1-1-2014

Algorithm 947: Paraperm-parallel generation of random permutations with MPI

Daniel Langr
Czech Technical University in Prague

Pavel Tvrdík
Czech Technical University in Prague

Tomáš Dytrych
Louisiana State University

Jerry P. Draayer
Louisiana State University

Recommended Citation

Langr, D., Tvrdík, P., Dytrych, T., & Draayer, J. (2014). Algorithm 947: Paraperm-parallel generation of random permutations with MPI. *ACM Transactions on Mathematical Software, 41* (1) https://doi.org/10.1145/2669372

Algorithm 947: Paraperm—Parallel Generation of Random Permutations with MPI

DANIEL LANGR and PAVEL TVRDÍK, Czech Technical University in Prague
TOMÁŠ DYTRYCH and JERRY P. DRAAYER, Louisiana State University

An algorithm for parallel generation of a random permutation of a large set of distinct integers is presented. This algorithm is designed for massively parallel systems with distributed memory architectures and the MPI-based runtime environments. Scalability of the algorithm is analyzed according to the memory and communication requirements. An implementation of the algorithm in a form of a software library based on the C++ programming language and the MPI application programming interface is further provided. Finally, performed experiments are described and their results discussed. The biggest of these experiments resulted in a generation of a random permutation of 2^{41} integers in slightly more than four minutes using 131072 CPU cores.

Categories and Subject Descriptors: G.2.1 [**Discrete Mathematics**]: Combinatorics—*Permutations and combinations*; G.3 [**Probability and Statistics**]: *Random number generation*; G.4 [**Mathematical Software**]: *Algorithm design and analysis*; *parallel and vector implementations*

General Terms: Algorithms, Design

Additional Key Words and Phrases: C++, distributed memory, implementation, MPI, parallel computing, random permutation

ACM Reference Format:
Langr, D., Tvrdík, P., Dytrych, T., and Draayer, J. P. 2014. Algorithm 947: Paraperm—Parallel generation of random permutations with MPI. *ACM Trans. Math. Softw.* 41, 1, Article 5 (October 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2669372

1. INTRODUCTION

Suppose that we need to generate a random permutation of a set of integers $\{0, \ldots, n-1\}$ on a massively parallel computer system using the MPI parallel programming

This work was supported by the Czech Science Foundation under Grant No. P202/12/2011, by the U.S. National Science Foundation under Grant No. OCI-0904874, and by the U.S. Department of Energy under Grant No. DOE-0904874. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. The authors acknowledge the Louisiana Optical Network Initiative (LONI) and the Louisiana State University (LSU) Center for Computation & Technology for providing HPC resources. The access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the program Projects of Large Infrastructure for Research, Development, and Innovations (LM2010005) is acknowledged. The access to the CERIT-SC computing and storage facilities provided under the program Center CERIT Scientific Cloud, part of the Operational Program Research and Development for Innovations, reg. no. CZ. 1.05/3.2.00/08.0144 is acknowledged.
Authors' addresses: D. Langr (corresponding author) and P. Tvrdík, Czech Technical University in Prague, Department of Computer Systems, Faculty of Information Technology, Thákurova 9, 160 00, Praha, Czech Republic; T. Dytrych and J. P. Draayer, Louisiana State University, Department of Physics and Astronomy, Baton Rouge, LA 70803, USA; email: langrd@fit.cvut.cz.

DOI: http://dx.doi.org/10.1145/2669372

library. From our point of view, an implementation of an algorithm providing such a functionality would ideally require a valid MPI communicator and a permutation size n as input parameters and would return randomly permuted integers $\{0, \ldots, n-1\}$ distributed in local arrays of MPI processes as an output. Having such an "ideal implementation", one should be able to obtain large-scale random permutations with minimal expended effort.

To our best knowledge, such an implementation has been neither described in the literature nor published as a piece of software, considering not only the MPI library, but also any other message-passing library or any other distributed-memory runtime environment.

Several algorithms for parallel generation of random permutations have been developed. Some of them were designed for shared-memory runtime environments only, mostly for their abstract model called the *parallel random-access machine* (PRAM) [JaJa 2011]. These algorithms are based on one of four different techniques, referred to as *shuffling*, *dart throwing*, *integer sorting*, and a *network simulation*. Miller and Reif [1985, Section 4.2] present a dart-throwing PRAM algorithm. Reif [1985] shows a PRAM algorithm based on integer sorting. Czumaj et al. [1998] presents network-simulation based PRAM algorithms. Hagerup [1991] describes three different PRAM algorithms based on the dart throwing, integer sorting, and shuffling technique, respectively. Anderson [1990] presents a parallel implementation of a shuffling algorithm [Durstenfeld 1964; Knuth 1997] for a *small parallel machine*, which represents a more realistic model of a shared-memory machine with a small number of processors (4–30) in comparison with PRAM. Some of the previously mentioned algorithms were discussed and empirically compared by Cong and Bader [2005]. The common feature of all the mentioned algorithms is that all processors generally need access to some global data structure, which makes these algorithms impractical for distributed-memory runtime environments, since their straightforward implementation would introduce a significant communication overhead.

There have also been developed several distributed-memory algorithms for parallel generation of random permutations. Goodrich [1997] shows a distributed algorithm designed for a *bulk synchronous parallel* (BSP) abstract computer model [Tiskin 2011]. This algorithm is, in fact, an adaptation of the integer-sorting based PRAM algorithm of Reif [1985], wherein the comparison-based optimal bulk-synchronous sorting algorithm [Goodrich 1996] is utilized.

Sanders [1998] provides a distributed-memory algorithm and also mentions its SIMD implementation, but only for small and limited permutations—particularly for cases where the permutation size equals the number of processors. Sanders also proves that his algorithm generates permutations with a uniform distribution, that is, each permutation is generated with the same probability. The Sanders' algorithm is discussed more in detail in Section 2.2.

Lassous and Thierry [2000] present three different algorithms for the *coarse grained model* (CGM) [Dehne et al. 1993]. The first one called *independent choice of processors* is practically identical to the previously mentioned Sanders' algorithm. The second algorithm called *division by packets* ensures equal distribution of data among processors, that is, perfect load balancing. However, this algorithm does not generate permutations with a uniform distribution (it cannot generate all possible permutations if applied only once). Finally, the third algorithm called *point-to-point communications* was designed to be efficiently implemented by point-to-point communication operations.

Perhaps a permutation generator nearest to our requirements is provided as a part of the parXXL software library [Gustedt et al. 2007], which supports MPI as a communication interface. However, this library was not primarily designed to generate

random permutations in parallel and there is no simple application programming interface (API) available for it. The usage of parXXL might hence require deep understanding of its documentation and source code. Moreover, only small-scale results on permutation generation—up to 440 processors—has been reported for this library [Gustedt 2008].

We have designed an algorithm for parallel generation of random permutations called Paraperm. We present this algorithm in terms of the MPI parallel programming library [Gropp et al. 1998; Snir et al. 1998]—the mainstream communication library for distributed-memory parallel programming.

We have not developed the Paraperm algorithm entirely from scratch. Instead, we elaborated the previously mentioned Sanders' algorithm—which he defined in a highly abstract way—up to the details of particular communication operations and particular data structures with the aim of maximizing the overall performance and scalability of the generation process. (We chose the Sanders' algorithm since it seems, according to the previous analysis, to be the most prospective algorithm for our needs.) Furthermore, we implemented the proposed algorithm in a form of a software library—which we called Paraperm as well—written in the C++ programming language on top of the MPI API. We show that utilizing our library, its users can embed the parallel generation of random permutations into their programs by adding only several lines of code.

The structure of this article is organized as follows. In Section 2, we show the Paraperm algorithm and discuss its properties including space and communication complexity. In Section 3, we describe the Paraperm software library and mention some implementation details related mainly to the memory management and random number generation. In Section 4, we present the performed experiments and analyse their results. We aim especially at the scalability with regard to the permutation size and to the number of processing units. Finally, concluding remarks are presented in Section 5.

2. ALGORITHM DESIGN

By a *permutation* of a set X, we mean a bijection from X to itself. Having a permutation π, we can write

$$\pi(x) = x', \quad x \in X, \quad x' \in X.$$

Then, we say that x' is a *permutation value* for the *argument* x, or, conversely, that x is an argument for the *permutation value* x'. We call the elements of X *permutation elements* or simply *elements* if the context is clear.[1]

2.1. Problem Definition

Let $X = \{0, \ldots, n-1\}$ and let $\pi : X \to X$ denote an arbitrary permutation of the elements of X. Let $P_0, \ldots, P_{N-1}$ denote particular MPI processes, which we will further call simply *processors*. Let $perm_r$ denote an integer array local to processor P_r.

We are looking for an algorithm that generates a random permutation π such that, on the algorithm output, $perm_r$ contains permutation values

$$\pi(pos_r), \ldots, \pi(pos_r + count_r - 1). \tag{1}$$

The resulting permutation values are hence distributed to the local $perm_r$ arrays of particular processors, where pos_r determines the argument of the first local permutation value while $count_r$ determines their count.

