I N F O R M A T I K

Asynchronous Parallel Disk Sorting

Roman Dementiev[1] and Peter Sanders[1].

MPI–I–2003–1–001 February 2003

FORSCHUNGSBERICHT RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Stuhlsatzenhausweg 85 66123 Saarbrücken Germany

Authors' Addresses

Roman Dementiev, Peter Sanders
Stuhlsatzenhausweg 85
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany
email: `{dementiev,sanders}@mpi-sb.mpg.de`

Abstract

We develop an algorithm for parallel disk sorting, whose I/O cost approaches the lower bound and that guarantees almost perfect overlap between I/O and computation. Previous algorithms have either suboptimal I/O volume or cannot guarantee that I/O and computations can always be overlapped. We give an efficient implementation that can (at least) compete with the best practical implementations but gives additional performance guarantees. For the experiments we have configured a state of the art machine that can sustain full bandwidth I/O with eight disks and is very cost effective.

Keywords

external memory, sorting, large data sets, overlapping I/O and computation

1 Introduction

Sorting is one of the most important operations performed on computers. In particular, sorting is a crucial tool when it comes to processing large volumes of data in secondary memory. Since a single disk is much cheaper than a high performance computer, a high performance external sorting algorithm needs to be able to exploit many disks. Interestingly, parallel disk sorting is a nontrivial problem. Asymptotically I/O optimal deterministic algorithms [16, 17] are complicated and have rather large constant factors. There are relatively simple randomized algorithms that approach the lower bound of $2N/DB \log_{M/B} N/B$ I/Os for sorting N elements using D disks, fast memory of size M, and blocks of size B [11]. These algorithms are so close to algorithms used in practice that theory and practice seem to be in harmony here. However, at least two issues remain before we can claim that the best randomized theoretical algorithms are also good in practice: We need a high performance implementation and we have to reconsider the model of computation when talking about constant factors. Perhaps the main issue for sorting is that I/O and internal work are completely separate issues in the I/O model of Vitter and Shriver [28]. In this paper we therefore design an algorithm that overlaps I/O and computation and give an efficient implementation.

Perhaps the most widely used external sorting algorithm is k-way merge sort. In Section 2, we explain how its best known randomized parallel disk version [11] can be adapted to allow almost perfect overlapping of I/O and computation. The idea is straight-forward: Use an additional *overlap buffer* that decouples I/O and computation. We prove that $k + \Theta(D)$ blocks buffer size are necessary and sufficient to allow almost perfect overlap. This is nontrivial to prove because we have to take into account the interactions between input, disk load balancing, merging, and output.

Whereas Section 2 uses synchronized parallel disk I/O steps to obtain a simple cost model, Section 3 explains how to implement the algorithm portably and efficiently in a fully asynchronous manner. The implementation is part of `<stxxl>`, a C++ library for external memory algorithms that we are currently developing. `<stxxl>` implements algorithms and data structures from the standard template library STL for massive data sets. The *I/O layer* — the lowest layer of `<stxxl>` — supports efficient asynchronous I/O that is currently implemented using multi-threading and unbuffered blocking file system I/O.

In Section 4 we describe how to achieve 375 MByte/s measured I/O bandwidth for about 3000 € using a dual-Xeon server board with multiple PCI busses, four cheap IDE disk controllers, and eight 80 GByte disks. This is about one third of the measured main memory bandwidth of this system so that one can conclude that on machines configured for high bandwidth I/O, the I/O bandwidth is hardly a limiting factor even if cost is an issue. Although the particular hardware configuration is a very dated result, we believe that a detailed description exemplifies an approach to configure hardware that will be valid for some time to come.

Section 5 summarizes the results of more than 1000 hours of experiments. `<stxxl>` is up to three times faster than previous libraries sustaining an I/O bandwidth of up to 315 MByte/s overlapped with sorting. It turns out that optimizing the number of I/O steps using the prefetching algorithm [11] does *not* optimize the overall execution time. Rather, the best results use an additional overlap buffer. The blocks sizes needed for good performance are several MBytes so that for large inputs, we enter the area where supposedly theoretical algorithms outperform plain striping that increases the block size requirement by another factor D. Perhaps the best way to characterize the bottom line performance of our system bought in July 2002 is to note that it sorts more cost effectively than the system that won the April 2002 *Penny*[2] *Sort Benchmark* but it does that about 6.5 times[3] faster than this low end system with two disks.

Related Work

The design of the `<stxxl>` library owes a lot to the previous external memory libraries TPIE [26, 4], LEDA-SM[10], and JavaXXL [24, 25]. `<stxxl>` adds emphasis on high performance, i.e., parallel disks, overlapping of I/O and computation, large inputs, and low internal overhead. None of the above libraries explicitly handles parallel disks and overlapping I/O and computation relies largely on the operating system. We view this as problematic[4] for leading edge performance because prefetching and caching of the operating system knows less about the application, leaves less memory for sorting itself, and often requires additional copies of the data. LEDA-SM, TPIE, and [9, 8] allow only 2GByte input size.

Barve and Vitter [6] implement a parallel disk algorithm [5] that can be viewed as the immediate ancestor of our algorithm. Innovations with respect to this work are: A different allocation strategy that enables better theoretical I/O bounds [12, 11]; a prefetching algorithm that optimizes I/O steps and never evicts data previously fetched; overlapping of I/O and computation; a completely asynchronous implementation that reacts flexibly to fluctuations in disk speeds; and an implementation that sorts many GBytes and does not have to artificially limit internal memory size to obtain a nontrivial number of runs.

Chaudhry and Cormen [9, 8] give a sophisticated external memory implementation of column sort that uses parallel disks and processors. The algorithm also has the theoretical advantage of being deterministic. This theoretical advantage translates into the practical benefit that disk access patterns are very regular and easy to overlap with computation. A drawback of column sort is that even in its most sophisticated form, it needs about 50 % larger I/O volume than multi-way merge sort (three versus two passes over the data). Another drawback is that column sort seems to need rather fine grained I/O because the maximum possible block size for about half of the I/Os is about $N^{1/3}$. For example, for 2GByte of 128 byte records this would be blocks of 32KByte which is far from the optimal block sizes that are nowadays measured in MBytes (see Figure 13). A theoretical disadvantage is that the maximal input size for which the three-pass algorithm works is $\mathcal{O}\left(M^{3/2}\right)$. For comparison, multi-way merge sort allows $\mathcal{O}(M^2/B)$ elements for a two-pass algorithm and $\mathcal{O}(M^3/B^2)$ for a three-pass algorithm. Column sort can be applied to larger inputs using recursion but this does not lead to asymptotically optimal performance.

[2]The cost of the hardware is spread over three years. Then it is measured how much data can be sorted in an interval of time that costs one US-cent. See `http://research.microsoft.com/barc/sortbenchmark/`.

[3]We sort the same amount of 125 million 100 byte elements but use 8 byte keys rather than 10 byte keys. We believe that for a tuned implementation and random keys this makes little difference.

[4]We observed an extreme example on an earlier Solaris based experimental platform: The system discarded cached blocks more slowly than they came in from parallel disks. The result was that all the application memory was swapped out in favor of cached disk blocks that were not needed at all . . . .

Rajasekaran [21] gives another asymptotically suboptimal deterministic parallel disk sorting algorithm that runs in three passes for not too large inputs.

Prefetch buffers for disk load balancing and overlapping of I/O and computation has been intensively studied [20, 7, 3, 13, 12, 11]. But we have not seen results that guarantee overlapping of I/O and computation during parallel disks merging of arbitrary runs.

There are many good practical implementations of sorting (e.g. [18, 1, 29, 19]) that address parallel disks, overlapping of I/O and computation, and low internal overhead. However, we are not aware of fast implementations that give theoretical performance guarantees on achieving asymptotically optimal I/O. Most practical implementations use a form of striping that requires $\mathcal{O}(N/DB \log_{\Theta(M/\mathbf{DB})} N/B)$ I/Os rather than the optimal $\mathcal{O}(N/DB \log_{\Theta(M/\mathbf{B})} N/B)$. This difference is usually considered insignificant for practical purposes. But on our system we already have to go somewhat below the block sizes that give best performance in Figure 13 if the input size is 128 GBytes. Another reduction of the block size by a factor of eight could increase the run time significantly. We are also not aware of high performance implementations that guarantee overlap of I/O and computation during merging for inputs such as the one described in the beginning of Section 2.3.

On the other hand, many of the practical merits of our implementation are at least comparable with the best actual implementations: We are close to the peak performance of our system and its price performance ratio is better than anything we have seen. Our library should also be easy to use since it is based on the well known interface of the STL.

