A Study of Sort Algorithms for Multiprocessor Database Machines

Jai Menon

IBM Almaden Research Center
San Jose, California 95120-6099

Abstract

This paper presents and analyzes algorithms for parallel execution of sort operations in a general multiprocessor architecture. We consider both internal and external sorting algorithms. For the latter, we study the performance of sorting algorithms that are derived from sorting algorithms that only do comparison and exchange by replacing each comparison-exchange with a B-way merge. In particular, we propose a new algorithm called the modified block bitonic sort. We then present the results of analyzing the performance of these different parallel external sorting algorithms. We show that the modified block bitonic sort is the fastest of the algorithms over a wide range of values of interest to us.

Introduction

Several multiprocessor database machines [GARDA81] [BABBE79] [DEWIT79]. have been or are currently being developed. Two of the most demanding operations that must be performed by such multiprocessor database machines are sorting and join. This paper presents a study of various algorithms for performing the first of these two operations on a general model of a multiprocessor database machine.

We will begin by considering ways to use parallel processors to sort files stored in random access memory (***parallel internal sorting***). In particular, we will show how to use the bitonic merge in order to do parallel internal sorting ([HSIAO80], [BITTO84]). We have presented the ***bitonic merge*** principle before, but the particular algorithm described here, and its analysis are presented for the first time.

Due to memory limitations, sorting of large files cannot be done in memory, and ***external sorting algorithms*** need to be used. The study of the external sorting algorithms is the main focus of this paper.

For external sorting algorithms, we study the performance of sorting algorithms that are derived from sorting networks that only do comparison and exchange by replacing each comparison-exchange with a B-way merge. We are interested in this class of algorithms because of the result of [BITTO83] where a sorting algorithm that is derived by replacing each comparison-exchange with a 2-way merge was presented and shown to be superior to all other algorithms presented in that paper. In this study, we are interested in the use of B-way merges, where B is significantly larger than 2. This is important because of the continuing drop in the costs of semiconductor memory, making it feasible to build multiprocessors with large amounts of semiconductor memory. For this class of external sorting algorithms, we investigate the impact of larger amounts of memory. While [BRATS84] analyzed the impact of large amounts of main memory on uniprocessor sorting algorithms, we believe our work is the first such investigation for multiprocessors.

For the class of external sorting algorithms that are derived as described above, we also consider several general techniques for further improving their performance. We feel that there are three techniques that have general applicability. We give examples of use of two of these three techniques to improve the performance of algorithms in the class of interest to us. The two techniques we consider are the use of ***pipelining*** and the use of ***parallel internal sorting***. The application of these techniques leads us to the discovery of an algorithm we call the ***modified block bitonic sort***.

We then present the results of analyzing the performance of these different parallel external sorting algorithms. We show that the modified block bitonic sort is the fastest of the algorithms over a wide range of values of interest to us and that it makes the best use of additional main memory buffer space.

The Architectural Model

In this paper, we are concerned with the parallel execution of sorting algorithms on multiprocessor database machines that do not have any special-purpose hardware for execution of the sorting operation [BITTO83] [VALDU84]. Such machines will have several general-purpose processors linked through a contention-free interconnection network of some sort. Each processor will have its own local memory, and all the processors also share some amount of global memory. The processors exchange data via this shared global memory which may be accessed simultaneously by several processors.

The database machines also use conventional disk drives for secondary storage. Relations (files) to be sorted are stored on these disk drives as fixed-size pages. The shared global memory is assumed to be the cache for accesses to secondary store. Then, any page stored in secondary store may be transferred and stored in any page frame in the cache. The local memory of the processors is also assumed to be page-oriented.

In general, our algorithms assume that each processor has B pages of memory associated with it. We may think of this as B pages of local memory, making a total of BP pages of local memory spread across the P processors, or we may think of it as B pages of global memory, making a total of BP pages of global memory shared by the P processors.

The general organization of our multiprocessor database machine is shown in Figure 1. The local memories of each processor are not shown in the figure.

Parallel Internal Sorting Algorithms Using Bitonic Merge

Most of the work on sorting using parallel processors [VALIA75] [PREPA78] [MULLE75] [HIRSC78] [THOMP77] [NASSI78] assume that P processors will be used to sort P records. We are more interested in considering methods which can use P processors to sort MP records, where M, which is very large, is the number of records that will fit in the local memory of each one of the P processors (alternatively, the space for MP records may be in the shared global memory).

[BAUDE78] was the first paper to consider this problem and present algorithms for sorting MP records using P processors. The class of algorithms presented in their paper was obtained by replacing every comparison-exchange step (in a sorting algorithm consisting of comparison-exchange steps) by a two-way merge. The merged sequence is split two ways, with the "lower" half sent to one destination processor, and the "upper" half sent to another destination processor.

The problem with the approach taken by [BAUDE78], is that their algorithms require each processor to have 4M memory. Thus, in order to sort MP records using P processors, they use 4MP memory. We present, below, a class of algorithms that can sort MP records using P processors, with (M+1)P memory [HSIAO80].

Our class of algorithms is also obtained from sorting algorithms that do comparison-exchanges. However, rather than replace each comparison-exchange with a two-way merge, we propose that we replace each comparison-exchange with a ***bitonic merge***. We have presented the ***bitonic merge*** principle before [BITTO84], but the particular algorithm described here, and its analysis are presented here for the first time.

An example of a bitonic merge is shown in Figure 2, where we show 2 processors, each with enough local memory to hold five records. The smallest record in P1 is compared with the largest record in P2, the smaller of the two is placed in P1's memory, the larger of the two is placed in P2's memory. Next, the second smallest record in P1's memory is compared with the second largest record in P2's memory. Once again, the smaller record is placed in P1's memory, and the larger record is placed in P2's memory. The process continues until no more exchanging is needed. At the end of these exchanges, the smallest 5 records are in P1 and the largest 5 records are in P2. The bitonic merge is complete, if P1 does a local sort of its memory and P2 does a local sort of its memory in parallel. The fact that the smallest records will be in P1 was proved by Alekseyev [KNUTH73]. He also showed that at the end of the exchanging, the M smallest records in P1 and the M largest records in P2 each form a bitonic sequence (a bitonic sequence is the concatenation of two sorted sequences, one sorted in ascending order, and one sorted in descending order). Clearly, such a bitonic sequence may be sorted by merging the two sorted subsequences from opposite ends.

