Data-adapted Parallel Merge Sort

Johannes Holke[1], Alexander Rüttgers[1], Margrit Klitz[1], and Achim Basermann[1]

German Aerospace Center (DLR), Simulation and Software Technology, Department
High-Performance Computing, Linder Höhe, 51147 Cologne, Germany
johannes.holke@dlr.de, alexander.ruettgers@dlr.de, margrit.klitz@dlr.de,
achim.basermann@dlr.de

Abstract. In the aerospace sciences we produce huge amounts of data. This data must be arranged in a meaningful order, so that we can analyze or visualize it. In this paper we focus on data that is distributed among computer processes and then needs to be sorted by a single root process for further analysis. We assume that the memory on the root process is too small to hold all sorted data at once, so that we have to perform the sorting and processing of data chunk-wise. We prove the efficiency of our approach in weak scaling tests, where we achieve a near constant bandwidth. Additionally, we obtain a considerable speed up compared to the standard parallel external sort. We also demonstrate the usefulness of our algorithm in a real-life aviation application.

Keywords: Parallel sorting, High-performance computing, Merge sort, Data analysis, Aerospace sciences

1 Introduction

In the German Aerospace Center (DLR - Deutsches Zentrum für Luft- und Raumfahrt) huge amounts of data arise day by day. On the one hand this data is produced by scientific and engineering simulations, e.g. from the full numerical simulation of an aircraft. On the other hand, lots of data is collected for Earth observation or the exploration of other planets.

Very often the accumulated data needs to be sorted to become useful. In this work, we focus on data that is at first distributed among different processes of a supercomputer but then needs to be processed in a sorted order by a single root process. Furthermore, we assume that the memory on the root process is too small to hold all sorted data at once, so that we have to perform the sorting and processing of data chunk-wise.

The contribution of this paper is as follows: We present a new parallel merge sort algorithm that can skillfully handle arbitrary unsorted data sets that are distributed on a large number of processes. We dynamically adjust the size of the buffer for each process depending on the distribution of data. Compared to a fixed buffer size, we reduce the number of necessary messages and can use the dynamic buffer very efficiently. In particular we optimize the routine to account for pre-sorted parts of the data and for imbalanced loads. In a benchmark study on the JUWELS supercomputer at FZ Jülich [4] we test our algorithm on up to 768 MPI ranks sorting up to 240 GiB of data. In weak scaling tests, we achieve a near constant bandwidth. Compared to the well-known parallel external sort [1,3] we demonstrate an up to 2 times speed-up on randomized data sets and up to 4.6 times on partly sorted data. In the latter case, we reduce the number of MPI messages to a constant that does not depend on the number of processes anymore.

Furthermore, on the DLR internal C^{2}A^{2}S^{2}E-2 cluster [2], we show that introducing our new algorithm in the DLR application code CODA [9] reduces the runtime of the complete mesh output operation by a factor of over 4 in the average case and by a factor of 100 in the (previous) worst case.

The remainder of this paper is organized as follows: In Section 2 we introduce the details of our data-adapted merge algorithm and point out the difference to the parallel external merge algorithm. Section 3 presents benchmark results with different sorting scenarios computed on JUWELS at FZ Jülich [4]. In addition, we focus on a specific case CFD application for which the algorithm was developed. For the scenario of a complete HDF5 export, we also show scaling results. We conclude our paper in Section 4 with a summary and suggestions for further research.

2 Data-adapted Merge Algorithm

Let $I = \{ id_0, \ldots, id_{N-1} \} \subseteq \mathbb{N}_0$ be a set of $N = |I|$ so called Ids (or keys) and $D = \{ d_0, \ldots, d_{N-1} \}$ be a set of N data items. We say that id_j is the Id of item d_j, writing $id_j = Id(d_j)$. The sets I and D are distributed across P processes $\{ p_0, \ldots, p_{P-1} \}$ in an arbitrary order. Thus, for each $0 \leq p < P$ we have a set $I_p \subseteq I$, such that

$$\bigcup_p I_p = I, \text{ and} \tag{1}$$

$$I_p \cap I_q = \emptyset, \text{ for } p \neq q, \tag{2}$$

and each process p holds the index set I_p and the corresponding data set $D_p = \{ d_j \in D \,|\, Id_j \in I_p \}$. We denote the number of items on process p by N_p.

