Parallel Sorting of Large Data Volumes on Distributed Memory Multiprocessors

Markus Pawlowski, Rudolf Bayer

Institut für Informatik, Technische Universität München
Arcisstr. 21, D 8000 München 2, Germany
e-mail: {pawlowsk | bayer}@informatik.tu-muenchen.de

Abstract. The use of multiprocessor architectures requires the parallelization of sorting algorithms. A parallel sorting algorithm based on horizontal parallelization is presented. This algorithm is suited for large data volumes (external sorting) and does not suffer from processing skew in presence of data skew. The core of the parallel sorting algorithm is a new adaptive partitioning method. The effect of data skew is remedied by taking samples representing the distribution of the input data. The parallel algorithm has been implemented on top of a shared disk multiprocessor architecture. The performance evaluation of the algorithm shows that it has linear speedup. Furthermore, the optimal degree of CPU parallelism is derived if I/O limitations are taken into account.

1 Introduction

Data sorting plays an important role in computer science and has been studied extensively [23]. The problem of sorting is easily understood. In the sequential case, sorting of N tuples (i.e. data items) has at least a complexity of $O(N \cdot \log(N))$. Sorting is frequently the basis of more complex operations. The use of multiprocessor architectures renders the parallelization of sorting indispensable. First approaches were sorting networks introduced by Batcher [3]. Since then a lot of literature concerning parallel sorting has been published. A complete survey of this literature is beyond the scope of this paper. However, three fundamental approaches can be recognized in order to parallelize a sorting algorithm. In the following these approaches are discussed to obtain a general knowledge of them:

- One possibility to implement parallel sorting algorithms is to implement them in hardware. The progress in VLSI makes this approach promising. Although the architecture of parallel sorting chips differs in details, the paradigm used is *vertical parallelism* (cf. section 2). In [17] simple sort-merge-cells are placed in a binary tree topology in order to build a powerful parallel sorting hardware. The scheme of the rebound-sorter of [13] is used in [18] to design a parallel sorting chip. A rebound-sorter consists of 2 coupled arrays passed by the data streams. A rebound-sorter is comparable to systolic algorithms.
- The second approach has its origin in abstract parallel machine models like the PRAM (Parallel Random Access Machine). A parallel implementation of the merge-sort algorithm is proposed by Cole [14]. It sorts N tuples in time $O(\log(N))$ using N processors of a CREW-PRAM (Concurrent Read Exclusive Write PRAM). Other authors (e.g. Bilardi et al. [10]) modify the so-called bitonic sorting algorithm (cf. Batcher [3]) to achieve a parallel sorting algorithm. It sorts N tuples in time $O(N/p \cdot \log(N/p))$ using p processors. In contrast to the CREW-PRAM algorithm of Cole [14], here the degree of parallelism is not dependent on the number of input tuples. The algorithms of this class can be implemented on a shared memory multiprocessor.

- Distributed memory multicomputers are the starting point of the third approach. Limitations of real systems are taken into consideration. Furthermore, the influence of a memory hierarchy (main memory vs. disk memory) is investigated. The literature contains lot of proposals (e.g. [4], [7], [16], [22] and [26]) on how to sort large volumes of data in parallel on top of a distributed memory multicomputer. The principle of the parallel sorting algorithms is similar and has its root in the well known external sorting algorithms as described by Knuth [23] or Aho et al. [1].

We designed a new parallel sorting algorithm belonging to the third class. The reasons are the following:

- Our prime premise is to use off-the-shelf hardware in order to explore possibilities of parallelism for database management systems. Therefore we do not consider the first approach. Another premise is to investigate the impact of massive parallelism on database management systems. Therefore shared memory multiprocessors (second approch) will not be used, as they do not scale well enough for our purposes.
- Sorting plays an important role during query evaluation [24]. The relational database system TransBase [33], which we use in our research project [25], implements duplicate elimination and a variant of join (sort merge join) by sorting the input operands. Since the database itself resides on secondary storage and query evaluation is done in main memory, the interaction between main memory and secondary memory has to be taken into account. This rules out the first and second approaches, too.
- A further reason is our cooperation with another research group providing the programming environment *MMK* [8] and *TOPSYS* [9] on top of a distributed memory multicomputer. In this way feedback can be obtained about the suitability of *MMK* and *TOPSYS* for developing parallel programs.

Our new parallel external sorting algorithm is based on data parallelism. Each processor sorts one data partition. Load imbalance is avoided by a new, efficient data partitioning algorithm. It does not sample the input data as in [16]. It samples the presorted initial runs. Hence it can limit the maximum load imbalance due to data skew which might occur. Therefore the partitioning algorithm is a component of our parallel sorting algorithm.

The remainder of the paper is organized as follows: Section 2 gives a brief introduction into the paradigms of parallel programming. In section 3 the horizontal paradigm is applied to the problem of parallel sorting. A new adaptive partitioning algorithm is introduced in section 4. Section 5 shows the implementation and performance results. The paper concludes in section 6 with a summary and future work. The appendix puts together the parameters used throughout this paper.

2 Paradigms of Parallel Programming

Horizontal parallelism is based on data-partitioning and works as follows: The input data is partitioned into independent data partitions (*splitting phase*). Each partition is subsequently processed by a single processor performing the usual sequential algorithm (parallel *processing phase*). Then the intermediate results computed by the parallel processing phase are collected into one final result (*integration phase*). Horizontal parallelism accellerates the sequential algorithm, if the overhead due to the additional splitting and integration phases is low and if load imbalance does not result from

unequally sized data partitions.

Vertical parallelism exploits the fact that a complex task can be divided into several subtasks which have to be processed in a pipelined manner (i.e. temporally overlapped). By that means a parallel execution is obtained. The parallel processing time is reduced, if the subtasks have nearly the same processing time and if the communication costs are low w.r.t. the processing costs of one subtask.

Figure 1 compares horizontal parallelism with vertical parallelism.

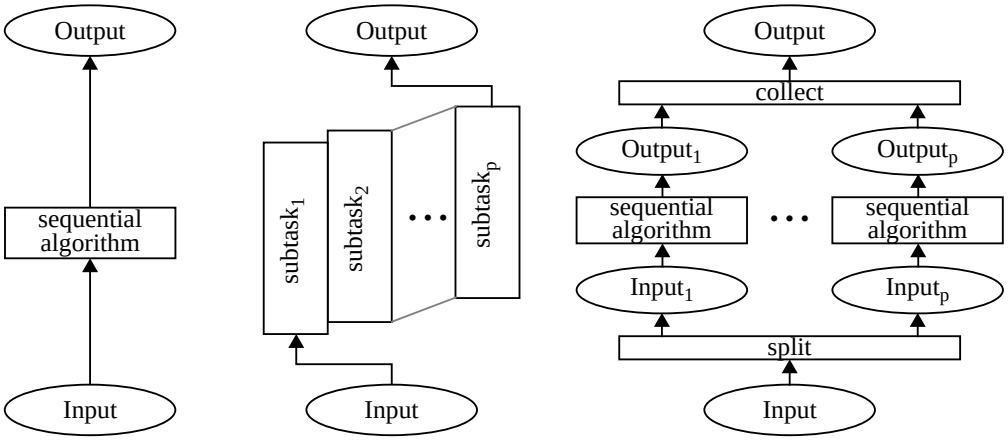

Fig. 1. Vertical and Horizontal Parallelism

3 Parallel External Sorting

The horizontal parallelization is applied to an algorithm performing external sorting. To have a common starting point, the usual sequential algorithm is described. After having presented the multiprocessor architecture of the target machine, first performance predictions of a parallel external sorting algorithm are presented. Finally the sequential sorting algorithm is parallelized horizontally. The paradigm of vertical parallelism is not applied, because it is not practicable.

