Library Support for Parallel Sorting in Scientific Computations

H. Dachsel[1], M. Hofmann[2,3], and G. Rünger[2]

[1] John von Neumann Institute for Computing, Central Institute for Applied Mathematics, Research Centre Jülich, Germany
[2] Department of Computer Science, Chemnitz University of Technology, Germany

Abstract Sorting is an integral part of numerous algorithms and, therefore, efficient sorting support is needed by many applications. This paper presents a parallel sorting library providing efficient implementations of parallel sorting methods that can be easily adapted to a specific application. A parallel implementation of the Fast Multipole Method is used to demonstrate the configuration and the usage of the library. We also describe a parallel sorting method which provides the ability to adapt to the actual amount of memory available. Performance results for a BlueGene/L supercomputer[4] are given.

1 Introduction

The task of sorting an arbitrary amount of data according to associated key values is an integral part of various algorithms and applications. As a consequence there has been active research in the past resulting in numerous contributions in sequential and parallel sorting [1,2]. Besides computational science in general, efficient implementations of sorting methods are very important in parallel and high performance computing. For instance, an integer sort is part of common benchmarks like NAS Parallel Benchmarks or SPLASH-2. The runtimes spent for sorting in real-world applications are diverse. For instance, in hierarchical N-Body methods sorting may require up to 10 percent [3], whereas it is the major part of the parallel spectral partitioner S-HARP [4]. Thus, the usage of efficient parallel sorting methods can be essential to obtain good parallel implementations.

While optimized and ready-to-use libraries exist for many common tasks in computational science, an appropriate support for parallel sorting is still missing. Only a few approaches like the POSIX routine `qsort` or the integer sort from the Zoltan library [5] using quicksort are available. Both of them are comparison based algorithms and, therefore, they are inappropriate for the common case of integer or floating point number sorting. Advanced implementations of radix

[3] Supported by Deutsche Forschungsgemeinschaft (DFG).

[4] Measurements are performed on the BlueGene/L system at the John von Neumann Institute for Computing, Jülich, Germany. http://www.fz-juelich.de/zam/ibm-bgl

sort methods are far more appropriate for that. But due to their fixed working patterns, they also lack the appropriate flexibility to become widely applicable.

This paper presents the main aspects of the configuration and the usage of a parallel sorting library. Primarily intended for applications in parallel scientific computing, this approach tries to combine two major objectives: (1) the generality of a library approach needed to become widely applicable and (2) the adaption to the actual application to obtain good efficiency. The library is written in C, widely configurable, and capable of adapting to various needs of the particular application or hardware environment. It features a separate configuration step to create new versions of the library routines, especially adapted to a certain application (e.g., the elements to be sorted). We also introduce a parallel radix sort method providing resource awareness in terms of memory usage. It is composed of various algorithms implemented in the library and capable of fulfilling the needs of our sample application. A parallel implementation of the Fast Multipole Method (FMM)[6] currently being developed at the John von Neumann Institute for Computing. Performance results are shown for sorting up to 1 billion elements on a BlueGene/L system, using random integer values as well as real data from the sample application.

The rest of this paper is organized as follows. Section 2 introduces the sample application and its need for efficient sorting support, followed by the description of the parallel sorting method in Section 3. Section 4 presents the main aspects of the library approach illustrated by the application. Section 5 presents performance results of the sorting method in a high scaling parallel environment and Section 6 concludes the paper.

2 Sorting within an FMM implementation

As a sample application requiring efficient parallel sorting support we consider the Fast Multipole Method, an $\mathcal{O}(n)$ hierarchical N-Body algorithm. The specific parallel implementation used is a three-dimensional FMM for calculating classical coulomb interactions. The main input data is a system of n point charges given by coordinates x_i, y_i, z_i and corresponding charge values q_i, $i = 1, \ldots, n$. The result of the computation is the energy E and the gradient G of the system as well as the potentials p_i. Apart from the energy (which is only a single scalar value), the amount of input and output data depends on the number of particles n of the system.