In this paper we are concerned with the following task:

Task 1 *A single designated process r, which we call the* root process, *needs to access all data items in D once in ascending order of their Ids. Thus, first d_0, then d_1, and so on. Furthermore, we assume that due to limited memory resources the number of Ids and data items that r can store simultaneously is bounded and significantly smaller than N.*

Our particular application is in the context of computational fluid dynamics. Here, the Ids correspond to the elements of a computational mesh and the data represents the state of the simulation. The j-th data item is associated with the j-th mesh element and can for example store coordinates, fluid velocities, concentrations, or other relevant data [1]. We then want to output this simulation state to a file, for example to visualize the simulation or to have a checkpoint to restart the simulation in the future. In order to have this output independent of the partition, we want to store our data in order, sorted by the Ids. Furthermore, in some situations we are limited to using serial file I/O through a single process r. Therefore, we are in the setting described by the task above.

Of the different approaches to solve these kinds of sorting problems, we aim for one similar to a parallel merge sort, where the data is sorted locally on the processes and then merged to sorted data on the root[2].

From now on we assume that the processes' local sets D_p and I_p are sorted.

2.1 The parallel external merge sort

The parallel external merge sort algorithm is a common method [3, 7, 10] that proceeds as follows. Given a chunk size C (divisible by P), the root process allocates memory for C many data items and assigns each process a portion of C/P many items. Each process then sends its next C/P items to the root. In a P-way merge step, the root process merges the data from the P buffers into a sorted array of size C, the *output chunk*. As soon as the root has merged all data items from process p a new communication is requested and process p sends its next C/P data items to the root. If the output chunk is full, this chunk of data can be passed on for further processing (serial data analysis, file I/O, etc.) and the root clears the output chunk to sort the next C items; see Figure 1.

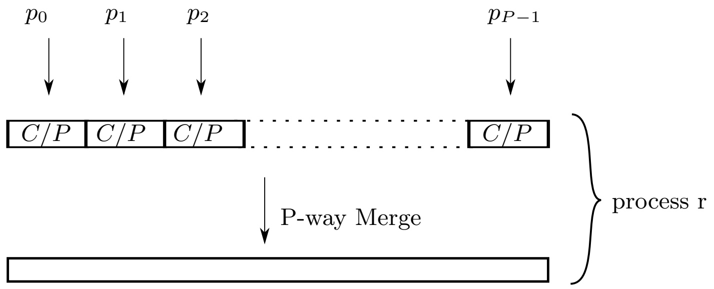

Fig. 1. In the classical external merge sort, the root allocates a buffer of size C/P for each process and receives the current portion of data from each process into this buffer. These buffers are then merged into the sorted output chunk. As soon as all C/P items from one process are merged, this process sends its next C/P items to the buffer.

[1] Data may also reside on a subset of the mesh elements, in which case gaps in the Ids I occur.

[2] A related problem is the so called external sort problem. Here the data resides unsorted on a hard drive and has to be written back to the hard drive in sorted order, while only a limited amount of data can fit into the memory of the calculating process [10].

We observe two drawbacks of this method.

First, the number of messages that each process sends to the root is always the same, regardless of the distribution of the data. We can calculate this number for a process p as the amount of data on p divided by the size of p's buffer on the root, C/P:

$$|\{\, \text{Messages from } p \text{ to } r \,\}| = \frac{N_p P}{C}. \tag{3}$$

In particular, this number increases with the number of processes P.

However, if large portions of the data on a process p are contiguous, we could use much fewer messages. In the extreme case that the data is already sorted on the processes. In this situation the optimal strategy is to send packs of C data items from process 0 to the root until all items on process 0 are processed, then continue with packs of C many items from process 1, and so on. Each process then sends $\frac{N_p}{C}$ messages, a reduction by a factor of P.