2 Multi-way Merge Sort with Overlapped I/Os

This section derives an parallel disk sorting algorithm that almost perfectly overlaps I/O and computation. More formally, the following theorem is shown.

Theorem 1. *Let L denote the time needed for accessing one block of size B on each of D disks. Let $T_{\text{sort}}(n)$ denote the time needed to sort n elements internally and $\ell(k)$ the time needed to produce one element of output in internal k-way merging. If I/O and computation can be overlapped, N elements can be sorted in time*

$$\max\left(k' T_{\text{sort}}\left(\frac{N}{k'}\right), \frac{2LN}{DB}\right) + \mathcal{O}\left(\frac{LM}{DB}\right)$$
$$+ \lceil \log_{\Theta(M/B)} k' \rceil \left(\max\left(\frac{2LN}{(1-\epsilon)DB}, \ell(k)N\right) + \mathcal{O}\left(L \min\left(\frac{M}{B}, \frac{D}{\epsilon}\log\frac{D}{\epsilon}\right)\right)\right)$$

where $k' = \mathcal{O}(N/M)$ is the total number of runs, $k = \Theta(M/B)$ is the merging degree used, and $\epsilon = \Theta(DB/M)$.

To help reading this complicated formula, one can note that in all practical cases, $k = k'$, $\lceil \log_{\Theta(M/B)} k' \rceil = 1$, and ϵ is some small constant. The first line corresponds to a *run formation* phase that reads batches of size $\Theta(M)$ from disk, sorts them internally, and writes them back to disk. Section 2.1 establishes that any internal sorting algorithm can be perfectly overlapped with I/O except for $\mathcal{O}(M/DB)$ I/Os at the beginning and at the end.

The second line of the time bound corresponds to merging the sorted runs into a single sorted run. Since this result is significantly more difficult to establish, it is obtained in three

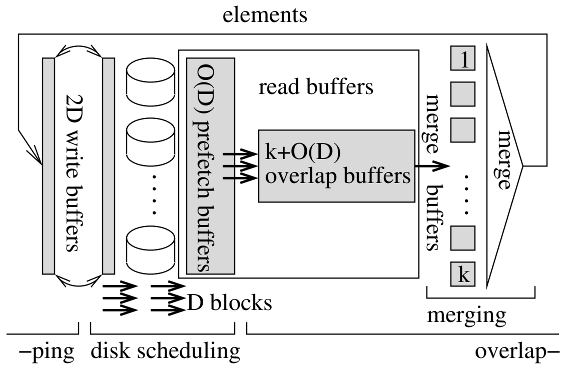

Figure 1: Data flow through the different kinds of buffers for overlapped parallel disk multi-way merging. Data is moved in units of blocks except between the merger and the write buffer.

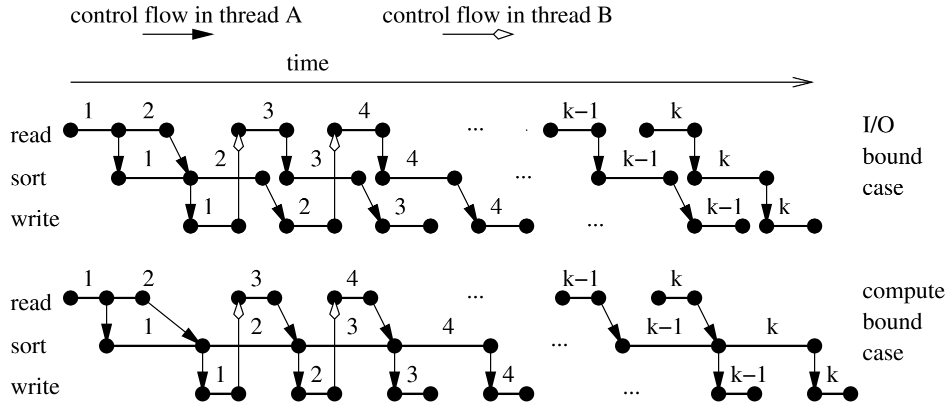

Figure 2: Overlapping I/O and computation during run formation.

steps. Section 2.2 describes merging from the point of view of a *merging thread* that reads blocks in an order predicted during run formation and writes individual elements. Each block has to be read exactly once using one *merge buffer* block for each run. Section 2.3 explains how an *I/O thread* interfaces this view with a I/O model that allows parallel access to D arbitrary blocks in an I/O step [2]. The I/O thread is responsible for overlapping I/O with computation. Using an *overlap buffer* the algorithm achieves perfect overlapping of I/O and computation up to a small overhead for filling and emptying the merge buffers.

Section 2.4 explains how a *prefetch* buffer can be used to implement this parallel access model on D parallel disks. This emulation costs a constant factor close to one in I/O overhead plus a logarithmic additive term. Figure 1 illustrates the data flow between these components of parallel disk multi-way merging.

2.1 Run Formation

There are many ways to overlap I/O and run formation. We start with a very simple method that treats internal sorting as a black box and hence can use the fastest available internal sorters.[5] Two threads cooperate to build k runs of size $M/2$:

```
post a read request for runs 1 and 2
thread A:                           |  thread B:
  for r:=1 to k do                  |  for r:=1 to k-2 do
    wait until run r is read        |    wait until run r is written
    sort run r                      |    post a read request for run r+2
    post a write request for run r  |
```

Figure 2 illustrates how I/O and computation is overlapped by this algorithm. We omit the proof of the following theorem that would essentially be a simple formalization of Figure 2.

Corollary 2. *An input of size N can be transformed into sorted runs of size $\Theta(M/2)$ in time $\max(2T_{\mathrm{sort}}(\frac{M}{2})\frac{N}{M}, \frac{2LN}{DB}) + \mathcal{O}(\frac{LM}{DB})$ where $T_{\mathrm{sort}}(n)$ denotes the time for sorting n elements internally and where L is the time needed for an I/O step.*

A natural question arising from this discussion is how long the runs can be if we want to overlap I/O and computation. Knuth [14, Section 5.4.1] describes an algorithm that achieves average run length $2M$. A recent implementation that even works for variable length records has been described by Larson and Graefe [15]. However, this algorithm is not cache efficient and requires an additional pointer for each element in the input. We therefore outline a relatively simple reformulation that is space efficient even for small records, cache efficient, and provably allows overlapping of I/O and computation.

An more abstract formulation is a good starting point: The algorithm maintains two priority queues Q and Q'. Initially, M elements are inserted into Q. The following operations are repeated until Q is empty:

$q :=$ deleteMinimum(Q)
read a new element q' from the input
if $q' < q$ **then** Q'.insert(q') **else** Q.insert(q')
write(q)

Then one run is finished, and a new run is started based on the now M elements in Q'. Although there are cache efficient priority queues [23], these have a too large worst case access time and we have to explain how to make the queues space efficient. The following representation solves both problems: Let ϵ denote some small constant. We represent the priority queues by collections of sorted sequences of size up to ϵM. Q additionally has a buffer priority queue Q_0 of size up to ϵM. Q' also has an insertion buffer Q'_0 that is an unsorted bag of up to ϵM elements. Insertions into Q or Q' go into these buffers. When they are filled, they are sorted and added to the collection of sorted sequences for the queues. This takes time $T_{\mathrm{sort}}(\epsilon M)$. Since the size of a run increases by ϵM whenever a

[5]If this method has not been published yet, we would still guess that it is folklore.

new sorted sequence is added, the average number of sequences in a run is $2/\epsilon$. Using a binary heap for Q_0 and multi-way merging for Q, a deleteMinimum can be implemented in time $\mathcal{O}(1/\epsilon)$. The average case insertion time into the buffers is $\mathcal{O}(1)$ even if binary heaps are used. Using $\mathcal{O}(\max(DB, \epsilon M))$ additional space for buffering input and output, perfect overlapping between I/O and computation is possible. The sorted sequences can be made space efficient by representing them as a linked list of small blocks of elements. As soon as the last element of a block is removed, the block is put into a free list that supplies empty blocks when building new sorted sequences.

In the full paper we will describe a further algorithm with similar properties that does not require binary heaps and replaces multi-way merging by distribution of elements to buckets and hence might be faster for "well behaved" integer keys.

