14(a)–(c) show the performance results for different number of processors, whereas Figs. 15(a)–(c) does for different buffer size. We also include the skew factor into the experimentation. As mentioned earlier, there are two types of skew: data skew and processing skew. In the experimentation, when there is data skew, processing skew is always assumed at the processing stage (i.e. merging phase). However, when there is no data skew, processing skew may or may not occur. Hence, we present three cases: no skew, processing skew only, and data and processing skews.

Fig. 14(a) shows that parallel partitioned sort performs slightly better than parallel redistribution merge-all sort (e.g. around 10%), when there is no skew. Since the processing phases for both methods are different, it is difficult to compare both methods phase by phase. However, it can be indicated that the scanning phase costs in parallel partitioned sort are minimal, compared with the first processing phase, namely local sorting phase in parallel redistribution merge-all sort, as in parallel partitioned sort, there is no sorting operation in the first phase. The major cost component in the scanning phase of parallel partitioned sort is the data loading from disk cost. The partitioning cost which is a data transfer cost from the scanning phase to the sorting phase is quite significant too, but not as large as the disk access cost.

Another thing that can be noted regarding parallel partitioned sort vs parallel redistribution merge-all sort is that the second phase of parallel partitioned sort where sorting is actually performed is quite similar to the first phase of parallel redistribution merge-all sort where local sort is carried out. Hence, the difference in term of performance is much or less determined by the scanning phase costs of parallel partitioned sort and the final merging phase costs of parallel redistribution merge-all costs.

Fig. 14(b) shows that when there is a high processing skew degree, parallel partitioned sort performs poorly. To show how poor that is, in Fig. 14(b) we also include a line for parallel merge-all sort, where no processing skew is involved. Notice that parallel partitioned sort can be as bad as parallel merge-all sort, particularly when the number of processors used is small (e.g. 4 processors).

Why parallel partitioned sort performs so badly, particularly in comparison with parallel redistribution merge-all sort. One reason is that the skew occurs in the second phase of processing, that is in the sorting phase of parallel partitioned sort and in the final merging of parallel redistribution merge-all sort. As mentioned above, the second phase of parallel partitioned sort is similar to the first phase of parallel redistribution merge-all sort. With processing skew exists,
the second phase of parallel partitioned sort now becomes so expensive, whereas the first phase of parallel redistribution merge-all sort remains the
same since no processing skew is involved in the first phase of processing. This results in an extreme overhead to parallel partitioned sort, and this is why the performance of parallel partitioned sort is degraded.

Fig. 15. (a) Varying buffer size – no skew; (b) varying buffer size – processing skew; (c) varying buffer size – data and processing skews.
With more processors are involved, the gap between the two parallel external sorting methods (i.e. parallel partitioned sort and parallel redistribution merge-all sort) and parallel merge-all sort becomes widen, since in parallel merge-all sort, the merging phase is purely done by one processor, whereas the other two methods the second phase is done by several processors. The latter is still better even when it is highly skewed.

Fig. 14(c) shows the performance when both data and processing skews exist. In this graph, we also include parallel merge-all sort with data skew and without skew. The reason for this inclusion is to compare the other two methods (i.e. parallel partitioned sort and parallel redistribution merge-all sort) with parallel merge-all sort as we did in the previous group. Since we now assume that data skew exists, it is fair to include parallel merge-all sort with data skew.

The performance of parallel partitioned sort is now slightly better than parallel redistribution merge-all sort. The main reason is that data skew now affect the first phase of parallel redistribution merge-all sort (i.e. local sorting phase) very badly. On the other hand, data skew effect in the first phase of parallel partitioned sort (i.e. scanning phase) is not as bad, since in the scanning phase, only a few operations are involved, particularly disk loading, reading, and partitioning. The first phase of parallel redistribution merge-all sort involves many more operations and they are all affected by data skew.

It is also notice from the performance graph that parallel redistribution merge-all sort performs slightly worse than parallel “without-redistribution-no-skew” merge-all sort, especially when the number of processors involved is less than 16. The reasons can be explained as follows. When there is processing skew, final merging phase costs of the redistribution method may be closed to those of “without redistribution” method, particularly when the number of processors is small. As there is data skew, local sorting phase costs of the redistribution method are far worse than the first phase of “without redistribution” method. Hence, overall, the total costs of the redistribution model exceed those of without redistribution model. When number of processors is large, the final merging phase costs of the redistribution model are far better than those “without” redistribution model, and subsequently improves overall performance.

