RC 19846 (87791) 11/22/94
Computer Science

IBM Research Report

A Performance Study of Workfile Disk Management for Concurrent Mergesorts in a Multiprocessor Database System

Kun-Lung Wu, Philip S. Yu and Jen-Yao Chung

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, NY 10598

James Z. Teng

IBM Programming Systems
P.O. Box 49023
San Jose, CA 95161

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

A Performance Study of Workfile Disk Management for Concurrent Mergesorts in a Multiprocessor Database System

Kun-Lung Wu, *Philip S. Yu* and *Jen-Yao Chung*

IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

James Z. Teng

IBM Programming Systems
P.O. Box 49023
San Jose, CA 95161

Abstract

This paper studies the performance of concurrent mergesorts in a multiprocessor database system. Specifically we examine the impacts of workfile disk allocation and data striping on the average sort response time. Workfile disks are disks that provide the external working space for the mergesorts. Concurrent mergesorts in a multiprocessor system can create severe I/O interference in which a large number of sequential write requests are continuously issued to the same workfile disk and block other read requests for a long period of time. We examine through detailed simulations a logical partitioning approach where workfile disks are logically partitioned into equal-sized groups and an arriving sort job selects one group to do the mergesort. We first study the impact of partition size on the sort response time assuming that data striping is not used (i.e., each intermediate sorted run from the sorting phase is stored on one disk). Then, we examine the impact of data striping. The results show that (1) without data striping, the best performance is achieved by using the entire workfile disks as a single partition if there are abundant workfile disks (or system workload is light); (2) however, if there are limited workfile disks (or system workload is heavy), the workfile disks should be partitioned into multiple groups and the optimal partition size is workload dependent; (3) data striping is beneficial only if the striping unit size is properly chosen; and (4) with a proper striping size, the best performance is generally achieved by using the entire disks as a single logical partition.

Index Term: workfile disk allocation, I/O parallelism, prefetching, concurrent mergesorts, data striping, performance evaluation.

1 Introduction

One of the most time-consuming operations in query processing is the sorting of a large table (relation), which typically uses an external sort or mergesort. A mergesort involves two phases: sorting phase and merge phase. During the sorting phase, tuples in a relation are first sorted into multiple *runs* according to a certain sort key [8]. These initial sorted runs are referred to in this paper as the *input runs* of a merge job. The input runs are temporarily stored on the workfile disks, which are external working spaces of a database system. During the merge phase, the input runs of a merge job are read into the buffer from the workfile disks, merged and then the merged data are written back to workfile disks in the final sorted order, referred to as the *output run* of the merge job. The disks for the output run may or may not be the same disks used for the input runs, depending on the total number of workfile disks.

As coupling multiple microprocessors to form a high performance machine becomes popular, more and more multiprocessor systems have been used for database query processing [1, 15]. In this paper, we study the performance of concurrent mergesorts in a multiprocessor database system where multiple workfile disks are shared by the processors. In concurrent mergesorts, the allocation of workfile disks to a sort job can have a significant performance impact. In order to achieve better load balancing among the disks, we want the runs of each mergesort to be spread evenly among all the disks. However, by sharing the disks, the progress of a mergesort can be blocked by a different mergesort due to an I/O interference on the same disk. For example, Job 1 can be blocked waiting for a read from a disk which is currently serving a write request from Job 2. Unlike the interference in general file system I/O's where the requests are generally homogeneous in size and random in access pattern, the workfile disk I/O interference in concurrent mergesorts is unique and can be rather severe. Since multiple disks can be fetching the input runs in parallel and only one disk is writing the output run, many relatively large sequential write requests can be continuously issued to the same disk during the merge phase. If a read request from another job is issued to the same disk, it can be blocked for a very long time. Moreover, once one of the read requests gets blocked during a merge phase, the entire merge process gets blocked even though other read requests have been serviced by other workfile disks (more details on the implementation of concurrent mergesorts will be provided in Section 3).

In order to contain interference between different sort jobs, the concept of logical partitioning of workfile disks is introduced to assign a sort job to one of the partitions. This would eliminate the interference among jobs assigned to different partitions. However, it may limit the load balancing capability of the system as a sort job can spread its runs among the disk in one partition. (Note that the best load balancing can be achieved by simply assigning a run to the least loaded workfile disk. But, the I/O interference between different jobs cannot be effectively controlled.) As disk array systems have become popular [14, 13, 17], we also examine the impact of data striping on the performance of concurrent mergesorts. With data striping, the continuously issued large sequential write requests, which are previously on the same disk, can now be spread among multiple disks. As a result, the blocking time can be substantially reduced.

Detailed simulations are conducted to evaluate the performance of concurrent mergesorts. We first study the impact of partition size on the sort response time and assume that no data striping is used. Namely, each input run is stored on one disk. However, since the length of the output run of a mergesort is the sum of those of its input runs, we assume that the output run is partitioned into multiple segments and spread among multiple disks. Each segment is assumed to be about the size of an input run. The results show that, without data striping, the best performance is achieved by using a single partition consisting of all workfile disks if there are abundant workfile disks (or the system workload is low). However, in disk-limited cases (or the system workload is high), better performance can be achieved by using multiple partitions of workfile disks and the optimal partition size is workload dependent. We also examine the impact of data striping on the average sort response time. The results show that data striping may not help the sort response time because data striping creates more I/O requests. It is beneficial only if the striping unit size is properly chosen.

There exist many papers that mainly focus on the performance of the merge phase of an external sort [8, 2, 5, 7, 9, 12, 18]. However, most of these papers have only dealt with a single merge job in a single system, and have not considered the interactions of concurrent merges in a multiprocessor system. There are also existing work related to allocating buffer space for general relational database queries [11, 6, 4, 24, 16]. However, their emphases were mostly different from ours and again they did not study the run placement and disk allocation problems. The run placement issue in a single system was addressed in [23] and the buffer allocation and thrashing control issues were considered in [22]. But the sharing of workfile disks by multiple processors as well as the disk allocation strategy were not discussed. As will be shown later on, the performance of concurrent merges in a multiprocessor database system depends to a very large degree on the disk allocation strategy.