2.2 Multi-way Merging

We want to merge k sorted sequences comprising N' elements stored in N'/B blocks (In practical situation, where a single merging phase suffices, we will have $N' = N$). In each iteration the merging thread chooses the smallest remaining element from the k sequences and hands it over to the I/O thread. Prediction of read operations is based on the observation that the merging thread need not access a block until its smallest element becomes the smallest unread element. We therefore record the *smallest* keys of each block during run formation. By merging the resulting k sequences of smallest elements, we can produce a sequence σ of block identifiers that indicates the exact order in which blocks are logically read by the merging thread. The overhead for producing and storing the prediction data structure is negligible because its size is a factor at least B smaller than the input.

The prediction sequence σ is used as follows. The merging thread maintains the invariant that it always buffers the k first blocks in σ that contain unselected elements, i.e., initially, the first k blocks from σ are read into these *merge buffers*. When the last element of a merge buffer block is selected, the now empty buffer frame is returned to the I/O thread and the next block in σ is read.

The keys of the smallest elements in each buffer block are kept in a tournament tree data structure [14] so that the currently smallest element can be selected in time $\mathcal{O}(\log k)$. Hence, the total internal work for merging is $\mathcal{O}(N' \log k)$. To establish that this strategy correctly merges the sequences, we have to show that the smallest element not selected yet resides in a block that is buffered.

Lemma 3. *At any point during multi-way merging, the smallest element among the elements in the k merge buffer blocks is minimal among all elements not yet selected by the merging thread.*

Proof. Suppose there is an unselected element e that is smaller than all unselected elements in the merge buffer blocks. Element e must be the smallest element of some block b in some sequence j such that none of the blocks of sequence j are in a merge buffer block. Since there are only k input sequences, there must be another sequence j' for which at least *two* blocks b' and b'' are buffered. Call the first element of the second block e''. Since b'' was read before b we must have $e'' \leq e$. Furthermore, there must be an unselected element e' in b' and we have $e' \leq e'' \leq e$. This contradicts the assumption that e is smaller than any buffered unselected element. $\square$

We have now defined multi-way merging from the point of view of the sorting algorithm. Our approach to merging slightly deviates from previous approaches that assign each merge buffer block to one input sequence and where the *last* key in the *previous* block decides about the position of a block in σ. With respect to performance, both approaches should be similar. Our approach is somewhat simpler however — the merging thread need not know anything about the k input runs and how they are allocated. Its only input is the prediction sequence σ. In a sense, we are merging blocks and the order in σ makes sure that the overall effect is that the input runs are merged. A conceptual advantage is that data *within* a block decides about when a block is read.

2.3 Overlapping I/O and Merging

Although we can predict the order in which blocks are read, we cannot easily predict how much internal work is done between two reads. For example, consider k identical runs storing the sequence $\boxed{1^{B-1}2}\boxed{3^{B-1}4}\boxed{5^{B-1}6}\cdots$. After initializing the merge buffers, the merging thread will consume $k(B-1)$ values '1' before it posts another read. Then it will post one read after selecting each of the next k values (2). Then there will be a pause of another $k(B-1)$ steps and another k reads quickly following each other, etc. We explain how to overlap I/O and computation despite of this irregularity using the I/O model of Aggarwal and Vitter [2] that allows access to D *arbitrary* blocks within one I/O step. To model overlapping of I/O and computation, we assume that an I/O step takes time L and can be done in parallel with internal computations. We maintain an *overlap buffer* that stores up to $k+3D$ blocks in a FIFO manner. Whenever the overlap buffer is nonempty, a read can be served from it without blocking. Writing is implemented using a *write buffer* FIFO with $2DB$ elements capacity. An *I/O thread* inputs or outputs D blocks in time L using the following strategy: Whenever no I/O is active and at least DB elements are present in the write buffer, an output step is started. When no I/O is active, less than D output blocks are available, and at least D overlap buffers are unused, then the next D blocks from σ are fetched into the overlap buffer.

The following theorem states that this simple strategy allows almost perfect overlapping of I/O and computation.

Theorem 4. *Merging k sorted sequences with a total of N' elements can be implemented to run in time*

$$\max\left(\frac{2LN'}{DB}, \ell N'\right) + \mathcal{O}\left(L\left\lceil\frac{k}{D}\right\rceil\right)$$

where ℓ is the time needed by the merging thread to produce one element of output and L is the time needed to input or output D arbitrary blocks.

The most basic tool for the proof of Theorem 4, is the following sufficient condition for the availability of input for the merging thread.

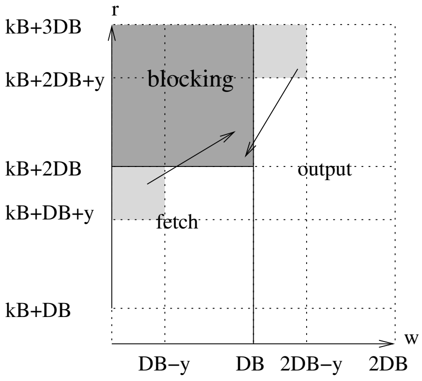

Figure 3: Proof of Lemma 6.

Lemma 5. *Whenever the overlap buffer and merge buffer together contain at least kB elements, then at least one element can be merged without fetching additional blocks.*

Proof. Suppose to the contrary that a new block needs to be fetched. This can only be the case if the overlap buffer is empty. But this implies that all k merge buffers are full. This contradicts the assumption that no elements can be merged. □

The key to the proof of Theorem 4 are the following two lemmas that represent the I/O bound respectively the compute bound case.

Lemma 6. *If $2L \geq DB\ell$ then the I/O thread never blocks until all input blocks are fetched.*

Proof. We describe the state of the system by the pair (w, r) where w is the number of elements in the write buffer and r is the total number of elements in the overlap buffer and the merge buffers. Let $y = \lfloor L/\ell \rfloor$ denote the number of elements that can be merged during one I/O step. Since $2L \geq DB\ell$, we have $y \geq DB/2$. If $y \geq DB$, Lemma 5 implies that r can never exceed $kB + DB$ so that the overlap buffer always has enough space to fetch D additional blocks. The interesting case is $DB/2 \leq y < DB$.

We want to show that the system never enters a state where the I/O thread can block. This can only happen if $w < DB$ and $r > kB + 2DB$ because otherwise we can either output or fetch D blocks. The dark shaded area in Figure 3 defines this area. If $r > kB + y$ there are two sub-cases: If $w < DB$, a fetch step is executed leading to the state transition $(w, r) \rightsquigarrow (w + y, r + DB - y)$. If $w \geq DB$, an output step leads to the state transition $(w, r) \rightsquigarrow (w - DB + y, r - y)$. With the help of Figure 3 it is now easy to see that only the light shaded regions can lead to a transition into the blocking region. But there are *no* transitions into the light shaded regions. This remains true for $r \leq kB + y$ because from there we cannot get to a state with $r > kB + DB$. □

Lemma 7. *If $2L < DB\ell$ then after $k/D + 1$ I/O steps, the merging thread never blocks until all elements are merged.*

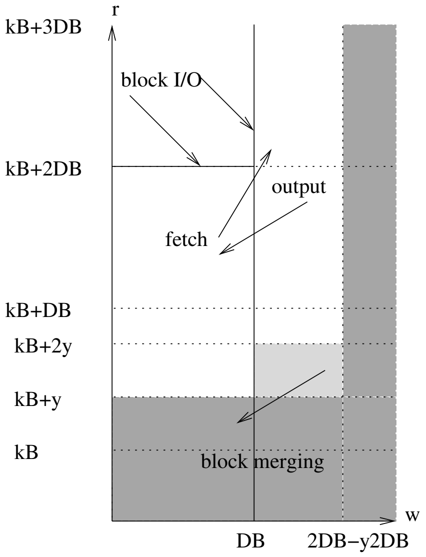

Figure 4: Proof of Lemma 7.

Proof. Define w, r, and y as in the proof of Lemma 6. Since $2L < DB\ell$, we have $y < DB/2$. We want to show that the system never enters a state where the merging thread can block. This can only happen if $w > 2DB - y$ or $r < kB + y$. Otherwise, we can distinguish three cases illustrated in Figure 4. If the I/O thread is active, we have the same state transitions as in Lemma 6, $(w, r) \rightsquigarrow (w + y, r + DB - y)$ if $w < DB$ and $r < kB + 2DB$ and $(w, r) \rightsquigarrow (w - DB + y, r - y)$ if $w \geq DB$. Otherwise, the I/O thread blocks and the merging thread moves elements to the write buffer until there is room for fetching or writing another D blocks. These transitions imply that the only region in the state space that can lead to a state where the merging thread is blocked, is $w \geq DB$ and $r \in [kB + y, kB + 2y)$. But this region cannot be reached from a state where the merging thread is active. □

Now it is easy to establish Theorem 4.