However, it is not fair to compare both parallel external sorting methods (i.e. parallel partitioned sort and parallel redistribution merge-all sort) where both skew exist, and parallel merge-all sort with no skew. To make a fairer comparison, we need to compare the previous two methods with parallel merge-all sort method with data skew. The performance graph shows that in this case, parallel partitioned sort and parallel redistribution merge-all sort are always better than parallel merge-all sort, when all involve data skew.

Figs. 15(a)–(c) show the three cases (i.e. no skew, processing skew only, and data and processing skews) with different buffer sizes. Fig. 15(a) shows that
parallel partitioned sort outperforms parallel redistribution merge-all sort when no skew is involved. The difference can be up to 15%. The difference is reduced when buffer size is large. This is because parallel redistribution merge-all sort takes advantages of large buffers particularly in the final merging phase, and thus reducing the gap with parallel partitioned sort.

Fig. 15(b) shows that parallel partitioned sort performs badly when processing skew is involved. In comparison with parallel redistribution merge-all sort, total processing costs of parallel partitioned sort are more than double. This fact has been exhibited previously when number of processors is varied and processing skew exists. In comparison with parallel merge-all sort, both methods (i.e. parallel redistribution merge-all sort and parallel partitioned sort) are far better. Only when the buffers are large enough, the gap between parallel merge-all sort and the other two methods is getting closer. One of the reason why both parallel redistribution merge-all sort and parallel partitioned sort are always better than parallel merge-all sort despite of processing skew is that the processors used is quite large (e.g. 64 processors) and the impact of processing skew to 64 processors is not as bad as if it were to use just one processor, particularly in the case of parallel merge-all sort.

Fig. 15(c) shows that parallel partitioned sort performs slightly better than parallel redistribution merge-all sort, especially when data skew and processing skew exist. This phenomenon has also occurred in the previous experimentation results. One major difference between this graph and the graph shown previously in Fig. 14(c) is that the performance of parallel merge-all sort does not come too close with those of the other two methods. The reason is also being a relatively large number of processors used (e.g. 64 processors).

From this section we can learn several things: One is that parallel partitioned sort is suitable for only when no skew or both skew are involved. When processing skew exists without data skew, parallel partitioned sort does not perform as well as parallel redistribution merge-all sort. In most cases, parallel “without-redistribution” merge-all sort does not deliver a better performance.

5.5. Overall comparison and speed up

In this section, we perform a comparison among the five parallel external sorting methods. Like in the previous section, here we cover three cases: no skew, processing skew, and data and processing skews. We also present speed up graphs for the five parallel external sorting methods. There are some of the results.

Fig. 16(a) shows a comparative performance of all methods when skew does not exist. It clearly shows that the tall bars are parallel “without-redistribution”
Fig. 16. (a) Varying number of processors – no skew; (b) varying number of processors – processing skew; (c) varying number of processors – data and processing skews.
methods. The other three, especially the two “with-redistribution” methods and parallel partitioned sort have lower processing costs. Based on the results presented in the previous sections, we know that in this case parallel partitioned sort is the best.

Fig. 16(b) shows the performance of all methods when processing skew is involved. Since parallel merge-all sort and parallel redistribution binary-merge sort do not have processing skew, they are not applicable. The graph shown in Fig. 16(b) indicates that parallel partitioned sort and parallel redistribution binary-merge sort are worse than parallel redistribution merge-all sort. Processing skew affects the second phase (i.e. sorting phase) of parallel partitioned sort greatly, and it also affects the performance of pipeline merging in parallel redistribution binary-merge sort. On the other hand, processing skew in the final merging phase of parallel redistribution merge-all sort does not give as much impact as the other methods.

Fig. 16(c) shows performance of all methods when both skew (data and processing skews) exist. The graph shows that only parallel binary-merge sort is increased, where others are decreased when more processors are involved in the processing. The best performance is achieved by parallel partitioned sort. And parallel redistribution merge-all sort comes second.

Figs. 17(a)–(c) show performance speed up of the five parallel external sorting methods. The graph in Fig. 17(a) shows that parallel partitioned sort and parallel redistribution merge-all sort are closed to linear speed up. Others, particularly the two “without-redistribution” methods are far from the target of linear speed up. For example, in the case of parallel merge-all sort, with 4 processors and 256 processors, it achieves a speed up of only 1.6 and 2.8, respectively. This indicates that number of processors do not offer much performance improvement. One main obvious reason being the final merging phase done by one processor only. Like parallel merge-all sort, the speed up of parallel binary-merge sort is also around the same figures; that is 1.3 for 4 processors and 1.6 for 256 processors. Here it shows that pipelining is totally unacceptable.