The paper is organized as follows. Section 2 describes the logical partitioning approach to workfile disk allocation. Section 3 describes the system model. Section 4 describes the simulation model and its implementation. Section 5 then presents the performance results from the simulations.

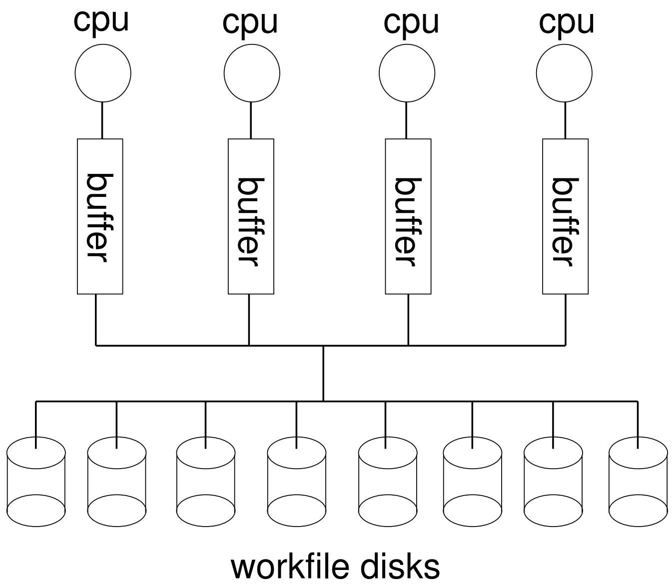


Figure 1: System architecture for concurrent mergesorts.

2 Workfile disk allocation

In this section, we describe the workfile disk allocation strategies for performing concurrent mergesorts in a multiprocessor system. Fig. 1 shows the system architecture where 8 workfile disks are shared by 4 processors. Each processor has its own buffer for performing the mergesorts. Sort jobs are assumed to arrive independently at each processor. During the sorting phase, the data to be sorted, stored in other data disks not shown in Fig. 1, are fetched into the buffer, sorted into runs, and then written each run to one of the workfile disks (or multiple disks, if data striping is used). After the sorting phase, these runs are fetched back to the buffer, merged and then written back to the workfile disks.

The disk allocation strategy determines the way the workfile disks are used by a sort job. In this paper, we study a logical partitioning approach to disk allocation. The workfile disks are logically partitioned into groups of equal size. Each sort job selects one group that is least loaded and uses the group of disks to do the mergesort.

In order to have better load balancing, runs should be distributed evenly among all the disks. Namely, the entire workfile disks are viewed as a single partition. Fig. 2(a) shows an example of one partition. Assume that there are two sort jobs J and K currently being executed. Each job has 5 runs after the sorting phase, denoted by $J.0$, $\cdots$, $J.4$ and $K.0$, $\cdots$, $K.4$, respectively. The

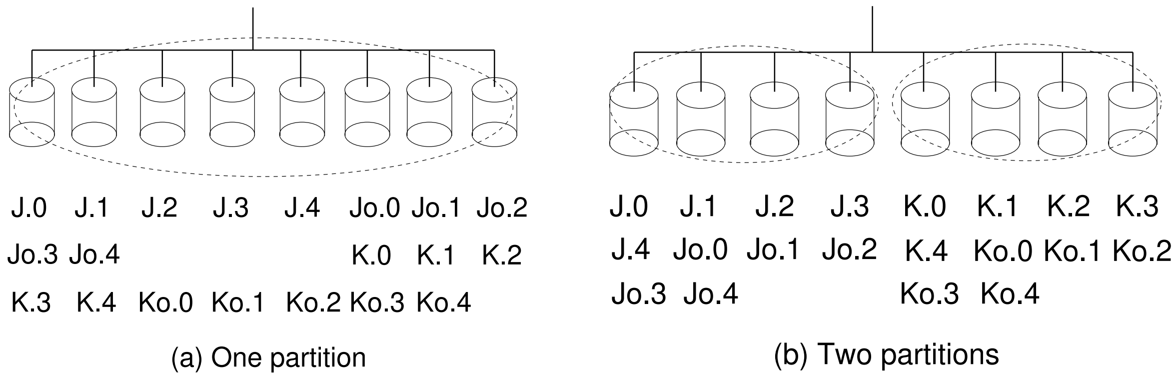


Figure 2: Logical partitioning of workfile disks.

output runs of J and K are denoted by Jo and Ko, respectively. Because the length of Jo is the sum of $J.0$, $\cdots$, $J.4$, we assume that it is partitioned into multiple segments, denoted by $Jo.0$, $\cdots$, $Jo.4$, with each segment about the size of one of its input runs. The output run is spread among multiple disks. Note that even though the output run Jo is placed on multiple disks, at any instant during the merge phase, there is only one disk that is actively performing the I/O for this output run. Similarly, during the sorting phase, there is only one disk that is actively writing for an input run. Moreover, none of the disks used for the output run is active during the sorting phase. As a result, we place the input runs of all the mergesorts in a round-robin fashion and the first segment of the output run of a mergesort job is placed right next to the disk that is used for its last input run. For example, in figure 2(a) the first input run of job K and the first segment of the output run of job J are both placed right next to the one for $J.4$, the last input run of job J.

Even though using a single partition can achieve the best load balancing among workfile disks, it can create severe I/O interference between different sort jobs. For example, in Fig. 2(a) job J can be blocked waiting for a read I/O from the disk storing run $J.2$ because the disk is currently serving a write I/O request from job K. As will be explained in more detail in Section 3, write requests are batched requests consisting of substantially larger amount of data than each read request. Moreover, many such write requests may be issued continuously to the same disk. If the read request from job J is blocked, it can be blocked for a long time. Once this read is blocked, job J is blocked because it cannot proceed without the data. This I/O interference problem is unique in the context of concurrent mergesorts in a multiprocessor system.