Proof. If $2L \geq DB\ell$, Lemma 6 implies that after time LN'/DB, all blocks have been fetched. It remains to merge $\mathcal{O}((k + D)B)$ elements from the merge and overlap buffer and to output them. This takes time $\mathcal{O}(\ell(k + D)B + L\lceil k/D \rceil) = \mathcal{O}\big(L \left\lceil \frac{k}{D} \right\rceil\big)$.

If $2L < DB\ell$, Lemma 7 implies that after $k/D + 1$ I/O steps (in time $\mathcal{O}(L\lceil k/D \rceil)$, the merging thread will merge all elements in time $\ell N'$. Then at most two further I/O steps suffice to flush the write buffer. The overall time needed is $\ell N' + L\mathcal{O}(\lceil k/D \rceil)$. □

2.4 Disk Scheduling

The I/Os for run formation and for the output of merging are perfectly balanced over all disks if all sequences are *striped* over the disks, i.e., sequences are stored in blocks of B elements each and the blocks numbered $i, \ldots, i + D - 1$ in a sequence are stored on different disks for all i. In particular, the original input and the final output of sorting can use any kind of striping.

The situation is more complicated during merging. Although each run is striped over the disks, the order σ prescribed by the smallest elements in the runs can lead to highly irregular access patterns. Vitter and Hutchinson [27] have shown that *Randomized Cyclic Allocation (RC)* makes the accesses in σ at least as well balanced as independent accesses to random disks. In RC allocation, the i-th block of a run is stored on disk $\pi(i \bmod D)$ where π is a random permutation that is chosen independently for each run. In [11] it is then shown that an optimal *prefetch order* σ' that uses a *prefetch buffer* of size $m = \Theta(D)$ blocks can be computed from σ by simulating a simple optimal writing algorithm for the reverse sequence σ^R. It is also shown that after a startup phase of $\min(k + \frac{N'}{DB}, \frac{m}{D}\log m)$ input steps, $(1 - \mathcal{O}(\frac{D}{m}))D$ blocks from σ can be fetched per input step on the average (k is the number of runs).

This is not quite sufficient for our purposes because overlapping I/O and computation requires "uniform" progress during each I/O step. But going back to the probabilistic core of the above analysis in [22] we see that the result can be strengthened: In almost every input step, $(1 - \mathcal{O}(D/m))D$ blocks from σ can be fetched. The failure probability is exponentially small in D.

The bottom line is that a prefetch buffer of m blocks allows us to emulate the model assumed in Section 2.3 except for a short startup phase, a reduction of the effective number of disks by D/m, and possibly occasional "hiccups" that affect a negligible fraction of the I/O steps. We obtain the following refined version of Theorem 4

Corollary 8. *For any $\epsilon > 0$ and $D = \Omega(1/\epsilon)$,[6] there is a prefetch buffer of size $m = \Theta(D/\epsilon)$ such that merging k sorted sequences with a total of N' elements can be implemented to run in time*

$$\max\left(\frac{2LN'}{(1-\epsilon)DB}, \ell N'\right) + \mathcal{O}\left(L\left(\frac{k}{D} + \min\left(k, \frac{1}{\epsilon}\log\frac{D}{\epsilon}\right)\right)\right)$$

where ℓ is the time needed by the merging thread to produce one element of output, L is the time needed to input or output D arbitrary blocks, and m is the size of the prefetch buffer.

A further remark is necessary for the (unrealistic) case of very large inputs where several merging phases are needed. In that case, a prefetching sequence σ' for all merging operations in a phase should be computed. The additive term $\mathcal{O}\big(L(\frac{k}{D} + \min(k, \frac{1}{\epsilon}\log\frac{D}{\epsilon}))\big)$ then only occurs once per phase.

3 Implementation

Our implementation of sorting is part of a new C++ library `<stxxl>` for external computing that is designed for maximum compatibility with the *standard template library (STL)*. Another goal of the library is very high performance with support for parallel disks and overlapping of I/O and computation. We started with an implementation of sorting because it already tests many of these properties and since an efficient sorter is a key ingredient for many external algorithms.

[6] We believe that the last restriction is an artifact of the analysis in [22, 11] but a formal proof that lifts it might be much more complicated without yielding much additional insight.

The I/O layer of `<stxxl>` implements asynchronous parallel block I/O. This level supports the minimum functionality needed to abstract from details of the file system and the operating system. Our current implementation runs on Linux using unbuffered synchronous file system I/O and POSIX threads for supporting asynchrony: There is one thread for each disk which maintains a read queue and a write queue. It arbitrates between these queues using a strategy chosen by the higher levels of the library. In our sorting algorithm, writing is prioritized, i.e., when the thread returns from an I/O operation, it first checks the write queue and posts the next request if it is nonempty. Only if the write queue is empty it tries the read queue. Later implementations might use completely different mechanism like the high performance asynchronous I/O supported by DAFS[7].

Run Formation. We build runs of size close to $M/2$ but there are some differences to the simple algorithm from Section 2.1. Overlapping of I/O and computation is achieved using a call-back mechanism supported by the I/O layer of `<stxxl>` rather than by multi-threading. Thus, the sorter remains portable over different operating systems with different interfaces to threading.

To limit the memory bandwidth requirements for large elements with small key fields, we implement *key sorting*, i.e., after reading elements using DMA, we extract pairs (key, pointerToElement), sort these pairs, and only then move elements in sorted order to write buffers from where they are output using DMA.

Furthermore, we exploit random keys. We use two passes of MSD (most significant digit) radix sort of the key-pointer pairs. The first pass uses the m most significant bits where m is a tuning parameter depending on the size of the processor caches and of the TLB (translation look-aside buffer). This pass consists of a counting phase that determines bucket sizes and a distribution phase that moves pairs. The counting phase is fused into a single loop with pair extraction. The second pass of radix sort uses a number of bits that brings us closest to an expected bucket size of two. This two-pass algorithm is much more cache efficient than a one-pass radix sort.[8] The remaining buckets are sorted using a comparison based algorithm: Optimal straight line code for $n \leq 4$, insertion sort for $n \in \{5..16\}$, and quicksort for $n > 16$.

Multi-way Merging. We have adapted the tuned multi-way merger from [23].

Overlapping I/O and Computation. We integrate the prefetch buffer and the overlap buffer to a *read buffer*. We distribute the buffer space between the two purposes of minimizing disk idle time and overlapping I/O and computation indirectly by computing an optimal prefetch sequence for a smaller buffer space.

Asynchronous I/O. I/O is performed without any synchronization between the disks. The prefetcher described in Section 2.4 computes a sequence σ' of blocks indicating the order in which blocks should be fetched. As soon as a buffer block becomes available for prefetching, it is used to generate an asynchronous read request for the next block in σ'. All I/O is implemented without superfluous copying. Blocks are moved by DMA (direct memory access) directly to user memory. A fetched block then travels to the prefetch/overlap buffer

[7] `http://www.dafscollaborative.org/`

[8] On our system we get a factor 3.8 speedup over one pass radix sort and a factor 1.6 over STL's sort which in turn is faster than a hand tuned quicksort (for sorting 2^{21} pairs storing a random four byte key and a pointer).

and from there to a merge buffer simply by passing a pointer. Similarly, when an element is merged, it is directly moved from the merge buffer to the write buffer and a block of the write buffer is passed to the output queue of a disk simply by passing a pointer to the the I/O layer of `<stxxl>` that then uses `write` to output the data using DMA.

4 Hardware

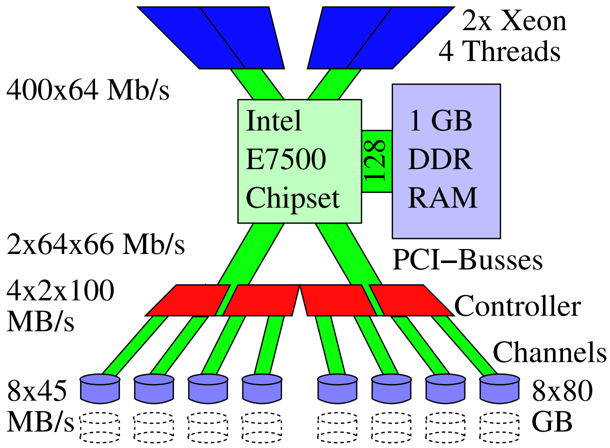

Figure 5: Simplified scheme of our experimental I/O-platform.