Several reasons make the bitonic merge superior to the two-way merge of [BAUDE78]. First, we only require that each processor have enough memory to hold M+1 records, whereas the two-way merge requires each processor to have enough memory to hold 4M records. Second, the bitonic merge is more suitable for implementation on parallel computers that require a high degree of synchronization between their processors. Third, the two-way merge requires an entire block of data to be transferred to a processor's memory before the merge operation is initiated, whereas the bitonic merge only requires the first record to be transferred before the merge operation is initiated.

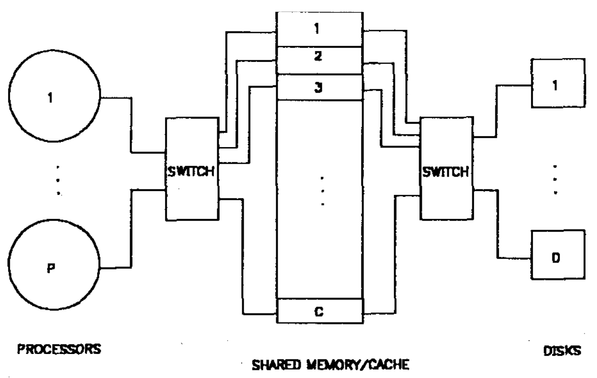

Figure 1: General Organization of a Multiprocessor Database Machine

Parallel Internal Shuffle Sort

A fast parallel internal sort can be derived from Stone's algorithm [STONE71] to sort P elements using P processors in $\log_2 P$ steps, where P must be a power of two. To describe the algorithm, we use the following notation. Let EXCHANGE(i,j) represent the procedure adopted to exchange records between processor i and processor j, so that the smallest M records are in processor i's memory and the largest M records are in processor j's memory. Also, let us give each processor a binary index - with four processors, processor 0 is '00', processor 1 is '01', ... , processor three is '11'. Then, we define the ***shuffle processor*** for processor k as processor l, if l is k left-circularly shifted. Finally, we say that during a ***perfect shuffle***, each processor sends the records in its memory to the memory of its shuffle processor.

The parallel internal shuffle sort is described below for the case of P=4. In general, the algorithm consists of $\log_2 P$ stages, and each stage has $\log_2 P$ steps.

- STAGE I
  1. Perform the perfect shuffle.
  2. Perform the perfect shuffle. EXCHANGE(0,1),EXCHANGE(3,2) in parallel.
- STAGE II
  1. Perform the perfect shuffle. EXCHANGE(0,1),EXCHANGE(2,3) in parallel.
  2. Perform the perfect shuffle. EXCHANGE(0,1),EXCHANGE(3,2) in parallel.
- FINAL STEP - Do localized sorts.

Analysis of Parallel Internal Shuffle Sort

Let us use the following notation for the analysis.

P	Number of processors
M	Number of records per processor

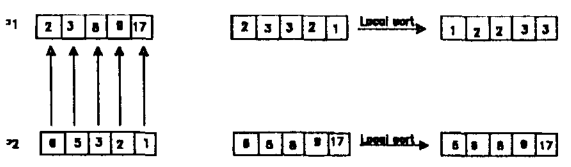

Figure 2: An Example of the Bitonic Merge

B	Number of pages per processor local memory
k	Number of records per page, M=Bk
C	Time to do a compare of two keys
V	Time to move a record in memory (or time for a complex move)

We note that the algorithm consists of one exchange step in the first stage, two exchange steps in the second stage, etc., and log P exchange steps in the final stage. Thus, there are a total of $(\frac{1}{2})(\log P)(1+\log P)$ exchange steps. We also note that there is one final localized sort step.

For our analysis, we will assume that the k tuples inside a page are in sorted order to begin with. Then, for the first exchange, each local processor may sort the Bk records into sorted order by performing a B-way merge of the B sorted pages. The time for the B-way merge is $BkV + Bk(\log_2 B)C$. In order to complete the first exchange, processors must compare and move Bk corresponding records requiring Bk(C+V) time. Therefore, the total time for the first exchange is $BkV + Bk(\log_2 B)C + Bk(C+V)$. For each of the remaining

$$(\frac{\log_2 P}{2} + \frac{\log_2^2 P}{2} - 1)$$

exchanges, the sort step is simpler, since the sequence to be sorted is bitonic and may be sorted by merging from the two ends. The time for each of these exchanges is 2Bk(C+V). The time for the final sort step is also Bk(C+V). So, the total time for the execution of the algorithm is

$$Bk(C+V)\left[\log_2 P + \log_2^2 P\right] + BkV + Bk(\log_2 B)C$$

Parallel External Sorting Algorithms

Let us now turn our attention to parallel external sorting algorithms. These are algorithms which use P processors, each with B pages of memory (and an additional page for holding output tuples), to sort N pages, where N is much greater than B (not necessarily much greater than BP). Just as parallel internal sorting algorithms can be derived from sorting algorithms that only do compare and exchange, so also can parallel external algorithms. This fact was pointed out in [BITTO83]. In that paper, a parallel external sorting algorithm called a ***block bitonic sort*** is derived from Batcher's bitonic sort [BATCH68] by replacing each comparison-exchange with a two-way external merge of two runs of size $\frac{N}{2P}$.

Using this same idea for generating parallel external algorithms, we first present an external sorting algorithm based on the odd-even transposition sort [KNUTH73]. For the odd-even external sort, we will show how to use pipelining to arrive at a ***pipelined odd-even external sort*** which is superior in performance to the odd-even external sort. We will then examine the block bitonic sort ([BITTO83]). Using the technique of pipelining on the block bitonic sort, we will derive a ***pipelined block bitonic sort*** which, unfortunately, has inferior performance to the block bitonic sort. From that, we will draw some conclusions about the efficacy of pipelining as a general technique for performance enhancement.

In order to improve the performance of the block bitonic sort, we will the examine the idea of using parallel internal sorts.