Thus, the average sort performance may be improved if the workfile disks are partitioned into multiple groups and each sort job chooses one group to perform the sorting. Fig. 2(b) shows an example of partitioning the workfile disks into two groups. In this case, jobs J and K use a different group and thus I/O interference from these two jobs is eliminated. However, in general there may be some disk groups that are not being used for sorting at some point in time while other groups are overloaded. For instance, job K may not be active until job J is almost done in Fig. 2(b). As a result, during the execution of job J, only 4 disks are active while the other 4 are idle, resulting in load imbalance among the disks.

Note that besides the severe blocking between different jobs, there may also be some self-blocking. But, the self-blocking of a read I/O by a write I/O from the same job, such as a read I/O for $J.0$ and a write I/O for $Jo.3$ in Fig. 2, is substantially less severe because without the data read from $J.0$, there is no further data to be written for $Jo.3$. Thus, the time for self-blocking is only limited. But, it can be very long if the blocking occurs between different jobs.

3 The system model

In this section, we describe the process of completing a mergesort job at one processor. Jobs are assumed to arrive independently at each processor. The data to be sorted are stored in separate data disks and are not shown in Fig. 1. A mergesort starts with a sorting phase where data are prefetched from the data disks into the buffer and a tournament tree-like algorithm is used to generate intermediate sorted runs. We assume that there is little queueing delay on the data disks that store the original relation to be sorted as they are well balanced [21]. However, we model the workfile disks in detail. The intermediate sorted runs are written to the workfile disks one at a time. If data striping is not used, a run is written onto one disk. If data string is used, depending on the striping unit size it may be spread among multiple disks. After the entire data are sorted into runs, the sorting phase ends and the merge phase begins.

To merge M input runs, at least the M active pages (the pages currently being merged), one from each input run, have to be in the buffer of that processor. These active buffer pages are *pinned* in the buffer and cannot be replaced until *unpinned* after the data are depleted by the merge process. The process of locating the active page of a run in a buffer is referred to as a *GetPage*. When one of the active pages is depleted, a GetPage request is issued for the next page from the same run. If a page is already in the buffer, then the buffer page is pinned and the merge process continues. If not, a synchronous read is issued and the merge process enters into a wait state until the read I/O is completed.

Prefetching is used to speed up the fetching of data both in the sorting phase and in the merge phase. The number of pages prefetched from each disk in an I/O request is referred to as the prefetch quantity (PFQ). (We assume that a page is 4K bytes.) A prefetch request is issued after a prefetch triggering page becomes active and is located in the buffer. A prefetch triggering page is the page whose page number is a multiple of the current PFQ. For example, if PFQ is 8, then pages 0, 8, 16, ... are prefetch triggering pages. When page 8 becomes active and is located in the buffer, a prefetch request for pages 16-23 is issued. To facilitate the prefetching, extra buffer space is needed. For example, if M runs are to be merged in the merge phase, $2 \times M \times$ PFQ pages of space are needed in a buffer. In the simulations, we assumed PFQ can be 1, 2, 4, or 8 for the merge phase but PFQ is fixed at 32 for the sorting phase, similar to IBM's DB2 [19]. (The setting of PFQ during the merge phase is discussed in Appendix A.) The PFQ for the merge phase must be smaller because there may be M prefetch I/O's executing simultaneously for each job. On the other hand, for the sorting phase there is only one prefetch request for each job.

In addition to prefetching on reads, the I/O efficiency can also be improved by deferring and batching the disk write requests. This is referred to as the *deferred write*, in contrast to the *synchronous write* where the write request must be completed before any further processing of the mergesort can be continued. During both the sorting and merge phases, dirty pages are generated in the buffer, similar to IBM's DB2 [20]. The buffer manager at each processor maintains a deferred write queue for the output of each ongoing sorting and merge and an LRU chain linking all the buffer pages. An asynchronous write request is issued when a certain number of dirty pages have been generated for a job. The number of dirty pages written out in each asynchronous write I/O is referred to as the *deferred write quantity* (DWQ). In this paper, we use 32 as the DWQ for both the sorting and merge phases.

At each processor, a merge job is scheduled for execution if enough buffer space is available (details of the scheduling of merge jobs are provided in Appendix A). Buffer availability affects the PFQ which in turn affects the disk I/Os. Tables 1 and 2 summarize the notation and definitions for the workload and system parameters, respectively. The number in the parentheses at the end of a definition is the default value used in the simulation if not otherwise specified.

4 The simulation model

Here we outline the simulation model and discuss the confidence intervals that were obtained from the simulations. The default values used in the simulation are given in Table 1 if not otherwise

Notation	Definition (Default values)
λ	sort job arrival rate at each processor, Poisson interarrival time distribution (0.015 jobs/sec)
R	mean base relation size, uniform distribution between $R/2$ records and $3R/2$ records (200,000 records)
r	record size (512 bytes)
T	tournament tree size (512 pages)
f	filtering factor, i.e., percentage of qualifying records in the base relation for sorting (0.4)
L	input run length (1024 pages)
PFQ	prefetch quantity, can be 1, 2, 4, or 8 for the merge phase (8 pages) and 32 for the sorting phase
P_{max}	maximum prefetch quantity at one processor for the merge phase (8)
DWQ	deferred write quantity (32 pages)

Table 1: Workload parameters.

Notation	Definition (Default values)
B	buffer size at each processor (40,000 pages)
D	total number of workfile disks
N	total number of processors (4)
P	total number of partitions
$Mips$	CPU MIPS (40)
C_{set_up}	CPU cost for initiating an I/O request, such as a synchronous read, a prefetch, an asynchronous write, and a synchronous write (5,000 instructions)
C_{merge}	the CPU cost for generating a dirty page (10,000 instructions)
C_{mgm}	CPU cost for managing the page table when a GetPage is a buffer hit (100 instructions)
T_{seek}	average disk seek time (16 ms)
$T_{latency}$	average disk latency time (8.35 ms)
$T_{transfer}$	average disk transfer time for a page of 4K bytes (0.91 ms)

Table 2: System parameters.

specified.