Our starting point was the belief that the gap between theory and practice in external memory parallel disk sorting can only be closed by demonstrating close to peak performance on state of the art hardware. For us and probably other groups this was a nontrivial problem. When we started we had several year old parallel disk hardware with a factor of ten lower bandwidth than the state of the art. More recent alternatives were PCs with a 32bit 33MHz PCI bus that are hopelessly limited in I/O bandwidth; a file server that could not be used for experiments because it serves hundreds of researchers; and a high end compute server for which a matching equipments with disks would have cost a six digit amount of money.

We therefore decided to configure a hardware platform for testing external memory algorithms from scratch. The machine was bought in July 2002. The design objectives were high bandwidth at low cost, and the use of standard components. The first challenge was to find an affordable main-board that breaks out of the limitations of a 32bit 33MHz PCI bus. We decided on a Supermicro SUPER P4DPE dual processor board with two 2GHz Intel Xeon processors (512 KByte cache and 2 threads per processor) at a cost around $675 + 2 \times 415$€. The board supports several independent 64bit PCI busses. Although we have not explicitly parallelized the sorter yet, the second processor is probably useful because it makes overlapping of I/O and computation more effective. We bought 1GByte of RAM.

The next important design decision was to use IDE disks rather than SCSI disks because they have higher capacity and similar I/O bandwidth than SCSI disks but are much cheaper. We decided on IBM 120GXP disks that have 80 GByte capacity at 120 € each.There were two difficulties to overcome however. It turned out that 64bit controllers are very expensive.

Fortunately it turned out that dual channel Promise 100 TX2 controllers are very cheap (around 40 € each). They work with 32 bits and 66MHz. Four of them on two 66MHz PCI busses are sufficient to support eight disks at full bandwidth.[9]

The second problem was to find a casing that allows to connect eight IDE disks given the limited cable length of the ATA standard. We choose a casing that has the shape of a double-bigtower. It is cheaper than a comparable rack-mount casing and works with shorter disk cables because the motherboard in the middle. We also use *round* disk cables that are less bulky than the usual flat ones.

We installed Debian Linux with kernel version 2.4.20 on this machine. Then we began with basic performance tests. Originally we thought that disk access via raw devices would give maximal bandwidth. Interestingly, this was only true up to four disks. Beyond that, the system started thrashing. We traced this problem down to the fact that there is some software intervention for each chunk of 512 bytes. Apparently, this overwhelms the operating system for too many disks. Good performance is obtained using unbuffered I/O in the `ext2` file system where files are opened with the option `O_DIRECT` and where addresses and block sizes are multiples of the virtual memory page size. Only then is it possible on PC hardware to move data directly from disk to user memory using DMA. We also decreased the number of `inodes` (blocks with meta-data) to reduce file system overhead.

With these measures we obtain an input bandwidth of up to 375 MByte/s on eight disks using the outermost (fastest) zones[10] of the disks. This is 97 % of the peak bandwidth specified by IBM. It was possible to attach a ninth disk obtaining 418 MByte/s. Bandwidth scaling stopped with the tenth disk. Figure 5 outlines the configuration of our hardware.

The bottom line is that for a system that costs three to four times as much as a standard PC with a single disk, we obtain eight times the I/O bandwidth. We believe that such a system is a more likely candidate for running applications with massive data sets than an ordinary PC and should therefore be preferred for performance studies of external memory algorithms.

An interesting observation is, that measuring the main memory bandwidth with the stream benchmark[11] we see 1200 MBytes/s. This implies that any external memory algorithm that accesses four bytes of main memory for each byte of I/O may already be compute bound.

5 Experiments

If not otherwise mentioned, we use random 32 bit integer keys to keep internal work limited. Runs of size 256 MByte[12] are build using key sorting with an initial iteration of 10 bit MSD

[9] In reality, incompatibilities between Linux and the controllers forced us to use five controllers in the following configuration: three controllers with one disk each on PCI-bus 1, two controllers with two disks each on PCI-busses 2 and 3, and one disk on the on-board controller.

[10] Modern disks store data at a roughly constant density so that the higher absolute speed of the outer parts of the disk allow around twice as high bandwidth as the inner parts.

[11] `http://www.cs.virginia.edu/stream/`

[12] This leaves space for two runs build in an overlapped way, buffers, operating system, code, and, for large inputs, the fact that the current implementation of the `ext2` file system needs 1 byte of internal memory for

radix sort. We choose block sizes in such a way that a single merging phase using 512 MBytes for all buffers suffices. Input sizes are powers of two between 2 GByte and 128 GByte with a default of 16 GByte[13]. When not otherwise stated, we use eight disks, 2 MByte blocks, and the input is stored on the fastest zones. All programs are compiled with `g++` version 3.2 and optimization level `-O6`.

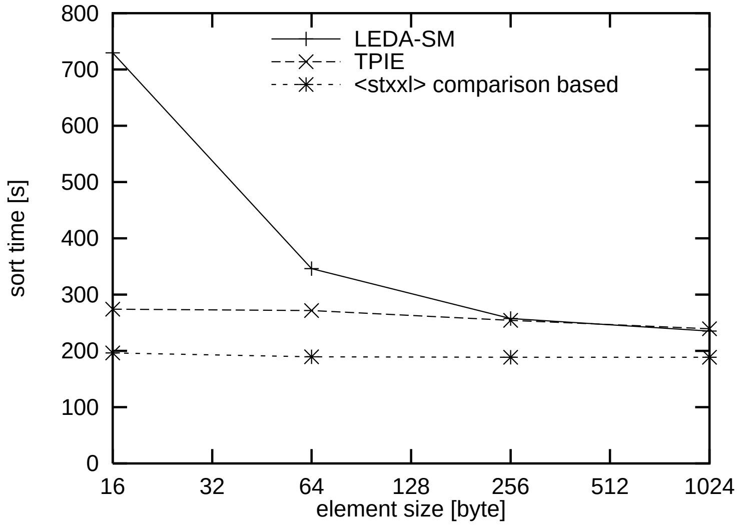

Figure 6: Comparison of the single disk performance of `<stxxl>`, LEDA-SM, and TPIE.

To compare our code with previous implementations, we have to run them on the same machine because technological development in recent years has been very fast. Unfortunately, the implementations we could obtain, LEDA-SM [10] and TPIE [25], are limited to inputs of size 2 GByte which for our machine is a rather small input. Figure 6 compares the single disk performance of the three libraries. Using the best block size for each library. The flat curves for TPIE and `<stxxl>` indicate that both codes are I/O bound even for small element sizes. This is even true for the fully comparison based version of `<stxxl>`. Still, `<stxxl>` is significantly faster than TPIE. This could be due to better overlapping of I/O and computation or due to higher bandwidth of the file system calls we use. `<stxxl>` sustains an I/O bandwidth of 45.4 MByte/s which is 95 % of the 48 MByte/s peak bandwidth of the disk at their outermost zone. LEDA-SM is compute bound for small keys and has the same performance as TPIE for large keys.

To get some kind of comparison for parallel disks, we run the other codes using Linux Software-RAID 0.9 and 8×128KByte stripes (larger stripes did not improve performance). Here `<stxxl>` is between two and three times faster than TPIE and and sustains an I/O bandwidth of 315 MByte/s for large elements. Much of this advantage is lost when `<stxxl>` also runs on the Software-RAID. Although we view as likely that the Software-RAID driver

each KByte of disk space accessed via `O_DIRECT`.

[13]We have a few measurements with 256 GBytes but the problem with `ext2` mentioned above starts to distort the results for this input size.

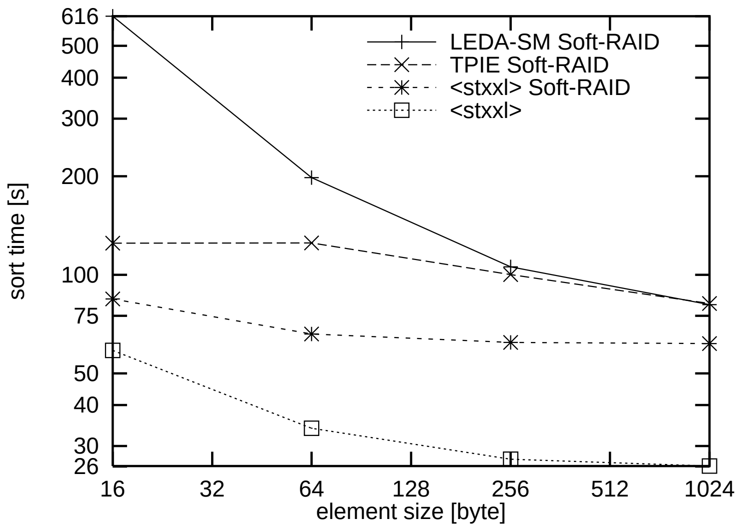

Figure 7: Comparison of of `<stxxl>`, LEDA-SM, and TPIE for eight disks.