The Parallel Odd-Even External Sort

Execution of the odd-even external sort for two processors (P = 2) and eight pages (N = 8) is illustrated in Figure 3. The class of algorithms suggested in [BITTO83] can process at most 2P runs with P processors. Therefore, a preprocessing stage is necessary when the number of pages to be sorted exceeds 2P. The function of this preprocessing stage is to produce 2P sorted runs of size $\frac{N}{2P}$ each. Since our external odd-even sort is an algorithm in the class of algorithms suggested in [BITTO83], it will also have a preprocessing stage. In our example, the preprocessing stage will produce four sorted runs of two pages each. Following this preprocessing stage, the odd-even external sort will have 2P more stages (this follows from the odd-even transposition sort), in each stage of which, the processors, in parallel, merge two runs of size $\frac{N}{2P}$.

Analysis of Parallel Odd-Even External Sort

We use the following notation, in addition to those developed for the analysis of the parallel internal sorting algorithms.

Cr	Time to read an external page
Cw	Time to write an external page
Cm	Time to merge two pages = 2k(C+V)
C_P^2	Time to read, merge and write two pages. This is equal to $2Cr + 2Cw + 2k(C+V)$.
C_P^B	Time to read, merge and write B pages. This is equal to $BCr + BCw + BkV + B(\log_2 B)kC$.

As described and analyzed in [BITTO83], the preprocessing stage consists of processors, in parallel, successively merging longer and longer pairs of runs, until the number of runs is twice the number of processors. It is the job of each processor to produce two runs of size $\frac{N}{2P}$. This will take

$$(\frac{N}{2P})\log_2(\frac{N}{2P})C_P^2$$

Then, each of the 2P steps of the odd-even external sort requires $(\frac{N}{2P})C_P^2$ steps. Therefore, the total time for the odd-even external sort is

$$((\frac{N}{2P})\log_2(\frac{N}{2P}) + N)C_P^2$$

Parallel B-ary Odd-Even External Sort

We now consider the following refinement. Until now, we had assumed that each processor had enough memory to hold 3 pages, where one page was for output, and the other two pages was to hold the input during a two-way merge. Now, let us assume that

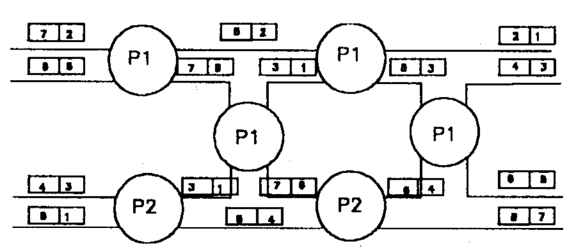

Figure 3: Illustration of the External Odd-Even Sort

each of the processors has more than 3 pages of buffer. Let each processor have B+1 pages of buffer, so that it may do a B-way merge, rather than a two-way merge. In other words, we are proposing a class of external parallel sorting algorithms that are derived by replacement of each comparison-exchange with a B-way external merge of B runs of size $\frac{N}{BP}$.

The preprocessing stage must now produce BP runs of size $\frac{N}{BP}$. Then, each of the 2P steps of the odd-even external sort requires $(\frac{N}{BP})C_P^B$ steps. Therefore, the total time for the odd-even external sort is

$$((\frac{N}{BP})\log_B(\frac{N}{BP}) + (\frac{2N}{B}))C_P^B$$

Consider the following example.

Example 1. Let N = 4096, P = 16, Cr = 6.4 msecs, Cw = 14.4 msecs, k = 40, C = .01 msecs, V = .20 msecs. Then, C_P^2 is 58.4 msecs and C_P^4 is 118.4 msecs. With these values, the time for the two-way odd-even sort is 291.5 seconds and the time for the four-way odd-even sort is 265.2 seconds. Therefore, it is attractive to do a four-way odd-even sort.

The values chosen for Cr, Cw, C and V above are those used in [BITTO83]. In general, increasing B helps to a point. Beyond that critical point, increasing B will actually hurt the performance of the odd-even external sort. The faster the CPU, and the slower the mass storage devices used, the higher the optimal value of B.

Pipelined B-ary Odd-Even External Sort

We wish to reduce the time taken to execute the 2P merge steps in the odd-even sort that follow the preprocessing stage. As it currently stands, the second merge step cannot be executed until the first step completes, the third step cannot be completed until the second step completes, and so on. However, if we had several more processors, then we could assign these extra processors to execute all the 2P steps in a pipelined fashion. Looking back at Figure 3, we see that two processors are used in the first step, one processor is used in the second step, two processors are used in the third step and one processor is used in the final step. Therefore, if we had six processors to do the odd-even sort, we could ***pipeline*** between the stages. As soon as the first pages of all the input runs to step 2 were available, step 2 would be started. Then, as soon as the first pages of all the input runs to step 3 were available, it would be started, and so on. This would speed up the algorithm, at the cost of additional processors.

Let us illustrate the difference between the odd-even sort and the pipelined odd-even sort by means of an example. Let P = 6 and B = 2. In the odd-even sort, we will have a preprocessing stage in which we will create 2P or 12 runs of size $\frac{N}{12}$. Then, we will execute 2P or 12 merge steps in a non-pipelined fashion. In the pipelined odd-even sort, we will have a preprocessing stage in which we will create four runs of size $\frac{N}{4}$. We will then organize the six processors in four steps (two in step 1, one in step 2, two in step 3, one in step 4). Then, we will execute four merge steps in a pipelined fashion.

Clearly, the pipelined odd-even sort takes longer during the preprocessing stage, because it needs to create longer runs. However, it makes up for the longer preprocessing stage by virtue of the fact that it only needs fewer merge steps and because these fewer merge steps can be executed in a pipelined fashion.

Analysis of Pipelined Odd-even Sort

We will do the analysis for B = 2. It is easy to show that with P processors, the pipelined odd-even sort executes the same number of merge steps as a normal odd-even sort with $K = (\frac{1+\sqrt{(1+8P)}}{4})$ processors. Thus, for example, with P = 6, the pipelined odd-even sort has the same number of merge steps as a normal odd-even sort with K = 2 processors.

In the first step of the pipelined odd-even sort, we will need to create 2K runs of size $\frac{N}{2K}$, using 2K out of the P processors. It is easy to see that $P \geq 2K$ as long as $P \geq 3$. So, if we only consider P to be three or greater, we can do this first step in

$$(\frac{N}{4K})\log_2(\frac{N}{2K})C_P^2$$

Now, we need to wait until the first pages reach the last step (there are 2K steps) of merging. This takes $(2K-1)C_P^2$ time units. Finally, in the last step, all processors will merge two runs of size $\frac{N}{2K}$, in $(\frac{N}{2K})C_P^2$ steps.