A discrete event-driven simulator consisting of four major components was developed. The first major component is a detailed buffer manager for each processor maintaining the LRU chain and the deferred write queues as described in Section 3. It tracks the status of each buffer page, whether it is pinned, unpinned, or not yet referenced. We explicit model the buffer component because buffer availability affects the merge scheduling algorithm (see Appendix A) and the PFQ, which in turns affects workfile disk contention. Insufficient buffer can result in smaller PFQ (i.e., more reads) and more frequent writes.

The second major component implements a job control algorithm. When a new job arrives, its input runs and output run are assigned to the disks in a logical partition. (Input runs are placed in each logical partition in a round-robin fashion, see Fig. 2.) Then it is scheduled for sorting. After the sorting phase, it is scheduled for merge if possible (see Appendix A for details on scheduling algorithms). If not, it waits on the *allocation queue* of that processor.

The third major component implements the sorting and merge process. For the sorting process, data are prefetched from the data disks storing the original relation, sorted and written to the workfile disks as intermediate sorted runs using deferred writes. The merge process depletes the active pages from each of the input runs and generates dirty pages in the buffer. After an active page is depleted, a GetPage for the next page of the same run is issued. If found in the buffer, the page is pinned and the merge process resumes. Otherwise, the merge process is blocked and waits until the page is read in from disk.

The fourth major component implements the disks. Each disk maintains a request queue and services the I/O requests FCFS. When data striping is used, multiple disk I/O requests may be issued to different workfile disks according to the striping unit size. Even when data striping is not used, the output run is still partitioned into multiple segment with each segment the similar size as that of an input run. Thus, it is equivalent to a striping size of an input run. In this paper, we assume that the average size of an input run is twice the size of the tournament tree size [7], and all the input runs are of equal size.

The CPU service times are constants that correspond to the CPU MIPS rating ($Mips$ MIPS) and the specific instruction pathlengths given in Table 2, including C_{merge} for generating a dirty page, C_{set_up} for initiating a read or write I/O request (both synchronous and asynchronous), and C_{mgm} for pinning a page. The I/O service time (not including the queuing time) is estimated as follows. If the previous I/O request and the current I/O request are both for the same run, then there is no seek time for this request. Thus, the service time for this I/O request is

$$T_{latency} + T_{transfer} \times N_{transfer}. \tag{1}$$

However, if the current I/O request is for a different run, then a seek time is added to the service time for this request. Thus, it becomes

$$T_{seek} + T_{latency} + T_{transfer} \times N_{transfer}. \tag{2}$$

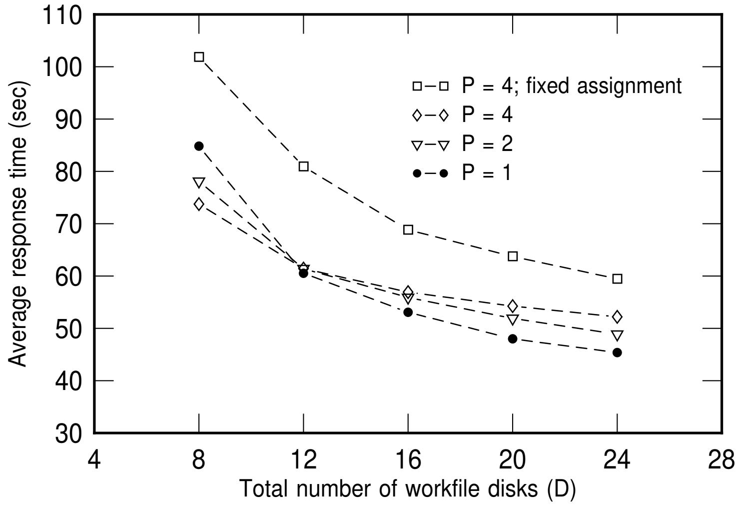
Here, T_{seek}, $T_{latency}$ and $T_{transfer}$ are the average disk seek time, average latency time and average transfer time per page, respectively, and $N_{transfer}$ is the number of pages transferred.

The mergesort request arrival process at each processor is assumed to be Poisson with rate λ. Each request contains a certain number of records and each record contains a fixed number (512 bytes) of data. The number of records for a sort job is assumed to be uniformly distributed between $R/2$ and $3R/2$, where R is the average number of records. Since there is typically a qualifying clause in a sort query, a filtering factor is used to reduce the number of records qualifying for sorting. After the sorting phase, the sort key values are assumed to be uniformly distributed among all the input runs.

Finally, in all the simulation results reported in this paper, confidence intervals were obtained (but not shown on the graphs) using the method of batch means [10]. Assuming independence of the estimates derived from each batch, confidence intervals were computed. For most cases, the estimated confidence intervals are very tight, except at very high levels of disk utilization. Nevertheless, the 90% confidence interval is consistently within 10% of the point estimates for all the data-points shown.

5 Simulation results

In this section, we present the performance studies of concurrent mergesorts. We first examine in detail the impact of logical partition size on the average response time under the condition that data striping is not used. Then we examine the impact of striping unit size and the combined impact of data striping and logical partitioning. Various sensitivity analyses were conducted for different parameters. However, we only show a subset of the results in this section as they demonstrate similar messages. For all the performance figures shown in the following subsections, the default value for the number of processors N is 4, the buffer size B at each processor is 40,000 pages, and the relation size is 200,000 records, which is equivalent to 10 intermediate runs after the sorting phase. Other default workload and system parameter values are provided in Tables 1 and 2.



(N=4 processors; B=40,000 pages; R=200,000 records; r=512 bytes; f=0.4; λ = 0.015 jobs/sec; T=512 pages)

Figure 3: Performance impact of total number of workfile disks.

