CCF Transactions on High Performance Computing (2023) 5:245–257
https://doi.org/10.1007/s42514-023-00150-2

REGULAR PAPER

ddRingAllreduce: a high-precision RingAllreduce algorithm

Xiaojun Lei[1] · Tongxiang Gu[2] · Xiaowen Xu[2,3]

Received: 8 March 2023 / Accepted: 24 April 2023 / Published online: 5 July 2023

Abstract
For complex problems in scientific computing, parallel computing is almost the only way to solve them, in which global reduction is one of the most frequently used operations. Due to the existence of floating-point rounding errors, the existing global reduction algorithm may result in inaccurate or different between two runs, which are difficult to meet the needs of complex applications. Since the communication cost of `RingAllreduce` is a constant, independent of the number of processes, it is an effective algorithm when a large amount of data needs to be communicated. However, it faces the same problem as the general global reduction operation, and it is necessary to develop a high-precision `RingAllreduce` algorithm. In this paper, by combining double-double arithmetic and `RingAllreduce` algorithm, we propose a high-precision `RingAllreduce` algorithm, called `ddRingAllreduce` algorithm. The theoretical error of the proposed algorithm is analyzed and the compact error bounds are derived. We have carried out a large number of parallel numerical experiments and obtained numerical results consistent with the theoretical analysis, and `ddRingAllreduce` is accurate in the case that `RingAllreduce` is inaccurate or miscalculated. At the same time, we also analyze the relationship between the problem size and the cost of using double-double arithmetic through experiments, at a small scale, the `ddRingAllreduce` algorithm can achieve higher accuracy with relatively less time overhead.

Keywords RingAllreduce · ddRingAllreduce · Collective communication · Double-double arithmetic · High precision

1 Introduction and motivation

In the past decade or so, there has been rapid development in high-performance computing. The number of supercomputers has been increasing and their computing speeds have been getting faster. High-performance computing has been widely used in fields such as laser fusion, oil exploration and weather forecasting (Xiaowen etal. 2009; Dogru etal. 2011; Kimura 2002), etc.

Parallel computing uses high-performance computers as hardware platforms to solve scientific computing problems by utilizing multiple computers working together in a coordinated manner. Compared to serial computing, parallel computing can solve problems of the same scale in a shorter time without losing accuracy, or solve larger-scale problems in the same amount of time. In large-scale parallel processing computers, MPI (The MPI 2008) (message passing interface) is currently the most widely used parallel programming interface.

Message passing interface is a message-passing parallel programming technique. The MPI standard defines a set of portable programming interfaces, and there are several major implementations of MPI interfaces, such as OpenMPI, MPICH, IntelMPI and MVAPICH. They are implemented according to the MPI interface standard with different internal implementations. MPI_Allreduce is a global reduction operation in MPI, which is equivalent to first performing MPI_Reduce, and then performing MPI_Bcast. MPI-defined reduction operations include sum, dot product, maximum

✉ Tongxiang Gu
txgu@iapcm.ac.cn

Xiaojun Lei
leixiaojun19@gscaep.ac.cn

Xiaowen Xu
xwxu@iapcm.ac.cn

1 Graduate School of Chinese Academy of Engineering Physics, 6 Huayuan Rd, Beijing 100193, China

2 Laboratory of Computational Physics, Institute of Applied Physics and Computational Mathematics, 6 Huayuan Rd, Beijing 100088, China

3 CAEP Software Center for Numerical Simulation, 6 Huayuan Rd, Beijing 100088, China

value, minimum value, maximum value and its position, minimum value and its position, etc. MPI_Reduce allows all processes in a communication group to participate in the reduction operation on the same variable and output the reduced result to a specified process. Generally, the master process has the reduced result. MPI_Scatter distributes vector data to each process. Therefore, compared to MPI_Reduce, MPI_Allreduce has reduced results for each process, while MPI_Reduce has reduced results only for the specified process.

When using computers with floating-point operations, some numerical problems arise. First, computers use binary to store floating-point numbers, and floating-point operations produce rounding errors, so the calculated results deviate from the real results, especially when the results of the previous operation are used in subsequent operations, there is an accumulation of rounding errors, which can lead to unreliable results in some cases. Second, the floating-point addition does not satisfy the associative law, and when parallel computation uses different number of processes to do the reduction operation, it may produce different results, i.e., non-reproducible results. To address the above phenomena, it is a proven way to study the numerical algorithms and implementations for high precision.