Consider the following example.

Example 2. Let N = 24, P = 6, K = 2, and Cr, Cw, k, C and V are as in example 1. Then, the time for the odd-even sort is 26 time units, whereas the time for the pipelined odd-even sort is 3 log 6 + 9, which is 16.75 time units, and hence, better.

External Block Bitonic Sort

Next, let us consider the Block Bitonic Sort which had been described in [BITTO83]. It is derived from Batcher's bitonic sorter in the same manner as we derived the odd-even external sort from the odd-even transposition sort. The algorithm is illustrated in Figure 4, for P = 2 and N = 8. The preprocessing step is identical to that for the odd-even external sort. However, instead of 2P merge steps, the algorithm only needs $(\frac{1}{2})(\log_2 2P)(1+\log_2 2P)$ merge steps, so that its total execution time, as analyzed in [BITTO83] is

$$\left[\log_2 N + \frac{\log_2^2 2P}{2} - \frac{\log_2 2P}{2}\right](\frac{N}{2P})C_P^2$$

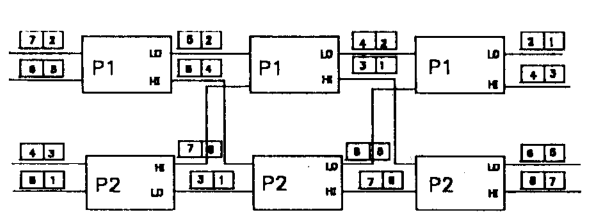

Figure 4: Illustration of the Block Bitonic Sort ([BITTO83])

We begin by improving the performance of this algorithm using B buffers, so that the total execution time now is

$$\left[\log_B\left(\frac{N}{BP}\right) + \frac{\log_2^2 2P}{2} + \frac{\log_2 2P}{2}\right]\left(\frac{N}{BP}\right)C_P^B$$

Consider the following example.

Example 3. Let N = 512, P = 8, Cr, Cw, k, C and V as in Example 1. Then, the time for the block bitonic sort with B = 2 is 28.8 seconds, whereas the time with B = 8 is 21.12 seconds.

Improving the Performance of the Block Bitonic Sort

Once again, our first avenue of exploration in the search for a better sorting algorithm is to consider the idea of pipelining. We will do the analysis for the case of B=2. The pipelined block bitonic sort will have a preprocessing stage in which 2K runs of size $\frac{N}{2K}$ are created. Subsequently, the pipelined block bitonic sort will do the merge steps in a pipelined fashion, using K processors at each stage of the pipeline. It is not difficult to see that K can be calculated by solving the equation

$$\left(\frac{K}{2}\right)(\log_2 2K)(1 + \log_2 2K) = P$$

To take an example, consider P=6, K=2. In this example, processors 3 and 4 will wait a single time unit (C_P^2) before they begin merging, and processors 5 and 6 will wait two time units before they begin merging. In general, the last processors must wait $(\frac{1}{2})(\log_2 2K)(1 + \log_2 2K) - 1$ time units before beginning merging.

To complete the analysis, let us assume that we are interested only in cases where P, the number of processors, is 6 or greater. Then, it is easy to see, from the formula for K above, that 2K is less than or equal to P. In the first step of the pipelined block bitonic sort, we will need to create 2K runs of size $\frac{N}{2K}$, using 2K out of the P processors. This will take

$$\left(\frac{N}{4K}\right)\log_2\left(\frac{N}{2K}\right)C_P^2$$

Then, we must wait

$$\left(\left(\frac{1}{2}\right)(\log_2 2K)(1 + \log_2 2K) - 1\right)C_P^2$$

time units until the last processors get pages to start merging. Finally, in the last step, all processors will merge two runs of size $\frac{N}{2K}$, in $(\frac{N}{2K})C_P^2$ steps.

Consider the following example.

Example 4. Let N = 1920, P = 80, K = 8, and Cr, Cw, K, C and V as in example 1. Then, the time for the block bitonic sort is 444 time units, whereas the time for the pipelined block bitonic sort is 609 time units.

We see that the pipelined block bitonic sort is worse than the block bitonic sort. Pipelining, as a general technique, is clearly not always rewarding - it helped the odd-even sort, but not the block bitonic sort. We have explored the idea of pipelining on other algorithms and have come to the conclusion that it is only useful when the number of merge steps in the original algorithm is high.

There are two other possible avenues of attack that may now be explored in order to improve the performance of the Block bitonic sort further. We may either improve the performance of the preprocessing stage without changing the number of merge steps needed, or we may try to decrease the number of merge steps needed without hurting the performance of the preprocessing stage. In the following sections, we will discuss the former. Details of the latter technique may be found in [MENON86].

Improving the Performance of the Preprocessing Stage

In order to improve the performance of the preprocessing stage, we suggest the use of the parallel internal sort that we developed in the section titled "Parallel Internal Shuffle Sort". Consider the case when N is very large and when $\frac{N}{BP}$ is equal to BP. Then, we may make one pass over the file, bring in BP records at a time, and sort these BP records using the parallel internal sort developed previously. After BP such internal sorts, we have completed the preprocessing stage and created BP runs of size $\frac{N}{BP}$.

Consider the following example.

Example 5. Let N = 4096, P = 16, B = 4, k = 40, Cr = 17 msecs, Cw = 17 msecs, C = 0.01 msecs and V = 0.1 msecs. In this case BP is equal to $\frac{N}{BP}$. Using the method of iterated merging, the preprocessing stage will take 35.33 seconds. This consists of approximately 26.12 seconds of I/O time and 9.21 seconds of CPU time. On the other hand, if we use the parallel internal sorts, we can accomplish the preprocessing in 34.3 seconds (25.6 CPU and 8.7 I/O). In other words, the parallel internal sort can be used to improve the performance of the preprocessing stage. The improvement is achieved by trading off CPU time for I/O time.

If the CPU is any faster or the disks are any slower than in the example, the use of the internal parallel sort for the preprocessing stage is recommended. On the other hand, if the CPU is any slower, or the disks are any faster, the method of iterated merging remains superior.