During the computations the system of particles is hierarchically subdivided into boxes which are enumerated according to a space filling curve scheme. Depending on the positions of the particles, each one is located in a certain box and labeled with the corresponding box number. By sorting the particles according to their box numbers, the locality of the subsequent computations is increased resulting in a more efficient processing of the input data. To preserve the initial ordering of the system, the original indices of the particles (addresses) are stored. These address values can be used to restore the initial ordering of the particles by sorting them according to their addresses. This allows for an integration of the FMM implementation as a flexible subroutine in various simulations. Table 1 summarizes the data associated with each particle with the particular data types and sizes.

	particle data	*data types and sizes*	*bytes per particle*
input	positions	3× `double`	24
	charges	1× `double`	8
output	gradient	3× `double`	24
	potentials	1× `double`	8
administrative	box numbers	1× `integer`	8
	addresses	1× `integer`	8

Table 1. Input, output, and administrative data per particle.

Regarding the number of bytes per particle, one can see that the size of the system is limited by the amount of memory available. For example, a system with about 1 billion particles occupies about 64 GB memory only for input/output data and at least 80 GB including the required administrative data. Even though it is possible to perform these computations with a serial implementation in reasonable time, a parallel shared memory system with about 128 GB main memory has to be used because of the memory requirements. To avoid a further limitation of the size of the system to be computed, the parallel sorting method should be able to work with limited memory usage.

3 A parallel radix sort algorithm

The choice of a suitable sorting algorithm is strongly influenced by the requirements of the specific application. Because the FMM is an $\mathcal{O}(n)$ algorithm, it is desirable that the sorting algorithm has time complexity $\mathcal{O}(n)$, too. Due to the big amount of data to be sorted, the algorithm should not rely on the availability of a second fully sized output buffer. Moreover, it should be able to operate *in-place* and to adapt to the actual amount of memory available. These requirements have to be met for the sequential as well as for the parallel case and limit the number of methods available. ZZ-sort [7] for example, as a true parallel in-place sorting method, has a larger time complexity than required. In general, comparison based methods are unsuitable, because they provably require $n \log n$ operations in the worst case [1]. Integer sorting in linear time can be achieved using radix sort methods. But recent parallel radix sort methods [8,9] pay no attention to limited memory usage. Due to probabilistic partitioning strategies and all-to-all communication schemes they can hardly be implemented in-place.

A parallel sorting method meeting the requirements described above is a *merge-based* parallel sorting algorithm as described by Tridgell et al. [10]. Figure 1(a) illustrates the two major steps:

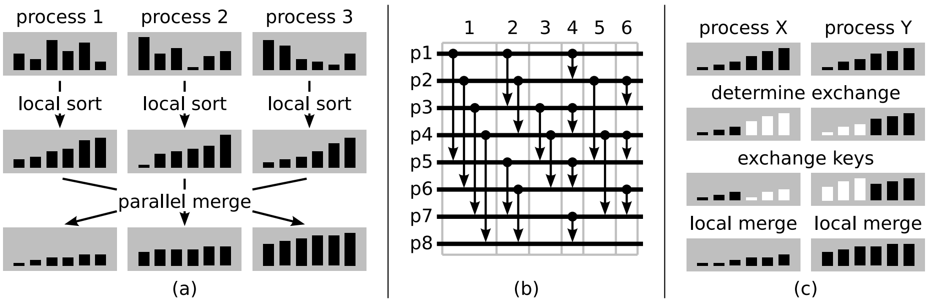

Figure 1. (a) merge-based parallel sorting with 3 processes, (b) batchers-merge-exchange network for 8 processes, (c) one single merge-exchange operation

1. An arbitrary sequential sorting method executed by all processes in parallel creates locally sorted sequences.
2. The sorted sequences are merged in parallel to form the globally sorted order.

For the in-place sequential sorting method a recursive *most-significant-digit-first* radix sort based on *American Flag* sort [11] is used in this paper. In every recursion step, a set of contiguous keys (starting with all keys) is sorted into bins according to a specific part of the bits of the key values. This is repeated with the keys in the single bins using the radix width r as the number of bits processed in one step. The recursion stops if the number of keys in a bin is below a certain threshold value t. The sorting is finished with an algorithm that is faster for small numbers of keys. The time complexity of this sequential sorting method results from the number of exchange operations for every key. Sorting b-bit integers results in a maximum depth of recursion of $\lceil \frac{b}{r} \rceil$. Each key is exchanged in every recursion step and at most t times according to the fast algorithm finishing the sorting. This results in $\lceil \frac{b}{r} \rceil + t$ exchange operations per key and time complexity $\mathcal{O}(n_s(\lceil \frac{b}{r} \rceil + t))$ for sorting a set of n_s keys. The constants r and t have a strong effect on the performance of the sorting method and their optimal values strongly depend on the hardware system.