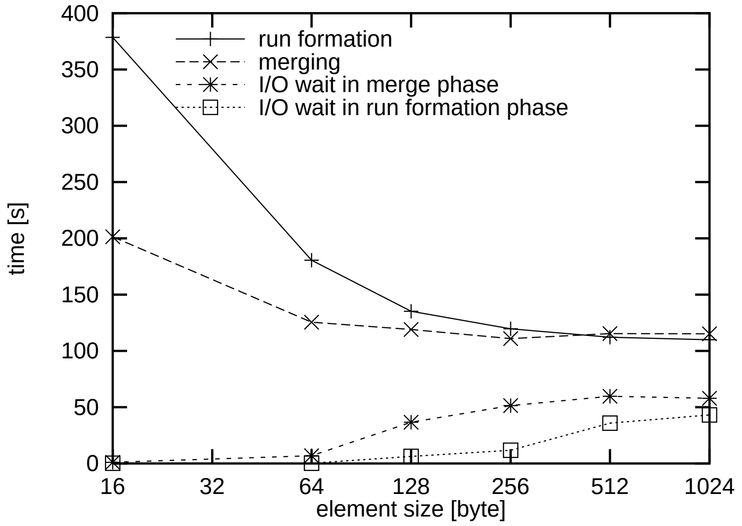

Figure 8: Dependence execution time and I/O wait time on the element size.

can be improved, this performance difference might also be an indication that treating disks as independent devices is better than striping (as predicted by theory).

Figure 8 shows the dependence of performance on element size in more detail. For element sizes ≥ 64, the merging phase starts to wait for I/Os and hence is I/O bound. The run formation phase only becomes I/O bound for element sizes around 128. This indicates areas for further optimization. For small elements, it should be better to replace key sorting by sorters that always (or more often) move the entire elements. For example, we have observed that the very simple loop that moves elements to the write buffer when the key-

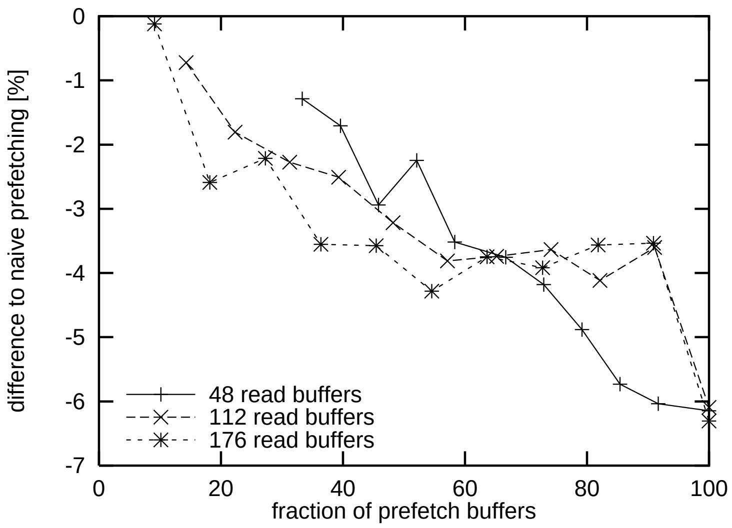

Figure 9: Change in input time due to optimal prefetching.

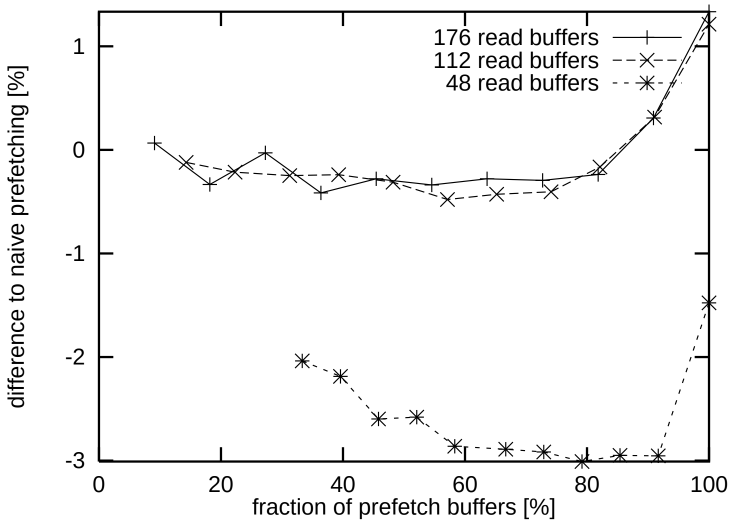

Figure 10: Change in total merge time due to "optimal" prefetching.

pointer pairs are already sorted can take up to 45 % of the CPU time of run formation. For small keys it looks also promising to use parallelism. Already our cheap machine supports four parallel threads.

We now turn to a more detailed analysis of prefetching and overlapping of I/O and computation. We first focus on the read buffers and hence fix the write buffer size to $4D$ blocks in Figures 9–11. Figure 9 compares the I/O time of the naive algorithm that tries to fetch blocks in the order specified by σ with optimal prefetching. It varies the fraction of the read buffer devoted to prefetching. As one would expect from the theoretical analysis in [11],

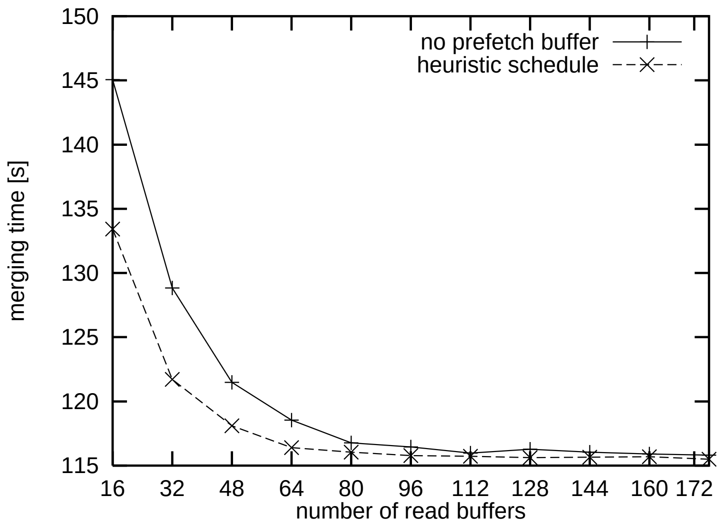

Figure 11: Impact of prefetch and overlap buffers on merging time.

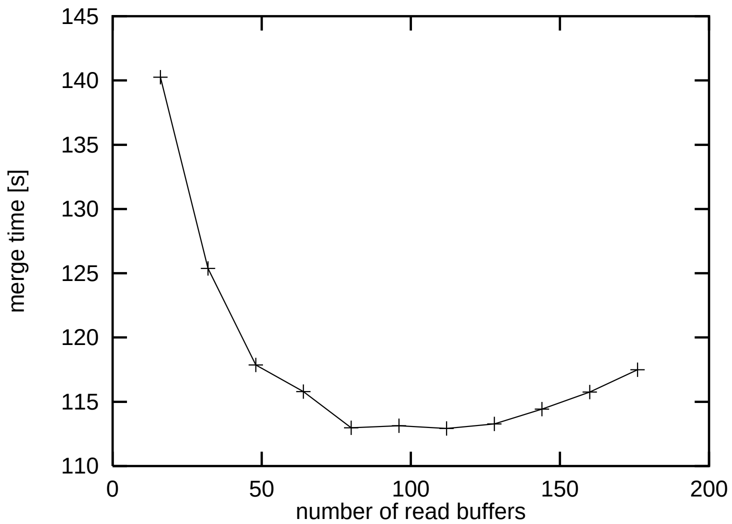

Figure 12: Tradeoff: write buffer size versus read buffer size.

the I/O time decreases as this fraction grows. However, Figure 10 indicates that the overall time needed for merging is best if most of the read buffer is dedicated to overlapping I/O and computation. Only for very small read buffers there is a significant difference between the naive algorithm and optimal prefetching.

In Figure 11 we compare the overall merging time for the naive algorithm and the following heuristics for choosing the prefetch buffer size w as a function of the read buffer size r: $w = 2D + \frac{3}{10}(r - 2D)$. We have not shown the empirically optimal choice because it is very close to this heuristics.