We conclude that it is possible to improve the performance of the preprocessing stage by using parallel internal sorts rather than iterated merging, as long as our processors are fairly powerful. The example shown above considered the case when the file to be sorted was larger than BP. We may also consider the case when the file to be sorted is smaller than BP. In that case, the modified bitonic sort would simply read the file into local memory, do a parallel internal sort, and then write the sorted file back to disk. On the other hand, the unmodified block bitonic sort would need to perform $(\frac{1}{2})(\log_2 2P)(1+\log_2 2P)$ merge steps.

Consider the following example.

Example 6. Let N = 64, P = 16, B = 4, k = 40, and Cr, Cw, k, C and V as in Example 1. In this case, BP is equal to N. Using the unmodified block bitonic sort, requires 7.6032 seconds for the execution to complete. Using the modified block bitonic sort, we would need only .52 seconds. Therefore, the modified block bitonic sort can be an order of magnitude faster than the unmodified block bitonic sort. We conclude that there are situations where, even with a slow CPU and fast disks, the modified bitonic sort is still the preferred algorithm.

Analyzing the Resultant Modified Block Bitonic Sort

The algorithm consists of a preprocessing stage in which we create BP runs of size $\frac{N}{BP}$. This is followed by the merge steps which we have analyzed before. We need to analyze the time to execute the preprocessing stage, since we have modified the technique for preprocessing. We recall that the purpose of the preprocessing stage is to create BP runs of size $\frac{N}{BP}$. Let us consider three cases, and analyze the preprocessing time for each of the three cases.

The first case is when N (the number of elements to be sorted) is less than BP (the number of elements that can be sorted internally). Using the formula developed in "Parallel Internal Shuffle Sort", it is easy to see that the internal sorting time is:

$$(\frac{N}{P})k(C+V)\left[\log_2 P+\log_2^2 P\right]+(\frac{N}{P})kV+(\frac{N}{P})k(\log_2 B)C$$

In addition, we must add the time to read and write the N pages to be sorted (each processor reads and writes $\frac{N}{P}$ pages) which is $(\frac{N}{P})(Cr+Cw)$. So, the total time for the preprocessing stage for the case when N is less than BP is

$$(\frac{N}{P})k(C+V)\left[\log_2 P+\log_2^2 P\right]+$$

$$(\frac{N}{P})kV+(\frac{N}{P})k(\log_2 B)C+(\frac{N}{P})(Cr+Cw)$$

In this case, there is no need for a subsequent merge stage. Therefore, the total time for sorting is as above.

Next, let us consider the case when N is greater than BP, but it is less than or equal to B^2P^2. In this case, the preprocessing stage will consist of BP passes, and each pass will consist of a parallel internal sort of $\frac{N}{BP}$ elements. At the end of these BP passes, we will have the required BP runs of size $\frac{N}{BP}$. Since N is less than or equal to B^2P^2, so $\frac{N}{BP}$ is less than or equal to BP. Assume $\frac{N}{BP}$ is equal to XP. Then, the time for each pass is

$$Xk(C+V)\left[\log_2 P+\log_2^2 P\right]+$$

$$XkV+Xk(\log_2 B)C+X(Cr+Cw)$$

The total sort has BP passes and must include the time for merging and is

$$(\frac{N}{P})k(C+V)\left[\log_2 P+\log_2^2 P\right]+$$

$$(\frac{N}{P})kV+(\frac{N}{P})k(\log_2 B)C+(\frac{N}{P})(Cr+Cw)$$

$$+(\frac{1}{2})(\log_2 2P)(1+\log_2 2P)C_P^B$$

Finally, we consider the case when N is greater than B^2P^2 or $\frac{N}{BP}$ is greater than BP. In this case, we will begin by creating $\frac{N}{BP}$ runs of size BP each. This is done by making $\frac{N}{BP}$ internal sorts, each internal sort creating runs of size BP. Now, each processor can take runs of size BP and create runs of size B^2P, then take runs of size B^2P and create runs of size B^3P, and so on until runs of size $\frac{N}{BP}$ are created. It can be shown that the total time, including merging is

$$(\frac{N}{P})k(C+V)\left[\log_2 P+\log_2^2 P\right]+$$

$$(\frac{N}{P})kV+(\frac{N}{P})k(\log_2 B)C+(\frac{N}{P})(Cr+Cw)$$

$$+(\log_B(\frac{(\frac{N}{BP})}{BP}))(\frac{N}{BP})(C_B^P)+(\frac{1}{2})(\log_2 2P)(1+\log_2 2P)C_P^B$$

Comparative Analysis of External Sorting Algorithms

In this section, we present the results of a comparative analysis of the different algorithms that we have presented. The numerical results for the execution times of the different sorting algorithms are obtained from APL programs that were written to calculate them based on the equations derived in the previous sections. For all the results presented in the following sections, we assumed that Cr and Cw, the times to read and write a page from the disk was 17 msecs, that k the number of tuples (records) per page was 40, that C, the time to compare two keys was .01 msecs, and that V, the time to move a record in memory was .1 msecs.

First, we compare the pipelined odd-even sort, the block bitonic sort, and the modified block bitonic sort. Since the pipelined odd-even sort was only analyzed with B the number of buffers equal to two, we only present results for B=2.

Comparing the Pipelined Odd-Even Sort With the Bitonic Sorts

The results of our comparison are shown in Figures 5, 6, 7, and 8. In these graphs, 1 is the block bitonic sort, 2 is the modified block bitonic sort and 3 is the pipelined odd-even sort.

From the first two graphs, we see that the sorting time for all three methods increases with the size of the relation being plotted. These graphs also tell us that the pipelined odd-even sort is inferior to the two block bitonic sorts except for very low number of processors when all the three methods are approximately equivalent in performance.