The second issue arises during the algorithm if a process p has sent all its elements to the root, but other processes still have data left. In this situation the segment of the root's buffer associated to p is not used in the remaining part of the algorithm, having the same effect as shrinking the buffer size and thus increasing the effective runtime of the algorithm. This effect is of particular interest when the data is not distributed evenly among the processes or, in the extreme case, when some processes do not have any data at all.

2.2 The data-adapted parallel merge

We propose a new data-adapted parallel merge algorithm to overcome these issues. Instead of assigning a fixed buffer on the root for each process we dynamically adjust its size depending on the distribution of the data. Thus, we ensure that the buffer on the root is utilized more efficiently and that fewer messages have to be send for sorted parts of the data.

For a thorough understanding of our approach, we provide pseudo-code in Algorithms 2.1, 2.2, and 2.3: In a setup step '`InitNextID`' each process sends the smallest element of I_p to the root, where these Ids are stored in an array `NextIDs`. During the algorithm `NextIDs[p]` will be updated to always contain the smallest Id on process p for which no data was sent to the root yet.

The algorithm then enters its main loop. In each iteration the root collects data from the other processes to fill the output chunk with the next C many data items. To achieve this, we proceed in three steps.

`UpdateDataRange` (see Algorithm 2.2): The root determines the Ids of the data items that will be processed in this iteration. This range starts at the smallest non-processed Id $m := \min \texttt{NextIDs}$ and ends at $m + C$. This range is then broadcasted across all processes.

`GatherData` (see Algorithm 2.3): In this step, the root collects the data from the processes. Each process whose next Id is within the data range, sends all data in the data range to root. Thus, process p sends d_j if and only if $m \leq id_j < m+C$. In particular, processes with no data in the data range do not send messages

Algorithm 2.1: data_adapted_parallel_merge (ID_Array I_p, Data_Array D_p, Chunk_size C, root r)

```
pos ← 0                                  /* Current position in $I_p$ and $D_p$ */
if $p == r$ then
    Allocate Id and data buffer to hold $C$ items respectively.
    Allocate Output chunk to hold $C$ data items.
InitNextID()
while Data left on any process do
    UpdateDataRange()
    GatherData()
    MergeData()
                 /* Application on $r$ processes data in ouput chunk */
```

to the root. If a process sends data to the root it additionally sends its next unprocessed Id to the root or, if no data is left, an End of data flag.

Algorithm 2.2: UpdateDataRange()

```
if $p == r$ then                                  /* This process is root */
    minId ← min { NextIDs }                          /* Smallest next Id */
    data_range[0] ← minId
    data_range[1] ← minId + $C$
    Broadcast data_range to all other ranks
else
    Receive data_range from root
```

At the end of this routine the root process determines from which processes it receives messages using the information in NextIDs and the data range, and then receives the data into its receive buffer. For these processes the root also updates the NextIDs array. If the sending process is the root itself ($p == r$), we do not need to send an MPI message here, but instead copy the data locally.

MergeData: In GatherData, the root received $k \leq P$ messages of sorted data. These are now merged in a k-way merge step into the output chunk.

After the MergeData step the output chunk represents the next sorted portion of the complete data set and can be processed by the calling application.

In Figure 2 we depict the first two loop iterations of a small example with three processes p_0, p_1, and p_2. The first items of the sets I_p are given as

$$I_0 = \{ 0, 3, 100, \dots \}, \; I_1 = \{ 1, 4, 7, 101, \dots \}, \; I_2 = \{ 5, 6, 110, \dots \}, \qquad (4)$$

and the sets D_p contain the corresponding data items. As chunk size we choose $C = 4$ (Certainly, C would be much larger in realistic applications; see Section 3). We show the Ids and data of each process on the left hand side and the data on