The parallel merge step is comprised of several single *merge-exchange* operations with two participating processes at a time. These pairs of processes are determined using classical sorting networks like *batchers-merge-exchange* network shown in Figure 1(b) for 8 processes. The arrows represent single merge-exchange operations executed from left to right. The network consists of 6 consecutive stages denoting the maximum number of merge-exchange operations for every process. For a number of p processes, batchers-merge-exchange network consists of $\frac{1}{2}\lceil \log_2 p \rceil (\lceil \log_2 p \rceil + 1)$ consecutive stages [1].

The merge-exchange operation between two processes is shown in Figure 1(c). The exact number of keys to exchange is determined using a bisection method, followed by the exchange of the keys with point-to-point communication (e.g., in-place with `MPI_Sendrecv_replace`). This reduces the merge-exchange operation to two independent local merge operations, one for each process. Table 2 lists several algorithms with varying memory requirements implemented for the local merge. Depending on the size of the subsequences (n_0 and n_1, $n_0 + n_1 = n_s$) to be merged and the amount of memory available the fastest one is chosen.

algorithm	*space*	*time*
Two-way Merge [1]	$\mathcal{O}(\min(n_0, n_1))$	$\mathcal{O}(n_0 + n_1)$
Tridgell & Brent [10]	$\mathcal{O}(\sqrt{n_0 + n_1})$	$\mathcal{O}(n_0 + n_1)$
Huang & Langston [12]	$\mathcal{O}(1)$	$\mathcal{O}(n_0 + n_1)$

Table 2. Space and time complexity of algorithms implemented for the local merge.

With each of these algorithms, one merge-exchange operation has time complexity $\mathcal{O}(n_s)$. In every stage of the sorting network, the processes execute their merge-exchange operations in parallel resulting in $\mathcal{O}(n_s \lceil \log_2 p \rceil^2)$ for the parallel merge step. A total number of n keys, equally distributed over p processes results in $n_s = \frac{n}{p}$ keys per process. For the overall parallel sorting method, consisting of local sorting and parallel merging, this results in time complexity $\mathcal{O}(\frac{n}{p} \lceil \log_2 p \rceil^2)$.

4 A parallel sorting library approach

The previous description shows that efficient and suitable parallel sorting is an expensive and complex task. Instead of a one-fits-all method there is a need for a variety of efficient implementations of different algorithms. For example, the parallel sorting method described in Section 3 consists of (1) a sequential sorting algorithm, (2) a sequential merge algorithm, (3) a sorting network, and (4) functions combining these parts in an appropriate way. The main purpose of the sorting library presented in this paper is to provide implementations of algorithms that can be easily used as or assembled to complete parallel sorting methods.

The sorting library supports sorting of generic elements consisting of a key component and associated data components. Each of these components can be organized in a separate array with a distinct location in memory. A list of elements is fully specified by the number of elements and the base addresses of the single component arrays. The sequential parts of the library rely on simple memory access only, while the parallel routines involve calls to MPI. A distributed list of elements is comprised of the arbitrarily sized local lists of processes participating in a parallel operation, where a global order is determined by their ranks within a given MPI communicator.

Because sorting a list of elements mainly consists of comparing and copying elements, it is necessary to perform these two operations as efficiently as possible. Since this relies on application specific properties (e.g., the type of the elements to be sorted), the sorting library features a separate configuration step before compile time to create new versions of the library especially adapted to the element type of the current application. The library functions can handle the key and data components simultaneously. Besides the instruction to copy a single key value additional instructions are inserted to copy the associated data components as well. While the library is widely configurable, this approach is far more efficient than a configuration at runtime with user-defined copy/comparison functions or many conditional statements. The code generated by the configuration step is less parametrized providing good conditions for compiler optimizations.