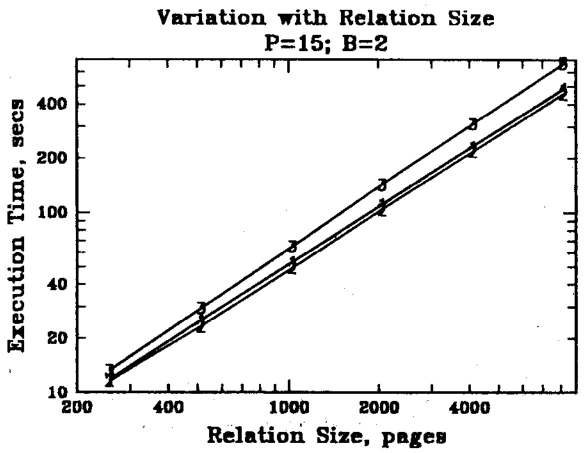

Figure 5: Execution Time Versus Relation Size for Sorting Methods

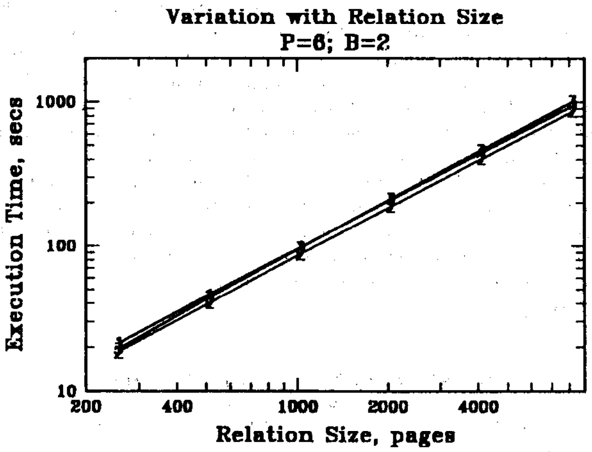

Figure 6: Execution Time Versus Relation Size for Sorting Methods

Looking at the latter two graphs, we see that all the sorting algorithms improve in performance when they can use more processors. Once again, it is clear that the pipelined odd-even sort is inferior to the other two sorting methods, except for very low number of processors when it actually outperforms the block bitonic sort.

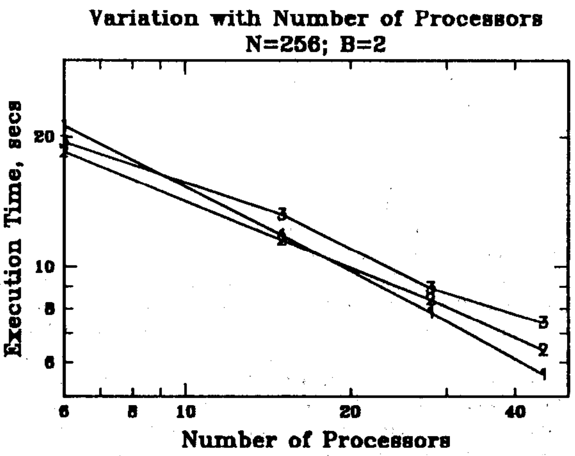

Figure 7: Execution Time Versus Number of Processors

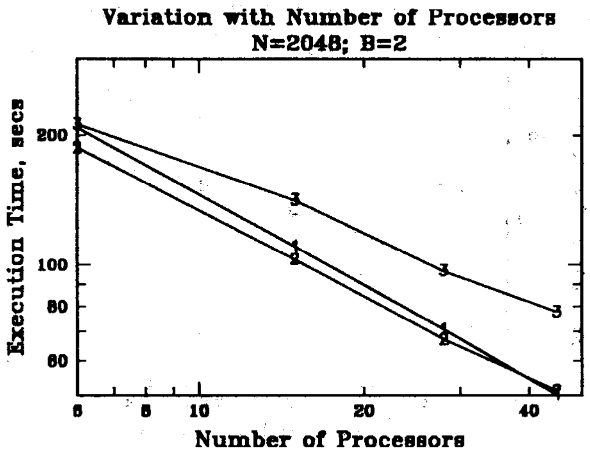

Figure 8: Execution Time Versus Number of Processors

For the chosen values of the parameters, there does not appear to be much difference in performance between the block bitonic sort and the modified block bitonic sort. The modified block bitonic sort is slightly better for small number of processors (less than 20 when the relation size is small (256 pages) and less than 40 when the relation size is large (2048 pages). It also appears better when the relation sizes to be sorted are large. The block bitonic sort appears to be slightly better for large number of processors and small relation sizes.

Comparing the Odd-Even Sort With the Block Bitonic Sorts

In this section, we show the odd-even sort, rather than the pipelined odd-even sort, for a couple of reasons. First, we were able to analyze the pipelined odd-even sort only for B=2, whereas the other sorting methods were analyzed for all values of B. Secondly, we found that while the pipelined odd-even sort was better than the odd-even sort, when compared against the performance of the block bitonic sorts, it looked as bad as the odd-even sort.

In Figures 9, 10 and 11, we show the variation in execution time with the size of the relation being sorted. In these graphs, 1 is the block bitonic sort, 2 is the odd-even sort, and 3 is the modified block bitonic sort. The odd-even sort is clearly the most inferior of the three algorithms. The relative performance of the bitonic sorts is more interesting. In Figure 9, we see that the modified block bitonic sort outperforms the block bitonic sort when the size of the relation to be sorted is less than about 1000 pages, and in Figure 10, we see that the modified block bitonic sort outperforms the block bitonic sort when the size of the relation to be sorted is less than about 8000 pages. For other values of the relation size, the two methods are approximately equal in performance. In general, it is seen that the modified block bitonic sort outperforms the other methods when the size of the relation to be sorted is smaller than, equal to, or slightly greater than the product of the number of processors P and the number of buffers B. For other values of the various parameters, the performance of the modified block bitonic sort is almost equal to that of the unmodified block bitonic sort. In summary, the modified block bitonic sort is superior for very large values of P and B, or very small values of N. From the previous section, we are also aware that the modified block bitonic sort is superior for very large values of N, and very small values of P.