The all-reduce operation combines values from all processes and distributes the results to all processes. It is commonly used in parallel computing. In the MPI standard (The MPI 2008), the routine for this operation is MPI_Allreduce. Currently, the most widely used all-reduce scheme is the butterfly-like algorithm (Rabenseifner 2004; Rabenseifner and Traff 2004; van de Geijn 1994). When the network can support the butterfly communication pattern without contention, this algorithm is optimal both in the latency term (using the minimum number of communication rounds needed) and in the bandwidth term. The problem with the butterfly-like algorithm is that the butterfly communication pattern can cause network contention in many contemporary clusters. Therefore, Patarasuk and Xin implement an effective all-reduce operation for large data sizes. The ring-based all-reduce operation they proposed is bandwidth-optimal (Patarasuk and Yuan 2009), the communication overhead of the `RingAllreduce` algorithm is a constant and independent of the number of processes. However, in the presence of rounding errors, the reduction result of the `RingAllreduce` algorithm may cause the above-mentioned accuracy problems.

Zhou (1980) proposed a calculation formula for the relationship between computer word length, speed, and memory matching by establishing a probability model for the accumulation of computer rounding errors, which was an early research on floating-point rounding errors in China. If the high-precision summation algorithm is not used, some applications will be inaccurate or incorrect. Demmel et al. proposed a fast and accurate floating-point summation algorithm in Demmel and Hida (2004) and applied it to computational geometry. They proposed a fast reproducible floating-point summation algorithm in Demmel and Nguyen (2013), after which they proposed a parallel reproducible summation algorithm in Demmel and Nguyen (2015) based on the algorithm of Demmel and Nguyen (2013). Higham's monograph (Higham 2002) made a very comprehensive introduction to the accuracy and stability of numerical algorithms. They proposed a class of fast and accurate high-precision floating-point summation algorithms in Blanchard et al. (2020), and also performed some theoretical analysis. Rump proposed a variety of high-precision summation algorithms, and made a very detailed theoretical analysis of them, such as (Rump et al. 2008a, b; Rump 2009). Muller's monograph (Muller etal. 2010) introduces the relevant knowledge of floating-point arithmetic in great detail. We published an article (Lei et al. 2021) with reference to Rump's work, proposing a new fast parallel high-precision summation algorithm, which is based on MPI_Allreduce high-precision, and carried out theoretical analysis and experimental verification on it. We also implemented a reproducible BiCGSTAB (Lei etal. 2023) based on Demmel's ReproBLAS (Ahrens etal. 2020) and Riakymch's ExBLAS (Iakymchuk etal. 2015).

The remaining sections of this paper are structured as follows: In Sect. 1, we provide an explanation of the symbolic representation and introduce the `RingAllreduce` algorithm. In Sect. 3, we introduce the double-double format and its basic operations (Li et al. 2002; Hida etal. 2001). Next, we propose our high-precision `RingAllreduce` algorithm, which combines the double-double format and the `RingAllreduce` algorithm. We also analyzed the error bounds of the proposed algorithm, which allows us to confirm that it achieves approximate double-double precision results. In Sect. 4, we present the experimental results, compare the accuracy and performance of the `RingAllreduce` algorithm, and verify that the theoretical error bound is tight. Finally, we conclude the paper in Sect. 5 and suggest some future work.

2 RingAllreduce algorithm

2.1 Notation

In this section, the meaning of the symbols used in the paper is introduced, as shown in Table 1, which the first column is the symbol, and the second column indicates the meaning it represents.

Table 1 Notation meaning

Symbol	Meaning
N	Total number of processes
P_i	ith process
K	Actual array length per process
n	Number of sums
$\mathbb{N}$	Set of natural numbers
p	Vector of sums

Table 2 Scatter-reduce data transfers

Process	Send	Receive
P_0	$chunk_0$	$chunk_2$
P_1	$chunk_1$	$chunk_1$
P_2	$chunk_2$	$chunk_0$

2.2 RingAllreduce algorithm

In the `RingAllreduce` algorithm, the processes are arranged in a logical ring. Each process should have a left neighbor and a right neighbor, it will only ever send data to its right neighbor, and receive data from its left neighbor. The algorithm proceeds in two steps: first, a scatter-reduce, and then, an allgather. In the scatter-reduce step, the processes will exchange data such that every process ends up with a chunk of the final result. In the allgather step, the processes will exchange those chunks such that all processes end up with the complete final result.

Let the N processes be $P_0, P_1, \ldots, P_{N-1}$, using the `RingAllreduce` algorithm, the scatter-reduce operation is performed as follows: Assuming each process has K values, first, the K values in each process is partitioned into N chunks, all chunks having $\lceil \frac{K}{N} \rceil$ values except the last chunk, which has a chunk size of $K-(N-1)\lceil \frac{K}{N} \rceil$. Let us number the chunks by $chunk_0, chunk_1, \ldots, chunk_{N-1}$. The scatter-reduce operation is carried out by performing the logical ring pattern $N-1$ iterations.