5.1 Performance impact of logical partitioning

Fig. 3 shows the average response time of a mergesort for three different numbers of partitions, 1, 2 and 4, for various workfile disks. There are four processors in the system, thus we limit the number of partitions to 4. For the purpose of comparison, we also show the case of fixed assignment with 4 partitions, where each processor uses a fixed partition for the entire simulation. As shown in this figure, the fixed assignment performs the worst compared with all other cases where a mergesort can be dynamically assigned to the least loaded logical partition. This is due to load imbalance in the case of fixed assignment. In general, the performance improves as the number of workfile disks increases. Most importantly, the performance is better for a larger logical partition size (or smaller number of partitions) if the number of workfile disks is large enough. This is indicated in the cases for D = 16, 20 and 24 disks in Fig. 3. However, the reverse is true if the number of workfile disks is small, as indicated for the case of D = 8 disks in Fig. 3. This is due to the fact that I/O interference from different mergesorts can be overcome by the benefits of increased load balance if there is enough workfile disks. But if the number of workfile disks is small, the benefits of load balance may not be significant enough to overcome the disadvantages of increased I/O interference. Thus, for disk-limited cases, it is better to partition the workfile disks into multiple logical partitions.

Note that for a given workload there is a minimum requirement for the total number of workfile disks and for the size of a logical partition. Below such a minimum requirement, the system cannot

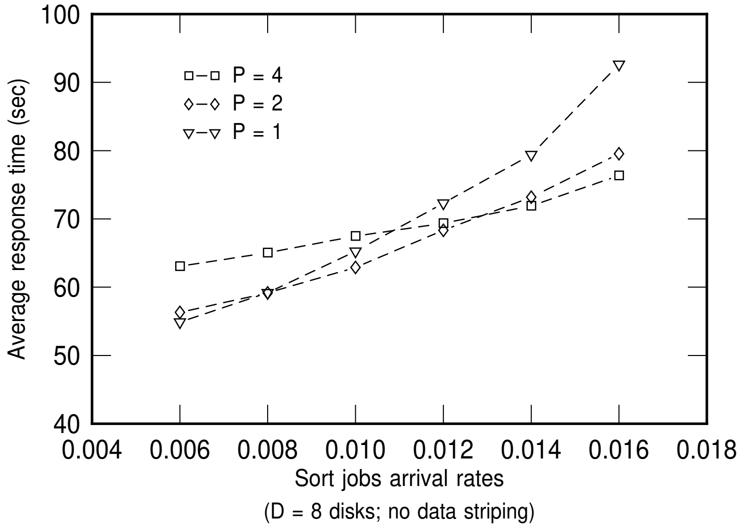


Figure 4: Performance impact of logical partitioning for a disk-limited case.

operate in a normal condition. The cases for only 4 total workfile disks are therefore not shown in Fig. 3. For the rest of this paper, the performance data are obtained under the assumption that the system has reasonably sufficient workfile disks.

To further examine the impact of logical partitioning, we look at various rates of sort job arrivals under two different disk capacities. The first one is with a total of 8 workfile disks and the second one with 16 disks. The 8-disk case represents a disk-limited case while the 16-disk case represents a disk-abundant case. Fig. 4 shows the performance of three different partitioning policies. For the case of $P = 1$, the entire 8 disks is fully shared by all the coming sort jobs. For the case of $P = 4$, the 8 disks are partitioned into 4 groups, each with 2 disks, and an incoming sort job chooses the least loaded group to perform its mergesort. As shown in this figure, different partition sizes behave differently as the arrival rate increases. For the cases where the system workload is very low, the policy of $P = 1$ is the best. As the workload increases, the policy of $P = 2$ becomes better than that of $P = 1$. But, as the system workload continues to increase, the policy of $P = 4$ becomes better than that of $P = 2$. This again demonstrates the trade-off between I/O interference and load balancing.

Fig. 5 shows the performance of the three partitioning policies (i.e., $P = 1, 2$ and 4) for the case of $D = 16$ disks. Similar to Fig. 4, it shows that full sharing is in general better when the system workload is low and partitioning into multiple groups is better when the system workload is high. Note that the performance of $P = 2$ in Fig. 5 is close to the worst of the three cases for the entire range of arrival rates. But, in Fig. 4, it is close to the best of the three cases. Thus, the proper partition size is workload dependent.

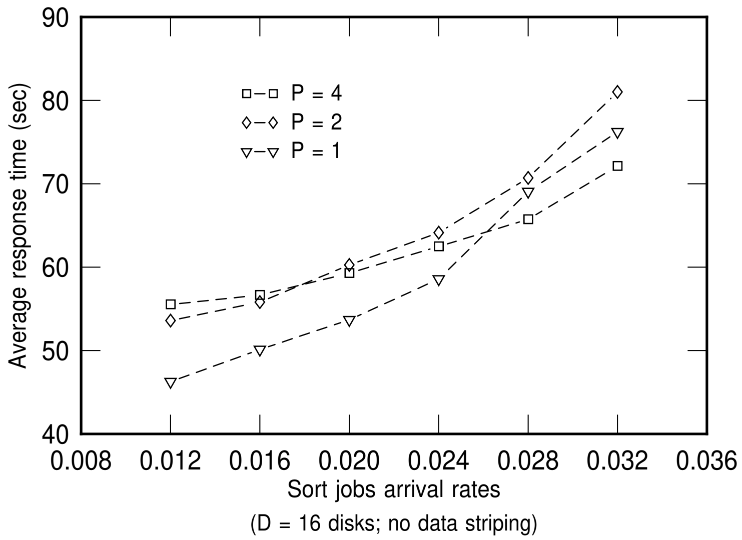


Figure 5: Performance impact of logical partitioning for a disk-abundant case.

From Fig. 4 and 5, it suggests that in order to achieve consistent good response times for concurrent mergesorts, it may be beneficial to implement an adaptive logical partitioning approach, where the partition size changes as the workload changes. A simplified version of this adaptive approach can use only either $P = 1$ or $P = N$ partitions, where N is the total number of processors. When the system workload is low, $P = 1$ is used. But as the system workload reaches a threshold, it can change to $P = N$. Certainly this threshold for changing P is workload dependent.