Now looking at the best two methods. The speed up of parallel partitioned sort is 3.1 for 4 processors and 248 for 256 processors. For parallel redistribution merge-all sort, the speed up is 2.9 for 4 processors and 224 for 256 processors. These two indicates that partitioning technique is very much in favor, except when pipeline is used as in parallel redistribution binary-merge sort.

Fig. 17(b) shows that none of the five methods come closed to linear speed up. This is due to the skew, which in this case is processing skew. To make the matter more clearly, here is some of the speed up figures. For parallel partitioned sort, the speed up is 1.7 for 4 processors and 5.5 for 256 processors. For parallel redistribution binary-merge sort, the speed up figures are quite the same with those of parallel partitioned sort, that is 1.8 for 4 processors and 2.6 for 256 processors. For parallel redistribution-merge all sort, the speed up
Fig. 17. (a) Speed up – no skew; (b) speed up – processing skew; (c) speed up – data and processing skews.
is slightly better that the other two methods, that is 2.3 for 4 processors and 16.3 for 256 processors. In this experimentation, processing skew is set to be very high. Using the Zipf model, the degree of skewness indicated by $\theta$ is equal to 1.

Fig. 17(c) shows a speed up graph of all parallel external sorting methods with data and processing skews. For parallel merge-all sort, since processing skew is not applicable, it has data skew only. The performance of all methods is even worse, due to a high degree of data skew as well as processing skew. The best achievement is parallel partitioned sort with a speed up of 1.6 for 4 processors and 4.8 for 256 processors. Parallel redistribution merge-all sort comes second with similar speed up.

The three speed up graphs show that skew, especially a high degree of skewness (i.e. $\theta = 1$), has a great impact on performance, in which an acceptable level of performance cannot be attained.

5.6. Discussions

From the experimentation results we have gathered, we have learned a few things:

- Skew is a major problem in parallel processing performance. Whether the skew exists at a processing level or at a data placement level, their existence degrades performance. When the degree of skewness is very high (i.e. $\theta = 1$), speed up curve shows that parallel processing is worthless.
- Data skew, which is caused by an uneven data placement, may always lead to processing skew. Therefore, solving data skew problem is essential, in order to minimize the possibility of skew problem exists at a later stage of processing.
- Even with a uniform data placement distribution in the initial stage of processing, processing skew may occur due to an uneven workload distribution during the processing. Careful thought and strategy need to be taken to solve processing skew problem.
- Our proposed parallel redistribution merge-all sort and parallel partitioned sort outperform the other three parallel external sorting methods in most cases. Depending on whether skew exists or not, we can determine which one of these two methods is more suitable. The rule is very simple.

\begin{verbatim}
If processing skew degree is high Then
    Use Parallel Redistribution Merge-All Sort
If both data and processing skew degrees are high
Or no skew Then
    Use Parallel Partitioned Sort
\end{verbatim}
6. Conclusions and future work

In this paper, we have presented a taxonomy for parallel external sorting in high performance database systems. The taxonomy consists of five sorting methods, namely parallel merge-all sort, parallel binary-merge sort, parallel redistribution binary-merge sort, parallel redistribution merge-all sort, and parallel partitioned sort. The first two sorts are widely known through previous publications. We added with the other three methods, through the use of redistribution and repartitioning concepts.

For the performance evaluation, we have shown the superiority of the last two sorting methods above. This is due to a combination between parallelism approaches at all phases of the process, and the reduction in height of the processing tree to just one level. The proposed methods however suffer from skew problems, due to the nature of partitioning and redistribution of the workloads. However, even when the degree of skewness is extremely high, the proposed models still outperform the traditional methods.

Our future work is to investigate into solving skew problems in parallel external sorting, in particular data skew and processing skew. Processing skew may be able to be resolved using a version of bucket tuning, whereas data skew may be able to be neutralized using both parallel I/O and overlapping between I/O operations and non-I/O operations. Thorough investigations need to be conducted to solve these issues. We also plan to explore the use of cluster architectures and their implementation of parallel external sorting methods. Some of the cost models will then have to be adjusted, and further performance evaluation needs to be carried out.

References