[1]Note that permutation values (and their arguments) are permutation elements as well. Each permutation element corresponds to a particular permutation value, thus, these terms may be exchangeable in some cases.

ALGORITHM 1: Sanders' algorithm

```
1 for each processor P_r do in parallel
2     send each element from {rn/N, ..., (r+1)n/N − 1} to a random processor
3     store incoming elements in [t_0, ..., t_{k−1}]
4     randomly permute [t_0, ..., t_{k−1}] locally
5     Δ ← Σ_{i<r} k_i                                   /* with k_i equal k at P_i */
6     put t_j into P_{r'} at position Δ − nr'/N + j, where r' = ⌊N(Δ + j)/n⌋
```

Moreover, we require that:

(1) the permutation values are stored in $perm_r$ in a successive manner, that is,

$$perm_r[i] = \pi(pos_r + i) \quad \text{for} \quad r \in 0, \ldots, N-1 \quad \text{and} \quad i = 0, \ldots, count_r - 1; \tag{2}$$

(2) each permutation value is stored exactly once, that is,

$$\bigcup_{r=0}^{N-1} \{perm_r[i] : i \in 0, \ldots, count_r - 1\} = X \quad \text{and} \quad \sum_{r=0}^{N-1} count_r = n.$$

Note that:

— if we refer to data local to a particular processor, we add the processor number as a subscript to the data name;
— we use the 0-based indexing for processor numbers and array indexes.

2.2. Sanders' Algorithm

The idea of the Sander's algorithm is simple—permutation elements are first randomly placed among processors, then they are locally permuted (shuffled), and finally their placement is balanced so that each processor has the same amortized number of elements on output. Sanders [1998] presents his algorithm using only 6 lines of pseudocode and he proves, by Theorem 1, that every possible permutation is generated with probability $1/n!$.

Algorithm 1 shows the original Sanders' algorithm written in our notation.[2] We may observe that the algorithm is presented at a very high level of abstraction. Communication between processors is carried out at line 2 as well as at line 6 (in case that $r' \neq r$), however, Sanders gives no details about particular communication operations to be involved (he only suggests to use an "all-to-all exchange" for large n). Due to randomness, processors do not know how many permutation elements they will actually receive at line 2 and correct termination of this process might be complicated and prone to deadlocks.

2.3. Paraperm Algorithm

We present the pseudocode of the proposed Paraperm algorithm as Algorithms 2–5. The pseudocode is designed with the following assumptions.

(1) We suppose that *comm* is a valid MPI communicator and that $n > 0$.
(2) We suppose that all MPI functions are performed over the *comm* communicator.
(3) We adopt the naming conventions for MPI functions and their parameters as they were introduced by Snir et al. [1998] for the C programming language binding.
(4) We emphasize MPI functions using a typewriter font face.

[2]Sanders [1998] also accompanies his algorithm with the following footnote: "Throughout the paper we omit tedious rounding where it does not give additional insight."

ALGORITHM 2: Paraperm algorithm

Input: *comm*, *n*
Output: *perm*, *pos*, *count*

```
1  for all processors do in parallel
2  |  N ← integer variable                                   /* number of processors */
3  |  call MPI_Comm_size(): size ← address of N
4  |  r ← integer variable                                   /* actual processor number */
5  |  call MPI_Comm_rank(): rank ← address of r
6  |  m ← ⌈n/N⌉          /* amortized number of permutation elements per processor */
7  |  pos ← rm                          /* argument of the first local permutation value */
8  |  count ← m                        /* number of local permutation values */
9  |  if (r + 1)m > n then count ← n − pos                   /* trim at the end if necessary */
10 |  if pos ≥ n then count ← 0
11 |  execute PHASE1                                         /* random distribution of elements */
12 |  execute PHASE2                                                          /* local shuffle */
13 |  execute PHASE3                                         /* final balancing of elements */
14 end
```

(5) We use the following syntax for MPI function calls:

$$\texttt{MPI\_Function()} : argument_1 \leftarrow value_1,\ argument_2 \leftarrow value_2,\ \ldots,$$

where, for sake of readability, we specify only arguments relevant to the algorithm.
(6) We suppose that all data are local to particular processors. We hence omit the processor-number subscript for data names in the pseudocode as well as in the accompanying text if the context is clear.

We split the Paraperm algorithm into three phases that match the original concept of the Sanders' algorithm as follows.

— Within PHASE1, permutation elements are randomly distributed among all processors.
— Within PHASE2, local permutation elements are shuffled by each processor.
— Within PHASE3, the placement of permutation elements is balanced among all processors.

Algorithm 2 only computes some basic algorithm parameters and then executes particular phases, which are defined by Algorithms 3–5. We further assume that all variables introduced by Algorithm 2 have global scope, that is, that they are available within the pseudocode of the phases as well.

A simple worked example using a very small data set is provided as an electronic appendix to this article; readers may find this helpful in understanding the workings of the Paraperm algorithm.

2.3.1. PHASE1. The task of all processors in this phase is to distribute mutually exclusive portions of permutation elements so that each element is sent to a randomly selected destination processor. Algorithm 3 shows the pseudocode of this phase, where, generally, processor P_r distributes elements $rm, \ldots, (r+1)m - 1$.

To achieve maximum performance, it is important to avoid sending elements one-by-one using a point-to-point communication operation. All the elements that target the same destination processor should be sent at once and therefore, they first need to grouped together. We resolved this problem using the following scenario.

(1) Permutation elements to be send are put into the *sendbuf* array.

ALGORITHM 3: Paraperm algorithm: PHASE1

```
1  sendbuf ← integer array of size count + 1                    /* permutation elements */
2  destprocs ← integer array of size count + 1                  /* destination processor numbers */
3  for k ← 0 to count − 1 do
4  |  sendbuf[k] ← pos + k                                      /* permutation element */
5  |  destprocs[k] ← random number from 0,...,N − 1             /* destination processor number */
6  end
7  sendbuf[count] ← 0
8  destprocs[count] ← N                                         /* terminator */
9  sort both sendbuf and destprocs at once according to destprocs in ascending order
10 sendcnts ← zero-initialized integer array of size N          /* send counts */
11 k ← 0
12 for r′ ← 0 to N − 1 do
13 |  while r′ = destprocs[k] do
14 |  |  sendcnts[r′] ← sendcnts[r′] +1
15 |  |  k ← k + 1
16 |  end
17 end
18 release destprocs                                            /* no longer needed */
19 sdispls ← zero-initialized integer array of size N           /* send displacements */
20 for r′ ← 1 to N − 1 do
21 |  sdispls[r′] ← sdispls[r′ − 1] +sendcnts[r′ − 1]
22 end
23 recvcnts ← integer array of size N                           /* receive counts */
24 call MPI_Alltoall(): sendbuf ← address of sendcnts[0], sendcount ← 1, recvbuf ← address
   of recvcnts[0], recvount ← 1
25 rdispls ← zero-initialized integer array of size N          /* receive displacements */
26 for r′ ← 1 to N − 1 do
27 |  rdispls[r′] ← rdispls[r′ − 1] +recvcnts[r′ − 1]
28 end
29 total ← 0                                                    /* total number of elements to be received */
30 for r′ ← 0 to N − 1 do
31 |  total ← total + recvcnts[r′]
32 end
33 temp ← integer array of size total                           /* receive buffer */
34 call MPI_Alltoallv(): sendbuf ← address of sendbuf[0], sendcounts ← address of
   sendcnts[0], sdispls ← address of sdispls[0], recvbuf ← address of temp[0],
   recvcounts ← address of recvcnts[0], rdispls ← address of rdispls[0]
35 release sendbuf, sendcnts, sdispls, recvcnts, and rdispls   /* no longer needed */
36 call MPI_Barrier() to synchronize processors
```

(2) Random destination processor numbers for these elements are generated and put into the *destprocs* array.
(3) Both *sendbuf* and *destprocs* arrays are sorted at once in ascending order using the values of *destprocs* as sort keys.

This process is performed within Algorithm 3 at lines 1–9. The role of the *terminator* (lines 7–8) is to simplify further manipulation with the *sendbuf* and *destprocs* arrays at lines 12–17.

When the permutation elements are grouped together in the *sendbuf* array, they can be distributed all at once using a variable-length all-to-all collective communication operation (variable-length since each source processor sends, generally, a different number of elements to each destination processor). In MPI, such communication

ALGORITHM 4: Paraperm algorithm: PHASE2

```
1 if size of temp > 1 then
2 |   for k ← (size of temp) − 1 downto 1 do
3 |   |   l ← random number from 0, ..., k
4 |   |   swap temp[k] and temp[l]
5 |   end
6 end
7 call MPI_Barrier() to synchronize processors
```

operation is represented by the `MPI_Alltoallv()` function. However, prior to its execution at line 34, the following four auxiliary arrays local to each processor need to be constructed:

— the *sendcnts* array containing the number of elements to be sent to each destination processor (lines 10–17),
— the *sdispls* array containing the displacements of elements to be sent to each destination processor (lines 19–22),
— the *recvcnts* array containing the number of elements to be received from each source processor (lines 23–24),
— the *rdispls* array containing the displacements of elements to be received from each source processor (lines 25–28).