3.1 Sequential External Sorting Algorithm

Sorting of tuples normally takes place in main memory. But if the amount of data to be sorted exceeds the capacity of the available main memory, the use of secondary storage is inevitable. The number of I/O accesses should be minimized. Most of the sorting process still takes place in main memory. For the sake of simplicity, we assume that the input tuples to be sorted are located in a file on secondary storage (input file) and that the sorted result tuples are stored in a file (result file), too. External sorting consists of two phases [23]. Refering to figure 2, the two phases are explained in detail.

During a so-called *presort phase* main memory is organized as a heap providing space for h tuples. All input tuples are pumped through main memory once, in the course of which they are presorted and stored in m so-called *initial runs* IR_i $(1 \le i \le m)$. The size of each initial run is as large as the heap in main memory at least. We can estimate m by the following formula:

$$m \le N / h \qquad (1)$$

The CPU-time complexity of the presort phase is $O(N \cdot \log(h))$. This results from the fact that each tuple has to be inserted into and deleted from the heap exactly once. The I/O complexity of $O(N)$ during the presort phase is caused by the need of loading (storing) each tuple from (into) secondary storage exactly once.

During a so-called *merge phase* the m initial runs IR_i are merged to compute the sorted result file. We use m-way merging to minimize the I/O accesses. Details can be found in [5] and [6]. The I/O complexity of the merge phase is equal to the I/O complexity of the presort phase, since again each tuple has to be loaded (stored) from (into) secondary storage exactly once. The CPU-time complexity of the merge phase is $O(N \cdot \log(m))$. It is not linear in the problem size, since the number of initial runs depends linearly on the problem size.

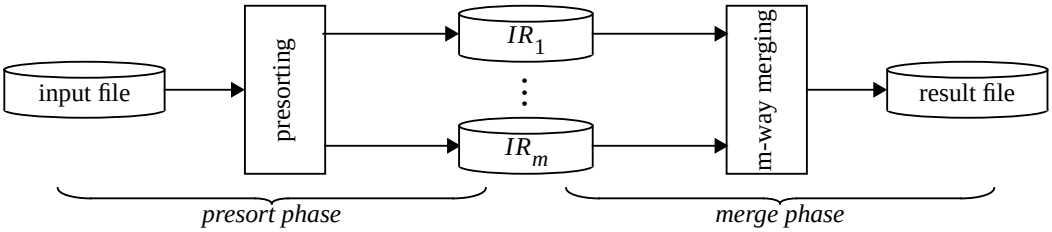

Fig. 2. External Sorting (Sequential Version)

3.2 Computer Architecture

Our target architecture is a shared disk multiprocessor. This architecture belongs to the family of distributed memory multiprocessors, which can be divided into shared disk [30] and shared nothing multiprocessors [31]. The structure of the two distributed memory multiprocessor architectures is shown in figure 3. Distributed memory multiprocessors consist of multiple nodes connected by a fast communication network. A processor and local main memory belong to each node. The processors can only access their local memory directly. They cannot access remote memory of other nodes. A global shared memory does not exist. Interprocessor communication is realized by message passing. Shared disk and shared nothing multiprocessors differ in the way how the secondary storage is connected to the processing elements.

Unlike main memory, secondary storage of shared disk multiprocessors is a subsystem of its own. Shared disk multiprocessors are characterized by the location and access transparency of the secondary storage [12]. The access to secondary storage is done by the fast communication network. This property makes it possible to think of the secondary storage as one large virtual device, even if it consists of many small devices. The access times to secondary storage are independent of the processors location. Our parallel external sorting algorithm exploits this transparency to a large extent.

The properties of access and location transparency cannot be found at shared nothing multiprocessors. This architecture is characterized by the fact that each node has its dedicated secondary storage. Each processor can only access this directly attached secondary storage. Prominent representatives of shared disk multiprocessors in the database area are systems like Bubba [11], Gamma [15], Prisma [2] and Teradata [32].

The way in which our sorting algorithm exploits the access and location transparency of secondary storage distinguishes it from the algorithms in [4], [7], [16] and [26], which are based on shared nothing architectures. Our algorithm discards the shared

nothing paradigm, as it is not open to high performance new disk technologies like RAIDs (cf. [28] and [19]). RAIDs offer the location and access transparency needed by our algorithm.

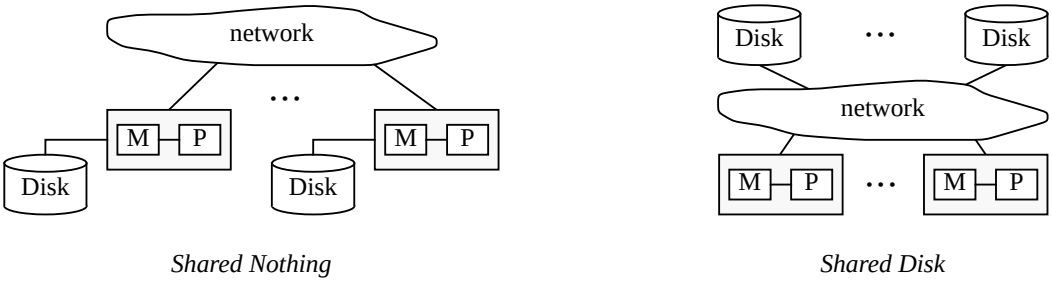

Fig. 3. Shared Disk vs. Shared Nothing

3.3 Optimal Parallel Run Time

In the sequential case, the run time of an external sorting algorithm has a lower bound which is determined by the transfer rate to secondary storage. Even in the parallel case, the run time of sorting is at least as high as the (parallel) transfer time needed to read and write all input tuples twice (once for the presort phase and once for the merge phase). An optimal parallel external sorting algorithm should have a run time near to that lower bound.

The parallel external sorting algorithm should exploit the I/O and CPU power of multiprocessors. But neither a significant I/O nor a significant CPU bottleneck can be accepted. In the following we assume that the transfer rate to secondary storage is a fixed system parameter whereas the degree of CPU parallelism can be chosen freely within the capability of the given system. This reflects the target architecture chosen for our research. In general, the I/O-subsystem of a shared disk multiprocessor has such a high performance that CPU parallelism is inevitable to yield a system without bottleneck. The CPU parallelism should be increased until a run time balance between CPU and I/O is obtained. This degree of CPU parallelism is called *balanced parallelism* (d in figure 4). The problem is now how to design a parallel external sorting algorithm which is effectively scalable. The term *effective scalability* expresses the property of a parallel external sorting algorithm having linear speedup characteristics up to the point of balanced parallelism. Figure 4 illustrates the above discussion.