We use a specific example to illustrate the scatter-reduce step: Suppose we have three processes, in the first iteration, the chunks sent and received by the three processes are shown in Table 2. After each process receives the data, it performs a reduction operation on the received data chunk with its corresponding data chunk (the chunk with the same chunk index), and replaces its own data with the (partial) reduction results. Figure 1 shows that the scatter-reduce step is implemented using three logical rings of processes.

After the scatter-reduce step is complete, every process has a chunk are the final results which include contributions from all the processes. In order to complete the all-reduce operation, the processes must exchange those chunks, so that all processes have all the necessary results.

The allgather proceeds identically to the scatter-reduce (with $N-1$ iterations of sends and receives), except instead of accumulating values the processes receive, they simply overwrite the chunks.

The `RingAllreduce` algorithm pseudocode is shown in Listing 1.

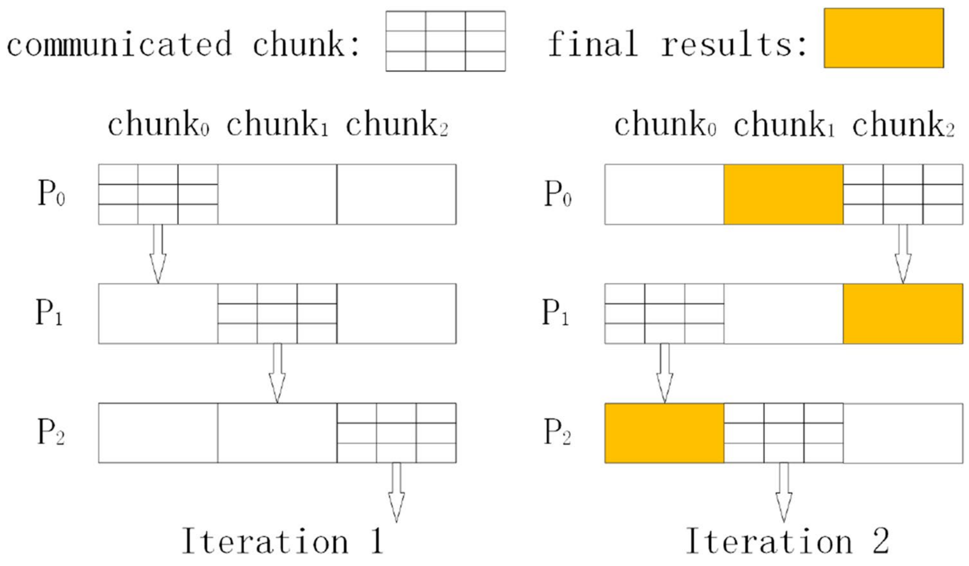


Fig. 1 Logical ring scatter-reduce algorithm

```
const size_t segment_size = length / size;
const size_t recv_from = (rank - 1 + size) % size;
const size_t send_to = (rank + 1) % size;
//scatter-reduce
for (int i = 0; i < size - 1; i++) {
     int recv_chunk = (rank - i - 1 + size) % size;
     int send_chunk = (rank - i + size) % size;
     datatype* segment_send = &(output[segment_ends[
    send_chunk] - segment_sizes[send_chunk]]);

     MPI_Irecv(buffer, segment_sizes[recv_chunk],
    datatype, recv_from, 0, MPI_COMM_WORLD, &recv_req)
    ;

     MPI_Send(segment_send, segment_sizes[send_chunk],
     datatype, send_to, 0, MPI_COMM_WORLD);

     datatype *segment_update = &(output[segment_ends[
    recv_chunk] - segment_sizes[recv_chunk]]);

     MPI_Wait(&recv_req, &recv_status);

     reduce(segment_update, buffer, segment_sizes[
    recv_chunk]);
     }
//allgather
//allgather is similar to scatter-reduce except
    without reduce
```

Listing 1: `RingAllreduce`.

Next we analyze the communication cost of `RingAllreduce` algorithm. We assume that the number of data owned by each process is K, in the `RingAllreduce` algorithm, each of the N processes will send and receive values $N-1$ times for the scatter-reduce, and $N-1$ times for the allgather. Each time, the processes will send $\lceil \frac{K}{N} \rceil$ values. Therefore, the total amount of data transferred to and from every process is

$$\textit{Data Transferred} = 2(N-1)\left\lceil \frac{K}{N} \right\rceil,$$

which, crucially, is independent of the number of processes.

Baidu has successfully applied the `RingAllreduce` algorithm to deep learning training, they also released their `RingAllreduce` algorithm implementation as a library https://github.com/baidu-research/baidu-allreduce.