To fill up the *recvcnts* array, one fixed-length all-to-all communication operation needs to be executed (line 24), which is in MPI represented by the `MPI_Alltoall()` function.

The randomly distributed permutation elements are finally received by each processor to its local array called *temp*, which is prepared at lines 29–33.

The side effect of grouping the permutation elements and sending them using a single communication operation is that they are, generally, placed in the *temp* arrays in different order than they would be placed in case of sending them one-by-one. However, this does not break the overall randomness of the algorithm, since these elements are then randomly shuffled in PHASE2, which results (or should result) in each possible permutation of the elements of *temp* with the same probability independently of their initial order.

*2.3.2. P*HASE2. The second phase of Paraperm is presented as Algorithm 4. In this phase, all processors randomly permute (shuffle) elements in their temporary arrays *temp*. We use the *modern shuffling algorithm*—known also as the *Knuth algorithm*—for this task [Durstenfeld 1964; Knuth 1997]. However, any other algorithm with the same functionality may be utilized here as well.

*2.3.3. P*HASE3. At the end of PHASE2, the resulting permutation has already been generated, however, the placement of its values is generally imbalanced. Processor P_r has, in its local array $temp_r$, permutation values

$$\pi(first_r), \ldots, \pi(first_r + size_r - 1),$$

where

$$size_r = \text{size of } temp_r \quad \text{and} \quad first_r = \sum_{r'=0}^{r-1} size_{r'}. \tag{3}$$

We request that, according to (1), on output of the Paraperm algorithm, processor P_r has, in its output array $perm_r$, permutation values

$$\pi(pos_r), \ldots, \pi(pos_r + count_r - 1), \tag{4}$$

ALGORITHM 5: Paraperm algorithm: PHASE3

```
1  size ← size of temp
2  first ← integer variable
3  call MPI_Scan(): sendbuf ← address of size, recvbuf ← address of first, count ← 1,
   op ← MPI_SUM                                   /* parallel prefix sum over size */
4  first ← first − size                           /* argument of the first local permutation value */
5  last ← first + size − 1                        /* argument of the last local permutation value */
6  r′ ← ⌊first/m⌋                                 /* destination processor for the first-th permutation value */
7  first′ ← first                                 /* argument of the first permutation value that belongs to P_r′ */
8  remains ← count                                /* number of permutation elements to process */
9  perm ← integer array of size m                 /* final array for permutation values */
10 requests ← empty dynamic array of MPI_Request elements
11 request ← variable of type MPI_Request
12 buf ← integer array of size 2
13 repeat
14     last′ ← (r′ + 1)m − 1                      /* argument of the last perm. value that belongs to P_r′ */
15     if last′ > last then last′ ← last          /* argument of the last perm. value to be sent to P_r′ */
16     count′ ← last′ − first′ + 1                /* number of permutation elements to be sent to P_r′ */
17     if r = r′ then                             /* copy instead send to myself */
18         for k ← first′ to last′ do
19             perm[k − pos] ← temp[k − first]
20         end
21         remains ← remains − count′             /* count′ elements processed */
22     else
23         buf[0] ← first′
24         buf[1] ← count′
25         call MPI_Isend(): buf ← address of buf[0], count ← 2, dest ← r′, tag ← 1,
           request ← address of request
26         append request into requests
27         call MPI_Isend(): buf ← address of temp[first′ − first], count ← count′, dest ← r′,
           tag ← 2, request ← address of request
28         append request into requests
29     end
30     r′ ← r′ + 1                                /* next destination processor */
31     first′ ← first′ + count′                    /* count′ elements processed */
32 until first′ > last                            /* until all elements from temp have been sent or copied */
33 while remains > 0 do                           /* receive sent elements */
34     status ← variable of type MPI_Status
35     call MPI_Recv(): buf ← address of buf[0], count ← 2, source ← MPI_ANY_SOURCE,
       tag ← 1, status ← address of status
36     first′ ← buf[0]
37     count′ ← buf[1]
38     call MPI_Recv(): buf ← address of perm[first′ − pos], count ← count′,
       source ← status.MPI_SOURCE, tag ← 2
39     remains ← remains − count′                 /* count′ elements processed */
40 end
41 for k ← 0 to size of requests − 1 do            /* complete unblocking sends */
42     call MPI_Wait(): request ← address of requests[k]
43 end
44 release requests and temp                      /* no longer needed */
45 call MPI_Barrier() to synchronize processors
```

where the values of pos_r and $count_r$ are defined at lines 6–10 of Algorithm 2 (see Section 2.4 for a discussion about the choice of these paremeters). The process that achieves such a requested state is presented as Algorithm 5. In the most general sense, it may be carried out using the variable-length all-to-all communication operation. However, let us suppose that:

— we use a decent-quality random number generator throughout the algorithm,
— we generate so-called *large-scale permutations* defined by the following condition[3]:

$$n \gg N > 1. \tag{5}$$

Then, we may assume that all processors receive in PHASE1 similar number of permutation elements, which further implies that:

— the sizes of the $temp_r$ arrays are approximately the same for all processors (we verified this assumption by experiments as discussed in Section 4);
— after PHASE1 (and PHASE2), most permutation elements are already stored in local memories of the processors, where they finally belong to;
— we need to transfer in PHASE3 among processors minority of permutation elements;
— processors need to communicate only with their neighbors.

For these reasons, it would be inefficient to use the `MPI_Alltoallv()` function here, since we would need, in addition, to allocate and fill four auxiliary arrays of size N. We hence decided to accomplish this phase using point-to-point communication operations as described in this article. However, to achieve maximum performance, it is important to send all elements that target the same processor at once, that is, not to send elements one by one.

Algorithm 5 works as follows. The *first* and *size* values from (3) are calculated at lines 1–4, which determine permutation values stored in the *temp* array of each processor. According to (3), a prefix sum operation is involved here, which is available in MPI through the `MPI_Scan()` function (line 3).

Then, the algorithm iterates over the *temp* array (lines 13–32) and collects all permutation elements that target the same destination processor (lines 14–16). If the destination processor is equal to the actual processor ($r = r'$), then the collected permutation elements are simply copied into the corresponding position in the output *perm* array (lines 18–20). Otherwise, these elements are sent to the destination processor (line 27). However, on the destination processor, the information about how many elements will be received and where to put them in the output *perm* array is unknown. Hence, prior to sending the elements themselves, this positional information is sent at lines 23–25. Within this phase, all processors generally need both to send and to receive elements, therefore, a nonblocking sending operation is utilized here, which is represented in MPI by the `MPI_Isend()` function.

Finally, after all elements from the *temp* arrays have been processed (either copied locally or sent to another processor), the elements that target the actual processor are received (lines 33–40). This process terminates as soon as the *perm* array contains all permutation elements that correspond to the desired permutation values (1), which is checked via the *remains* variable.

The last thing that needs to be done is to wait for the completion of all the executed nonblocking sending operations (lines 41–43).

[3]It would make no sense to use the Paraperm algorithm if this condition is not met.

2.4. Distribution of Permutation Values

When the algorithm completes, processor P_r has, in its array $perm_r$, permutation values

$$\pi(rm), \ldots, \pi\big((r+1)m-1\big),$$

where

$$m = \lceil n/N \rceil. \tag{6}$$

Such uniform distribution has a major advantage—not only do we know which permutation values are stored on a particular processor, but we also know where a particular permutation value is located. Namely,

$$\pi(i) = perm_r[j], \quad \text{where} \quad r = \lfloor i/m \rfloor \quad \text{and} \quad j = i \bmod m. \tag{7}$$

The downside of this approach is that some processors may end up with no permutation values at all. Substituting i with $n-1$ in (7), we get the last processor that stores some permutation values P_{last}, where

$$last = \left\lfloor \frac{n-1}{\lceil n/N \rceil} \right\rfloor. \tag{8}$$

Thus, when $last < N-1$, none of the processors $P_{last+1}, \ldots, P_{N-1}$ store any permutation values—we call these processors *free*. We can analyze the number of free processors by expressing n as

$$n = kN + l, \quad \text{where} \quad l \in \{0, \ldots, N-1\} \quad \text{and} \quad k \geq 0, \tag{9}$$

and substituting (9) into (8). In the case when N divides n exactly ($l = 0$) there are no free processors at all. Otherwise ($l \neq 0$), the number of free processors is given by

$$N - 1 - last = \left\lfloor \frac{N+1-l}{k+1} \right\rfloor - 1,$$

which implies the following.

— There are no free processors if $k + l \geq N$.
— There are no free processors if $k \geq N-1$, independently of l. This condition is also met when $n \geq N^2$.
— The number of free processors is maximized when $l = 1$ and k approaches 0. However, the latter condition is contrary to the requirement $k \gg 1$ needed to satisfy (5).

