Parallel Sorting with Minimal Data

Christian Siebert[1,2] and Felix Wolf[1,2,3]

[1] German Research School for Simulation Sciences, 52062 Aachen, Germany
[2] RWTH Aachen University, Computer Science Department, 52056 Aachen, Germany
[3] Forschungszentrum Jülich, Jülich Supercomputing Centre, 52425 Jülich, Germany
{c.siebert,f.wolf}@grs-sim.de

Abstract. For reasons of efficiency, parallel methods are normally used to work with as many elements as possible. Contrary to this preferred situation, some applications need the opposite. This paper presents three parallel sorting algorithms suited for the extreme case where every process contributes only a single element. Scalable solutions for this case are needed for the communicator constructor `MPI_Comm_split`. Compared to previous approaches requiring $O(p)$ memory, we introduce two new parallel sorting algorithms working with a minimum of $O(1)$ memory. One method is simple to implement and achieves a running time of $O(p)$. Our scalable algorithm solves this sorting problem in $O(\log^2 p)$ time.

Keywords: MPI, Scalability, Sorting, Algorithms, Limited memory

1 Introduction

Sorting is often considered to be the most fundamental problem in computer science. Since the 1960s, computer manufacturers estimate that more than 25 percent of the processor time is spent on sorting [6, p. 3]. Many applications use sorting algorithms as a key subroutine either because they inherently need to sort some information, or because sorting is a prerequisite for efficiently solving other problems such as searching or matching. Formally, the *sequential sorting problem* can be defined as follows:[1]

Input: A sequence of n items $(x_1, x_2, \ldots, x_n)$, and a relational operator $\leq$ that specifies an order on these items.

Output: A permutation (reordering) $(y_1, y_2, \ldots, y_n)$ of the input sequence such that $y_1 \leq y_2 \leq \cdots \leq y_n$.

This problem has been studied extensively in the literature for more than sixty years. As a result, many practical solutions exist, including sorting algorithms such as *Merge sort* (1945), *Quicksort* (1960), *Smoothsort* (1981), and *Introsort* (1997). Since there can be $n!$ different input permutations, a correct sorting algorithm requires $\Omega(n \log n)$ comparisons. Some of the previously mentioned solutions achieve a worst-case running time of $O(n \log n)$, which makes them therefore asymptotically optimal.

[1] To avoid any restrictions, this paper focuses on comparison-based sorting algorithms.

Published in "Proc. of EuroMPI 2011", LNCS 6960, pp. 170–177, Springer, 2011.
The original publication is available at www.springerlink.com

Single-core performance has been stagnant since 2002 and with the trend to have exponentially growing parallelism in hardware due to Moore's law, applications naturally demand a parallel sorting solution, involving p processes. We assume that each process can be identified by a unique rank number between 0 and $p-1$. A necessary condition for an optimal parallel solution is that the n data items are fully distributed over all processes. This means that process i holds a distinct subset of n_i data items, so that $n = \sum_{i=0}^{p-1} n_i$. Usually neglected, our paper investigates the extreme case where every process holds exactly one data item, thus n_i is always 1 and $n = p$. This *parallel sorting problem with minimal data* can be formulated as an extension to the sequential sorting problem:

Input: A sequence of items distributed over p processes $(x_0, x_1, \ldots, x_{p-1})$ so that process i holds item x_i, and a relational operator $\leq$.
Output: A distributed permutation $(y_0, y_1, \ldots, y_{p-1})$ of the input sequence such that process i holds item y_i and $y_0 \leq y_1 \leq \cdots \leq y_{p-1}$.

The communicator creator `MPI_Comm_split` in the Message Passing Interface requires an efficient solution for the parallel sorting problem with minimal data. Existing implementations as in *MPICH* [5] and *Open MPI* [4] need $O(p)$ memory and $O(p \log p)$ time just to accomplish this sorting task. This paper offers three novel parallel sorting algorithms as suitable alternatives:

1. An algorithm similar to Sack and Gropp's approach [7] in terms of linear resource complexity. Its advantage is simplicity, making it an ideal candidate to implement `MPI_Comm_split` efficiently for up to $100,000$ processes.
2. A modification of the first algorithm to reduce its $O(p)$ memory complexity down to $O(1)$, eliminating this bottleneck at the expense of running time.
3. A scalable algorithm which also achieves this minimal memory complexity, and additionally reduces the time complexity to $O(\log^2 p)$. Experiments prove this method to be the fastest known beyond $100,000$ processes.