Algorithm 2.3: GatherData()

```
if $I_p[pos] \geq data\_range[1]$ then
    return                                  /* This process does not send data */
Find j, such that $I_p[\text{pos} + j] < \text{data\_range}[1] \leq I_p[\text{pos} + j + 1]$
$S_p^0 \leftarrow \{ I_p[\text{pos}], I_p[\text{pos} + 1], \ldots, I_p[\text{pos} + j] \}$
$S_p^1 \leftarrow \{ D_p[\text{pos}], D_p[\text{pos} + 1], \ldots, D_p[\text{pos} + j] \}$
$\text{pos} \leftarrow \text{pos} + j + 1$                       /* Update position in $I_p$ */
$\text{NextID}_p \leftarrow I_p[\text{pos}]$                          /* The next unused Id */
Send $S_p^0$, $S_p^1$ and $\text{NextID}_p$ to the root process.
if $p == r$ then                                /* This process is root */
    for $q \in \{ \hat{q} \mid NextID[\hat{q}] < data\_range[1] \}$ do
        Receive $S_q^0$ and $S_q^1$ and store into Id- and Data-Chunk.
        Receive $\text{NextID}_q$
        $\text{NextIDs}[q] = \text{NextID}_q$
```

the root on the right hand side of Figure 2. The initial `InitNextID` step is not depicted and was already performed.

We observe that only the processes that hold data of the current requested chunk send data to the root and that additionally the size of the messages may differ. The root receives all data for the current chunk in one go and does not need to wait for multiple sends from the same process.

By determining the minimum of the next Ids, gaps in the Id array can be skipped. Observe that from step 2 to step 3 the requested data range jumps from $[4, 8)$ to $[100, 104)$, since the root knows that there are no Ids in between 8 and 100. However, gaps in the Id range within the currently requested data range lead to less than C items received on the root. We observe this in step 1 where one slot in the chunk remains unused. Nevertheless, this drawback has only minor influence on the runtime of our algorithm as we demonstrate in Section 3.1.

Remark 1. In our description of Task 1 we explicitly assume that different data items have different Ids. We use this in the algorithm when we determine the next data range. It is possible to adapt the algorithm to cope with duplicated Ids if we know a bound $\hat{n}$ on the number of usages of the same Id beforehand. In this case, `UpdateDataRange` may request a range from m to $m + \frac{C}{\hat{n}}$ instead.

3 Results and Discussion

3.1 Data-adapted Parallel Merge

In this section we test our algorithm for four scenarios. The first is a random distribution of Ids, in the second the Ids are sorted, in the third blocks of 10,000 contiguous Ids are randomly distributed, and in the fourth the data is not distributed evenly among the processes and has large gaps (half the processes have twice as much data as the others). The random distributions are generated by

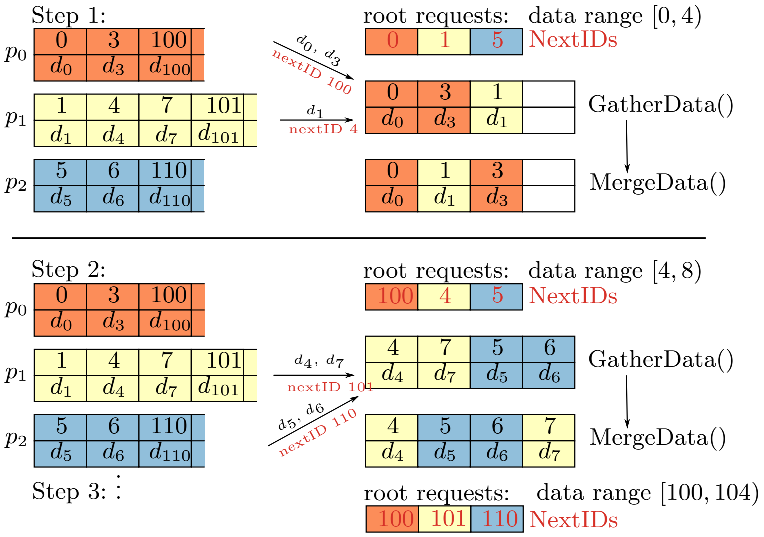

Fig. 2. Graphical description of our proposed data-adapted parallel merge algorithm. We show the first two loop iterations of an example with three processes and chunk size $C = 4$. On the left hand side we depict the Ids and data sets I_p and D_p of the processes. On the right hand side, we show how the root process receives and stores the different messages. Note that the root process will be one of p_0, p_1, p_2.