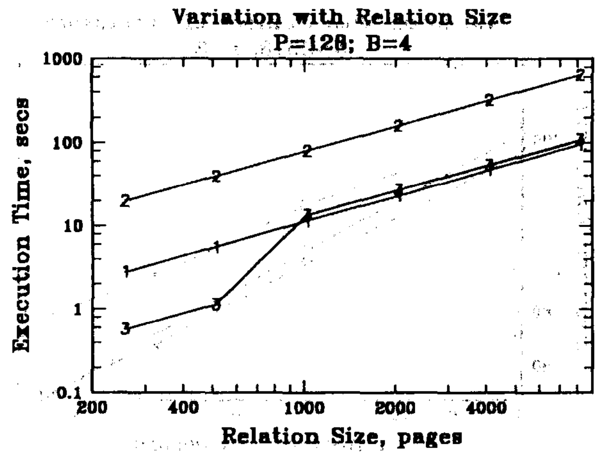

Figure 9: Execution Time Versus Relation Size for Sorting Methods

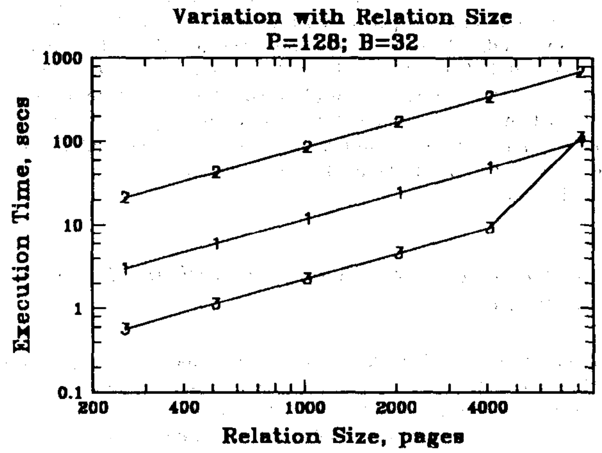

Figure 10: Execution Time Versus Relation Size for Sorting Methods

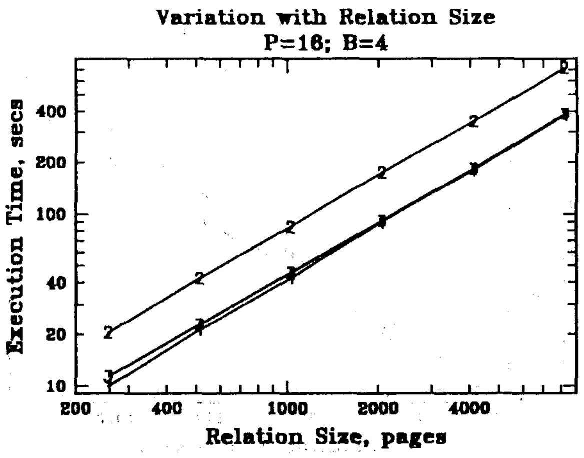

Figure 11: Execution Time Versus Relation Size for Sorting Methods

In Figure 12, we show the variation in execution time with the number of processors involved in the sorting. Once again, it may be seen that the modified block bitonic sort and the unmodified block bitonic sort are almost equal in performance until a threshold number of processors is reached. Beyond this threshold number of processors, the modified block bitonic sort is the best performer. The threshold is reached when there are enough processors so that the total size of the records to be sorted only slightly exceeds the total capacity of the combined memories in the multiprocessor given by B times P. The performance of the modified block bitonic sort is seen to improve even when the number of processors is increased beyond the threshold number of processors we described above.

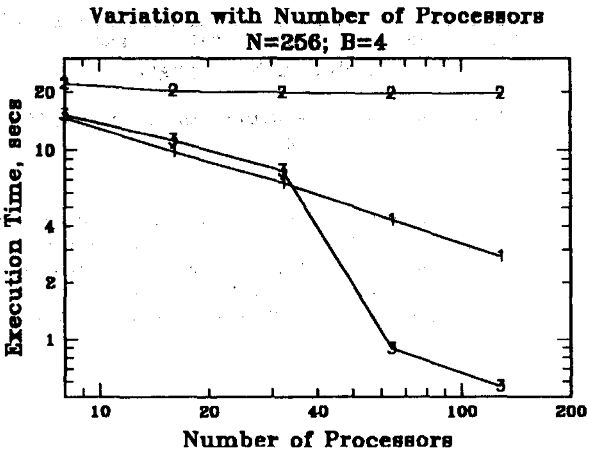

Figure 12: Execution Time Versus Number of Processors

Finally, in Figures 13 and 14, we show the variation in execution time with the number of buffers available per processor. It is seen that the odd-even sort and the unmodified block bitonic sorts exhibit a very mild bowl-shaped behavior - that is, increasing the number of buffers helps to a point. Increasing the number of buffers beyond that point actually hurts the performance of these algorithms.

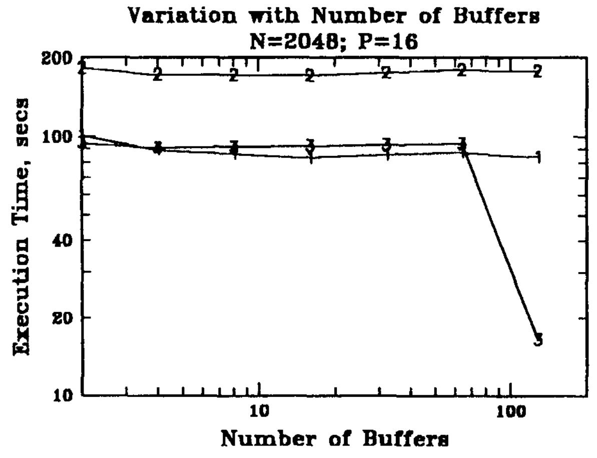

Figure 13: Execution Time Versus Number of Buffers

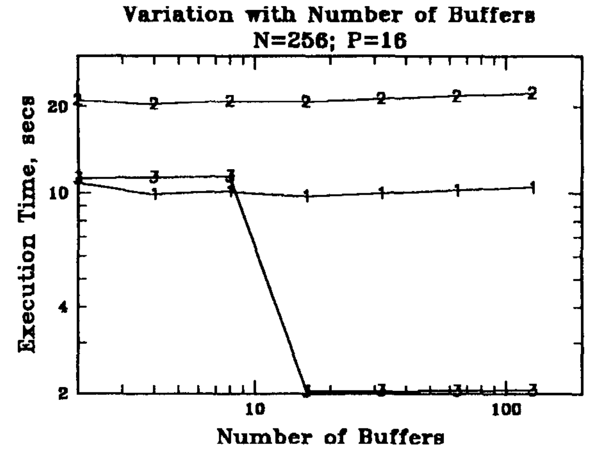

Figure 14: Execution Time Versus Number of Buffers

The modified block bitonic sort exhibits a very different kind of behavior. It begins with the same bowl-shaped behavior. However, beyond a certain buffer size, its performance improves dramatically. Once again, this buffer size beyond which the dramatic performance improvement takes place is when the total size of the records to be sorted only slightly exceeds the total capacity of the combined memories in the multiprocessor given by B times P.