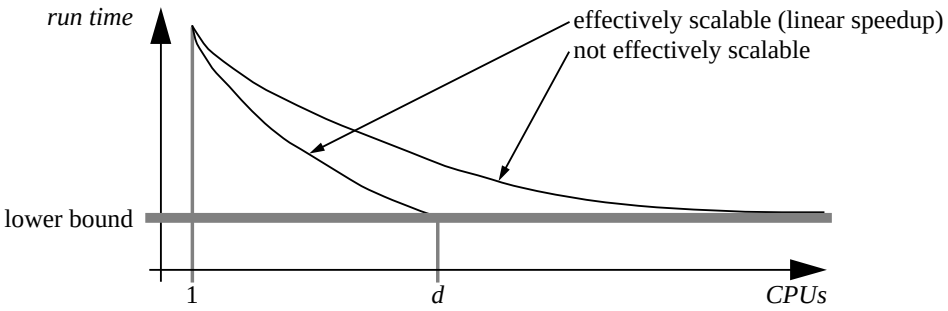

Fig. 4. Run Time of a Parallel External Sorting Algorithm

3.4 Parallelization of External Sorting

In this paper we do not apply the paradigm of vertical parallelism to parallelize the external sorting algorithm. The reason is that in a distributed memory environment the costs of interprocessor communication have a high threshold value. In order to minimize the amount of communication it is necessary to build packets of tuples which are sent from one subtask of the pipeline to the next one, instead of sending the tuples alone. Each subtask has to unwrap the incoming packets in order to extract the tuples to be processed. Furthermore after processing the tuples, they have to be wrapped up by each subtask for the outgoing packets which are sent to the next subtask. The handling of the communication packets is very CPU intensive and cannot be parallelized, because each subtask has to deal with all tuples and their packaging. Measurements have shown that in case of external sorting the processing of the communication packets takes up 80% of total CPU time. The ratio becomes even worse if the degree of parallelism increases, as the processing time for the communication packets in each subtask is invariant with the degree of vertical parallelism.

Therefore we apply the paradigm of horizontal parallelism. The parallel processes can do their jobs without communication between each other, because their jobs are independent from each other. Thus the performance of the parallel algorithm does not depend on interprocessor communication costs in such a crucial way as it is the case with vertical parallelism.

However, a similar packaging mechanism as described in the previous paragraph must be implemented in order to access the tuples stored on secondary storage. In contrast to vertical parallelism this kind of packaging can be easily parallelized, because the processes of the horizontally parallelized external sorting algorithm just work on a data partition instead of on all tuples.

Because the presort phase and the merge phase of the external sorting algorithm cannot be overlapped[1)], the two phases of the parallel version are explained in distinct subsections. The paradigm of horizontal parallelism is applied to transform the sequential algorithm (cf. section 3.1) to a parallel one.

3.4.1 Presort Phase. The presort phase is performed by p processors P_j $(1 \le j \le p)$ in parallel. In accordance with figure 1, we describe the three phases (splitting, parallel processing and integration phase) of the horizontally parallelized presort phase in figure 5.

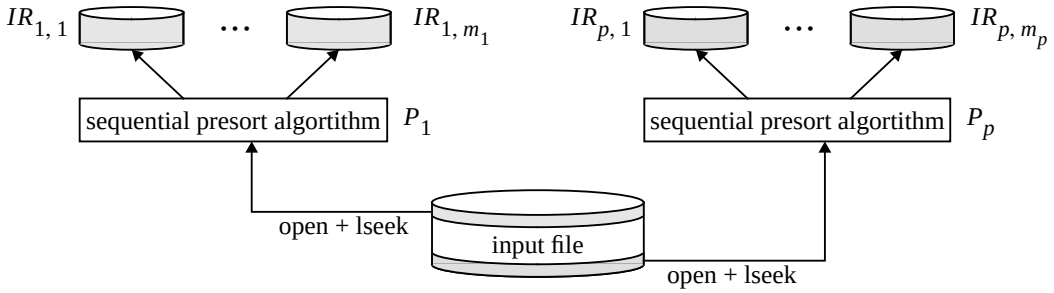

Fig. 5. Horizontally Parallelized Presort Phase

1) Recall: In order to merge the initial runs during the merge phase, the presort phase has to be completed.

The input data must be split into p equally sized data partitions. The input file is stored on secondary storage having the property of access and location transparency, therefore each processor P_j can access the input file directly. The splitting of the input data can be done logically by simple commands of the operating system. The start and end positions of the data partitions are computed by simple arithmetic. It is not necessary to move the input data physically during the splitting phase. Furthermore this simple splitting phase guarantees that all data partitions have equal size.

Now the p data partitions of the input file can be processed in parallel and independently. Each processor P_j applies the usual sequential algorithm to construct its initial runs $IR_{j,i}$ $(1 \le i \le m_j)$. The initial runs are stored on secondary storage.

The final integration phase can be omitted, since the intermediate results are already in a form (set of initial runs) which can be processed by the following merge phase of the external sorting algorithm. The omission of the integration phase is due to the access and location transparency of the secondary storage.

Taking a closer look on the theoretical run time of the parallelized presort phase, we recognize a linear speedup. The parallel presort phase does not bear any additional costs because of the splitting and integration phases. The linear complexity of the presort phase deduced in section 3.1 together with the property that the data partitions are equally sized guarantees this linear speedup.

3.4.2 Merge Phase. The merge phase of the external sorting algorithm is performed by p processors P_j $(1 \le j \le p)$ in parallel. The m initial runs IR_i $(1 \le i \le m)$ form the input. In order to parallelize the merge phase horizontally, the initial runs IR_i must be split into p independent data partitions D_j. Two methods are possible to accomplish the data partitioning:

SETPART. This method is very simple and straightforward. It divides the set of the m initial runs IR_i into p disjunct subsets D_j. Each subset D_j contains m/p initial runs (cf. figure 6, left part).

KEYPART. This method is explained by figure 6 (right part), too. It splits each initial run IR_i into p regions $R_{i,j}$ $(1 \le i \le m, 1 \le j \le p)$. The borders of the regions are determined by the value of $(p+1)$ splitting keys k_j. They must obey the following condition:

$$k_0 \le k_1 \le k_2 \le \ldots \le k_p \tag{2}$$

The value v of the sorting attribute of each tuple residing in region $R_{i,j}$ lies between k_{j-1} and k_j. Each region $R_{i,j}$ itself is a small initial run. Now define a data partition D_j to be the set of m small initial runs:

$$\forall\,(1 \le j \le p) : D_j = \{R_{1,j}, \ldots, R_{m,j}\} \tag{3}$$

The splitting phase does not move any data neither by the *SETPART* nor by the *KEYPART* method. Because of the access and location transparency of the secondary storage the computation of the p data partitions D_j can be performed logically. Even the splitting of the initial runs into smaller ones can be done logically.

After having determined the independent sets D_j of initial runs, each of the p parallel merge processes P_j merges the partition D_j assigned to it. The result of each merge process P_j is the sorted file O_j (cf. again figure 6).