These algorithms represent self-contained parallel sorting solutions for our case. In combination, they resolve all scalability problems for `MPI_Comm_split`.

2 Communicator Construction

MPI is an established standard for programming parallel applications, and is especially suited for distributed-memory supercomputers at large scale. Every communication in MPI is associated with a *communicator*. This is a special context where a group of processes belonging to this communicator can exchange messages separated from communication in other contexts. MPI provides two predefined communicators: `MPI_COMM_WORLD` and `MPI_COMM_SELF`. Further communicators can be created from a group of MPI processes, which itself can be extracted from existing communicators and modified by set operations such as inclusion, union, and intersection. All participating processes must perform this procedure and provide the same full (i.e., global) information, independently

of whether a process is finally included in the new context or not. This way of creating a single communicator is not suited for parallel applications targeting large supercomputers because the resource consumption (i.e., memory and computation) scales linearly with the number of processes in the original group.

The convenience function `MPI_Comm_split` (MPI-2.2 p. 205f) provides an alternative way to create new communicators and circumvents the MPI group concept. It is based on a *color* and *key* input and enables the simultaneous creation of multiple communicators, precisely one new communicator per color. The key argument is used to influence the order of the new ranks. However, such an abundant functionality comes at a cost: any correct implementation of `MPI_Comm_split` must sort these <color, key> pairs in a distributed fashion. Every process provides both integers separately. Internally, they can however be combined into a single double-wide value, such as in `value=(color<<32)+key` for architectures with 32 bit integers, before executing the parallel sorting kernel. The output value together with the original rank number is sufficient for subsequent processing in `MPI_Comm_Split`, as segmented prefix sums (e.g., exemplified in MPI-2.2 p. 182f) can efficiently compute an identifier offset for the new communicator and the new rank number in $O(\log p)$ time and $O(1)$ space.

Open question: Can `MPI_Comm_split` be implemented in a scalable way, in particular with a memory complexity significantly smaller than $O(p)$?

3 Related Work

Parallelizing sorting algorithms has shown to be nontrivial. Although a lot of sequential approaches look promising, turning them into scalable parallel solutions is often complex and typically only feasible on shared-memory architectures [2].

Many popular parallel sorting approaches for distributed memory are based on *Samplesort* (1970) [3]. This algorithm selects (for example at random) a subset of $O(p^2)$ input items called "samples", which need to be sorted for further processing. Unfortunately, methods using this approach do not work for $n < p^2$, and offer as such no solution for our special case with 1 item per process. Even today these samples are still sorted sequentially [8] causing this to be the main bottleneck at large scale. In fact, a scalable solution to the sorting problem with minimal data might even help to eliminate this bottleneck in Samplesort.

Current implementations of `MPI_Comm_split` based on *Open MPI* as well as *MPICH* do not sort the <color, key> arguments in parallel. Instead, they simply collect all arguments on all processes using `MPI_Allgather`, then apply a sequential sorting algorithm, and finally pick the resulting value that belongs to the corresponding process. The ANSI C standard library function `qsort` is used if available, which has an average running time of $O(n \log n)$ but can exhibit the $O(n^2)$ worst case for unfavorable inputs. Both implementations fall back to a slow $O(n^2)$ *Bubblesort* algorithm if *Quicksort* is not provided by the system. This naive approach results in a memory consumption that scales poorly with $O(p)$ and a running time of $O(p \log p)$ or even $O(p^2)$, which is to be avoided.

Sack and Gropp (2010) identified and analyzed this scalability problem for `MPI_Comm_split` in foresight of the exascale era [7]. Asking for a parallel sorting solution, they proposed to utilize the exact splitting method of Cheng and colleagues (2007) [1] to improve upon the scalability of `MPI_Comm_split`. Instead of a single sorting process, they propose to partially gather the input items on multiple sorting roots. The exact splitting method is used to partition the gathered data equally for these roots, which then sort the resulting smaller data sets sequentially. The authors evaluated this intricate approach for up to 64 sorting processes and projected an expected speedup of up to 16, representing communicators for 128 million MPI processes. This limited scaling in the order of $\log n$ already reduces the complexities of `MPI_Comm_split` by a factor of $\log p$ down to $O(p)$ in terms of time and $O(p/\log p)$ in terms of memory. We propose further solutions to this problem in Section 4 to improve upon both complexity terms.