using the random number generator from [8]. For the results in Table 1 and Figure 3 we have 8.388.608 Ids per process, 4 `double` entries ($4 \times 8 = 32$ Byte) per Id and a chunk size of $C = 32.768$ on the root process. Since we keep the problem size per process constant while increasing the number of processes, this can be seen as a weak scaling study. In a further step, we compare our algorithm with a reference implementation of the external parallel merge sort.

Our results were obtained on JUWELS at FZ Jülich [4]. Each node consists of a Dual Intel Xeon Platinum 8168 with $2 \times 24 = 48$ cores at 2.7GHz each and $12 \times 8 = 96$ GB of RAM.

Table 1 lists the results of our new data adapted parallel merge algorithm with 48 to 768 processes on JUWELS for the first three scenarios. Since the problem size per process is kept constant, the total amount of data to be merged in GiB increases with the number of processes. In particular, we have a 15 GiB per compute node throughout our tests.

If we compare the number of messages per process between our algorithm and the external sorting algorithm in the second column of Table 2, we see that the number of messages per process is distinctly lower for our algorithm than for the external sorting algorithm in the scenarios *Sorted* and *Contiguous*. We achieve this advantage by the dynamic chunk size in our algorithm. In the scenario *Random*, the numbers of messages sent per process are exactly the same for our algorithm and the classical external sorting algorithm. Here, the number of messages is given by Equation (3). For this scenario, the advantage of a dynamic chunk size in our algorithm can not be exploited. In most application use cases, however, we do not expect a totally random distribution of IDs.

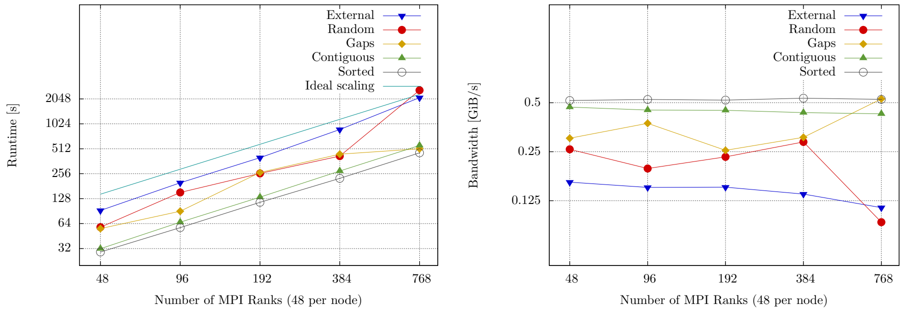

Fig. 3. Runtime (left) and bandwidth (right) for our four test scenarios compared with the parallel external merge.

Figure 3 compares runtimes and bandwidths achieved for all four scenarios between our algorithm and the external sorting algorithm for 48 to 768 processes on JUWELS. In the scenarios *Sorted* and *Contiguous* the runtimes of our algorithm are significantly shorter than the runtimes of the external sorting algorithm. The main reason is the distinctly reduced number of messages per process by exploiting a dynamic chunk size in our algorithm. In addition the number of messages in our algorithm stays nearly constant with increasing process number, cf. Table 2. In the sorted case, we even have an exactly constant number of messages, 256.

