DEMSort — Distributed External Memory Sort

Mirko Rahn, Peter Sanders, Johannes Singler[*] Tim Kieritz
Karlsruhe Institute of Technology

Abstract

We present the results of our DEMSort program in various categories of the SortBenchmark. DEMSort is a sophisticated and highly tuned implementation of a merge-sort-based algorithm. It makes use of several libraries to support at the same time distributed-memory parallelism and shared-memory parallelism, in addition to very efficient disk I/O. This makes it excellent for sorting huge volumes of data.

Our timings beat previous result by more than a factor of three, albeit using a much smaller machine.

1 Algorithm

Based on merge sort and a multiway selection algorithm[1], we developed an algorithm that is I/O-optimal up to lower order terms. The detailed algorithm analysis will follow in a more theoretical paper, here we content ourselves with a rough description of the algorithm and our environment used for the SortBenchmark.

The algorithm works in three stages, run formation, distribution of data to the final node, and local merge.

Run formation During run formation, so-called *runs* of the size of the global internal memory are sorted. For each of the R runs, a node first fills the local internal memory with data from the local external storage. This data is then sorted globally (i. e. across all nodes), and written back to the disks. The sequential (but disk-parallel) algorithm that serves as a basis here is described in [2].

Distribution We use a multiway selection algorithm to get the exact split positions in the runs, and redistribute the elements to their final target node accordingly. The transfer volume could we very well larger than the internal memory, but we can communicate from/to internal memory only. Therefore, we split the distribution stage into many rounds, each one fitting into internal memory.

[*]Partially supported by DFG grant SA 933/3-1.

[1]A multiway selection consists of finding the element of a certain global rank among several sorted sequences.

Local Merging After distribution, each node stores R (possibly empty) sorted sequences in its external memory. This sequences are then merged together in a completely node-local operation.

1.1 Analysis

- During run formation, each record is read and written once.
- During the distribution, in the best case input (e. g. random input) only a very small amount of data is redistributed. However, there are worst case inputs for which nearly all records are on the wrong node after run formation. In that case, all data has to be redistributed, so every record is read and written once, again.
- During the local merge, again each record is read and written anyway.

Altogether, the algorithm reads and writes each record between four and six times, which is optimal in the best case.

Of course, there is also communication involved. Again, for best case inputs, a record is communicated in run formation only. For worst case inputs, each record is communicated once more in the distribution stage.

1.2 Output scheme

The output is a sorted permutation of the input records. Each node has one file, nodes with smaller numbers having smaller records. Thus, the concatenation of files in the order of the nodes gives the completely sorted sequence. The output files are created during the sort, they do not exist before.

2 Implementation Details

We have implemented the algorithm in C++ using the STXXL, the standard template library for XXL data sets [1]. This library handles asynchronous block-wise access on local disks[2] in a very efficient way. To sort and to merge data internally, we used routines of the parallel mode in GCC, which exploits multi-core parallelism, and is based on the Multi-Core Standard Template Library [4]. Communication between nodes is done using the message passing interface MPI [3]. Unfortunately, data volumes are specified using 32-bit signed integers in MPI, so no data volume greater than 2 GiB can be passed to MPI-routines. We have re-implemented one of them, namely `MPI_Alltoallv` (basing on ordinary `Isends` and `Ireceives`), to break this barrier.

2.1 Overlapping

In all stages, we overlap disk I/O with internal work and communication: During run formation, for instance, we sort run i while run $i-1$ is written to disk, and run $i+1$ is read from disk, subsequently. Here is room for further improvement as we do not overlap communication and sorting.

Particularly important for the "internal case" (all data fits into the global main memory) is the very tight overlapping of I/O and computation in the run formation stage. We cannot overlap runs as stated above since there is only *one* run. Instead, we first sort each block individually, immediately after been read from disk. When all blocks are fetched, a multiway merge instead of a full sort suffices to get the local data sorted. The global internal sort then further processes this data. As a special optimization for the internal case, we prefill the output file as far as possible while doing the global sort computation and communication. Disk operation is cancelled immediately when the computation is finished. This trick allows for best disk performance even though the output files are newly created. For the external case, this is not useful, since there are no disk idle times thanks to overlapping.

2.2 In-place operation

Our implementation has another valuable feature: It works (nearly) in-place. To sort a certain volume of data, we need only a very small amount of additional external space, independent of the input size. This feature pays off particularly on huge input volumes. If you can store 10^{14} bytes, can you also store 2×10^{14}? We only had less space than 2×10^{14} available.

To keep the data in place we use several tricks:

[2]In our tests, we used a RAID seen as a single disk.

- We read the input files from end to begin and shrink their size accordingly.
- As the result of the run formation, we write many small files (one file per block). Thus, the disk space can be freed later independently of the block access order.
- The distribution uses a newly developed Alltoall algorithm that works (nearly) in-place with these files.
- The small files are read and deleted one after the other during merging, as the program produces a single final output file per node.

3 Results

First, we would like to thank the people from the Steinbuch Centre for Computing in Karlsruhe. Their support was terrific, permitting the great results presented here.

3.1 Machine

The testing machine was a 200-node Intel Xeon Cluster running Suse Linux Enterprise 10 SP 1 with kernel 2.6.22.19. Each node consists of two Quad-Core Intel Xeon X5355 processors clocked at 2667 MHz with 16 GiB main memory and 2 × 4 MiB Cache, and has attached 4 Seagate Barracuda 7200.10 disks. So in total, the machine has 3 TiB of internal and about exactly 200 TB of disk space, but only 117 TiB were usable to us. The typical maximum I/O rate of the disks is about 80 MiB/s for reading as well as for writing (outermost tracks). Most of the disks are of type ST3250820AS, some of them (below 2 %) are of type ST3250310AS, the latter being faster. For the contest runs we selected the 195 fastest out of 198 nodes available to us. The nodes are connected by an 288-port InfiniBand 4xDDR switch. The theoretical point to point bandwidth between two nodes is more than 1300 MiB/s, which goes down to an average of 500 MiB/s when all nodes are communicating.