Next, we analyze the error bounds of the `RingAllreduce` algorithm, following (Higham 2002), we define γ_n as

$$\gamma_n := \frac{nu}{1-nu}, n \in \mathbb{N},$$

when using γ_n, we implicitly assume that $nu < 1$.

Let $p = (p_1, \dots, p_n)^T \in \mathbb{F}^n$. Then it holds that (Higham 2002)

$$\tilde{s} := fl\left(\sum_{i=1}^{n} p_i\right) \Rightarrow \mid \tilde{s} - \sum_{i=1}^{n} p_i \mid \leq \gamma_{n-1} \sum_{i=1}^{n} \mid p_i \mid. \tag{1}$$

Note that (1) is valid for any order of addition in the summation.

Let us denote s and S by

$$s := \sum_{i=1}^{n} p_i, S := \sum_{i=1}^{n} \mid p_i \mid.$$

The condition number of the summation of the vector p is defined by

$$cond\left(\sum_{i=1}^{n} p_i\right) := \frac{S}{|s|}, s \neq 0.$$

The error bounds of the result `res` by `RingAllreduce` are given as follows:

Theorem 1 *Let* `res` *be the result obtained by* `RingAllreduce`*, then*

$$\texttt{res} - s \leq \gamma_{n-1} S. \tag{2}$$

Moreover, if $s \neq 0$, then

$$\frac{\texttt{res} - s}{s} \leq \gamma_{n-1} cond\left(\sum p_i\right). \tag{3}$$

From the error bounds, we can see that the relative error of a summation problem is related to both the number of summations and the condition number of the problem. This theorem allows us to assess the accuracy of the `RingAllreduce` algorithm. Assuming that we need to find the sum of 100 numbers, if the condition number of the summation problem is 10^{13} order of magnitude, then the relative error between the result given by the algorithm and the exact solution is 1.

3 ddRingAllreduce algorithm

3.1 Double-double formats

In this section, we use the same notation as in Yamanaka et al. (2008). Let $\mathbb{F}$ be a set of floating-point numbers. Throughout this paper, we assume floating-point arithmetic adhering to IEEE standard 754 (ANSI 2019). Let $p = (p_i) \in \mathbb{F}^n$ and let $fl(\cdot)$ be the result of floating-point operations, where all operations inside parentheses are executed by ordinary floating-point arithmetic in rounding-to-nearest. We denote by $\mathtt{u}$ the machine epsilon. In IEEE standard 754 double precision $\mathtt{u} = 2^{-53}$.

The basic double-double precision arithmetic operation is composed of some algorithms in the QD multi-part format software library (Hida etal. 2001) developed by Hida, Li and Bailey. It can make the numerical calculation result approximate to double-double precision.

Suppose a double-double precision number is x, which is represented by the combination of two non-overlapping double precision floating-point numbers x_h and x_l, that is, $x = x_h + x_l$, and satisfies $|x_l| \leq \frac{1}{2}\mathrm{ulp}(x_h) \leq \mathtt{u}\,|x_h|$, the definition of $\mathrm{ulp}(x_h)$ is the gap between the two nearest floating-point numbers around a real number x_h (Jiang 2013).

The following describes the addition of double-double precision. We first introduce the error-free transformation algorithm for the addition of two floating-point numbers, assuming that a and b are two floating-point numbers and $fl(a\ op\ b) \in \mathbb{F}$, according to the fundamental properties of floating-point arithmetic, the error of a floating-point number is still a floating-point number. Therefore, we can obtain:

$$x = fl(a \pm b) \Rightarrow a \pm b = x + y, y \in \mathbb{F},$$
$$x = fl(a \times b) \Rightarrow a \times b = x + y, y \in \mathbb{F}.$$

An error-free transformation is a transformation that converts a floating-point number pair (a, b) into another floating-point number pair (x, y), where y is the error. After accumulating the errors, the result is compensated back to its original value.

Algorithm 1 `FastTwoSum` [5]

```
Input: $a, b$
Output: $x, y$
  $x = \mathrm{fl}(a + b)$
  $q = \mathrm{fl}(x - a)$
  $y = \mathrm{fl}(b - q)$
```

FastTwoSum is an error-free transformation used for adding two floating-point numbers, which requires the condition $| a | \geq | b |$ to be satisfied.

```
Algorithm 2 TwoSum [11]
Input: $a, b$
Output: $x, y$
  $x = \mathrm{fl}(a + b)$
  $z = \mathrm{fl}(x - a)$
  $y = \mathrm{fl}((a - (x - z)) + (b - z))$
```

TwoSum has no conditional requirements and is still valid in the case of underflow.