4 Algorithm Designs

All algorithms discussed in this section can be used for the implementation of `MPI_Comm_split`. They expect one input value per process, sort all values in parallel, and return one output value per process, according to the definition of the parallel sorting problem with minimal data in Section 1.

4.1 Sequential Algorithm

Existing implementations of `MPI_Comm_split` simply collect all input values on all processes and do the actual sorting work in a redundant sequential fashion.

```
MPI_Comm_rank(comm, &rank);
tmparray = malloc(sizeof(type)*p);
MPI_Allgather(input, 1, type, tmparray, 1, type, comm);
qsort(tmparray, p, sizeof(type), cmpfunc);
output = tmparray[rank];
free(tmparray);
return output;
```

Listing 1.1. Sequential implementation

The `MPI_Allgather` operation has a time complexity of $O(p)$, but the sequential sorting functionality encapsulated in `qsort` uses $O(p \log p)$ comparisons on average. Therefore the latter becomes the dominating factor in Listing 1.1, leading to an overall time complexity of $O(p \log p)$. A temporary array capable of holding all p input values is needed, resulting in a memory complexity of $O(p)$.

4.2 Counting Algorithm

An interesting observation helps us to remove the redundant executions of `qsort`: It is sufficient to count how many values are smaller or equal than a process' own value as the destination for the input value arises directly from this information.

```
tmparray = malloc(sizeof(type)*p);
MPI_Allgather(input, 1, type, tmparray, 1, type, comm);
dest = -1;
for (i = 0; i < p; i++) { if (tmparray[i] <= input) dest++; }
free(tmparray);
MPI_Sendrecv(input, 1, type, dest, tag, output, 1, type,
             MPI_ANY_SOURCE, tag, comm, status);
return output;
```

Listing 1.2. Counting implementation

This parallel sorting algorithm can be made stable by splitting the loop into two parts and using different comparators. To ignore a process' own value, Listing 1.2 initializes `dest` to -1 instead of 0. After counting, each process knows the corresponding process it has to send its value to. The for loop over p values reduces the total time complexity by a factor of $\log p$ down to $O(p)$. Since this not only makes `MPI_Comm_split` much faster but also simplifies its implementation by removing the dependencies to external functions such as `qsort` and own *Bubblesort* implementations, we recommend immediate integration into MPI libraries. The memory requirements do not change and therefore stay $O(p)$.

4.3 Ring Algorithm with $O(1)$ Memory

When memory requirements become a concern (e.g., with huge number of cores), our counting algorithm can be adapted to avoid additional memory. The idea is to mix the gathering and visiting of all values, so that this can be done in smaller chunks—in the extreme case with only a single value. We created a virtual ring of processes by using `MPI_Cart_create` to embed a one-dimensional and periodic Cartesian topology into the underlying network topology. The convenience function `MPI_Cart_shift` identifies the left and right neighbor in the ring.

```
dest = 0;
prev = input;
for (i = 1; i < p; i++) {
   MPI_Sendrecv(prev, 1, type, left, tag, next, 1, type,
                right, tag, ring_comm, status);
   if (next <= input) dest++;
   prev = next;
}
MPI_Sendrecv(input, 1, type, dest, tag, output, 1, type,
             MPI_ANY_SOURCE, tag, comm, status);
return output;
```

Listing 1.3. Ring implementation with O(1) memory

We utilize $p-1$ iterations in Listing 1.3 to ignore a process' own value. The time complexity remains $O(p)$, although the hidden constant[2] is potentially

[2] The counting solution employs only one `Allgather` which can be implemented to induce $O(\log p)$ network latencies as opposed to $O(p)$ for individual communications.

much higher than in Listing 1.2. Fortunately, only a fixed number of variables are needed, reducing the memory complexity down to the minimum of $O(1)$.