		Runtime [s]					Bandwidth [GiB/s]				
# ranks	GiB	External	Random	Sorted	Contiguous	Gaps	External	Random	Sorted	Contiguous	Gaps
48	15	92.3	58.0	29.1	31.9	55.8	0.16	0.26	0.52	0.47	0.30
96	30	198.8	152.3	57.3	66.5	90.3	0.15	0.20	0.52	0.45	0.37
192	60	399.7	258.0	115.6	133.6	264.8	0.15	0.23	0.52	0.45	0.25
384	120	873.8	417.9	225.0	275.8	440.5	0.14	0.29	0.53	0.44	0.31
768	240	2,118.0	2,604.3	457.6	561.7	511.6	0.11	0.09	0.52	0.43	0.53

Table 1. Scaling results for our four test cases *Random*, *Sorted*, *Contiguous*, and *Gaps* compared with the original parallel external sort. We show the runtimes (left) and bandwidth (right) for our experiments for $P = 48$ up to $P = 768$ processes on JUWELS.

Note that we expect a linear increase in the total runtime for both algorithms since the total amount of data rises and all data has to be processed on the root process. This linear increase is indicated by the 'Ideal scaling' line in Figure 3. We also observe a clear advantage of our algorithm in the case of not evenly distributed data, scenario *Gaps*. Here, the dynamic chunk size is of particular

	# messages/proc			
# ranks	External	Random	Sorted	Contiguous
48	12,288	12,288	256	1,060.85
96	24,576	24,576	256	1,078.35
192	49,152	49,152	256	1,086.90
384	98,304	98,304	256	1,089.76
768	196,608	196,608	256	1,093.00

	Runtime [s]	
Chunk size	External sort	Data-adapted
32,768	2,118.0	2,601.0
131,072	1,909.6	1,572.8
262,144	1,866.9	1,358.0

Table 2. Left: Number of messages per process for the external sorting algorithms and our algorithms for the first three scenarios (*Random*, *Sorted*, *Contiguous*). Right: Runtimes for different chunk sizes with 768 MPI ranks of the external sorting algorithms and our proposed algorithm for the *Random* scenario.

advantage, since with increasing process number more and more processes become idle after some iterations due to the load imbalance in this scenario. For the scenario *Random*, our algorithm still shows superior runtime behavior compared with the external sorting algorithm except for 768 processes. In the latter case the overhead of managing the parallel messages is a possible explanation for the slower runtimes. However, if we increase the chunk size as in Table 2, we can also for 768 processes achieve distinctly shorter runtimes with our algorithm than with external sorting algorithm. Larger chunk sizes improve the computation to communication ratio and are advantageous for both algorithms, but can be more efficiently exploited in our data-adapted parallel merge implementation.

Figure 3 displays the bandwidth behavior of our algorithm in comparison to the external sorting algorithm. We observe that with increasing processor numbers the bandwidth stays more or less constant for our algorithm in the scenarios *Sorted* and *Contiguous*, while the bandwidth of the external sorting algorithm decreases. Moreover, the bandwidth of our algorithm is distinctly higher than that of the external sorting algorithm, in the best case by a factor of about 5. The only exception is again scenario *Random* with 768 processes, but as for the runtime this can be changed by adapting the chunk size according to Table 2.

3.2 Application: File I/O with FlowSimulator

One of DLR's ongoing goals in aviation is the virtual design of an aircraft. A key element in the aerodynamic design process is the numerical flow simulation for which the DLR develops its next-generation CFD (computational fluid dynamics) software code CODA [9].

CODA is developed as part of the FlowSimulator (FS), which is an HPC platform for the integration of multiple parallel components into a process chain. All components ("plug-ins") are integrated via a Python interface so that the whole simulation process chain can be controlled by a Python script; see Figure 4, left. For a detailed description of FS, we refer to [11] and [12].