The overall conclusions is that the larger permutation we generate, the less processors will be free.[4]

For illustration, we evaluated the average number of free processors for all $n \in \{100N, 100N+1, \ldots, N^2-1\}$ and for the selected numbers of processors N. The results are presented in Table I.

Note that if users are not satisfied with our distribution of permutation values to processors, they may easily redistribute these values more evenly to all the processors $P_0, \ldots, P_{N-1}$ by exploiting the same procedure that was used in PHASE3.

2.5. Complexity

The evaluation of memory, communication, and computational complexity of the Paraperm algorithm is an ambiguous task, because these complexities depend on

[4]The proof of the mentioned statements is beyond the scope of this text. However, it may be found easily by substitution of (9) into (8).

Table I. The Average Number of Free Processors for $n \in \{100N, 100N+1, \ldots, N^2-1\}$

N	Absolute average number of free processors	Relative average number of free processors [%]
10^3	0.82	$8.20 \cdot 10^{-2}$
10^4	1.86	$1.86 \cdot 10^{-2}$
10^5	2.99	$2.99 \cdot 10^{-3}$

Table II. Memory Requirements per Processor in Integer Units of the Paraperm Algorithm for its Particular Phases

Phase	Memory requirements per processor
PHASE1	$2n/N + 4N$
PHASE2	n/N
PHASE3	$2n/N$

randomness and each algorithm execution generally results in different numbers. We hence further suppose that:

(1) we have an ideal random number generator,
(2) we perform the *large-scale* runs of the algorithm defined by condition (5),
(3) consequently all elements are distributed (almost) evenly to all processors in PHASE1.

We evaluate the memory requirements of data structures as well as lengths of transfered messages in *integer* units, that correspond to permutation elements (and/or processor numbers). This approach makes the presented numbers independent of the actual data types used within implementations.[5] Further, we omit the integer units if the context is clear.

We express all the presented quantities using n and N, since these are the input parameters of the algorithm.

2.5.1. Memory Requirements. The memory requirements of the Paraperm algorithm are presented in Table II. However, note the following remarks.

— We present memory requirements using order-of-magnitude numbers rather than exact numbers. We do not use the O-notation here, since *hidden constants* are important and would be lost in that case.
— We suppose that the sorting algorithm running in PHASE1 is in-place and does not need any additional memory.
— We do not make any allowance within these calculations for memory that might be allocated by the MPI library.

The overall memory requirements per processor are hence determined by PHASE1, where we need two arrays of size n/N and four arrays of size N for the execution of the `MPI_Alltoallv()` function.

2.5.2. Communication Complexity. The numbers of elements transferred by MPI routines for each phase of the algorithm are presented in Table III. As in Section 2.5.1, we present these as order-of-magnitude quantities.

[5]Translation into bytes can be done simply by multiplying the numbers by the byte-size of the actually used data types.

Table III. Amortized Number of Transferred Elements of the Paraperm Algorithm for Its Particular Phases

Phase	Number of transferred elements
PHASE1	$n - n/N$
PHASE2	0
PHASE3	≥ 0

In PHASE1, all elements need to be transferred to destination processors, except those that target the processor they are sent from. In PHASE2, no elements are transferred at all, since only a local shuffling procedure is performed over the *temp* arrays. The number of transferred elements in PHASE3 varies for each execution of the algorithm, since it depends on the particular random processor numbers generated within PHASE1. The communication complexity for PHASE3 can thus only be evaluated statistically. The results of our experiments are presented in Section 4 (let us state that during all our numerical experiments the number of elements transferred in PHASE3 was always less than $0.01n$, that is, $< 1\%$ of their total count).

Note that, in addition to the transmission of permutation elements, the following communication operations are also performed by the algorithm:

— a fixed-size all-to-all communication operation over the arrays of size N in PHASE1,
— a parallel prefix sum in PHASE3,
— synchronization barriers at the end of each phase.

2.5.3. Computational Complexity. The computationally expensive operations are the following.

— Generation of random destination processor numbers for n/N permutation elements in PHASE1. The computational complexity of this task is $O(n/N) \cdot O(\varphi)$, where $O(\varphi)$ represents the computational complexity of generating a single random number. The term $O(\varphi)$ is implementation-dependent since it is determined by the particular random number generator used within a particular implementation of the algorithm.[6]
— Local sorting of n/N permutation elements and their destination processor numbers in PHASE1. We assume this has computational complexity $O\big(n/N \cdot \log(n/N)\big)$.
— Local shuffling of approximately n/N permutation elements in PHASE2. The computational complexity of this task is also $O(n/N) \cdot O(\varphi)$.

We state in advance that our experiments presented in Section 4 indicate that these computationally intensive tasks contribute very little to the overall run time of the algorithm and they become negligible for the generation of very large permutations.

2.6. Load Balancing

There are two aspects that can cause a potential load imbalance among processors. The first one is deterministic and is given by the chosen type of the final distribution of permutation values among processors, which was discussed in detail in Section 2.4. We showed that this type of imbalance is generally insignificant if the large-scale condition (5) is satisfied, and it vanishes entirely when $n \geq N^2$ or N divides n.

[6]The computational complexity of (pseudo)random number generator is beyond the scope of this article. See, for example, the paper of Goldreich [2010] for details.

The second aspect is given by the random distribution of permutation elements to processors in PHASE1. Ideally, each processor would receive n/N permutation elements, however, due to randomness, it might happen that some processor will obtain significantly more elements than the others. The question is, what is the probability of such a situation? Sanders [1998] claims that "most elements are immediately sent to their final destinations with high probability" (which implies their uniform or nearly uniform distribution) and he supports this statement by Lemma 2, where the proof refers to a "standard allocation problem that can be solved using Chernoff bounds". Lassous and Thierry [2000] claim directly that their first algorithm (which corresponds to Sanders') "randomly permutes input data according to the uniform distribution with high probability". Moreover, they show, with reference to the proof presented by Reif [1985], that the probability that some processor will obtain more elements than some limit proportional to $\beta \cdot n/N$ is $1/e^{\beta \cdot n/N}$, where $\beta \geq 1$. Even for $\beta = 1$, the denominator of this probability becomes an extremely large number as the ratio n/N grows. Thus, the probability of an imbalanced random distribution of permutation elements is negligible if the large-scale condition (5) is satisfied. For illustration, the probability that all elements will target the same destination processor (a worst-case scenario) is $1/N^{n-1}$, which drops quickly to zero as n/N grows ($N \geq 2$). Note that if the large-scale condition is not satisfied, then n/N is small and a considerable load imbalance might happen, although this should not pose any problem.

We have also performed some experiments that support the presented statements; their results are presented and discussed in Section 4.5.

3. IMPLEMENTATION

We have implemented our proposed Paraperm algorithm as a C++ library. We chose the C++ programming language for the following reasons.

(1) C++ allowed us to create a clear and concise API based on a single short header file.
(2) C++ allowed us to code the algorithm as a class template with a generic integer data type used for permutation elements. Particular data types are selected by users of Paraperm at compile time, which can result in memory savings.[7]
(3) Within the code, we use some functionality of the Boost library [Abrahams and Gurtovoy 2004; Karlsson 2005] which is written in C++.
(4) The algorithm is completely implemented inside header files. Moreover the Boost functionality we used is also only defined in header files. Therefore, users can integrate Paraperm into their code by adding a single `#include` preprocessor directive and no additional linking is required.

The downside of choosing C++ is that the implemented algorithm is not directly available to C/Fortran users.[8]

The C++ implementation corresponds very closely to the pseudocode of the algorithm and its particular phases presented in Section 2. We mostly use the same variable names throughout the C++ code with the prime symbol replaced by underscore (so, for example, *count′* from the algorithm pseudocode translates into `count_` in the C++ code).

[7]For instance, when $n > 2^{32}$, a 64-bit data type must be used for the permutation elements. However, to hardcode the 64-bit data type into Paramerm would result in all permutation elements having at least half of their bytes zero when $n \leq 2^{32}$.

[8]This problem can be solved by wrapping Paraperm with a plain C API using, for example, the *adapter/façede design pattern* [Reddy 2011].

```
namespace paraperm {
  template <typename T = uintmax_t>
  class Paraperm : boost::noncopyable
  {
    public:
      typedef T value_type;
      typedef std::vector<T> vector_type;

      Paraperm();
     ~Paraperm();

      void generate(MPI_Comm comm, T n);

      const vector_type& perm() const;
      T pos() const;
      T count() const;

    private:
      struct Impl;
      Impl* pimpl_;
  };
}
```

Fig. 1. `Paraperm.h` header file (without preprocessor directives).