5.2 Performance impact of data striping

In this section, we examine the impact of data striping on the response times of concurrent mergesorts. Note that data are striped across all the disks in each logical partition. In order to facilitate maximum parallelism for prefetching the input runs during the merge phase, the striped data blocks of various runs are placed in a staggered manner [3]. Fig. 6 shows an example of data placement. For example, run 0 of job J starts from workfile disk 0 and run 1 of job J starts from disk 1. Thus, the second striping block of run $J.0$ is placed on the same disk as the first striping block of run $J.1$. Namely, block $J.0.1$ and block $J.1.0$ are both placed on disk 1.

Depending on the striping unit size (or block size), an I/O request from the sorting job may result in multiple smaller I/O requests to the disks. For instance, if the striping unit size is 4 pages then a prefetch request of 8 pages will generate 2 independent I/O requests each of 4 pages to two different disks. An asynchronous write of 32 pages will generate 8 independent write I/O's of 4 pages each to 8 different disks. So, data striping increases the total number of disk I/O's as compared with no

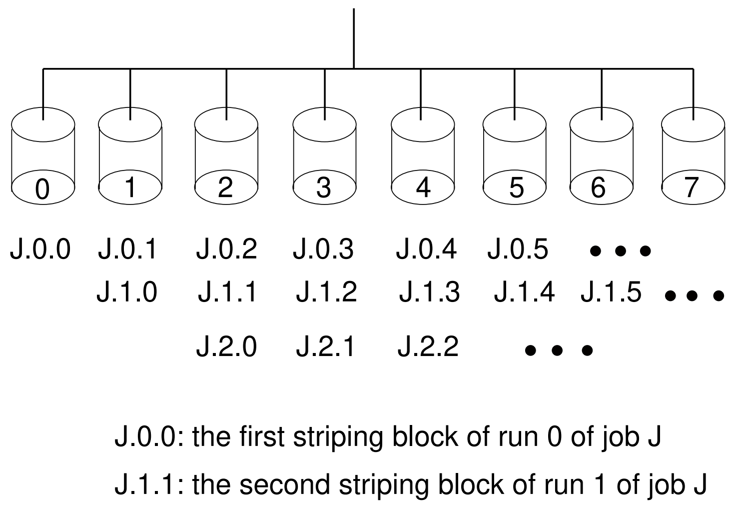


Figure 6: Data placement of runs when data striping is used.

striping. It increases the disk utilization because each I/O, no matter big or small, usually requires a seek time and seek time dominates an I/O response time.

Increasing the number of smaller I/O's certainly has a negative impact on the sort performance because disk utilization is increased. However, if there is enough disks, smaller I/O's can be more evenly spread to the disks, better balancing the workload. More importantly, data striping can also reduce the long blocking time because the batched sequential writes can be spread among multiple disks. Thus, there are benefits to be gained by data striping. To understand the performance impact of data striping, we also studied two cases, one with $D = 8$ and the other with $D = 16$ disks.

Fig. 7 shows the average response times and their corresponding average disk utilizations for various striping sizes, ranging from 8 to 1024 pages, for the case of $D = 8$ disks. Note that for the case of striping size of 1024 pages, there is no striping for the input runs because an input run is 1024 pages long. Nevertheless, the output run is partitioned into multiple of 1024-page segments (i.e., striped every 1024 pages). In general, striping does provide some performance improvement if the striping size is not smaller than 16 pages. However, if the striping size is 8 pages, the performance become significantly worse with striping than without striping. This can be clearly explained by the corresponding disk utilization graphs. Disk utilizations start to increase significantly as the

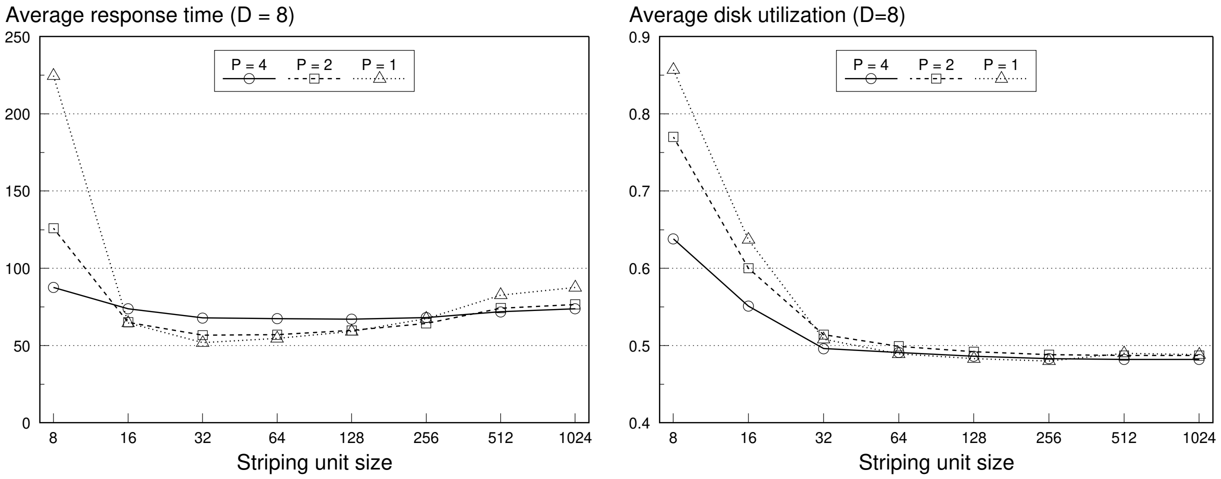


Figure 7: Impact of striping size on response time and disk utilization for D = 8.

striping size becomes smaller than 32. For the case of striping size of 8 pages, the disk utilization goes as high as 85% for the case of $P = 1$. This is because 32 is the deferred write quantity, i.e., an asynchronous write I/O is issued for every 32 dirty pages in a deferred write queue. For a striping unit size smaller than 32, multiple smaller I/O requests are generated for writes and disk utilization starts to increase noticeably.