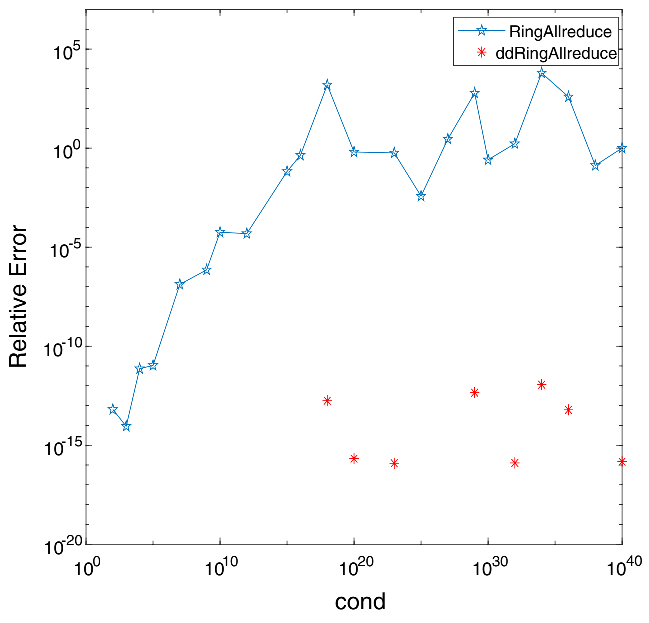
```
Algorithm 3 Split [5]:
Input: $a$
Output: $x, y$
  $c = factor \times a \quad \% factor = 2^s + 1$
  $x = c - (c - a)$
  $y = a - x$
```

The Split algorithm divides a floating-point number a with a precision of m into two floating-point numbers with a precision of up to $s - 1$ digits, where $s := \lceil m/2 \rceil$.

TwoProd is an error-free transformation algorithm for floating-point number multiplication proposed by Dekker (1971).

```
Algorithm 4 TwoProd [5]
Input: $a, b$
Output: $x, y$
  $x = a \times b$
  $[a_1, a_2] = Split(a)$
  $[b_1, b_2] = Split(b)$
  $y = a_2 \times b_2 - (((x - a_1 \times b_1) - a_2 \times b_1) - a_1 \times b_2)$
```

```
Algorithm 5 add_dd_dd [5]
Input: $a = (a_h, a_l)$, $b = (b_h, b_l)$
Output: $r = (r_h, r_l)$
  $[s_h, s_l] = TwoSum(a_h, b_h)$
  $[t_h, t_l] = TwoSum(a_l, b_l)$
  $s_l = s_l + t_h$
  $[t_h, s_l] = FastTwoSum(s_h, s_l)$
  $t_l = t_l + s_l$
  $r_h, r_l] = FastTwoSum(t_h, t_l)$
```



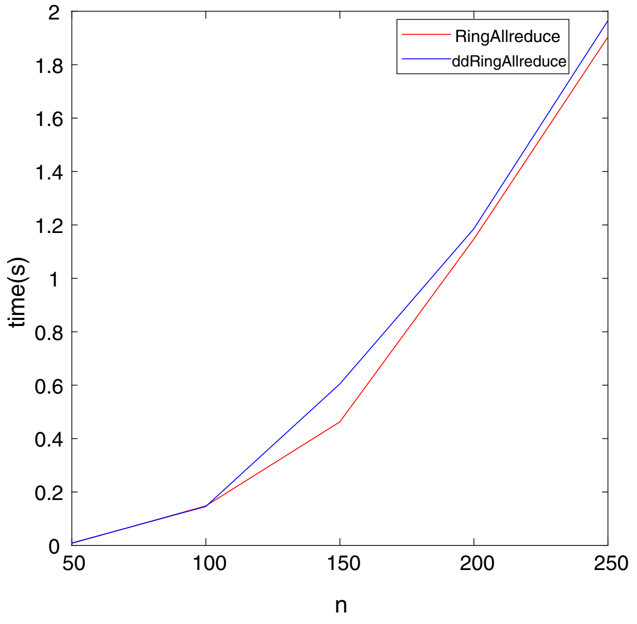


Fig. 2 Left: **a** Relative error image. Right: **b** CPU time image

`add_dd_dd` is the accumulation of 2 double-double type numbers. The application of double-double arithmetic can approximate a floating-point number with a precision (mantissa) of 106 bits, which satisfies the following properties (Li et al. 2002):

$$fl\,(a\,op\,b) = (a\,op\,b)(1+\delta),$$

where a and b are in double-double format, $op \in \{+,-,\times,\div\}$, satisfy

$$|\,\delta\,| \leqslant u_{dd},\ op \in \{+,-\};\ \ |\,\delta\,| \leqslant 2u_{dd},\ op \in \{\times,\div\},$$

where $\mathrm{u}_{dd} = 2\mathrm{u}^2 = 2^{-105}$.