3.1. API

We designed the algorithm to be a class template called `Paraperm`, which resides in the `paraperm` namespace. The definition of this class template is in the `Paraperm.h` header file, which is the only header file that users need to include in their code. The template parameter determines the data type used for permutation elements. The default value is `uintmax_t`.[9]

To demonstrate the simplicity of the Paraperm API, we show in Figure 1 the definition of the `Paraperm` class template.[10] Except for this definition, the `Paraperm.h` header file contains only preprocessor directives.

The generation of a random permutation is performed by calling the `generate()` member function. The `comm` and `n` function parameters match the *comm* and n algorithm input parameters, respectively. After the `generate()` member function ends, the generated permutation values are available through the `perm()`, `pos()`, and `count()` member functions that return, when called on processor P_r, the algorithm output parameters $perm_r$, pos_r, $count_r$, respectively. Inside the `generate()` function, the `comm` communicator is duplicated using the `MPI_Comm_dup()` function, which is a standard practice for library code that uses MPI.

More details about the Paraperm class template and its member functions can be found in the API documentation, which is a part of the Paraperm software library.

3.2. Sample Usage

As mentioned in Section 1, the ideal implementation of a parallel generator of random permutations should require only minimal effort to use. To prove that our implementation satisfies this requirement, we show in Figure 2 an example code that generates a random permutation of size $n = 2^{24}N$ for an arbitrary number of processors. There are only eight lines of code related to Paraperm. Thanks to the utilization of

[9]The unsigned integer data type of maximum bit width supported by a given system architecture.

[10]Note that we hide all non-public functionality of Paraperm using the *pointer-to-implementation* idiom (also known as *pimpl*) [Sutter 2000, Items 26–30; Sutter and Alexandrescu 2004, Item 43].

```
#include <mpi.h>
#include <cstdint>
#include <paraperm/Paraperm.h>

  typedef paraperm::Paraperm<uint64_t> Paraperm;

  int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int N;
    MPI_Comm_size(MPI_COMM_WORLD, &N);

    Paraperm paraperm;

    const Paraperm::value_type n = (1UL << 24) * N;
    paraperm.generate(MPI_COMM_WORLD, n);

    const Paraperm::vector_type& perm = paraperm.perm();
    const Paraperm::value_type pos = paraperm.pos();
    const Paraperm::value_type count = paraperm.count();

    // do whatever with the generated permutation

    MPI_Finalize();
    return 0;
}
```

Fig. 2. Example code for generation of random permutations of size $n = 2^{24}N$.

the `Paraperm::value_type` and `Paraperm::vector_type` type definitions, there is only a single point where the data type for permutation elements is specified.

3.3. Random Number Generation

Within the Paraperm library, we use the pseudorandom number generator (PRNG) from the Boost library. As an engine, we use the Boost implementation of the Mersene Twister PRNG [Matsumoto and Nishimura 1998]:

— `boost::random::mt19937_64` in case that the template parameter `T` matches the `uint64_t` data type,
— `boost::random::mt19937` otherwise.

As a distribution, we use the `boost::random::uniform_int_distribution`. While `mt19937_64`, `mt19937`, and `uniform_int_distribution` are all now part of the C++11 standard [ISO 2011], many compilers do not yet provide these and we, therefore, decided to use the Boost-based solution to aid portability.

There is a single instance of the PRNG on each processor. These instances cannot be seeded by the `time()` value as is usually done in sequential programs, since such an approach could result in the same seed on multiple processors. We exploit the parallel solution presented by Katzgraber [2010, Section 7.1], which combines the `time()` value together with the processor number for the generation of seeds.

3.4. Memory Management

In PHASE1 of the Paraperm algorithm, there are two arrays that need to be sorted at once according to the values of one of these arrays (see Section 2.3.1 for details). Unfortunately, this procedure cannot be performed by the `std::sort()` function provided

Table IV. Configuration of the Systems Used for Experiments

System:	Blue Waters	Hopper	Queen Bee	Tezpur	Zewura
Provider:	NCSA	NERSC	LONI	LSU	CERIT-SC
Cores:	362240	153216	5344	1440	1600
Nodes:	22640	6384	668	360	20
Cores per node:	16	24	8	4	80
Memory (TB):	1382	217	5.3	1.4	10.2
Memory per node (GB):	64	32/64	8	4	512
Memory per core:	4	1.33/2.66	1	1	6.4
Peak perf. in PFLOPS:	7.1	1.28	0.051	0.015	N/A
Interconnect:	Gemini	Gemini	Infiniband	Infiniband	Ethernet
Topology:	3D torus	3D torus	N/A	N/A	N/A
Per node bandwidth (GB/s):	9.6	20	N/A	N/A	N/A
C++ compiler:	GNU g++ 4.7.2	GNU g++ 4.7.1	Intel icpc 11.1	Intel icpc 11.1	GNU g++ 4.5.2
MPI library:	cray-mpich 5.6.4	cray-mpich 5.5.2	mvapich 1.1	mvapich 1.1	mpich2 1.4.1

by the C++ Standard Template Library[11] (STL) [Josuttis 2012]. There are two options how for circumventing this restriction:

— either abandon the STL-based sorting algorithm and use a different solution;
— or store each permutation element together with its destination processor number as a pair (`std::pair`), put these pairs into a separate array, and then sort this array with `std::sort()`.

Since the implementation of the sorting algorithm in STL is usually highly optimized, we chose the second option for the Paraperm library. However, such solution has one drawback—prior to sending the permutation elements to destination processors, these elements need to be extracted from the sorted pairs into a continuous chunk of memory (*sendbuf*). Consequently, the memory requirements grow at this point from $2n/N + N$ to $3n/N + N$. After the *sendcnts* array is created, the array of pairs can be released. The total memory requirements of PHASE1 in our implementation of the Paraperm algorithm is thus the minimum of $3n/N + N$ and $2n/N + 4N$ (the latter value still holds for the `MPI_Alltoallv()` function calls; see Section 2.5.1).

4. EXPERIMENTS

We have performed multiple experiments to evaluate the Paraperm algorithm and its implementation. We created a test program based on the code presented in Figure 2, wherein we added measurement and verification procedures.

We utilized several parallel systems that are listed in Table IV, where we list their parameters together with the C++ compilers and the MPI libraries used for compilation of the test program and its runtime execution. For the Blue Waters system, we present parameters of the XE cabinets only (the XK cabinets are intended primarily for GPU-based computations). Note that the presented parameters of the com-

[11]Sorting two arrays at once using the `std::sort()` function requires a writable random-access *zip iterator* (an iterator that can access multiple containers simultaneously). The STL does not provide such an iterator. The Boost library defines `boost::zip_iterator`, which is, however, read-only.

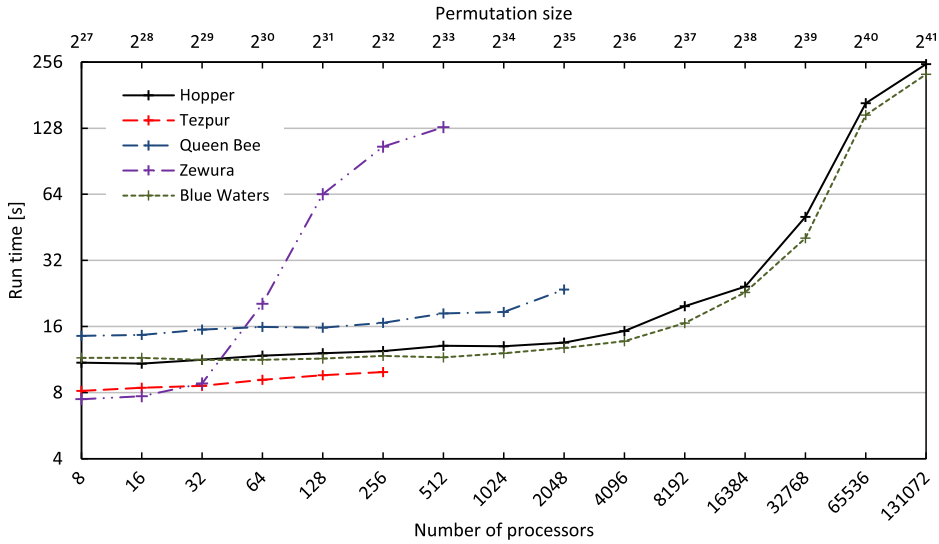

Fig. 3. Run times of the algorithm for constant per-processor permutation size $n/N = 2^{24}$ and increasing number of processors, on different systems.

puter system correspond to the moment when the experiments were performed (some parameters might have changed since then).

We compiled the test program with the `-O2` optimization option in all cases.

All the presented experiments were of the *one-process-per-core* type, that is, all runs of the algorithm were performed with the number of MPI processes equal to the number CPU cores.

4.1. Scalability with Constant Load per Processor

First, we measured run times of the algorithm for the case where the number of permutation elements per processor was constant—namely $n/N = 2^{24}$—while the number of processors N varied (so-called *weak scalability*). The overall permutation size n thus also varied proportionally to N, since $n = 2^{24}N$. The results are shown in Figure 3.