Fig. 8 shows the similar data for the case of $D = 16$, with the same workload. Compared with Fig. 7, the most significant difference is found in the case of striping size of 8 pages. In Fig. 8, the performance is still better for striping size of 8 pages than for no striping (or striping size of 1024 pages), but it is not the case in Fig. 7. This is due to the fact that there are twice as many workfile disks in Fig. 8 (16 disks) as in Fig. 7 (8 disks) and thus the disk utilization does not goes as high as it does in Fig. 7. However, as the striping size continues to decrease to 4 pages, the performance of data striping becomes significantly worse. With striping size of 4 pages, not only the deferred write requests but also most of the prefetch requests result in multiple smaller I/O requests. Note that the maximum PFQ for the merge phase is 8 pages in our simulations. Thus, a striping size that is less than 8 pages will increase disk utilization dramatically.

From both Fig. 7 and 8, we can also observe that with a proper striping size, such as 32 pages, performance is better if the entire workfile disks are used as a single partition. With a striping size of 32 pages, no additional I/O requests are generated due to data striping in our simulations. Thus, the benefits of load balancing can be achieved without the negative impacts of increased disk utilization. In the no striping case, the I/O interference between a read of one job and writes of another job tends to continue for a long time until all the sequential write requests are completed.

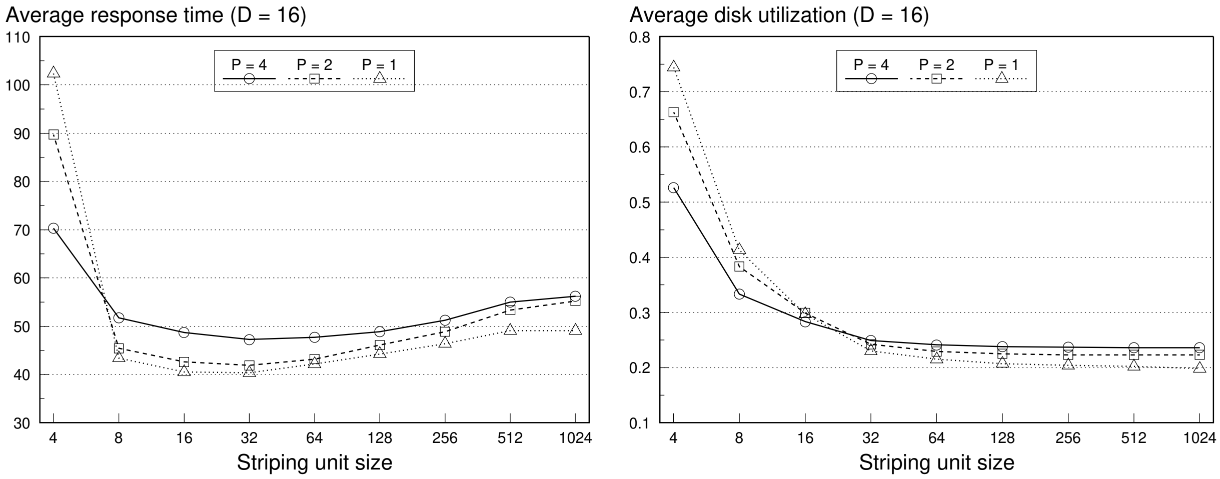


Figure 8: Impact of striping size on response time and disk utilization for D = 16.

On the other hand, for the striping case, the interference between two jobs usually does not persist. Fig. 9 further demonstrates the observation that better performance is achieved by using a single partition with a striping size of 32 pages. It achieves the best performance for the entire range of relation sizes.

6 Summary

In this paper, we examined an approach to logical partitioning of workfile disks for concurrent mergesorts in a multiprocessor database system. A detailed simulator was developed and extensive simulations were conducted to study and compare the performance of concurrent mergesorts for different logical partition sizes with and without data striping. The results show that, without data striping, it is best to use a single partition approach if there are abundant disks, i.e., the system is lightly loaded. However, if the system is disk-limited (i.e., system load is generally high), then it is beneficial to partition the workfile disks into multiple groups and assign a mergesort job to the least loaded group. Moreover, data striping can improve the overall performance only if the striping size is properly chosen. If the striping size is too small, data striping can make the overall performance worse. Finally, with a proper striping size, the best performance is generally achieved by using the entire workfile disks as a single logical partition.

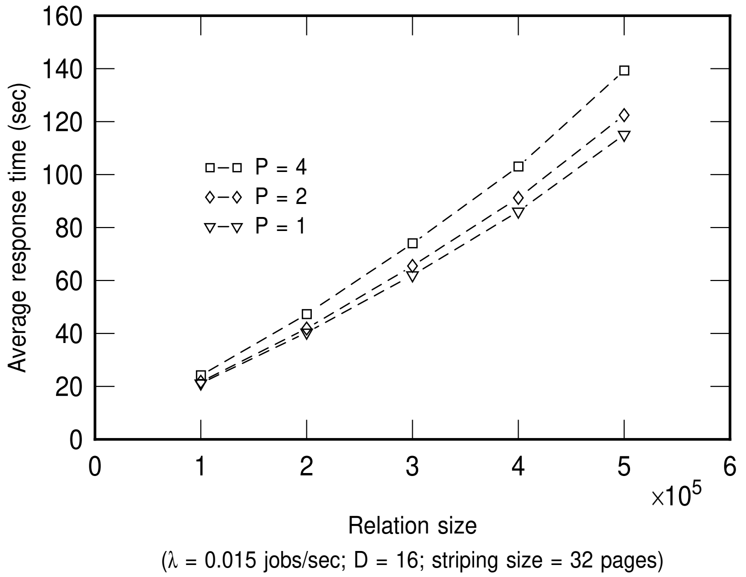


Figure 9: Performance impact of logical partitioning with striping size of 32 pages.