The storage and the parallel management of data in FS is performed by an HPC-library called FlowSimulator Data Manager (FSDM). FSDM stores data in a collection of C++ container classes that are all wrapped to Python. It has a wide range of import and export filters for the most common file formats such

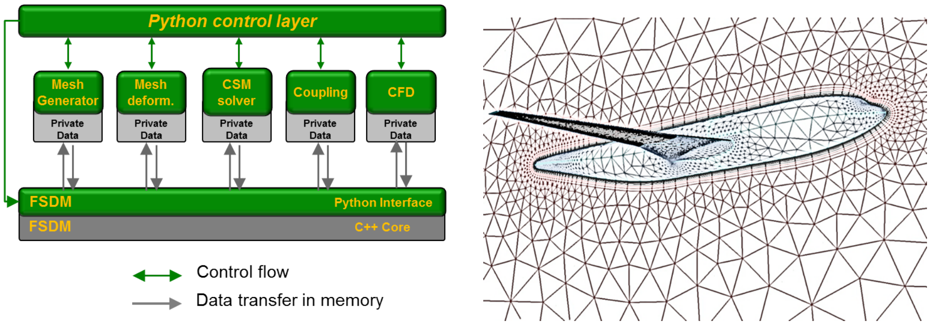

Fig. 4. Left: Basic architecture of the FlowSimulator framework. Right: Illustration of an unstructured grid that is used for a CFD simulation around an airplane.

as HDF5, CGNS, NetCDF and Tecplot. After the import, FSDM decomposes the data and distributes it over the different MPI domains. Here, FSDM makes use of popular partitioning algorithms such as ParMETIS or RGB (Recursive Graph Bisection) [5,6]. Other ingredients of FSDM include geometry operations, mesh deformation and interpolation to only name a few.

Due to the various export formats that are supported by FSDM, we often encounter a situation as described in Task 1 in the case that the export filter only supports sequential file I/O. In the following, we benchmark the file I/O of a CFD simulation into an HDF5 file using FSDM. Note that we are aware of the fact that the HDF5 library [13] supports parallel file I/O. However, the current HDF5 export is performed by the root process only.

In the following, we consider an unstructured mesh that models an airplane as illustrated in Figure 4 and that contains various simulation datasets, e.g. the velocity and the pressure field. The mesh is adaptively refined at the region of interest close to the airplane's wing and consists of nodes, surface elements (triangles, quadrangles) and volume elements (tetrahedrons, hexahedrons). Each mesh element is identified by a unique Id integer number. As an example, the color of the mesh elements in Figure 4 represent their associated Id number. Here, on the one hand the mesh elements that model the airplane have low Ids (colored in blue) and on the other hand the mesh elements of the far field have high Ids (colored in red).

Table 3 shows the results of the HDF5 file export in FS on the DLR $C^2A^2S^2E$-2 [2] cluster. Each cluster node consists of two Intel Xeon E5-2695v2 processors with $2 \times 12 = 24$ cores, 2.4GHz per core and $8 \times 16 = 128$ GiB of RAM. The exported dataset has a size of 7.2 GiB and consists of a mesh with 17 CFD subdatasets that are exported one after another. The table compares the runtime of the original file I/O implementation in FSDM with the proposed new algorithm described in Section 2. In this case, the chunk size of the new algorithm is $C = 10^6$. We explicitly note that file I/O is performed by the root process r in all cases so that the increase in runtime with larger processor numbers is to be expected.

	Runtime [s]		
#ranks	old export	old export + Ids invert.	new export
24	742	19,536	182
48	1,104	32,907	278
96	1,575	34,602	356
192	2,126	-	534
384	2,995	-	803
768	4,524	-	1,052

Table 3. Runtime comparison of the original HDF5 file export with the proposed algorithm in FlowSimulator.

In the old export routine, r performs the data exchange with one process after another and then writes a contiguous block of m data elements $\{\, d_j, d_{j+1}, \ldots, d_{j+m} \,\}$ with m as large as possible to file. Since the dataset D_p on each process p has been sorted locally, we usually obtain contiguous block sizes m in the order of several thousand elements. The runtime results for this case are listed in the second column of Table 3. The third column considers the situation that the local order on each process has been destroyed since we deliberately invert the list of local Ids. This reduces the number of elements m that can be written in one operation by r and increases the total runtime by a factor of 20-30. Due to the enormous increase in runtime, we have only computed the results up to four nodes on $\mathrm{C^2A^2S^2E}$-2. Finally, the last column states the results with our proposed new algorithm. In this case, there is always a local sort on each process so that the runtime results do not depend on the initial local order. We observe that the new implementation reduces the runtime compared to the old export (second column) by a factor of four. This underlines the usefulness of the algorithm for sequential file I/O on moderate processor numbers.