Since the number of permutation elements per processor was constant, ideally, we would like a constant run time (independent of the number of processors). However, recall that the all-to-all communication operation is executed within PHASE1. The number of messages thus grows quadratically with the number of processors and the run time depends mainly on the behavior of the `MPI_Alltoallv()` function for a given system.

Figure 3 shows that the run time increased only very slightly for the Blue Waters, Hopper, Queen Bee, and Tezpur systems up to around 4096 processors. On Blue Waters and Hopper, the run time then started to grow considerably faster, which was probably caused by the overall saturation of the interconnect subsystem.

On the Zewura system, the runtime increased considerably between 32 and 64 as well as between 64 and 128 processors. In our opinion, this was for the following reasons.

— Both algorithm runs for $N = 32$ and 64 were performed within a single node. Therefore, all the communication between processors was actually carried out over shared memory. The slowdown was thus probably caused by the increased number of conflicts when accessing physical memory units by local CPU cores.

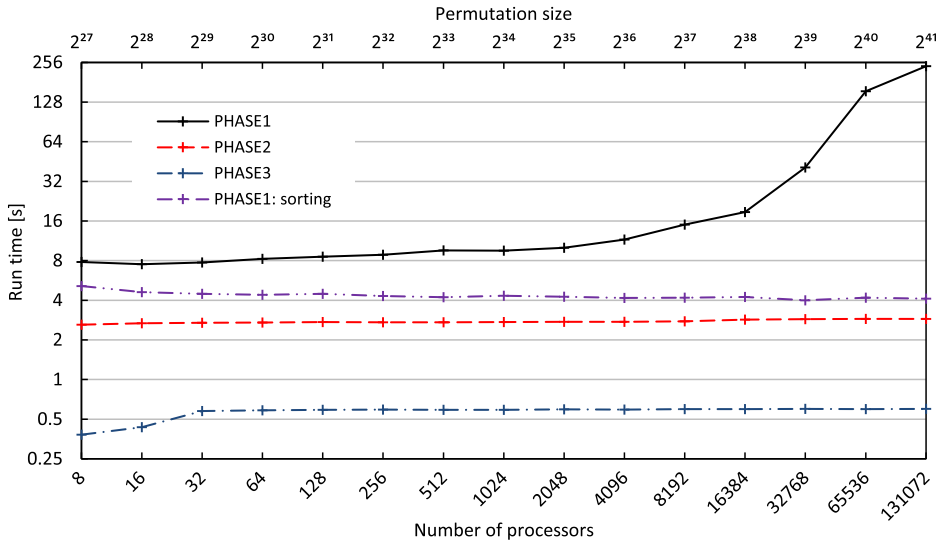

Fig. 4. Run times of the individual phases of the algorithm for constant per-processor permutation size $n/N = 2^{24}$ and increasing number of processors, on the Hopper system.

— For 128 processors, the algorithm already used processors from 2 nodes. However, the Zewura system is primarily intended only for shared-memory computations performed within individual nodes. The communication between nodes is much slower when compared with other systems, which utilize special interconnect subsystems and optimized MPI routines.

Next, we measured run times of the individual phases of the algorithm. The values obtained on the Hopper system are presented in Figure 4. The run times of both PHASE2 and PHASE3 were practically independent of the number of processors. Most of the run time was spent within PHASE1 and for higher number of processors, this phase dominated the overall run time of the algorithm.

PHASE1 consists of several potentially time-consuming steps. In the light of our experimental results, we can make the following comments.

(1) *Generation of n/N Random Numbers*. The very same task is performed within PHASE2. Thus, the run time of this task effectively became negligible as N increased.
(2) *Sorting the Array of Permutation Elements and Their Destination Processor Numbers*. We separately measured the run time of this task; the results are presented in Figure 4 as the data set PHASE1:sorting. The run time of this task was almost constant and it became effectively negligible for higher N.
(3) *Calling the* `MPI_Alltoall()` *and* `MPI_Alltoallv()` *functions*. Recall that the former function is performed over an array of size N, while the latter function over an array of size n/N. Within our experiments, $N \leq 2^{17}$ while $n/N = 2^{24}$. Thus, we dare to claim that the majority of the run time of this task was spent inside the `MPI_Alltoallv()` function.
(4) *Performing a Barrier Synchronization of All Processors*. The same task was performed within both PHASE2 and PHASE3. Thus, the run time of this task became negligible as N grew.

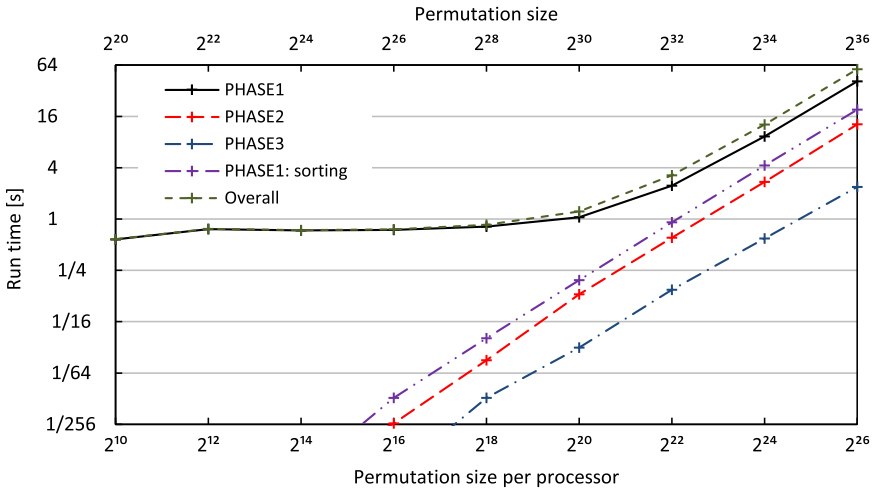

Fig. 5. Run times of the algorithm for constant number of processors $N = 1024$ and increasing permutation size, on the Hopper system.

The results imply that the more processors participate in generating the random permutation, the more the run time is dominated by the execution of the `MPI_Alltoallv()` function and the other parts of the algorithm become effectively negligible. Note that the `MPI_Alltoallv()` function itself is not a part of Paraperm and thus we cannot directly affect its performance.

4.2. Scalability with Constant Number of Processors

Second, we measured run times of the algorithm for the case where the number of processors was constant—namely $N = 1024$—while the permutation size n varied. The number of permutation elements per processor n/N thus also varied as well proportionally to n. The results are shown in Figure 5, from which we can observe the following.

— The runtimes of PHASE2, PHASE3, and the sorting part of PHASE1 grew proportional to n. Moreover, they all grew at the same rate.
— There was some overhead in PHASE1 that dominated its run time up to n/N being around 2^{20}. Beyond this point, the run time of PHASE1 started to grow proportionally to n with the same rate as the run times of the other two phases.

Such behavior is clearly desirable, since it predicates the linear data scalability of the algorithm in the case of constant number of processors.

4.3. Scalability with Constant Permutation Size

Third, we measured run times of the algorithm for the case where the permutation size n was constant while the number of processors N varied (so-called *strong scalability*). We performed two experiments: a small-scale one and a large-scale one. The results are presented in Figure 6(a) and Figure 6(b), respectively.

In the small-scale experiment, we can observe the linear speedup of both the algorithm and its particular phases. There was some small runtime overhead in PHASE1 (less than 1 second) that broke the linear speedup for $N = 1024$. (This overhead was also described in Section 4.2 and shown by Figure 5.)

In the large-scale experiment, the run times of PHASE2, PHASE3 and the sorting part of PHASE1 also exhibit linear speedup. As for PHASE1, using more than