3.2 ddRingAllreduce algorithm

In the `RingAllreduce` algorithm, we use double-double arithmetic for the reduce operation, so we get a high-precision `RingAllreduce` algorithm, called `ddRingAllreduce`. First, we convert the input data to double-double type. In the scatter-reduce stage, we use Algorithm 5 to add the two numbers of the adjacent process, and then send the obtained double-double result to the next process, which is say, we add the input data in the double-double format. After the final round of iteration is completed, we round the double-double result to the double type and then enter the allgather stage.

Theorem 2 *Let* `res` *be the result obtained by* `RingAllreduce`, *then*

$$|\,\mathtt{res} - s\,| \leqslant \frac{(n-1)\mathrm{u}_{dd}}{1-(n-1)\mathrm{u}_{dd}} S. \tag{4}$$

Moreover, if $s \neq 0$, then

$$|\,\frac{\mathtt{res} - s}{s}\,| \leqslant \frac{(n-1)\mathrm{u}_{dd}}{1-(n-1)\mathrm{u}_{dd}} cond\Big(\sum p_i\Big). \tag{5}$$

Proof The sum of two numbers in p is denoted as $T_i = p_k + p_j$, the sum $\hat{T}_i$ of floating-point calculations satisfies

$$\hat{T}_i = \frac{p_k + p_j}{1+\delta_i},\quad |\,\delta_i\,| \leqslant \mathrm{u}_{dd},\ i = 1 : n-1.$$

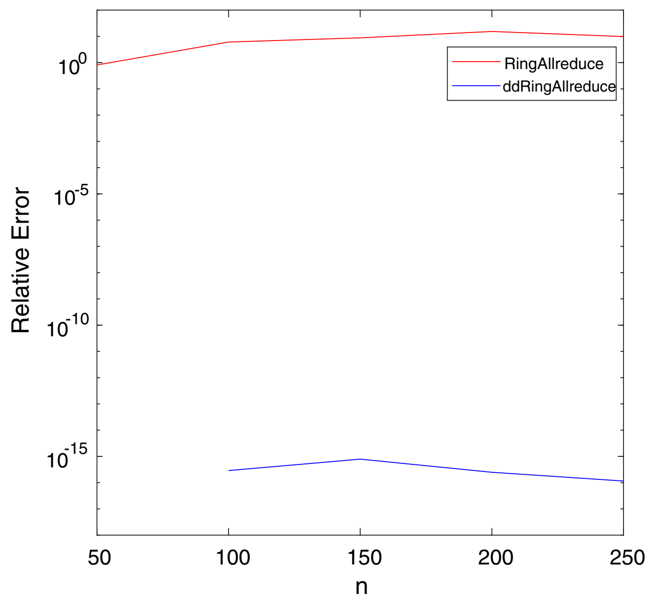
The local error introduced in calculating $\hat{T}_i$ is $\delta_i \hat{T}_i$, the overall error is the sum of the local errors (since summation is a linear process), therefore, we can get the overall error as

$$E_n := \mathtt{res} - s = \sum_{i=1}^{n-1} \delta_i \hat{T}_i.$$

Since $|\,\delta_i\,| \leqslant \mathrm{u}_{dd}$, we can get

$$|\,E_n\,| \leqslant \mathrm{u}_{dd} \sum_{i=1}^{n-1} |\,\hat{T}_i\,|,$$

and have $|\,\hat{T}_i\,| \leqslant \sum_{j=1}^{n} |\,p_j\,| + O(\mathrm{u}_{dd})$ for each i, therefore, we can get the upper bound



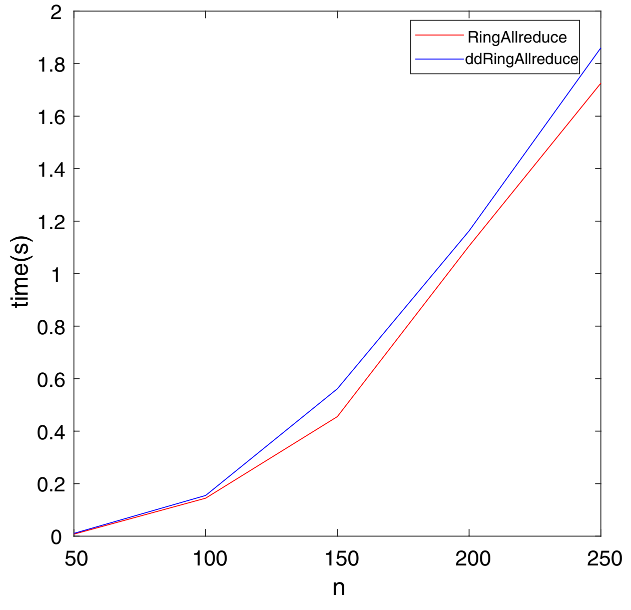


Fig. 3 Left: **a** Relative error image. Right: **b** CPU time image

$$| E_n | \leqslant (n-1)\mathrm{u}_{dd} \sum_{i=1}^{n} | p_i | + O({\mathrm{u}_{dd}}^2).$$