Finally, we note that the modified block bitonic sort is the only algorithm that can make use of buffer sizes in excess of about 8 or 16 pages (for example 64 pages in Figure 13).

Conclusions and Summary

In this paper, we have studied the problem of sorting using multiple processors. Both internal sorting and external sorting have been considered. For internal sorting, we presented an algorithm that uses the idea of ***bitonic merging***.

For external sorting algorithms, we have studied the class of sorting algorithms that are derived from sorting algorithms that only do comparison and exchange, by replacing each comparison-exchange with a B-way merge. For this class of algorithms, we have suggested three and studied two techniques for improving performance - the use of ***pipelining*** and the use of ***parallel internal sorting***. These are general techniques that may be used to improve the performance of any of the algorithms in the class of algorithms of interest to us.

For pipelining, we concluded that it may be used to improve the performance of those algorithms that have a large number of merge steps, but is less useful for those algorithms where the number of merge steps is not very large.

We have also shown how to improve the block bitonic sort by using parallel internal sorting, and called our new algorithm the ***modified block bitonic sort***.

We have analyzed and studied the performance of several sorting algorithms under various conditions. We believe that our work represents the first attempt to study the variation in performance of multiprocessor sorting algorithms with changing buffer size. This study is important given that memory costs are dropping, making it quite feasible to build multiprocessors with large buffers. We showed that the odd-even and the unmodified block bitonic sorts exhibit a very mild bowl-shaped behavior - that is, increasing the number of buffers helps to a point. Increasing the number of buffers beyond that point actually hurts the performance of these algorithms. The modified block bitonic sort, on the other hand, behaves differently. It initially exhibits the same, mild, bowl-shaped behavior. However, beyond a certain buffer size, its performance improves dramatically. This buffer size beyond which the dramatic performance improvement takes place is when the total size of the records to be sorted only slightly exceeds the total capacity of the combined memories in the multiprocessor.

We saw that the modified block bitonic sort is much better than all the other methods for very large values of P (the number of processors) and B (the number of buffers per processor minus 1), or very small values of N (the size of the relation being sorted). We also saw that the modified block bitonic sort is slightly better than the other methods for very large values of N, and very small values of P (less than 20 to 40). For other values of N, B, and P, the modified block bitonic sort is slightly worse in performance than the block bitonic sort which is the best sorting algorithm under those conditions.

Our results show that the modified block bitonic sort is the fastest or close to the fastest algorithm over a wide range of values among the class of algorithms of interest to us. Furthermore, of all the algorithms we considered in this paper, the modified block bitonic sort made the best use of additional main memory buffers.

Bibliography

[BABBE79] Babb, E., Implementing a Relational Database by Means of Specialized Hardware, *ACM Transactions on Database Systems* **4**, No.1 (March 1979) pp. 1-29.

[BATCH68] Batcher, K. E., Sorting Networks and their Applications, *Proceedings of the 1968 Spring Joint Computer Conference* **32** (Atlantic City, N. J., May 1968) pp. 307-314.

[BAUDE78] Baudet, G. and Stevenson, D., Optimal Sorting Algorithms for Parallel Computers, *IEEE-TC* C-27, No.1 (Jan. 1978).

[BITTO83] Bitton, D., Boral, H., Dewitt, D. J., and Wilkinson, W. K., Parallel Algorithms for the Execution of Relational Database Operations, *ACM Transactions on Database Systems* **8, No.** 3 (Sept 1983).

[BITTO84] Bitton, D., Dewitt, D. J., Hsiao, D. K., and Menon, M. J., A Taxonomy of Parallel Sorting, *Computing Surveys* **16, No.3** (Sep. 1984).

[BRATS84] Bratsbergsengen, K., Hashing Methods and Relational Algebra Operations, *Proceedings of Tenth International Conference on VLDB* (Singapore, August 1984).

[DEWIT79] Dewitt, D. J., Query Execution in DIRECT, *Proceedings of ACM SIGMOD Conference* (New York, May 1979).

[GARDA81] Gardarin, G., An Introduction to SABRE: a multimicroprocessor database machine, *Sixth Workshop on Computer Architecture for Non-numeric Processing* (Hyeres, France, June 1981).

[HIRSC78] Hirschberg, D. S., Fast Parallel Sorting Algorithms, *Communications of the ACM* **21, No.8** (Aug. 1978).

[HSIAO80] Hsiao, D. K., and Menon, M. J., Parallel Record Sorting Methods for Hardware Realization, Technical Report, OSU-CISRC-TR-80-7, Computer and Information Science Department, Ohio State University, Columbus, Ohio (July 1980).

[KNUTH73] Knuth, D. E., *The Art of Computer Programming*, Addison Wesley (Reading, 1973).

[MENON86] Menon, M. J., A Study of Sort Algorithms for Multiprocessor Database Machines, IBM Research report RJ-5047 (Feb. 1986).

[MULLE75] Muller, D. E., and Preparata, F. P., Bounds to Complexities of Networks for Sorting and for Switching, *Journal of the ACM* **22, No. 2** (Apr. 1975).

[NASSI78] Nassimi, D., and Sahni, S., Bitonic Sort on a Mesh Connected Parallel Computer, *IEEE Transactions on Computers* **C-27, No. 1** (Jan. 1978).

[PREPA78] Preparata, F. P., New Parallel Sorting Schemes, *IEEE Transactions on Computers* **C-27, No. 7** (July 1978).

[STONE71] Stone, H. S., Parallel Processing with the Perfect Shuffle, *IEEE Transactions on Computers* **C-20,2** (Feb. 1971).

[THOMP77] Thompson, C. D., and Kung, H. T., Sorting on a Mesh-Connected Parallel Computer, *Communications of the ACM* **20, 4** (Apr. 1977).

[VALDU84] Valduriez, P. and Gardarin, G., Join and Semi-Join Algorithms for a Multiprocessor Database Machine, *ACM Transactions on Database Systems* **9, No. 1** (March 1984).

[VALIA75] Valiant, L. G., Parallelism in Comparison Problems, *SIAM Journal of Computing* **3, No. 4** (Sept. 1975).