4.4 Scalable Algorithm

While the previous algorithms are simple to understand, we sketch now a more sophisticated approach to solve the parallel sorting problem with minimal data. It is based on the divide-and-conquer concept underlying *Quicksort*:

1. globally select a *pivot* value (preferably close to the median of all elements)
2. *divide*: partition all distributed values into the three sets consisting of (i) values that are less than *pivot*, (ii) values that are equal to *pivot* (important for duplicates and stability), and (iii) values that are greater than *pivot*
3. *conquer*: recursively proceed with the set the process belongs to

Assume an $O(\log p)$ time collective communication operation that returns an element close to the median of all provided values. Each process invokes this functionality with its own input value to get a suitable pivot value in return.

$$\vec{v_i} = \begin{cases} (1\ 0\ 0) & \text{if } x_i < \text{pivot}, \\ (0\ 1\ 0) & \text{if } x_i = \text{pivot}, \\ (0\ 0\ 1) & \text{if } x_i > \text{pivot}. \end{cases} \quad \begin{array}{l} \vec{y_i} = \text{PrefixSum}(\vec{v_i}) \\ \\ \vec{w} = \text{GlobalSum}(\vec{v_i}) \end{array} \quad d_i = \vec{v_i} \cdot \left(\begin{pmatrix} 0\ 0\ 0 \\ 1\ 0\ 0 \\ 1\ 1\ 0 \end{pmatrix} \cdot \vec{w}^T + \vec{y_i}^T \right)$$

Fig. 1. Calculating the new location in the divide step

The partitioning can be accomplished by utilizing parallel reduction operations. Each process compares its own value x_i with the ascertained pivot value. Depending on the outcome, it will initialize an array $\vec{v_i}$ as specified in Figure 1. This information is then processed in a prefix summation (cf. `MPI_Exscan`) and a global summation (cf. `MPI_Allreduce`) to enable a calculation of the new location d_i (i.e., recipient) of each process' value. A data shuffle via `MPI_Sendrecv` concludes a single partitioning round with an overall time complexity of $O(\log p)$.

In the conquer step, a process compares its received value against the pivot to decide where to proceed. This will divide the number of values in roughly two halves, causing $O(\log p)$ divide-and-conquer rounds. To avoid the use of $O(\log p)$ stack space, we implemented this tail-recursive conquer step iteratively. Altogether, the memory complexity is $O(1)$ and the running time becomes $O(\log^2 p)$.

Implementation Details Partitioning leads to subgroups of processes continuing independently in subsequent rounds. Since the algorithm uses collective operations, we could create new communicators. However, existing communicator creation is, with a complexity of $\Omega(p)$, too expensive. Instead, we designed special collective implementations that work on a sub-range of all processes in `MPI_COMM_WORLD`. In contrast to the hardware-tuned Blue Gene/P collectives, these range collectives slow down our scalable sorting method by a factor of roughly 28, but in exchange achieve the required time complexity of $O(\log p)$.

We use an efficient median-of-3 reduction scheme within a complete ternary tree topology to find an approximate median of all values. Each process provides its input value as one of the leaves. Inner nodes receive three values, determine their median, and forward the result to the next level. This procedure is repeatedly applied in $O(\log p)$ levels until the root gets the result. This single value delivers a good approximation of the median because the $2^{\log_3 p} - 1$ smallest as well as $2^{\log_3 p} - 1$ largest values out of $p = 3^k$ values will never be selected. Analysis reveals that a value close to the median is picked with very high probability.

5 Experimental Evaluation

All measurements were carried out on the full *Jugene* system located at the Jülich Supercomputing Centre in Germany. It consists of 73,728 compute nodes, each equipped with 2 GiB of memory and a 4-way SMP PowerPC processor running at 850 MHz. Executables were linked against the BG/P MPI library 1.4.2.