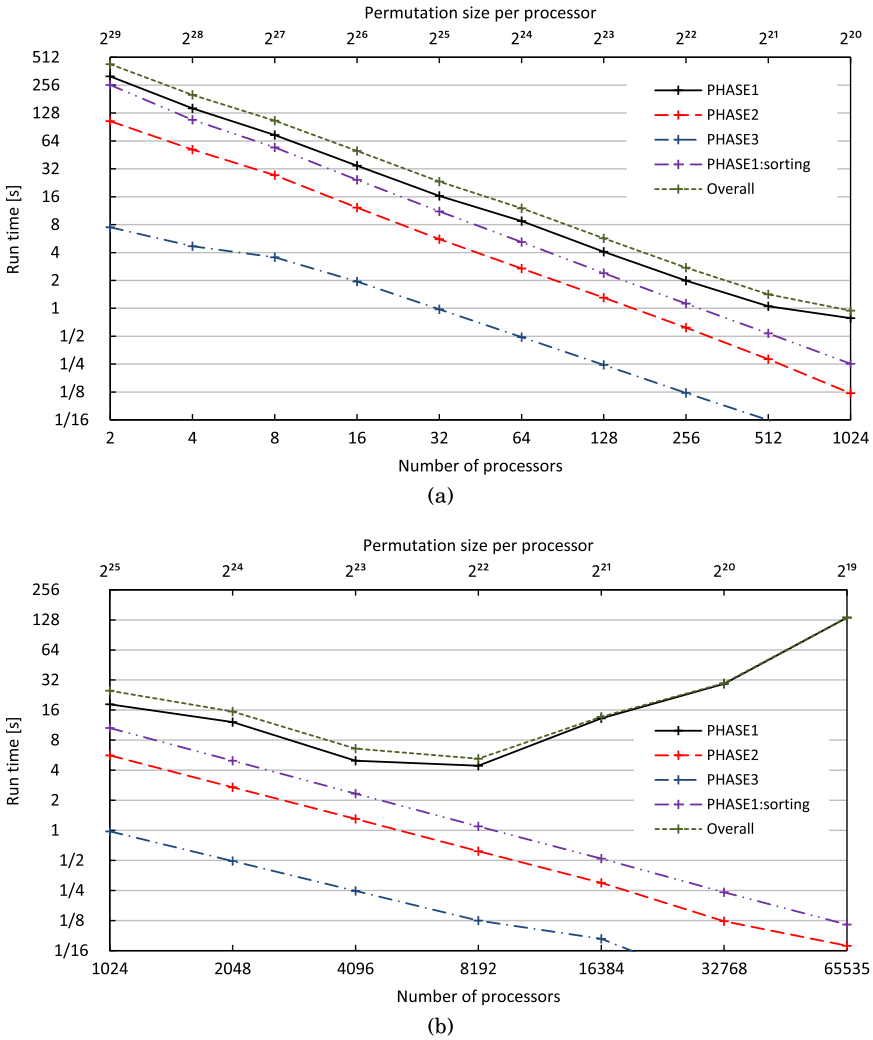

Fig. 6. Run times of the algorithm for constant permutation sizes $n = 2^{30}$ (a) and $n = 2^{35}$ (b) and increasing number of processors, on the Blue Waters system.

4096 processors results in run times that were dominated by the execution of the `MPI_Alltoallv()` operation, which corresponds to the results presented in Section 4.1 and by Figure 3.

4.4. Mapping of MPI processes

Since Paraperm is a communication-intensive algorithm (for large-scale runs, the run time is dominated by communication operations), the mapping of MPI processes to computational nodes and their CPU sockets/NUMA[12] nodes may have significant impact on the run time. Let M denote the number of MPI processes running on the same node. For collective communication, these processes share the same network interface(s). Consequently, we may assume that the per-process network bandwidth is

[12]Nonuniform memory access.

Table V. Run Times of the Algorithm for Different Numbers of MPI Processes per Node M for $n/N = 2^{24}$ and $N = 8192$, on the Blue Waters System

M:	16	8	4	2
Run time [s]:	14.6	11.3	10.5	12.6

Table VI. Statistical Results of 20 Run Times of the Algorithm in Seconds for $N = 128$, $n = 2^{31}$, and the Hopper System

Average	Median	Standard deviation	Maximum	Maximum to average [%]
11.94	11.91	0.12	12.43	104.01

Table VII. Statistical Results of the Measured Sizes of the *temp* Arrays After PHASE1

Ideal size	Average maximum	Average maximum [%]	Maximum maximum	Maximum maximum [%]
16777216	16791056	100.08	16795396	100.11

Data were collected from all processors from 20 algorithm runs for $N = 128$, $n = 2^{31}$, and the Hopper system.

inversely proportional to M. However, smaller M implies larger number of nodes (N/M), which entails larger average distance of data transfers.

We carried out a simple study to evaluate the influence of the mapping of MPI processes to nodes using the Blue Waters system. We measured the algorithm run time for the constant number of MPI processes $N = 8192$, while varying the number of MPI processes allocated per node M (recall that within the context of MPI, both terms *MPI process* and *processor* refer to the same entity). The result of this experiment is presented in Table V. Each node on the Blue Waters system contains two sockets/AMD Interlagos 6276 CPUs, each consisting of two NUMA nodes. We always set the mapping so that each socket/NUMA node runs the same number of MPI processes. The fastest run time was obtained when each MPI process is allocated on a single NUMA node.

Unfortunately, our results cannot be used to draw a general conclusion. If we use only a limited set of CPU cores per node, the overall algorithm performance will depend on utilization of the remaining cores (possibly running tasks from other users) and especially on their communication activities. The results of mapping experiments might also vary for different systems/architectures. Moreover, if all scheduled tasks of other users are defined to allocate whole nodes (which is a typical allocation policy), the execution of our task will result in underutilization of computational resources of the system. Users should always check the scheduling and resource-allocation policies for their systems to make proper mapping decisions.

4.5. Random Effects

We also evaluated the influence of (pseudo)randomness upon algorithm runs. We executed the test program 20 times using 128 processors on the Hopper system and

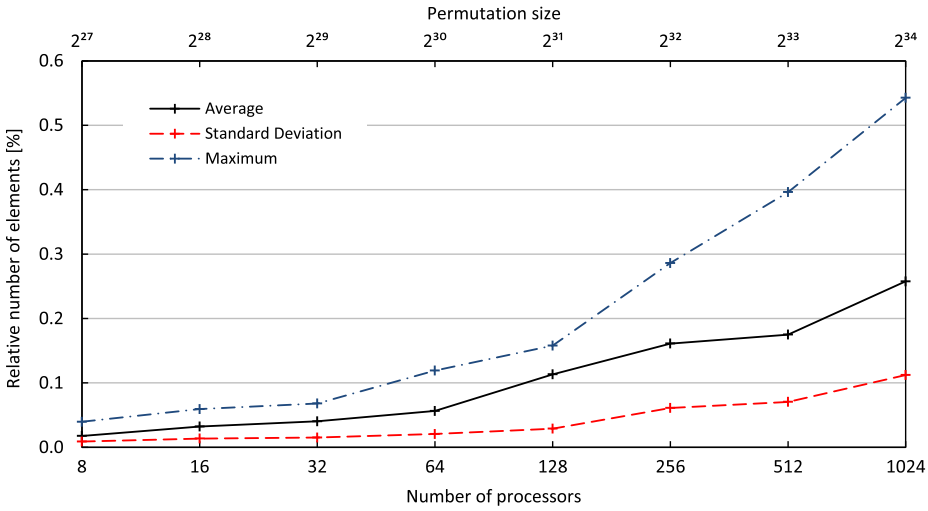

Fig. 7. Statistical values of the relative number of transferred permutation elements in PHASE3 for $n/N = 2^{24}$, different number of processors, and the Hopper systems. For each number of processors, the algorithm run was performed 20 times.

setting the permutation size $n = 2^{31}$. The results are shown in Table VI, Table VII, and Figure 7.

4.5.1. Run Time of the Algorithm. In Table VI, we show the statistical results for the run times of the algorithm. The longest run time was only 1.04 times longer than the average and the algorithm was thus practically independent of the randomness. Moreover, the variations could have been caused not only by randomness, but also by the actual state of the runtime environment (e.g., the actual utilization of its interconnect subsystem) and we cannot distinguish between these.

4.5.2. Random Distribution of Permutation Elements. In Table VII, we show the statistical results for the sizes of the *temp* arrays after PHASE1. Recall that due to the random distribution of permutation elements in PHASE1, each processor ends this phase with generally different sized *temp* arrays. The *ideal size* corresponds to the perfectly balanced distribution, where each processor would have exactly $n/N = 16777216$ permutation elements in its *temp* array.

We gathered the measured sizes for all 128 processors and all 20 algorithm runs. For each algorithm run, we first calculated the maximum *temp* size from all processors. We thus got 20 maximal values, from those we further calculated their average—called *average maximum*—and their maximum—called *maximum maximum*. The values are given as percentages of the ideal size.

Obviously, we can conclude that the random distribution of permutation elements in PHASE1 was very well balanced for all algorithm runs and all processors, since the maximum observed deviation from the ideal size of the *temp* array was only 0.11 %.

4.5.3. Final Balancing of Permutation Elements. Finally, we wanted to know how many permutation elements needed to be transferred in PHASE3 of the algorithm. In Figure 7 we present the statistical values of their relative numbers (with regard to their total

Table VIII. The Relative Numbers of Imbalanced Observed Cases in Percents from 10000 Trials, for Different Combinations of N and n and for Different Imbalance Limits Determined by the α Parameter

$\alpha = 1.02$

	n						
N	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16	42.82	6.83	0.25	0.00	0.00	0.00	0.00
128	100.00	100.00	100.00	100.00	97.04	28.34	0.53

$\alpha = 1.05$

	n						
N	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16	0.00	0.00	0.00	0.00	0.00	0.00	0.00
128	100.00	99.79	64.04	2.89	0.00	0.00	0.00

count n) gathered from all 20 algorithm runs. Clearly, these numbers are very low, which justifies the choice of point-to-point communication operations for PHASE3 as described in Section 2.3.3.