4.1 Configuration of the library

The configuration of the library is made by creating header files with appropriate definitions of preprocessor symbols (macros). At least one main header file is necessary to specify the type of the elements to be sorted. Listing 1.1 shows a small example of a configuration for sorting elements consisting of the input data of the particles from Table 1 according to their associated box numbers.

Listing 1.1. Sample configuration (`input.h`)

```
 1  #define SL_USE_MPI

 3  /* key section */                   /* box numbers */
 4  #define sl_key_type_c               long
 5  #define sl_key_type_mpi             MPI_LONG
 6  #define sl_key_size_mpi             1
 7  #define sl_key_integer

 9  /* data section */
10  #define SL_DATA0                    /* positions */
11  #define sl_data0_type_c             double
12  #define sl_data0_size_c             3
13  #define sl_data0_type_mpi           MPI_DOUBLE
14  #define sl_data0_size_mpi           3

16  ...
```

With `SL_USE_MPI` (line 1) the MPI based parallel parts of the library are enabled. The key component is specified with the `sl_key_...` symbols (lines 4-7) defining the appropriate C and MPI data types and sizes. The symbol `sl_key_integer` is used to signal an integer based key and enables routines like radix sort requiring bitwise operations on key values. Similar symbols are used to specify up to four associated data components that have to be rearranged in the same way as the key value they belong to. For example, the positions of the particles are located in a separate array (three consecutive `double` values per particle) and have to be rearranged together with the box numbers during the sorting. Symbol `SL_DATA0` (line 10) enables an associated data component and `sl_data0_...` (lines 11-14) are used to define the particular data types and sizes. Additional symbols can be used to adapt the comparison operation of key values or the copy operation for a certain component.

A second header file is used to define algorithm specific parameters that may depend on the configuration or the actual hardware. In this header file, performance critical parameters, like the radix width r or the threshold value t

for the radix sort method from Section 3, can be adapted. The exact values can be specified manually, for example, derived from runtime observations or specific knowledge about the hardware system.

By calling a separate configuration script, a new instance of the library is created incorporating the given configuration. This new version of the library contains code that corresponds directly to the specific type of the elements to be sorted. To support multiple elements with different types in a single application, it is possible to create multiple configurations in separate header files. For each one (e.g., `input.h` and `output.h`) a separate version of the library is created. Identifiers, like library function names etc., are prefixed with the name of the configuration file (e.g., `input_...` and `output_...`) to distinguish between the different library versions.

4.2 Assembling the parallel sorting method

After the configuration there exist one or several versions of the sorting library, each one especially adapted to a certain kind of elements to be sorted. The main interface for all library functions processing a list of elements is a structure called `elements`. It contains a field (`size`) holding the number of elements in the list and appropriate fields for the memory addresses of the key (`keys`) and the data components (`data0`, `data1`, ...). To distinguish between different kinds of elements, the name of the structure is prefixed too (e.g., `input_elements`, `output_elements`).

Because the FMM application from Section 2 is written in Fortran and, therefore, unable to support call-by-value or the `elements` structure, it is necessary to create appropriate wrapper functions in C. Listing 1.2 shows a sample routine implementing the parallel sorting method from Section 3.

Listing 1.2. Sample routine assembling the parallel sorting method

```
 1  #include "sl_input.h"

 3  void sort_input(long *n, long *box, double *xyz,
 4                  double *q, void *m, long *ms)
 5  {
 6    int size, rank;
 7    input_elements s, sx;

 9    MPI_Comm_size(MPI_COMM_WORLD, &size); MPI_Comm_rank(MPI_COMM_WORLD, &rank);

11    input_elements_alloc_from_block(&sx, m, *ms);

13    s.size = *n;     /* number of elements */
14    s.keys = box;    /* box numbers */
15    s.data0 = xyz;   /* positions */
16    s.data1 = q;     /* charges */

18    input_sort_radix(&s, NULL, -1, -1, -1);

20    input_mpi_mergek(&s, input_sn_batcher, NULL,
21      input_merge2_memory_adaptive, &sx, size, rank, MPI_COMM_WORLD);
22  }
```

In line 1 the automatically generated header file from the sample configuration in Section 4.1 is included. The parameters of the routine (`n`, `box`, `xyz`,

q) are used to initialize the number of elements as well as the key and the data components within an `input_elements` structure named `s` (lines 12-15). The merge-based parallel sorting is done by calling the local radix sort method (line 17) followed by the parallel merge step (lines 19-20) using batchers-merge-exchange network (`input_sn_batcher`). Two more parameters (`m`, `ms`) of the routine are used to allocate a second list of elements (line 10) serving as temporary buffer for the merge algorithms. Depending on the size of the subsequences to be merged and the amount of memory available, the given merge operation (`input_merge2_memory_adaptive`) selects one of the algorithms from Table 2 at runtime.