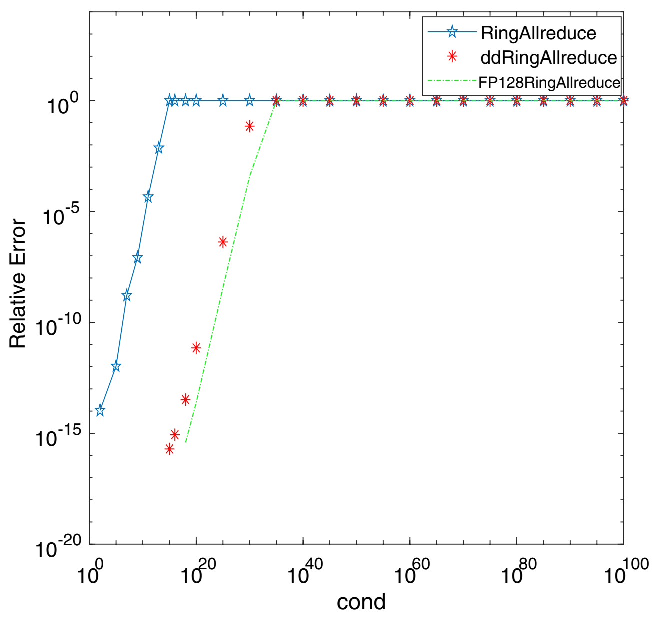
Then according to the series expansion, we can get

$$| \mathtt{res} - s | \leqslant \frac{(n-1)\mathrm{u}_{dd}}{1-(n-1)\mathrm{u}_{dd}} S.$$

Dividing both sides by s yields formula (5). □

4 Numerical results

The following experiment are performed on Sugon HPC cluster with 172 compute nodes (16 accelerator nodes), consisting of two 12-core processors each (24 cores per node). The MPI library used for this experiment is OpenMPI. Accuracy is evaluated by relative error $e =| \mathtt{res} - s | / | s |$, where $\mathtt{res}$ is an estimate of s. s is the exact value calculated



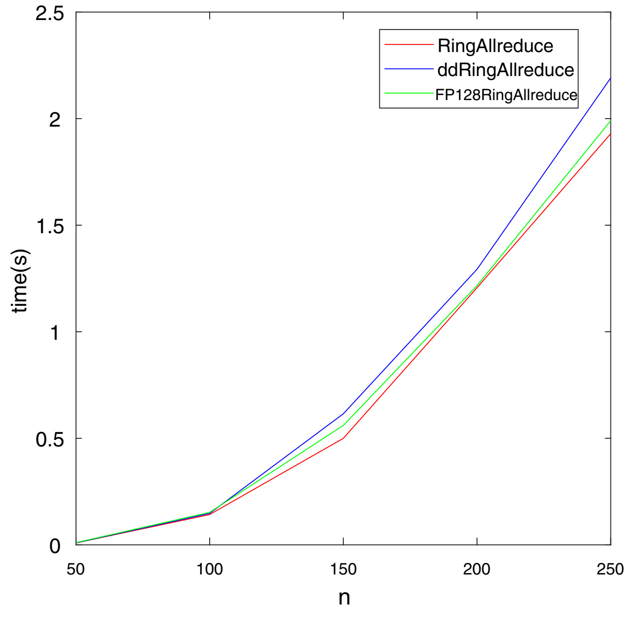


Fig. 4 Left: **a** Relative error image. Right: **b** CPU time image

by the MPFR library (Fousse etal. 2007) or known, MPFR is an arbitrary precision numerical library written in C language. The following are some explanations for the three calculation examples. In the three calculation examples, the data to be summed are serious positive and negative cancellations, and the final accurate values are all small. Such problems are prone to large condition numbers. It can be seen from the definition of the condition number that for this type of problem, the denominator in the calculation formula of the condition number for the summation problem is small while the numerator is relatively large. Therefore, the condition number is large, and ordinary recursive summation algorithms may not be able to provide accurate results. High-precision algorithms are needed instead.