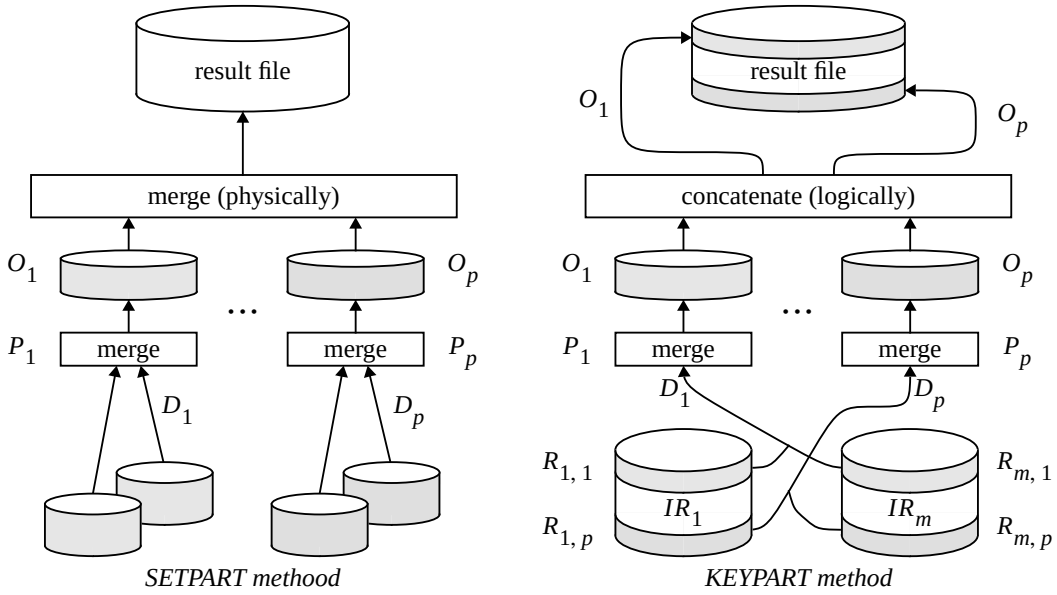

Fig. 6. Horizontally Parallelized Merge Phase

The final integration phase of the horizontally parallelized merge phase depends highly on the choice of the splitting method. If the p data partitions D_j are computed by the *SETPART* method, the integration of the p intermediate results O_j consists in merging them (cf. figure 6, left part). Although this merging can be temporally overlapped with the construction of the intermediate results O_j, it has nevertheless an unwanted effect of sequentialization. One single process has to merge all the files O_j. Because the processing time of merging depends linearly on the number of tuples to be merged, the total processing time of the parallelized merge phase is the same as in the sequential case. The result is, that no speedup at all is obtained. Therefore, the *SETPART* method should not be applied, although it is simple and straightforward.

If the p data partitions D_j are computed by the *KEYPART* method, the integration of the p intermediate results O_j consists in concatenating them (cf. figure 6, right part). This works, because equations (2) and (3) guarantee that the result of the concatenation is sorted. The concatenation can be done logically due to the access and location transparency of the secondary storage. There is no need at all to move the files O_j physically. Thus the processing time of the integration phase can be neglected. This is the reason why our new parallel external sorting algorithm uses a *KEYPART* method in order to split the m initial runs IR_i into the p data partitions D_j.

Before presenting the new splitting algorithm, which implements a *KEYPART* method, we will take a closer look at the run time of the horizontally parallelized merge phase. This is an interim calculation, which excludes the run time of the new splitting algorithm. Under the condition that the p data partitions D_j are equally sized (i.e. contain the same number of tuples), the speedup is linear like the speedup of the presort phase of the external sorting algorithm. This statement can easily be deduced from the observations made in section 3.1. However one crucial precondition of the linear speedup property is that the data partitions are equally sized and can be computed efficiently. This exactly is guaranteed by our data partitioning method even in the presence of

arbitrary data skew.

4 Data Partitioning

Data partitioning by means of splitting keys is one precondition that a horizontally parallelized external sorting algorithm will work in a reasonable manner. Therefore this section explores the possibilities of data partitioning using splitting keys. The discussion of data partitioning of this section is not done without reference to the parallel external sorting algorithm. Before describing and analyzing the new partitioning algorithm, first general requirements are postulated and related issues are discussed.

4.1 Requirements and Existing Approaches

A data partitioning algorithm should be efficient and robust against data skew in order to be useful for a *KEYPART* method. A trade-off exists between these two requirements. On the one hand, one can use a simple sampling algorithm to deduce the splitting keys. This sampling method is easy to implement and has good performance. But the quality of the splitting keys tends to become unacceptable in presence of data skew. The splitting keys may yield data partitions which differ a lot in their sizes. Thus one principle precondition is not fulfilled. Explorations by Quinn [29] confirm this statement.

On the other hand, one can be robust against data skew by paying an enormous computational overhead. An exact partitioning algorithm (like the percentile finding algorithm [21]) is based upon searching algorithms by bisection. The parallel external sorting algorithm proposed in [34] runs the percentile finding algorithm to compute independent data partitions needed for the parallel merge phase. But the I/O complexity, which depends linearly on the number of initial runs, and the random accesses to secondary storage show that the percentile finding algorithm is unsuited to implement a *KEYPART* method efficiently [27].

These observations are noticed by DeWitt et al. in [16], too. A compromise solution must be found in order to keep the trade-off between efficiency and robustness as low as possible. It is clear that the sizes of the data partitions do not need to be totally equal. A certain degree of imbalance can be tolerated. Therefore an exact splitting algorithm (like the percentile finding algorithm [21]) is superfluous. The compromise solution consists in an intelligent sampling method. The probabilistic splitting algorithm used in [16] samples the input file to be sorted. The maximum imbalance between the single data partitions can be estimated by means of probability theory. However, the drawback of the probabilistic splitting algorithm is that it never can guarantee the maximum imbalance, because a factor of uncertainty will always remain. To overcome this disadvantage, a new adaptive partitioning algorithm by sampling is developed in the next section.

4.2 Adaptive Partitioning by Sampling

The new adaptive partitioning algorithm of this section pays attention to the data distribution of the input file by sampling it during the presort phase. The distance between the samples determines the accuracy of the splitting keys (i.e. the maximum imbalance of the data partitions due to the splitting keys can be predicted). The possibility to vary the distance between the samples derived from the input file makes the algorithm adap-

tive. Thus, every given limit of maximum imbalance can be reached. The following parallel merge phase does not suffer from processing skew in presence of data skew.

The main difference to the probabilistic splitting algorithm [16] is, that the adaptive partitioning algorithm can predict the quality of the splitting keys exactly instead of estimating it. Nevertheless the adaptive partitioning algorithm remains a sampling method and avoids the performance drawback of the percentile finding algorithm [21]. Furthermore, the adaptive partitioning algorithm exploits the access and location transparency of secondary storage and therefore avoids interprocessor communication during the redistribution of input tuples unlike the probabilistic splitting algorithm [16].

4.2.1 Description. The essence of the adaptive partitioning algorithm can be described in a few words: Each n^{th} tuple of each initial run forms a sample. They mirror the data distribution of the input file. After having sorted these samples, the adaptive partitioning algorithm computes the splitting keys. They represent the data distribution as well. Therefore imbalance between the sizes of the data partitions determined by these splitting keys is limited.

In order to give a formal description of the adaptive partitioning algorithm, we need the following definitions:

Def. 1. The *step distance* n is a positive number. It is a parameter of the adaptive partitioning algorithm and determines its accuracy.

Def. 2. A *sample area* is a connected region within a file of tuples. Its size (in number of tuples) is equal to the step distance n. A sample area is a logical partition which is independent of the physical blocks of an I/O-device.

The adaptive partitioning algorithm consists of 6 basic steps described in the sequel. Assume, that the m initial runs $IR_i \ (1 \le i \le m)$ are given. The adaptive partitioning algorithm now computes p data partitions $D_j \ (1 \le j \le p)$ implementing a *KEYPART* method (cf. section 3.4.2).

(i) Each initial run $IR_i \ (1 \le i \le m)$ is divided into l_i sample areas $R_{i,l} \ (1 \le l \le l_i)$ (i.e. $IR_i = R_{i,1} \ldots R_{i,l_i}$). The following equations hold:

$$\forall (1 \le i \le m) : l_i = \lceil |IR_i| / n \rceil \tag{4}$$

$$\forall (1 \le i \le m) \, \forall (1 \le l \le l_i - 1) : |R_{i,l}| = n \tag{5}$$

$$\forall (1 \le i \le m) : |R_{i,l_i}| = |IR_i| - n \cdot (l_i - 1) \tag{6}$$

Equation (4) determines the number of sample areas each initial run is divided into. Equation (5) says that all sample areas except the last one of each initial run consist of as many tuples as prescribed by the step distance n. The last sample area of each initial run possibly contains fewer tuples as indicated by equation (6).