4.5.4. Probability of a Load Imbalance. To support the statements presented in Section 2.6, we also conducted the following additional experiment. We simulated 10000 cases of the random distribution of permutation elements among processors for some fixed combination of n and N. Throughout this simulation, we counted such cases where any processor obtained the number of elements higher than $\alpha \cdot n/N$, for some $\alpha \geq 1$. The relative number of these *imbalanced* cases then experimentally evaluates the probability that some processor obtains more permutation elements than α times their ideal count n/N. The parameter α hence represents a measure of a tolerance of their imbalance. The results of this experiment are presented in Table VIII. These results clearly confirm that the probability of imbalanced random distribution of permutation elements to processors in PHASE1 drops quickly as n/N increases.

4.6. Verification of Results

To verify that the Paraperm algorithm generates each possible permutation with the probability $1/n!$ is not computationally feasible. Even for an extremely small problem where $n = 10$, we would need to execute the algorithm at least 10^9 times to gather enough data to obtain any statistically relevant results. However, recall that Sanders [1998] proved this probability for his algorithm, and we showed that our elaboration of his algorithm does not break its randomness.

For large-scale problems, even to insure that the final permutation contains all elements (and each element only once) would be too computationally expensive. We successfully performed this type of verification for problems where $n < 100$.

For large-scale runs, we verified the final permutations by summing their values as follows:

$$S = s_0 + \cdots + s_{N-1}, \quad \text{where} \quad s_r = \sum_{i=0}^{count_r} perm_r[i]\,.$$

Since we can express this sum analytically as

$$S' = \sum_{i=0}^{n-1} i = n(n-1)/2,$$

we performed the verification simply by comparing S and S'. These two values were equal in all our tests.

5. CONCLUSIONS

The contribution of this article is an efficient scalable MPI-based algorithm Paraperm for parallel generation of random permutations of a set of integers $\{0, \ldots, n-1\}$, as well as the implementation of this algorithm and the evaluation of its scalability.

The Paraperm algorithm has the following characteristics:

— it is very fast in practice—the generation of a random permutation of 2^{41} elements took slightly over 4 minutes using 131072 CPU cores;
— it has a simple user interface that consists of only a small number of input and output parameters;
— the resulting permutations are distributed amongst processors in such a way that users may evaluate the location of a particular permutation value in a constant time.

However, users should also be aware that the algorithm needs, on each processor, considerably more memory compared with that required to store the resulting permutation values.

The development of the Paraperm algorithm was based on the Sanders' algorithm, which has been elaborated up to the level of particular MPI communication operations. Sanders expressed his original algorithm with only 6 lines of pseudocode; while Paraperm requires 102.

The implementation of the algorithm is provided in a form of a software library called Paraperm written in C++ and based on the MPI API. The Paraperm API is clear and concise—it consists of a single header file with a single definition of a class template. To generate random permutations with the Paraperm library, users only need to add a few lines of code to their programs. Moreover, they can choose the most appropriate data type for permutation elements, thus optimizing memory usage.

The outcomes of scalability studies were shown for several massively parallel computer systems using up to 131072 CPU cores. The number of cores installed on today's most powerful systems is an order of magnitude higher. The scalability study of the Paraperm algorithm for theses systems will be a subject of our future work.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

David Abrahams and Aleksey Gurtovoy. 2004. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, Boston, MA.

R. Anderson. 1990. Parallel algorithms for generating random permutations on a shared memory machine. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'90)*. ACM, New York, NY, 95–102. DOI:http://dx.doi.org/10.1145/97444.97674.

Guojing Cong and David A. Bader. 2005. An empirical analysis of parallel random permutation algorithms on SMPs. In *Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS'05)*. Michael J. Oudshoorn and Sanguthevar Rajasekaran (Eds.), ISCA, Winona, MN, 27–34.

Artur Czumaj, Przemyslawa Kanarek, Miroslaw Kutylowski, and Krzysztof Lorys. 1998. Fast generation of random permutations via networks simulation. *Algorithmica 21*, 1, 2–20. DOI:http://dx.doi.org/10.1007/PL00009206.

Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. 1993. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proceedings of the 9th Annual Symposium on Computational Geometry (SCG'93)*. ACM, New York, 298–307. DOI:http://dx.doi.org/10.1145/160985.161154.

Richard Durstenfeld. 1964. Algorithm 235: Random permutation. *Commun. ACM 7*, 7, 420–421. DOI:http://dx.doi.org/10.1145/364520.364540.

O. Goldreich. 2010. *A Primer on Pseudorandom Generators*. American Mathematical Society, Providence, RI.

Michael T. Goodrich. 1996. Communication-efficient parallel sorting (preliminary version). In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC'96)*. ACM, New York, 247–256. DOI:http://dx.doi.org/10.1145/237814.237870.

Michael T. Goodrich. 1997. Randomized fully-scalable BSP techniques for multi-searching and convex hull construction. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*. SIAM, Philadelphia, PA, 767–776. DOI:http://dl.acm.org/citation.cfm?id=314161.314442.

William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. 1998. *MPI—The Complete Reference, Volume 2: The MPI-2 Extensions*. MIT Press, Cambridge, MA.

Jens Gustedt. 2008. Engineering parallel in-place random generation of integer permutations. In *Proceedings of the 7th International Workshop on Experimental Algorithms (WEA'08)*. Catherine McGeoch (Ed.), Lecture Notes In Computer Science, vol. 5038, Springer Berlin, 129–141. DOI:http://dx.doi.org/10.1007/978-3-540-68552-4_10.

Jens Gustedt, Stéphane Vialle, and Amelia De Vivo. 2007. The parXXL environment: Scalable fine grained development for large coarse grained platforms. In *Proceedings of the 8th International Workshop on Applied Parallel Computing. State of the Art in Scientific Computing, (PARA'06)*. Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Wasniewski (Eds.), Lecture Notes in Computer Science, vol. 4699, Springer-Verlag, Berlin, 1094–1104.

Torben Hagerup. 1991. Fast parallel generation of random permutations. In *Automata, Languages and Programming*, Javier Leach Albert, Burkhard Monien, and Mario Rodriguez Artalejo (Eds.), Lecture Notes in Computer Science, vol. 510, Springer Berlin, 405–416. DOI:http://dx.doi.org/10.1007/3-540-54233-7_151.

ISO/IEC. 2011. *ISO/IEC 14882:2011 Information technology—Programming languages—C++*. ISO/IEC Copyright Office, Geneva, Switzerland.

Joseph F. JaJa. 2011. Parallel random access machines (PRAM). In *Encyclopedia of Parallel Computing*, David Padua (Ed.), Springer Berlin, 1608–1615. DOI:http://dx.doi.org/10.1007/978-0-387-09766-4_2093.

Nicolai M. Josuttis. 2012. *The C++ Standard Library—A Tutorial and Reference* (2nd ed.). Addison-Wesley-Longman, Boston, MA.

Björn Karlsson. 2005. *Beyond the C++ Standard Library: An Introduction to Boost*. Addison-Wesley Professional, Boston, MA.

Helmut G. Katzgraber. 2010. *Random numbers in scientific computing: An introduction*. arXiv:1005.41717 [physics.comp-ph], http://arxiv.org/abs/1005.4117 (accessed September, 2012).

Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* 3rd Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Isabelle Guérin Lassous and Eric Thierry. 2000. Generating random permutation in the framework of coarse grained models. In *Procedings of the 4th International Conference on Principles of Distributed Systems (OPODIS 2000) (Studia Informatica Universalis)*. Franck Butelle (Ed.), Suger, Saint-Denis, rue Catulienne, France, 1–16.

Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul. 8*, 1, 3–30. DOI:http://dx.doi.org/10.1145/272991.272995.

G. L. Miller and J. H. Reif. 1985. Parallel tree contraction and its application. In *Proceedings of the 26th Symposium on Foundations of Computer Science*. IEEE, 478–489. DOI:http://dx.doi.org/10.1109/SFCS.1985.43.

Martin Reddy. 2011. *API Design for C++*. Morgan-Kaufmann, Burlington, MA.

J. H. Reif. 1985. An optimal parallel algorithm for integer sorting. In *Proceedings of the 26th Symposium on Foundations of Computer Science*. IEEE, 496–504. DOI:http://dx.doi.org/10.1109/SFCS.1985.9.

P. Sanders. 1998. Random permutations on distributed, external and hierarchical memory. *Inform. Process. Lett.* 67, 6, 305–309. DOI:http://dx.doi.org/10.1016/S0020-0190(98)00127-6.

Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. 1998. *MPI—The Complete Reference, Volume 1: The MPI Core* 2nd Ed. MIT Press, Cambridge, MA.

Herb Sutter. 2000. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley Longman Publishing Co., Inc, Boston, MA.

Herb Sutter and Andrei Alexandrescu. 2004. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, Boston, MA.

Alexander Tiskin. 2011. Bulk Synchronous Parallelism (BSP). In *Encyclopedia of Parallel Computing*, David Padua (Ed.), Springer, Berlin, 192–199. DOI:http://dx.doi.org/10.1007/978-0-387-09766-4_2067.

Received February 2013; revised July 2013 and November 2013; accepted December 2013