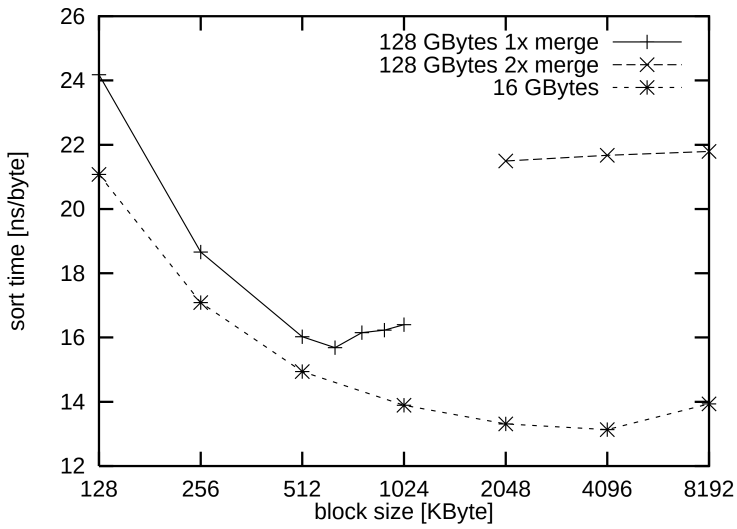

Figure 13: Dependence of sorting time on the block size.

Based on this heuristics for the read buffer, Figure 12 explores the tradeoff between read buffer size and write buffer size given a total buffer size of 188 blocks. Although we see the asymmetry between read buffer size and write buffer size predicted by the theoretical analysis, it turns out that write buffers much larger than $2D$ blocks can be profitable. A likely reason is that a write buffer of size $w = aD$ blocks leads to an effective output block size of $(a-1)B$ thus reducing seek times and perhaps also rotational delays. Based on this observation, we use the following heuristics for the write buffer size in the subsequent figures: $w = \max(t/4, 2D)$ when the total number of buffer blocks available for read and write buffers is t. The total number of blocks available in our measurements is $t = (M - kB)/B$ where $M = 512$ MByte and $k = \lceil 2N/M \rceil$ is the number of runs.

Figure 13 shows the dependence of the execution time on the block size. We see that block sizes of several MBytes are needed for good performance. The main reason is the well known observation that blocks should consist of several disk tracks to amortize seeks and rotational delays over a large consecutive data transfer. This figure is much larger than the block sizes used in older studies because the data density on hard disks has dramatically increased in the last years. This effect is further amplified in comparison to the SCSI disks used in most other studies because modern IDE disks have even higher data densities but larger rotational delays and less opportunities for seek time optimization.

Nevertheless, the largest possible block size is not optimal because it leaves too little room for read and write buffers. Hence, in most measurements we use the heuristics to choose half the largest possible block size that is a power of two.

For very large inputs, Figure 13 shows that we already have to go below the "really good" block sizes because of lack of buffer space. Still, it is not a good idea to switch to two merge passes because the overall time increases even if we are able to stick to large block sizes with more passes. The large optimal block sizes are an indicator that "asymptotically efficient" can also translate into "practically relevant" because simpler suboptimal parallel

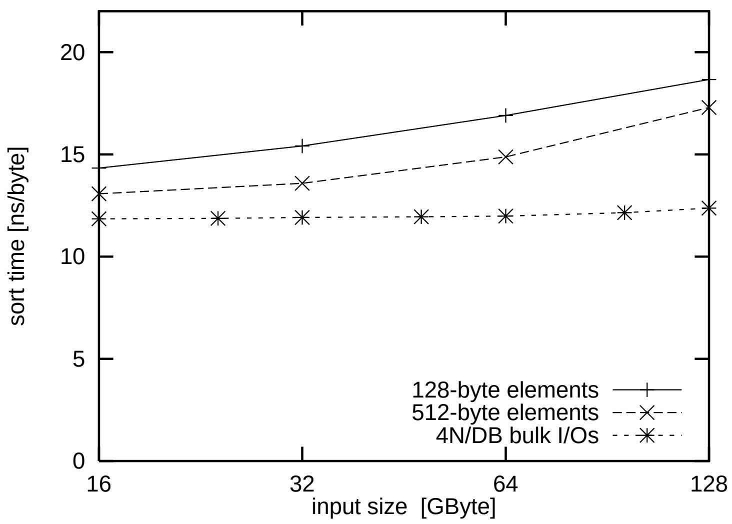

Figure 14: Dependence of sorting time on the input size.

disk algorithms often use logical blocks striped over the disks. On our system this leads to a further reduction of the possible block size by a factor of about eight.

Finally, Figure 14 shows the overall performance for different input size using all the heuristics introduced above. Although we can stick to two passes, the execution time per element goes up because we need to employ slower and slower zones, because the block sizes go down, and because the seek times during merging go up.

6 Discussion

We have engineered a sorting algorithm that combines very high performance on state of the art hardware with theoretical performance guarantees. This algorithm is compute bound although we use small random keys and a tuned linear time algorithm for run formation. Similar observations are likely to apply to all external memory algorithms that exhibit good spatial locality, i.e. those dominated by scanning, sorting, and similar operations. This indicates that bandwidth is no longer a limiting factor for external memory algorithms if parallel disks are used. Furthermore, the low price of our hardware platform indicates that whenever I/O bandwidth it an issue, the price performance ratio can actually *improve* by adding disks.

On the other hand, the fact that it is challenging to sustain peak bandwidth for eight disks on a dual processor system implies that using even more disks requires more aggressive use of parallel processing. Currently it is not clear however how to achieve that in a cost efficient way. Cheap networks with 100Mbit/s Ethernet support only about one fifth the bandwidth of a cheap disk. Even Gigabit Ethernet is not an answer.

Algorithmically, several promising improvements remain even for small cheap machines: There are several ways to speed up run formation for small elements. During merging, it would be good to reduce seek times for large inputs, either by some clever compromise between seek minimization and prefetching, or by switching to distribution sort that can be implemented to have inherently low seek overhead.

As `<stxxl>` will grow beyond the limits of the STL, it is even more important to integrate sorting tightly into the library. As in database systems, good implementations of external memory algorithms move data in a pipelined fashion between various scanning and sorting filters. This pipelining has to be supported in a robust way. For example, we need a memory management that works robustly even if several sorts go on at the same time.

Acknowledgements

We would like to thank Andreas Crauser, David Hutchinson, Lutz Kettner, and Jeff Vitter for valuable discussions. Soumyadeb Mitra and Nitin Rajput implemented a prototype parallel disk sorter during an internship. This experience helped with several design decisions. Our computer support group made several useful recommendations about configuring the machine. Andrew Morton helped with performance aspects of Linux.

References

[1] R. C. Agarwal. A super scalar sort algorithm for RISC processors. In *SIGMOD*, pages 240–247. ACM, 1996.

[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. Albers, N. Garg, and S. Leonardi. Minimizing stall time in single and parallel disk systems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 454–462, New York, May 23–26 1998. ACM Press.

[4] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing i/o-efficient data structures using TPIE. In *10th European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 88–100. Springer, 2002.

[5] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.

[6] R. D. Barve and J. S. Vitter. A simple and efficient parallel disk merge sort. In *11th ACM Symposium on Parallel Algorithms and Architectures*, pages 232–241, 1999.

[7] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[8] G. Chaudhry and T. H. Cormen. Getting more from out-of-core columnsort. In *4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, number 2409 in LNCS, pages 143–154, 2002.

[9] G. Chaudhry, T. H. Cormen, and L. F. Wisniewski. Columnsort lives! an efficient out-of-core sorting program. In *13th ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 2001.

[10] A. Crauser and K. Mehlhorn. LEDA-SM a platform for secondary memory computations. Technical report, MPII, 2000. draft.

[11] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with parallel disks. In *9th European Symposium on Algorithms (ESA)*, number 2161 in LNCS, pages 62–73. Springer, 2001.

[12] M. Kallahalla and P.J. Varman. Optimal prefetching and caching for parallel I/O systems. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 219–228, 2001.

[13] Tracy Kimbrel and Anna R. Karlin. Near-optimal parallel prefetching and caching. *SIAM Journal on Computing*, 29(4):1051–1082, 2000.

[14] D. E. Knuth. *The Art of Computer Programming — Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

[15] P. Larson and G. Graefe. Memory management during run generation in external memory. In *SIGMOD*, pages 472–484. ACM, 1998.

[16] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *5th ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, Velen, Germany, 1993.

[17] M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, 42(4):919–933, 1995.