The compiler was GCC 4.3.1, and the MPI-implementation MVAPICH 1.1.

3.2 Parameters

We used one process per node, i. e. 16 GiB main memory and 4 disks per process, combined as a RAID-0 (striping). The RAID was formatted using XFS with the options `agcount=1` and `unwritten=0`.

We transfer records from/to disk in blocks of a certain size, which is a tuning parameter. In the benchmark runs, each block contained 10×2^{14} records, i. e. about 16 MiB.

In order to use multi-core parallelism each process was allowed to use 4 cores.

3.3 Generation and Verification

We used the program `gensort` from Chris Nyberg to generate the data in parallel. Each node generates its own data, starting with an appropriate offset. During generation we calculated checksums[3] using the routines from libz. Each node stores the number of generated records as well as the checksum. The global checksum is calculated as the sum of all the local checksums and also stored.

After sorting, each node verifies the local order of the records and again calculates checksums and records counts. The checksums and record counts are collected and summed up to form a second global checksum. Additionally each node communicates its first and last record, whose ordering is then also verified.

So we have four things to check for: Whether or not the number of records is the same before and after sorting, whether or not the two global checksums are equal, whether or not all nodes have sorted data and whether or not the connecting records are sorted. If all answers are "Yes", we have good evidence that the output is indeed a sorted permutation of the input.

3.4 Timings

We measured the time from issuing the sort command on the head node until the control flow returns, e. g. `/usr/bin/time mpirun demsort ...` This includes the time to distribute the program itself, the MPI startup time and the MPI finalization time. We ensure that the input file is completely written to disk and that the system disk caches are flushed on startup time. Since the STXXL avoids file caching, we can be sure that the output files are completely written to disk when the control flow comes back.

All timings are given in seconds.

3.4.1 TeraByte

We made four attempts for the now outdated TeraByte category and got the following results:

#	complete time	variance among nodes		
		min	avg	max
1	66.02	56.27	59.61	61.90
2	63.57	56.23	59.26	62.13
3	64.83	56.25	59.39	63.49
4	66.75	56.27	59.54	62.64

[3]According to the rules, we actually sum up the CRC-32 checksums of the individual records.

The *complete time* includes all startup and finalization overhead, the sort time is measured by the program internally. All attempts were done on 192 nodes, sorting 318 blocks each. That is 52,101,120 records per node, summing up to 10,003,415,040 records. The checksum is `12A212A 4122242209`.

Our result for the TeraByte category is:

We have sorted more than 10^{10} records in less than 64 seconds.

3.4.2 Minute

We need less than 64 seconds to sort 10^{10} records. Our hope was to sort 10^{10} records in less than a minute. We tried hard and our program is very well able to do so when using another MPI Implementation. But on the other hand, there is some overhead in distributing and starting the program, which is important for such "small" inputs. After all, we gave up and tried to find out the maximal volume we can sort in less a minute on 195 nodes.

The following table summarizes our attempts:

Blocks			
per node	$\sum$	records	complete time
299	58,305	9,552,691,200	59.99
			63.85
			60.17

Our result for the Minute category is:

We have sorted 9,552,691,200 records in less than a minute.

The checksum is `11CB255282001DCCB`.

3.4.3 GraySort

We made three attempts using 195 nodes. The results are:

#	complete time
1	10,743
2	10,628
3	10,915

All running times are very close to three hours.

We sorted 31,301 blocks per node. That is 5,128,355,840 records per node, summing up to and 1,000,029,388,800 records. The checksum is `746B2FCE425B12DFB2`.

So our result for the GraySort is:

We have sorted more than 10^{12} records in less than 10,628 seconds.

According to the new metric, this reads as:

We have sorted more than 5,645,630,723 records per minute.

We have transferred 4.015 times the size of the input data. The overall transfer rate is $35.19\,\mathrm{GiB/s}$ in total, i. e. $46.19\,\mathrm{MiB/s}$ per disk.

3.5 Daytona or Indy

Our program works according to the Indy rules.

Our algorithm and implementation is generic. However, the record type and the block size is fixed at compile time, and the block size must be a multiple of the record size. So definition of the type and the comparison functor, followed by a recompilation, is needed to sort another type of data. However, this could be automatized, in particular if we restrict ourselves to natural sorting orders (ascending or descending lexicographic). Such a wrapper would induce some additional overhead, which is negligible for huge inputs but not for the TeraByte and the Minute category. After all, we do not have such a wrapper.

There are two more requirements for the Daytona category, namely to demonstrate a run that lasts at least one hour and does not overwrite the input. In our GraySort attempts, we fulfil the first, but we are unable to fulfil the second, since we do not have enough external space on our machine to store both the input and the output file at the same time. However, this is not a problem in principle. The program can be parametrized to keep the input file.

References

[1] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template Library for XXL data sets. *Software Practice & Experience*, 38(6):589–637, 2008.

[2] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, San Diego, 2003.

[3] MPI Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, May 1994.

[4] J. Singler, P. Sanders, and F. Putze. MCSTL: The multicore standard template library. In *13th International Euro-Par Conference*, volume 4641 of *LNCS*, pages 682–694. Springer, 2007.