5 Performance results

For demonstrating the efficiency of the parallel sorting algorithms implemented in the sorting library, we present performance results on a BlueGene/L system [13]. The BlueGene/L supercomputer is a massively parallel architecture with up to 65,536 dual-processor nodes and a peak performance of 360 teraflops. The system features several networks including a 3D torus interconnect for low-latency (100 ns per hop), high-bandwidth (175 MB/s per link and direction) point-to-point communication. Each node possesses two 700 MHz PowerPC based processors (one dedicated for communication) and 512 MB main memory.

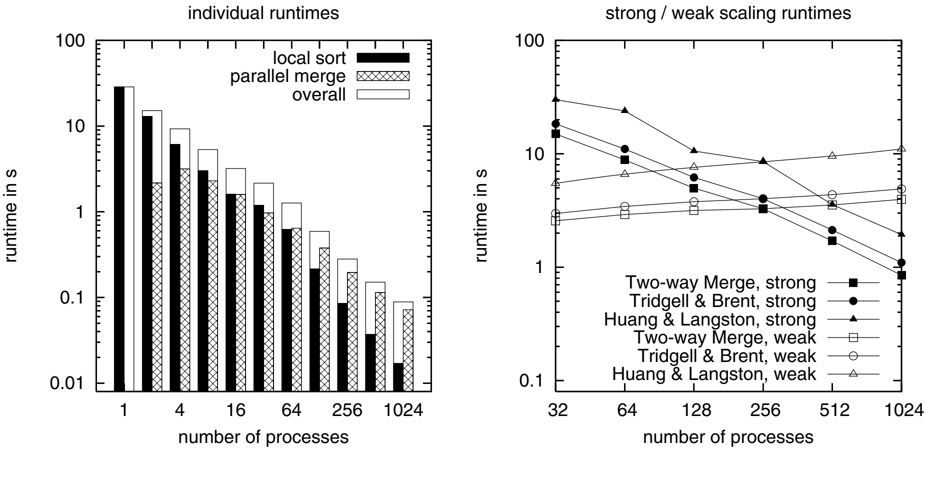

Figure 2. Runtimes for parallel integer sorting.

Figure 2 shows the results for sorting equally distributed random integer values. On the left, individual runtimes for the local radix sort and the parallel merge (with the two-way merge algorithm) are shown using 125 million values (maximum problem size on a single node). The runtimes of both parts of the parallel sorting method decrease for an increasing number of processes. For smaller numbers of processes the most time consuming part is the local radix sort, while for an increasing number of processes the overall runtime is more and more dominated by the parallel merge part. Scaling this problem size to 1024 processes, we obtain a parallel efficiency of about 32%.

Figure 2 (right) shows runtimes for strong scaling (constant problem size) with a total number of 2^{30} values and weak scaling (constant problem size per process) with 2^{22} values per process. In general, one can see that the parallel sorting method scales well. The strong scaling runtimes constantly decrease while for weak scaling there is a moderate increase resulting from the growing number of stages in the sorting network. Regarding the runtimes of the different merge algorithms used, a clear dependency on the memory usage exists. The fastest parallel sorting is done with the two-way merge algorithm requiring the biggest amount of memory. Using the algorithm of Tridgell et al., the runtimes slightly increase, while with the algorithm of Huang et al., the parallel sorting is at least two times slower. The same applies to the parallel efficiency when going from 32 to 1024 processes. While with the first two algorithms the efficiency remains above 50%, it falls below 40% with the algorithm requiring almost no additional memory.