4 Conclusion

In this paper we introduce a new algorithm to solve a parallel sorting problem, where data resides on distributed processes and needs to be accessed by a single root process in sorted order. Due to limited memory resources the root can only access this data chunk-wise. We optimize our network communication to automatically adapt to the data distribution among the processes. Compared to the common parallel external sort approach, we obtain speed-ups of factors 2 to 4. With our method we are able to exploit pre-sorted parts of the data and can handle unbalanced loads.

Additionally to our results in benchmark studies, we applied our approach to sequential file I/O in the DLR FlowSimulator environment. Here, we demonstrated speed-ups of the complete I/O routine of a factor of 4 in the general case and up to 100 in our previous worst case. We are certain that many more applications can benefit from our work, especially in the areas of data-analysis and visualization, and in situations where parts of a tool-chain are serial. Future work on the techniques presented in this paper may include improved handling of duplicated keys and a generalization of the algorithm to multiple root processes. The latter could be promising on clusters with many compute nodes and a limited number of I/O nodes.

Acknowledgments

This research was carried out under the project *Virtual Aircraft Technology Integration Platform* (VicToria) by the German Aerospace Center (DLR).

The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS at Jülich Supercomputing Centre (JSC).

References

1. Bitton, D., DeWitt, D.J., Hsiao, D.K., Menon, J.: A taxonomy of parallel sorting. Tech. rep., Cornell University (1984)
2. CASE-2: Sgi ice x, intel xeon e5-2695v2 12c 2.4ghz, inifiniband, fdr, https://www.top500.org/system/178196, last accessed: April 16, 2019
3. Friedland, D.B.: Design, Analysis, and Implementation of Parallel External Sorting Algorithms. Ph.D. thesis (1981), aAI8206830
4. JUWELS: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers-/JUWELS/JUWELS_node.html, last accessed: April 15, 2019
5. Karypis, G., Schoegel, K., Kumar, V.: ParMETIS – Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.1 (2013)
6. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell system technical journal **49**(2), 291–307 (1970)
7. Knuth, D.E.: The art of computer programming: sorting and searching, vol. 3. Pearson Education (1997)
8. Langr, D., Tvrdík, P., Dytrych, T., Draayer, J.P.: Algorithm 947: Paraperm—parallel generation of random permutations with mpi. ACM Trans. Math. Softw. **41**(1), 5:1–5:26 (Oct 2014). https://doi.org/10.1145/2669372, http://doi.acm.org/10.1145/2669372
9. Leicht, T., Jägersküpper, J., Vollmer, D., Schwöppe, A., Hartmann, R., Fiedler, J., Schlauch, T.: DLR-Project Digital-X - Next generation CFD solver 'Flucs'. In: Deutscher Luft- und Raumfahrtkongress 2016 (Februar 2016), https://elib.dlr.de/111205/
10. Leu, F.C., Tsai, Y.T., Tang, C.Y.: An efficient external sorting algorithm. Information processing letters **75**(4), 159–163 (2000)
11. Meinel, M., Einarsson, G.O.: The FlowSimulator framework for massively parallel CFD applications. In: PARA 2010 conference: state of the art in scientific and parallel computing. Citeseer (2010)
12. Reimer, L.: The FlowSimulator—a software framework for CFD-related multidisciplinary simulations. In: European NAFEMS Conference Computational Fluid Dynamics (CFD) – Beyond the Solve (Dezember 2015), https://elib.dlr.de/100536/
13. The HDF Group: Hierarchical Data Format, version 5 (1997-NNNN), http://www.hdfgroup.org/HDF5/