(ii) Each sample area $R_{i,l} \ (1 \le i \le m, 1 \le l \le l_i)$ obtains an area key $A_{i,l}$. The area key $A_{i,l}$ is the value of the sorting attribute of the first tuple of $R_{i,l}$. Now define $C(IR_i)$ to be the sequence of area keys of initial run IR_i.

$$\forall (1 \le i \le m) : C(IR_i) = (A_{i,1}, \ldots, A_{i,l_i}) \tag{7}$$

Each $C(IR_i)$ is sorted, because the initial runs are sorted. Furthermore, each $C(IR_i)$ mirrors the data distribution of IR_i. This observation is important, because it makes the computation of the maximum load imbalance during the

merge phase possible. The situation after the first two basic steps of the adaptive partitioning algorithm is shown in figure 7. Initial run IR_i and $C(IR_i)$ are shown.

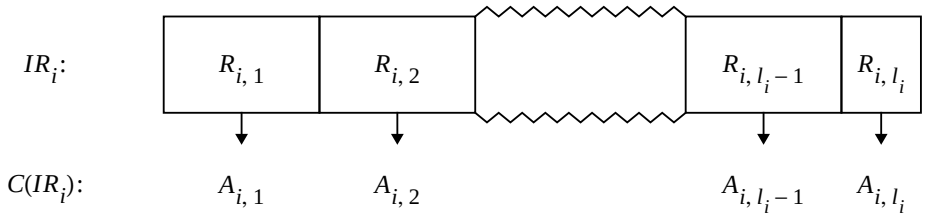

Fig. 7. Adaptive Partitioning Algorithm after the 2nd Step

(iii) The m sorted sequences of area keys $C(IR_i)$ are merged into the sorted sequence $C = (C_1, \ldots, C_L)$. It is $L = \sum_{i=1}^{m} l_i$. The merging can be realized by an usual m-way merging algorithm.

(iv) Now the sorted sequence C of area keys is divided into p equally sized partitions, thus yielding $(p-1)$ splitting keys k_j $(1 \le j \le p-1)$. The elements k_j are the splitting keys of the *KEYPART* method (cf. section 3.4.2), which divide the m initial runs IR_i into the p independent data partitions D_j. The value of k_j is defined as follows:

$$\forall (1 \le j \le p-1) : k_j = C_{j \cdot \lceil L/p \rceil} \tag{8}$$

The values k_0 and k_p can be defined arbitrarily, if they fulfill the following restriction:

$$k_0 < \min \{A_{i,1} \mid 1 \le i \le m\} \wedge k_p \ge \max \{A_{i,l_i} \mid 1 \le i \le m\} \tag{9}$$

(v) The splitting keys k_j are used to put the sample areas $R_{i,l}$ into the p groups G_j. The group G_j is defined as:

$$\forall (1 \le j \le p) : G_j = \{R_{i,l} \mid (1 \le i \le m) \wedge (1 \le l \le l_i) \wedge (k_{j-1} < A_{i,l} \le k_j)\} \tag{10}$$

Equation (10) says, that a group G_j contains exactly these sample areas $R_{i,l}$, whose area key lies in the interval bordered by the splitting keys k_{j-1} and k_j. Each group execpt G_p contains $\lceil L/p \rceil$ sample areas. The last group contains $L - ((p-1) \cdot \lceil L/p \rceil)$ sample areas.

Figure 8 shows the situation after the fifth step of the adaptive partitioning algorithm. The groups G_j approximate the searched data partitions D_j. The groups G_j are "nearly" disjunct. Only the borders of the groups G_j overlap. The last step has to compute the exact splitting positions within the initial runs.

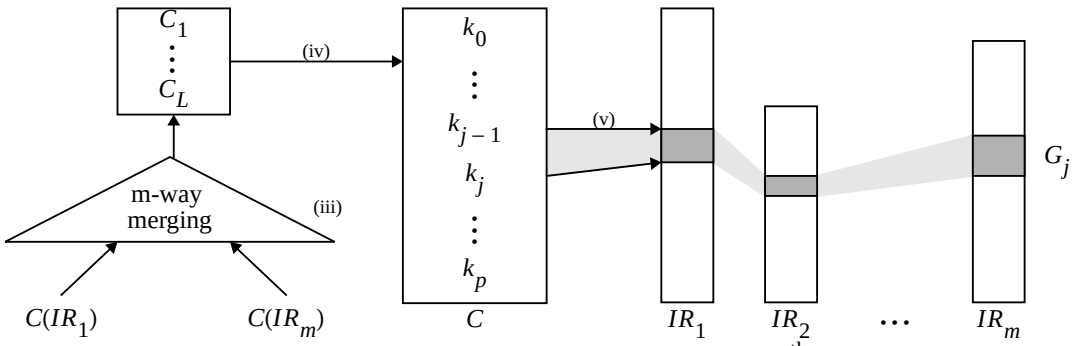

Fig. 8. Adaptive Partitioning Algorithm after the 5th Step

(vi) In order to compute the disjunct data partitions D_j from the groups G_j $(1 \le j \le p)$, the following observations are exploited:

· All tuples of the sample area $R_{i,\,l}$ have a sorting attribute, the value of which is not less than the area key $A_{i,\,l}$ belonging to $R_{i,\,l}$. This property can be deduced from the fact that initial runs are sorted.

· All tuples of the sample area $R_{i,\,l}$ have a sorting attribute, the value of which is not greater than the area key $A_{i,\,l+1}$ belonging to the sample area next to $R_{i,\,l}$. Again, this is true, because initial runs are sorted.

· The area keys $A_{i,\,l}$ of the sample areas $R_{i,\,l} \in G_j$ lie between the splitting keys determining the group G_j. More formally:

$$\forall\,(1 \le i \le m,\, 1 \le l \le l_i,\, 1 \le j \le p) : (R_{i,\,l} \in G_j \Rightarrow k_{j-1} \le A_{i,\,l} \le k_j) \tag{11}$$

Equation (11) limits the searching space for the exact splitting positions within the initial runs to compute the p searched data partitions D_j. The exact splitting positions are contained in the sampling areas bordering the groups G_j (cf. figure 9).

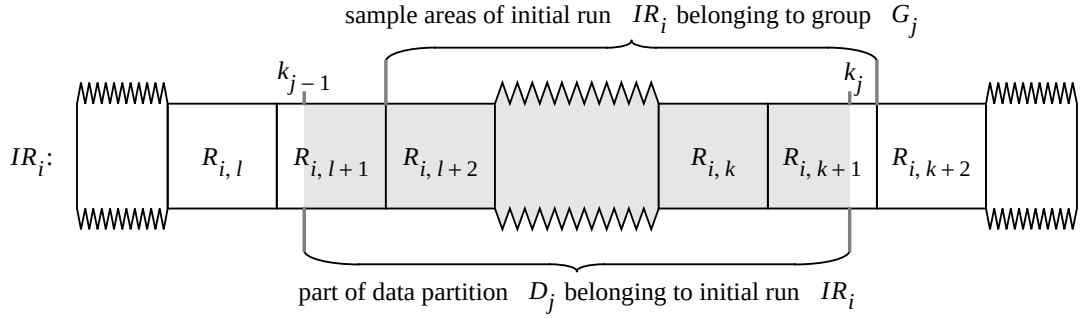

Fig. 9. Adaptive Partitioning Algorithm after the last Step

4.2.2 The Impact of Data Skew. Data skew might produce data partitions which have unequal size. We define the *imbalance* between two data partitions as the difference of the number of tuples in two data partitions:

$$Imbal\,(D_i, D_j) \;=\; \big||D_i| - |D_j|\big| \tag{12}$$

$$MaxImbal \;=\; \max\,\{Imbal(D_i, D_j) \,\big|\, 1 \le i, j \le p\} \tag{13}$$

A direct consequence of the description how to deduce the data partitions D_j from the groups G_j within step (vi) is the central proposition of this paper. It determines the maximum imbalance between the data partitions D_j.