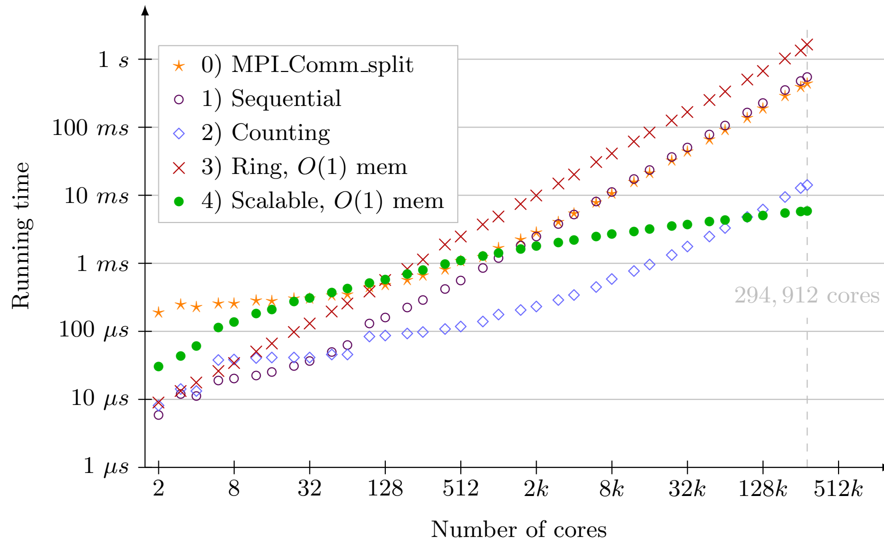

Fig. 2. Performance comparison of the presented algorithms

Figure 2 depicts the runtime of all presented methods for a varying number of cores. Except for the `MPI_Comm_split` operation which used `color=1` and `key=rank` as input, all sorting algorithms started with a randomly chosen 64-bit value per process. Compared to the extracted sorting kernel, the `MPI_Comm_split` operation shows some overhead up to 2048 processes, after which both performance curves converge. Our counting solution is up to 687% faster than all other methods to the point of 98,304 cores. As expected, the ring algorithm is the slowest candidate for larger communicators, but will never run out of memory. Contrary to the other approaches where the running time increases proportionally to the number of cores, the curve of our scalable algorithm flattens. This

makes it the fastest method beyond $100k$ cores, outperforming the current implementation by a factor of 92.2 at full scale. Its performance can be modelled by $t(p) = 17.5 \cdot (\log_2 p)^2\ \mu s$, giving a predicted running time of 12.7 ms for 128 million processes. As such it is a factor of 29.2 faster than Sack and Gropp's best proposed solution while requiring a million times less memory.

6 Conclusion

This paper approaches the problem of parallel sorting with minimal data. Being able to handle a single element per process in a scalable way is crucial for an efficient implementation of the MPI communicator creator `MPI_Comm_split`. Extending the work of Sack and Gropp, we introduced three novel algorithms to solve this problem. Our first approach is similar to their proposed method in terms of resource complexity, but is much simpler to implement and more efficient in practice, making it an ideal candidate for MPI libraries. In prospect to future systems, we reduced the $O(p)$ memory complexity down to the minimum of $O(1)$ at the expense of performance in our second algorithm. Finally, we sketched a scalable algorithm that solves the parallel sorting problem with minimal data. Measurements on the largest Blue Gene/P installation today showed that this method eventually outperforms all other methods, making it 92.2 times faster than current implementations and a hundred thousand times more memory efficient on $294,912$ cores. Since the algorithm's time complexity of $O(\log^2 p)$ yields excellent scalability without any additional memory, it provides a suitable solution to the tackled problem, at and beyond exascale—closing the open question of a scalable `MPI_Comm_split` implementation with a positive answer.

References

1. Cheng, D.R., Shah, V., Gilert, J.R., Edelman, A.: A Novel Parallel Sorting Algorithm for Contemporary Architectures. Tech. rep., University of California (2007)
2. Cole, R.: Parallel Merge Sort. SIAM Journal on Computing 17, 770–785 (1988)
3. Frazer, W.D., McKellar, A.C.: Samplesort: A Sampling Approach to Minimal Storage Tree Sorting. Journal of the ACM 17, 496–507 (1970)
4. Gabriel, Fagg, Bosilca, Angskun, Dongarra, Squyres, Sahay, Kambadur, Barrett, Lumsdaine, Castain, Daniel, Graham, Woodall: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Proc. of the 11th European PVM/MPI Users' Group Meeting. pp. 97–104 (2004)
5. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A High-Performance, Portable Implementation of the Message Passing Interface Standard. Par. Comp. 22, 789ff (1996)
6. Knuth, D.E.: The Art of Computer Programming, volume 3 - Sorting and Searching (2nd ed.). Addison Wesley Longman Publishing, Redwood City, CA, USA (1998)
7. Sack, P., Gropp, W.: A Scalable MPI_Comm_split Algorithm for Exascale Computing. In: Proc. of the 17th European MPI Users' Group Meeting. pp. 1–10 (2010)
8. Shi, H., Schaeffer, J.: Parallel Sorting by Regular Sampling. Journal of Parallel and Distributed Computing 14, 361–372 (1992)