[18] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC machine sort. In *SIGMOD*, pages 233–242, 1994.

[19] C. Nyberg, C. Koester, and J. Gray. Nsort: A parallel sorting program for NUMA and SMP machines, 2000. http://www.ordinal.com/lit.html.

[20] V. S. Pai and P. J. Varman. Prefetching with multiple disks for external mergesort: Simulation and analysis. In *ICDE*, pages 273–282, 1992.

[21] S. Rajasekaran. A framework for simple sorting algorithms on parallel disk systems. In *10th ACM Symposium on Parallel Algorithms and Architectures*, pages 88–98, 1998.

[22] P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *11th ACM-SIAM Symposium on Discrete Algorithms*, pages 849–858, 2000.

[23] Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[24] J. van den Bercken, B. Blohsfeld J-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL - a library approach to supporting efficient implementations of advanced database queries. In *27th International Conference on Very Large Data Bases*, pages 39–48. Morgan Kaufmann, 2001.

[25] J. van den Bercken, J-P. Dittrich, and B. Seeger. java.XXL: A prototype for a library of query processing algorithms. In *International Conference on Management of Data*, volume 29(2), page 588. ACM, 2000.

[26] D. E. Vengroff. *TPIE User Manual and Reference*. Duke University, 1995. `http://www.cs.duke.edu/~dev/tpie_home_page.html`.

[27] J. S. Vitter and D. A. Hutchinson. Distribution sort with randomized cycling. In *12th ACM-SIAM Symposium on Discrete Algorithms*, pages 77–86, 2001.

[28] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[29] J. Wyllie. SPsort: How to sort a terabyte quickly. `http://research.microsoft.com/barc/SortBenchmark/SPsort.pdf`, 1999.

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2003-2-001	P. Maier	Compositional Circular Assume-Guarantee Rules Cannot Be Sound And Complete
MPI-I-2002-4-002	F. Drago, W. Martens, K. Myszkowski, H. Seidel	Perceptual Evaluation of Tone Mapping Operators with Regard to Similarity and Preference
MPI-I-2002-4-001	M. Goesele, J. Kautz, J. Lang, H.P.A. Lensch, H. Seidel	Tutorial Notes ACM SM 02 A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2002-2-008	W. Charatonik, J. Talbot	Atomic Set Constraints with Projection
MPI-I-2002-2-007	W. Charatonik, H. Ganzinger	Symposium on the Effectiveness of Logic in Computer Science in Honour of Moshe Vardi
MPI-I-2002-1-008	P. Sanders, J.L. Träff	The Factor Algorithm for All-to-all Communication on Clusters of SMP Nodes
MPI-I-2002-1-005	M. Hoefer	Performance of heuristic and approximation algorithms for the uncapacitated facility location problem
MPI-I-2002-1-004	S. Hert, T. Polzin, L. Kettner, G. Schäfer	Exp Lab A Tool Set for Computational Experiments
MPI-I-2002-1-003	I. Katriel, P. Sanders, J.L. Träff	A Practical Minimum Scanning Tree Algorithm Using the Cycle Property
MPI-I-2002-1-002	F. Grandoni	Incrementally maintaining the number of l-cliques
MPI-I-2002-1-001	T. Polzin, S. Vahdati	Using (sub)graphs of small width for solving the Steiner problem
MPI-I-2001-4-005	H.P.A. Lensch, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2001-4-004	S.W. Choi, H. Seidel	Linear One-sided Stability of MAT for Weakly Injective Domain
MPI-I-2001-4-003	K. Daubert, W. Heidrich, J. Kautz, J. Dischler, H. Seidel	Efficient Light Transport Using Precomputed Visibility
MPI-I-2001-4-002	H.P.A. Lensch, J. Kautz, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing, Transmission, and Interactive Display of High Quality 3D Models on the Web
MPI-I-2001-4-001	H.P.A. Lensch, J. Kautz, M. Goesele, W. Heidrich, H. Seidel	Image-Based Reconstruction of Spatially Varying Materials
MPI-I-2001-2-006	H. Nivelle, S. Schulz	Proceeding of the Second International Workshop of the Implementation of Logics
MPI-I-2001-2-005	V. Sofronie-Stokkermans	Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators
MPI-I-2001-2-004	H. de Nivelle	Translation of Resolution Proofs into Higher Order Natural Deduction using Type Theory

MPI-I-2001-2-003	S. Vorobyov	Experiments with Iterative Improvement Algorithms on Completely Unimodel Hypercubes
MPI-I-2001-2-002	P. Maier	A Set-Theoretic Framework for Assume-Guarantee Reasoning
MPI-I-2001-2-001	U. Waldmann	Superposition and Chaining for Totally Ordered Divisible Abelian Groups
MPI-I-2001-1-007	T. Polzin, S. Vahdati	Extending Reduction Techniques for the Steiner Tree Problem: A Combination of Alternative-and Bound-Based Approaches
MPI-I-2001-1-006	T. Polzin, S. Vahdati	Partitioning Techniques for the Steiner Problem
MPI-I-2001-1-005	T. Polzin, S. Vahdati	On Steiner Trees and Minimum Spanning Trees in Hypergraphs
MPI-I-2001-1-004	S. Hert, M. Hoffmann, L. Kettner, S. Pion, M. Seel	An Adaptable and Extensible Geometry Kernel
MPI-I-2001-1-003	M. Seel	Implementation of Planar Nef Polyhedra
MPI-I-2001-1-002	U. Meyer	Directed Single-Source Shortest-Paths in Linear Average-Case Time
MPI-I-2001-1-001	P. Krysta	Approximating Minimum Size 1,2-Connected Networks
MPI-I-2000-4-003	S.W. Choi, H. Seidel	Hyperbolic Hausdorff Distance for Medial Axis Transform
MPI-I-2000-4-002	L.P. Kobbelt, S. Bischoff, K. Kähler, R. Schneider, M. Botsch, C. Rössl, J. Vorsatz	Geometric Modeling Based on Polygonal Meshes
MPI-I-2000-4-001	J. Kautz, W. Heidrich, K. Daubert	Bump Map Shadows for OpenGL Rendering
MPI-I-2000-2-001	F. Eisenbrand	Short Vectors of Planar Lattices Via Continued Fractions
MPI-I-2000-1-005	M. Seel, K. Mehlhorn	Infimaximal Frames: A Technique for Making Lines Look Like Segments
MPI-I-2000-1-004	K. Mehlhorn, S. Schirra	Generalized and improved constructive separation bound for real algebraic expressions
MPI-I-2000-1-003	P. Fatourou	Low-Contention Depth-First Scheduling of Parallel Computations with Synchronization Variables
MPI-I-2000-1-002	R. Beier, J. Sibeyn	A Powerful Heuristic for Telephone Gossiping
MPI-I-2000-1-001	E. Althaus, O. Kohlbacher, H. Lenhof, P. Müller	A branch and cut algorithm for the optimal solution of the side-chain placement problem
MPI-I-1999-4-001	J. Haber, H. Seidel	A Framework for Evaluating the Quality of Lossy Image Compression
MPI-I-1999-3-005	T.A. Henzinger, J. Raskin, P. Schobbens	Axioms for Real-Time Logics
MPI-I-1999-3-004	J. Raskin, P. Schobbens	Proving a conjecture of Andreka on temporal logic
MPI-I-1999-3-003	T.A. Henzinger, J. Raskin, P. Schobbens	Fully Decidable Logics, Automata and Classical Theories for Defining Regular Real-Time Languages
MPI-I-1999-3-002	J. Raskin, P. Schobbens	The Logic of Event Clocks
MPI-I-1999-3-001	S. Vorobyov	New Lower Bounds for the Expressiveness and the Higher-Order Matching Problem in the Simply Typed Lambda Calculus
MPI-I-1999-2-008	A. Bockmayr, F. Eisenbrand	Cutting Planes and the Elementary Closure in Fixed Dimension
MPI-I-1999-2-007	G. Delzanno, J. Raskin	Symbolic Representation of Upward-closed Sets
MPI-I-1999-2-006	A. Nonnengart	A Deductive Model Checking Approach for Hybrid Systems
MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-007	C. Burnikel, K. Mehlhorn, M. Seel	A simple way to recognize a correct Voronoi diagram of line segments

MPI-I-1999-1-006	M. Nissen	Integration of Graph Iterators into LEDA
MPI-I-1999-1-005	J.F. Sibeyn	Ultimate Parallel List Ranking ?
MPI-I-1999-1-004	M. Nissen, K. Weihe	How generic language extensions enable “open-world” desing in Java