Prop. 1. Assume that the p data partitions D_j are computed by our new adaptive partitioning algorithm from m initial runs using step distance n. Then the maximum imbalance $MaxImbal$ between two data partitions is limited by the following formula:

$$MaxImbal < 2 \cdot m \cdot n \tag{14}$$

It is an important observation that the maximum imbalance does neither depend on the degree of parallelism nor on the data skew. Using equation (1), one can deduce:

$$MaxImbal < 2 \cdot (N/h) \cdot n \tag{15}$$

Equation (15) is equivalent to:

$$MaxImbal/N < 2 \cdot n/h \tag{16}$$

The ratio $MaxImbal/N$ is called *normalized maximum imbalance*. It expresses the

maximum imbalance which might be obtained by the adaptive partitioning algorithm, related to the total size of the input file. This is necessary to give a worst case estimation of the parallel merge phase. In contrast to the probabilistic splitting algorithm [16], we are able to guarantee an upper limit of the run time needed to sort a given input file in parallel. Using the expected length of $2 \cdot h$ for initial runs [23], the above worst case imbalance can be guaranteed, but the expected normalized maximum imbalance will be less than n/h.

The following example shows that just a small fraction of the input file must be sampled to guarantee a low degree of normalized maximum imbalance. We assume that the input file contains $2 \cdot 10^6$ tuples (each having a size of 50 bytes yields a total size of the input file of 100 MByte). Furthermore, we assume that the main memory is limited to 2 MByte thus having a heap size of $4 \cdot 10^4$ tuples during the presort phase. Then table 1 says that only 0.25% of the input data must be sampled in order to limit the normalized maximum imbalance to 2 %.

Normalized Maximum Imbalance	0.5%	1.0%	2.0%	5.0%
Percentage of Sampled Input Tuples	1.0%	0.5%	0.25%	0.1%
Step Distance *n* (in tuples)	100	200	400	1000

Table 1. The Accuracy of the Adaptive Partitioning Algorithm

5 Implementation and Performance

In this section we describe the implementation of the horizontally parallelized external sorting algorithm as developed in section 3.4. The new adaptive partitioning algorithm of section 4.2 is used to implement the *KEYPART* method. This section concludes with performance results done by measurements and theoretical analysis.

5.1 Hardware and Software Environment

The new parallel external sorting algorithm has been implemented on top of an iPSC/2 (intel Super Personal Computer) [20]. The iPSC/2 is a distributed memory multiprocessor. Its nodes are connected in a hypercube manner. The CFS (Concurrent File System) provides access and location transparency of the secondary storage. Thus, the iPSC/2 in combination with the CFS belongs to the class of shared disk multiprocessors (cf. section 3.2). The configuration we used for the implementation and the performance measurements have had 16 processing nodes and two I/O processors equipped with 2 disks each.

The operating system used for the implementation is *MMK* [8]. It offers the programmer three different types of objects in order to design and develop his program. All objects have global names, which are unique within the system. A parallel *MMK*-program consists of a set of these objects which are mapped onto real processors of the iPSC/2 during run time.

TASK. A task performs a sequential program written in any usual sequential programming language like C or FORTRAN. The tasks are the active components of a *MMK*-program. They perform the parallel algorithm in its literal sense.

MAILBOX. Mailboxes are used for the intertask communication. A task can send a message to any mailbox whose name is known to it. The messages are stored in the

mailbox in a FIFO manner. Also, a task can receive a message from any known mailbox. The storing capacity of a mailbox is a start-up parameter which can be varied so that both synchronous and asynchronous intertask communication is possible.

SEMAPHORE. Semaphores deal with task synchronization.

5.2 Implementation

The set of tasks and mailboxes (semaphores are not used) of the parallel *MMK*-program implementing the new parallel sorting algorithm is shown in figure 10. The data flow in this figure is bottom up following the arcs.

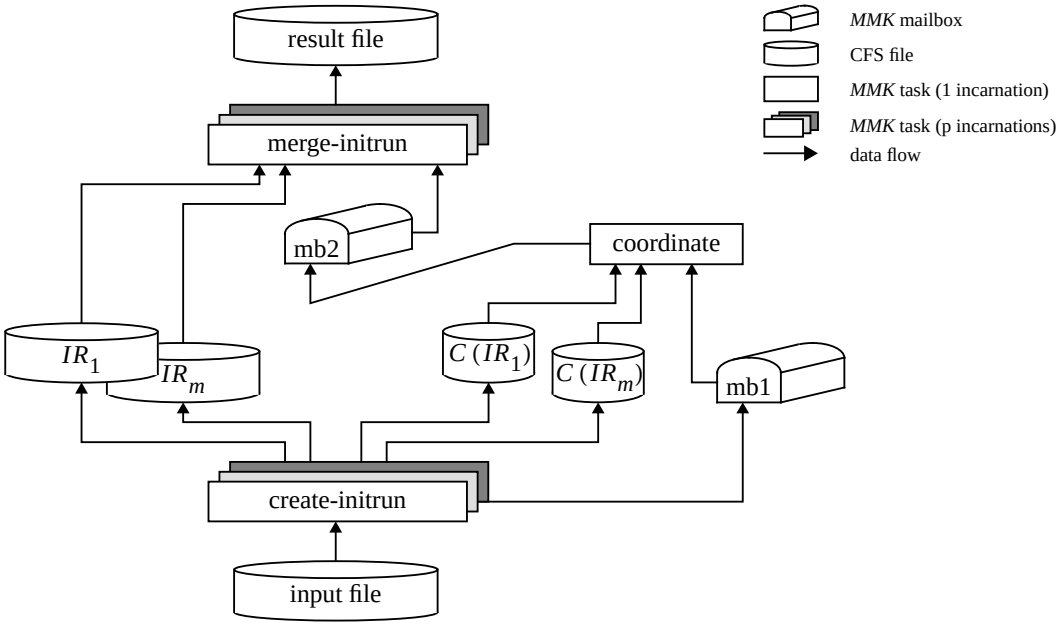

Fig. 10. The Parallel Sorting Algorithm on Top of *MMK*

The function of the task *create-initrun* is to generate initial runs from an unsorted input file which resides on the CFS (i.e. performing the presort phase). Furthermore it has to perform the basic steps (i) and (ii) of the adaptive partitioning algorithm. Therefore, its result is a set of initial runs IR_i and a set of area key sequences $C(IR_i)$ both stored in files of the CFS. After the completion of its job, the task *create-initrun* sends a message to the communication mailbox *mb1*. This message contains the filenames of the generated files. To process the task *create-initrun* in parallel, the paradigm of horizontal parallelism is applied and p incarnations of the task *create-initrun* are performed on p different nodes of the iPSC/2. The input file must be split logically as explained in section 3.4.1.

The function of the task *coordinate* is to perform the basic steps (iii) through (v) of the adaptive partitioning algorithm. It reads the files $C(IR_i)$. Their filenames are received from the mailbox *mb1*. The result of the task *coordinate* are the splitting keys k_j and the borders of the groups G_j. The result is sent to the mailbox *mb2*. This task cannot be performed in parallel, since it is the sequential part of the adaptive partitioning algorithm. Therefore, just one incarnation of it exists.