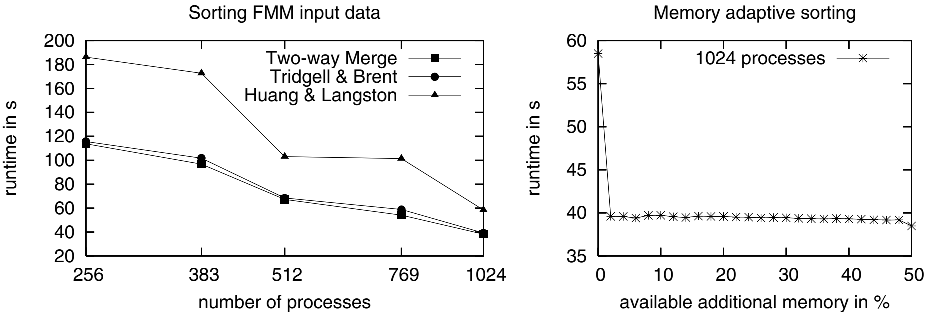

Figure 3. Runtimes for parallel sorting within the FMM application.

Figure 3 (left) shows runtime results for sorting the input data of the FMM application, using the configuration from Listing 1.1. Runtimes are added up for 10 subsequent sorting steps within one FMM calculation with 2^{30} particles. Sorting data within this application confirms the good scaling results as well as the dependency on the memory usage. Figure 3 (right) shows runtimes for varying amounts of memory available. Only in the worst case, where almost no additional memory is available, the runtime is very high. With already 2% of additional memory (100% corresponds to a second fully sized output buffer) the runtimes drop to a lower level and another slight decrease occurs only at 50%. Thus, the memory adaptive parallel sorting method provides complete sorting in the worst case while maintaining good performance in all other cases.

6 Summary

In this paper, we have presented a parallel sorting library, which adapts sorting algorithms to the types of the elements to be sorted and the amount of memory available. With this library we have assembled a parallel radix sort method consisting of various algorithms implemented in the sorting library. We have used an FMM application to exemplify the main aspects of the configuration and the usage of the sorting library. To provide good performance for very different applications, new versions of the sorting library can be created automatically in a separate configuration step before compile time. Each of them is especially adapted to a certain kind of elements to be sorted. For sorting random data as well as real data from our sample application, we have achieved good scaling results and have demonstrated the ability of the sorting method to adapt to the amount of memory available.

References

1. Knuth, D.E.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley (1973)
2. Akl, S.G.: Parallel Sorting Algorithms. Academic Press, Inc. (1990)
3. Hu, Y., Johnsson, S.L.: A data-parallel implementation of O(N) hierarchical N-body methods. In: Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing. (1996)
4. Sohn, A., Simon, H.: S-HARP: a scalable parallel dynamic partitioner for adaptive mesh-based computations. In: Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM). (1998)
5. Devine, K., Boman, E., Heapby, R., Hendrickson, B., Vaughan, C.: Zoltan data management service for parallel dynamic applications. Computing in Science and Engineering **4**(2) (2002) 90–97
6. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. Journal of Computational Physics **73** (1987) 325–348
7. Zheng, S.Q., Calidas, B., Zhang, Y.: An efficient general in-place parallel sorting scheme. The Journal of Supercomputing **14**(1) (1999) 5–17
8. Jiménez-González, D., Navarro, J.J., Larriba-Pey, J.L.: Fast parallel in-memory 64-bit sorting. In: ICS '01: Proceedings of the 15th international conference on Supercomputing. (2001) 114–122
9. Lee, S.J., Jeon, M., Kim, D., Sohn, A.: Partitioned parallel radix sort. Journal of Parallel and Distributed Computing **62**(4) (2002) 656–668
10. Tridgell, A., Brent, R.P.: A general-purpose parallel sorting algorithm. International Journal of High Speed Computing (IJHSC) **7**(2) (1995) 285–302
11. McIlroy, P.M., Bostic, K., McIlroy, M.D.: Engineering radix sort. Computing Systems **6**(1) (1993) 5–27
12. Huang, B.C., Langston, M.A.: Practical in-place merging. Communications of the ACM **31**(3) (1988) 348–352
13. Adiga, N.R., et al.: An overview of the BlueGene/L supercomputer. In: Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing. (2002) 1–22