Appendix A: Scheduling merge jobs

A merge job can take multiple steps to complete depending upon the buffer availability. For example, to merge 16 input runs, one can accomplish the task in one step by merging all 16 runs in one step. Alternatively, due to buffer limitation or other constraints, one can first merge 8 of the sorted runs to produce an intermediate output run, and then merge the remaining 8 sorted runs with the intermediate output run. Multiple step merges are undesirable as portions of the data are scanned multiple times. To minimize the response time, both the PFQ and the number of steps determined by the I/O parallelism have to be considered [22]. Although for a single merge sort some optimal trade-off may be devised, in a concurrent mergesort environment further complexity arises from the buffer contention among the multiple merge requests which can be of very different sizes. Note that the PFQ for the merge phase used by the buffer manager in this paper is assumed to be a system wide quantity, similar to IBM's DB2 [20]. It may be adjusted by system load, but cannot be tuned for each merge request separately.

Since it is generally undesirable to complete a merge in multiple steps, in this paper, we assume that each merge job is completed in a single step. For a merge job, a scheduling algorithm (or run allocation algorithm) is first executed to decide whether or not this job can be scheduled for execution. The scheduler maintains the total number of runs, $R_{allocated}$, allocated so far for merge and decides whether or not a merge can be scheduled based on the following criteria. Let P_{max} be the maximum PFQ allowed for the merge phase. We define $round\_2(y, P_{max})$ as a function that rounds y down to the nearest number that is an integer power of 2, but less than or equal to P_{max} if $y \geq 1$; $round\_2(y, P_{max}) = 0$ if $y < 1$. For instance, $round\_2(0,8) = 0$, $round\_2(5,8) = 4$, and $round\_2(15,8) = 8$. If R_i runs are requested by a merge job, then it is schedulable if

$$round\_2(\frac{B}{((2 \times (R_{allocated} + R_i))}, P_{max}) \geq 1, \tag{3}$$

where B is the buffer size. If not schedulable, the merge job simply waits. After a new merge job is scheduled for execution, PFQ for the merge phase may have to be reduced since more runs are allocated. The system-wide PFQ for merge changes using the following formula:

$$\mathrm{PFQ}' = round\_2(\frac{B}{2 \times R'_{allocated}}, P_{max}), \tag{4}$$

where $R'_{allocated}$ is the total number of runs allocated after the merge job is scheduled and PFQ$'$ is the new PFQ. In this paper, P_{max} for merge is assumed to be 8. Based on Equation 4, PFQ for merge can be 1, 2, 4 or 8. (Note that the use of an integer power of 2 as a possible value of PFQ is similar to IBM's DB2 [19].) The schedulability criterion expressed in Equation 3 simply says that so long as the new PFQ is at least 1 then the merge job can be scheduled. Otherwise, it waits.

References

[1] *System/390 MVS Sysplex Overview Introducing Data Sharing and Parallelism in a Sysplex.* IBM Corporation, 1994.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] S. Berson et al. Staggered striping in multimedia informantion systems. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 79–90, 1994.

[4] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. of Very Large Data Bases*, pages 127–141, 1985.

[5] D. J. DeWitt, D. Bitton, H. Boral, and W. K. Wilkinson. Parallel algorithms for relational database operations. *ACM Trans. on Database Systems*, 8(3):324–353, 1983.

[6] C. Faloutsos, R. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. In *Proc. of Very Large Data Bases*, pages 265–274, 1991.

[7] B. R. Iyer and D. M. Dias. System issues in parallel sorting for database systems. In *Proc. of Int. Conf. on Data Engineering*, pages 246–255, 1990.

[8] D. E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching.* Addison-Wesley, 1973.

[9] S. C. Kwan and J. L. Baer. The I/O performance of multiway mergesort and tag sort. *IEEE Trans. on Computers*, 34(4):383–387, 1985.

[10] S. S. Lavenberg, editor. *Computer Performance Modeling Handbook*. Academic Press, 1983.

[11] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 387–396, 1991.

[12] V. S. Pai and P. J. Verman. Prefetching with multiple disks for external mergesort: Simulation and analysis. In *Proc. of Int. Conf. on Data Engineering*, pages 273–282, 1992.

[13] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 109–116, 1988.

[14] A. L. N. Reddy and P. Banerjee. An evaluation of multiple-disk I/O systems. *IEEE Trans. on Computers*, 38(12):1680–1690, Dec. 1989.

[15] D. Reiner, J. Miller, and D. Wheat. The Kendall Square query decomposer. In *Proc. of 2nd Parallel and Distributed Information Systems Conference*, 1993.

[16] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. on Database Systems*, 11(4):473–498, 1986.

[17] K. Salem and H. Garcia-Molina. Disk striping. In *Proc. of Int. Conf. on Data Engineering*, pages 336–342, 1986.

[18] B. Salzberg. Merging sorted runs using large main memory. *Acta Informatica*, 27:195–215, 1989.

[19] J. Z. Teng. DB2 buffer pool management. Lecture notes in Dallas DB2 Users Group meeting, Jan. 1992.

[20] J. Z. Teng and R. A. Gumaer. Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.

[21] J. L. Wolf. The placement optimization program: A practical solution to the disk file assignment problem. In *Proc. of 1989 ACM SIGMETRICS*, pages 1–10, 1989.

[22] K.-L. Wu, P. S. Yu, and J. Z. Teng. Performance comparison of thrashing control policies for concurrent mergesorts with parallel prefetching. In *Proc. of 1993 ACM SIGMETRICS*, pages 171–182, 1993.

[23] K.-L. Wu, P. S. Yu, and J. Z. Teng. Data placement and buffer management for concurrent mergesorts with parallel prefetching. In *Proc. of Int. Conf. on Data Engineering*, pages 418–427, 1994.

[24] P. S. Yu and D. W. Cornell. Buffer management based on return on consumption in a mulitquery environment. *VLDB Journal*, 2(1), Jan 1993.