The function of the task *merge-initrun* is to compute the exact splitting positions within the initial runs IR_i according to the splitting keys k_j (i.e. performing the basic

step (vi) of the adaptive partitioning algorithm) and to merge the initial runs (i.e. performing the merge phase). The task *merge-initrun* gets its start-up parameters (i.e. filenames, splitting keys, etc.) by reading the mailbox *mb2*. To perform the task *merge-initrun* in parallel, p incarnations of the task *merge-initrun* are processed on p different nodes of the iPSC/2 (i.e. horizontal parallelism). Attention must be paid that the number of splitting keys and the degree of parallelism during the merge phase are equal.

5.3 Performance

In order to demonstrate performance results, the following testbed has been chosen: The unsorted input file has a size of 100 MByte. Each tuple within the input file has a size of 50 bytes. The heap of the tasks *create-initrun* can hold 10^4 tuples each to generate the initial runs. The step distance n during the adaptive partitioning algorithm is set to 400. Comparing this parameter setting with table 1, a normalized maximum load imbalance of 2% can be guaranteed.

Figure 11 shows the run time during the presort and merge phases. It demonstrates linear speedup until the I/O-subsystem (i.e. the CFS of the iPSC/2) becomes the bottleneck. Referring to the discussion of section 3.3, our new parallel sorting algorithm is effectively scalable. Because the access pattern to the secondary storage is different during the presort and merge phases, the two phases have different values of balance parallelism. During the presort phase the access to the secondary storage is sequential, whereas it is random during the merge phase. The degree of CPU parallelism which can be exploited effectively seems to be very low. The reason is the performance imbalance between the CPU and the I/O-subsystem of the iPSC/2 which we used for our measurements. During the *create-initrun* phase 100 MByte are read from an written to the disk, i.e. a total of 200 MByte. This takes a total time of about 250 sec at the point of balance parallelism. Thus the performance of the I/O-subsystem of the iPSC/2 is about 800 KByte/sec. During the *merge-initrun* phase disk performance is about 500 KByte/sec.

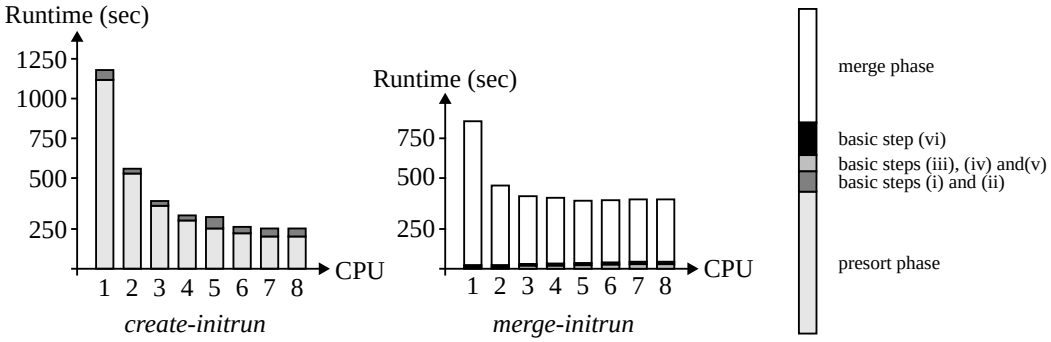

Fig. 11. Run Time of the Parallel Sorting Algorithm

Figure 11 expresses the additional run time due to the adaptive partitioning algorithm, too. It can be seen that the additional run time is very low w.r.t. the run time for sorting in its literal sense. In order to confirm this observation we give a theoretical analysis of the complexity of the adaptive partitioning algorithm.

The basic steps (i) and (ii) are performed during the presort phase by the task

create-initrun. The computation of the files IR_i and $C(IR_i)$ is done simultaneously. Furthermore, the paradigm of horizontal parallelization is applicable. The additional CPU costs to compute the area keys $A_{i,j}$ can be neglected, because the sorting attributes of the sampled tuples have to be extracted anyway during the presort phase. The I/O complexity of the basic steps (i) and (ii) are $O((N/n) \cdot \kappa)$ where κ is the ration $KeySize/TupleSize$.

The basic steps (iii), (iv) and (v) are performed by the task *coordinate*. This is the only sequential part of the adaptive partitioning algorithm. Therefore the additional costs must be explored very carefully. The additional I/O costs have again a complexity of $O((N/n) \cdot \kappa)$ like during the steps (i) and (ii). The additional CPU costs in order to merge the m sequences $C(IR_i)$ into the sequence $C = (C_1, \ldots, C_L)$ have a complexity of $O((N/n) \cdot \log(m))$. This may look expensive, but the measurements have shown that the sequential part of the adaptive partitioning algorithm can be performed quickly compared to the total run time needed to sort large volumes of data.

The final basic step (vi) of the adaptive partitioning algorithm can be performed in parallel again. Each process P_j which has to merge the data partition D_j assigned to it gets a prologue. During this prologue the exact splitting positions within the groups G_j are computed. The additional CPU and I/O both have a complexity of $O(n)$. The costs are due to the additional sample area, which must be loaded from secondary storage. Within this additional sample area the exact splitting position must be found according to the splitting keys. Table 2 summarizes the theoretical results about the additional costs due to the adaptive partitioning algorithm.

basic step	add. I/O costs	add. CPU costs	parallel processing
(i), (ii)	$O((N/n)\cdot\kappa)$	negligeable	yes
(iii), (iv), (v)	$O((N/n)\cdot\kappa)$	$O((N/n)\cdot\log(m))$	no
(vi)	O(n)	O(n)	yes

Table 2. Additional Costs of the Adaptive Partitioning Algorithm

6 Summary and Future Work

We have proposed a new parallel sorting algorithm for large data volumes. It applies the paradigm of horizontal parallelism. The two phases are treated separately. The presort phase turns out to have splitting and integration phases without any significant overhead. Furthermore, load imbalance cannot occur during the parallel processing phases. The subsequent merge phase gets a prologue in the course of which a *KEYPART* method computes splitting keys. They are used to partition the initial runs horizontally. Parallel merging becomes possible. Both phases show linear speedup characteristics until the I/O-subsystem becomes the bottleneck. By means of CPU parallelism, we are able to tune the multicomputer.

Our new adaptive partitioning algorithm implementing a *KEYPART* method fulfills the necessary requirement of efficiency (low overhead) and robustness against data skew (load balance). Although it performs sampling in order to guarantee efficiency, it can give a worst case estimation of the maximum imbalance which may occur. The basic idea, which distinguishes our algorithm from existing sampling algorithms, is the way how the samples are taken. Instead of sampling the unsorted input data stream, we

sample the initial runs. Thus, we have samples mirroring the data distribution. Therefore the splitting keys extracted from the samples are not sensitive to data skew. The accuracy of the splitting keys depends on the step distance (input parameter), which adapts the worst imbalance to given requirements. An other important requirement is that the accuracy of our adaptive partitioning algorithm does not depend on the degree of parallelism.

In the future, the following topics should be analyzed: The description of our adaptive partitioning algorithm provides various possibilities for tuning. Our current implementation sends the area keys $A_{i,j}$ via the CFS to the task *coordinate*. But the amount of data due to the area keys[2)] does not justify to store the area keys on secondary storage. Efficient main memory data structures and the intertask communication facility of *MMK* can be used to accelerate the algorithm.

Up to now, the performance measurements are done on a system which hardly provides parallelism within the I/O-subsystem. A highly parallel I/O-subsystem with fast access times and high transfer rates should be used to prove the ability of our new parallel sorting algorithm under this environment, too.

The most important challenge for the future is the integration of our new parallel sorting algorithm into the parallel query evaluation plans of our relational database system *TransBase* [33]. Its sequential query evaluation plans treat sorting as a monolithic operator. It will be necessary to divide this monolith into a presort operator and a merge operator. Thus a different degree of parallelism can be assigned to the operators ensuring the optimal degree of parallelism. Furthermore, this integration gives the processors more load due to other operators (e.g. selection) of the parallel query evaluation plans. This increments the potential of more parallelism compared to the case that only sorting has to be performed by the processors.

Appendix (List of used parameters)

- N: number of tuples to be sorted
- p: degree of parallelism (= number of processors)
- m: number of initial runs
- h: heap size used during presort phase
- n: step distance of our partitioning algorithm
- κ: ratio: $KeySize/TupleSize$

References

1. A. Aho, J. Hopcroft, J. Ullman; *Data Structures and Algorithms*; Addison Wesley Publ. Comp. Inc., 1983
2. P. Apers, M. Kersten, H. Oerlemans; *PRISMA Database Machine: A Distributed Main Memory Approach*; In: Proceedings of the 1st International Conference on Extending Database Technology, Venice, Mar 88
3. K. Batcher; *Sorting Networks and their Applications*; In: Proceedings of the 1968 Spring Joint Computer Conference, Vol. 32, 1968, pp. 307 - 314
4. B. Baugstø, J. Greipsland; *Parallel Sorting Methods for Large Data Volumes on a Hypercube Database Computer*; In: Proceedings of the 6th International Workshop on

2) If we assume that a key of a tuple has a length of 10 bytes, then the set of area keys extracted from the initial runs consumes 50 KByte of memory in the example chosen in section 5.3.

Database Machines, Deauxville, Jun 89, LNCS No. 368, pp. 128 - 141
5. R. Bayer, T. Härder; *Preplaning of Disk Merges*; Computing, Vol. 21, No. 1, pp. 1 - 16, 1978
6. R. Bayer, T. Härder; *A Performance Model for Preplaned Disk Sorting*; Computing, Vol. 21, No. 1, pp. 17 - 36, 1978
7. M. Beck, D. Bitton, K. Wilkinson; *Sorting Large Files on a Backend Multiprocessor*; IEEE Transactions on Computers, Vol. 37, No. 7, Jul 88
8. T. Bemmerl, T. Ludwig; *MMK - A Distributed Operation System Kernel with Integrated Loadbalancing*; In: Proceedings of the CONPAR 90 - VAPP IV, Zürich, Sept 90
9. T. Bemmerl, A. Bode, P. Braun, O. Hansen, T. Treml, R. Wismüller; *The Design and Implementation of TOPSYS*; Technical report, Technische Universität München, No. 342/16/91 A, Jul 91
10. G.Bilardi, A. Nicolau; *Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-Memory Machines*; SIAM J. Comput., Vol. 18, No. 2, Apr 89, pp. 216 - 228
11. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, P. Valduriez; *Prototyping Bubba, A Highly Parallel Database System*; IEEE Transactions an Knowledge and Data Engineering, Vol. 2, No. 1, Mar 90
12. U. Borghoff; *Catalogue of Distributed File/Operating Systems*; Springer Verlag, Berlin Heidelberg, 1992
13. T. Chen, V. Lum, C. Tung; *The Rebound Sorter: An Efficient Sort Engine for Large Files*; In: Proceedings of the 4th International Conference on Very Large Data Bases, West Berlin, pp. 312 - 318
14. R. Cole; *Parallel Merge Sort*; SIAM J. Comput., Vol. 17, No. 4, Aug 88, pp. 770 - 785
15. D. DeWitt, S. Ghandeharizadeh, D. Scheider, A. Bricker, H. Hsiao, R. R. Rasmussen; *The Gamma Database Machine Project*; IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, Mar 90
16. D. DeWitt, J. Naughton, D. Schneider; *Parallel Sorting on a Shared Nothing Architecture using Probabilistic Splitting*; In: Proceedings of the 1st International Conference on Parallel and Distributed Information Systems, Miami Beach, Florida, Dec 91
17. Y. Dohi, A. Suzuki, N. Matsui; *Hardware Sorter and its Application to Database Machine*; In: Proceedings of the 9th Conference on Computer Architecture, Austin, Apr 82, pp. 218 - 225
18. B. Edem, R. Helliwell, T. Johnston, E. Lary, R. Lary; *Sort Accelerator*; Technical Report, Database Research Group, DEC, May 90, Do. No.: DBS-TR-3 DEC-TR-691
19. G. Gibson; *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*; Technical Report, No. UCB/CSB 91/613, Computer Science Decision, University of California, Berkeley, Dec 90
20. Intel Scientific Computers; *Concurrent Supercomputing - The Second Generation, A technical Summary of the iPSC/2 Concurrent Supercomputer*; Reprinted from the Proc. of the ACM Third Hypercube Conference
21. B. Iyer, G. Ricard, P. Varman; *Percentile Finding Algorithm for Multipe Sorted Runs*; In: Proceedings of the 15th International Conference on Very Large Databases, Amsterdam, 1989, pp. 135 - 144
22. B. Kandler, M. Pawlowski; *SAM: A Sorting Toolbox - User's Guide*; Technical Report, Technische Universität München, No. 342/2/91 B, Jun 91 (in german)
23. D. Knuth; *Sorting and Searching. The Art of Computer Programming*; Addison Wesley Publ. Comp. Inc., 1973, Vol. 3
24. K. Lehnert; *Regelbasierte Beschreibung von Optimierungsverfahren für relationale Datenbankabfragesprachen*; Ph.D. Thesis, Technische Universität München, Dec 88 (in german)
25. E. Loibl, H. Obermaier, M. Pawlowski; *Towards Parallelism in a Relational Database System*; Technical Report, Technische Universität München, No. 342/10/91 A, Jun 91
26. R. Lorie, H. Young; *A Low Communication Sort Algorithm For a Parallel Database Machine*; In: Proceedings of the 15th International Conference on Very Large Data Bases,

Amsterdam, 1989, pp. 125 - 134
27. D. Menzel; *Paralleles Externes Sortieren auf Multiprozessoranlagen*; Master Thesis, Technische Universität München, Nov 1991 (in german)
28. D. Patterson, G. Gibson, R. Katz; *A Case for Redundant Arrays of Inexpensive Disks (RAID)*; In. Proceedings of the SIGMOD International Conference on Managemant of Data, Editors: H. Boral, P. Larson, ACM Press, Chicago, Jun 88, pp. 109 - 116
29. M. Quinn; *Parallel Sorting Algorithms for Tightly Coupled Multiprocessors*; Parallel Computing, Vol. 6, 1988, pp. 349 - 357
30. A. Reuter; *Database Sharing*; Informatik Spektrum, Vol. 8, No. 4, Apr 85, pp. 225 - 226
31. M. Stonebraker; *The Case for Shared Nothing*; Database Engineering, Vol. 9, No. 1, 1986
32. Teradata Corp.; *DBC/1012 Database Computer System Manual*; Doc. No. C10-0001-02, Nov 1985
33. TransAction Software GmbH; *TransBase Relational Database System*; System Guide, München, 1988
34. P. Varman, B. Iyer, S. Scheufler; *A Multiprocessor Algorithm for Merging Multiple Sorted Lists*; In: Proceedings of the International Conference on Parallel Processing, 1990, pp. III-22 - III-26