4.1 Example 1

We use Algorithm 6.1 in Ogita et al. (2005) to generate arbitrarily ill-conditioned sum data. First generating ill-conditioned dot product data, the ill-conditioned sum data length is $2n$ generated from the dot product data of length n, and then the algorithm `TwoProd` is used to convert the ill-conditioned dot product data of length n to the ill-conditioned sum data of length $2n$ through error-free transformation. Finally, randomly disturbing $2n$ summation data can generate ill-conditioned sum data with different condition numbers.

The following are some experimental results and analysis of experimental results.

Figure 2a shows the relative error graph for a total data size of 200, using 200 processes, with each process having only one number. The reason for doing this is that it allows us to calculate the exact values of these 200 numbers, and thus we can calculate the relative error between the results obtained by our algorithm and the exact values. In practical applications, each process has multiple chunks, and each chunk has multiple values. Then each chunk performs a global reduction operation. Our algorithm and implementation can perform this operation, but we cannot calculate the exact value after each chunk reduction, so we cannot calculate the relative error of our proposed algorithm. Therefore, we use one number per process to verify the accuracy of our algorithm. In the later examples comparing accuracy, we use this approach.

Figure 2a shows the relative error graph for two algorithms. The horizontal axis represents the condition number of the summation problem, while the vertical axis represents the relative error. The exact value of the summation problem is calculated using the MPFR library. From Fig. 2a, it can be observed that the `ddRingAllreduce` algorithm provides results of the same order of magnitude as the machine precision, and numerical calculations cannot be expected to yield results more accurate than the machine precision. However, the relative error of the `RingAllreduce` algorithm increases as the condition number of the summation problem increases. When the condition number of the summation problem is around 10^{15}, the `RingAllreduce` algorithm gives completely incorrect results. Based on Fig. 2a, we can conclude that the `ddRingAllreduce` algorithm is more accurate than the `RingAllreduce` algorithm when the condition number of the summation problem is large.

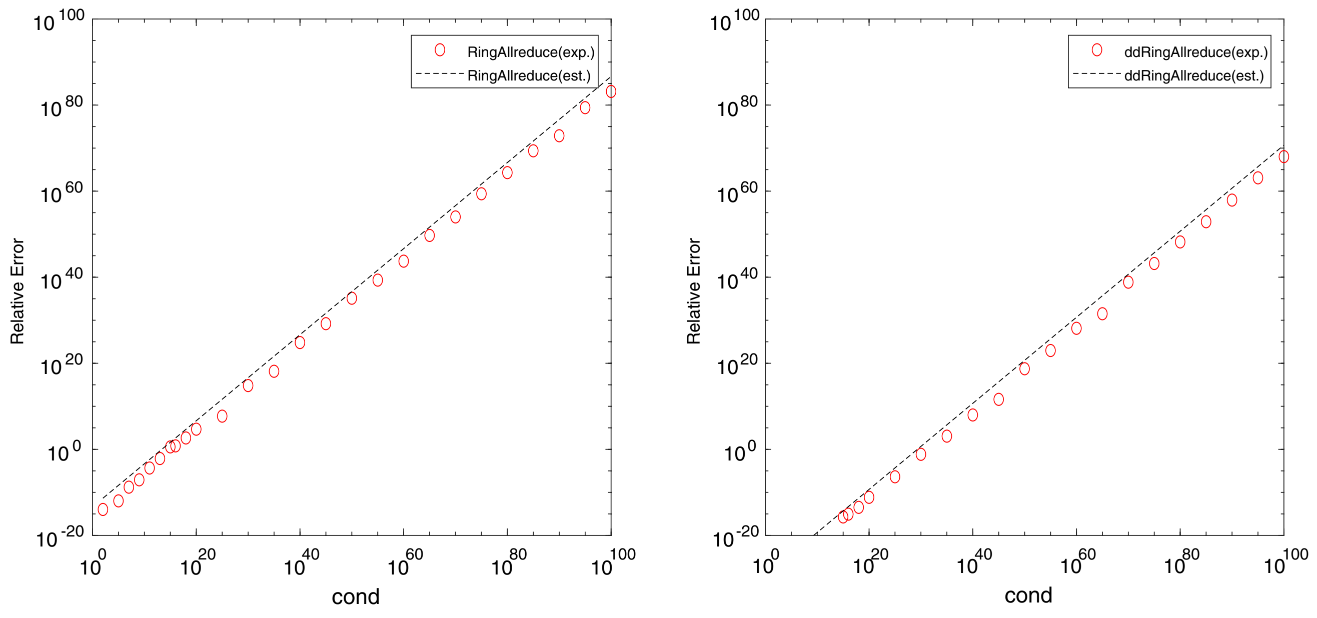


Fig. 5 Error bounds and true relative errors left: **a** RingAllreduce, right: **b** ddRingAllreduce

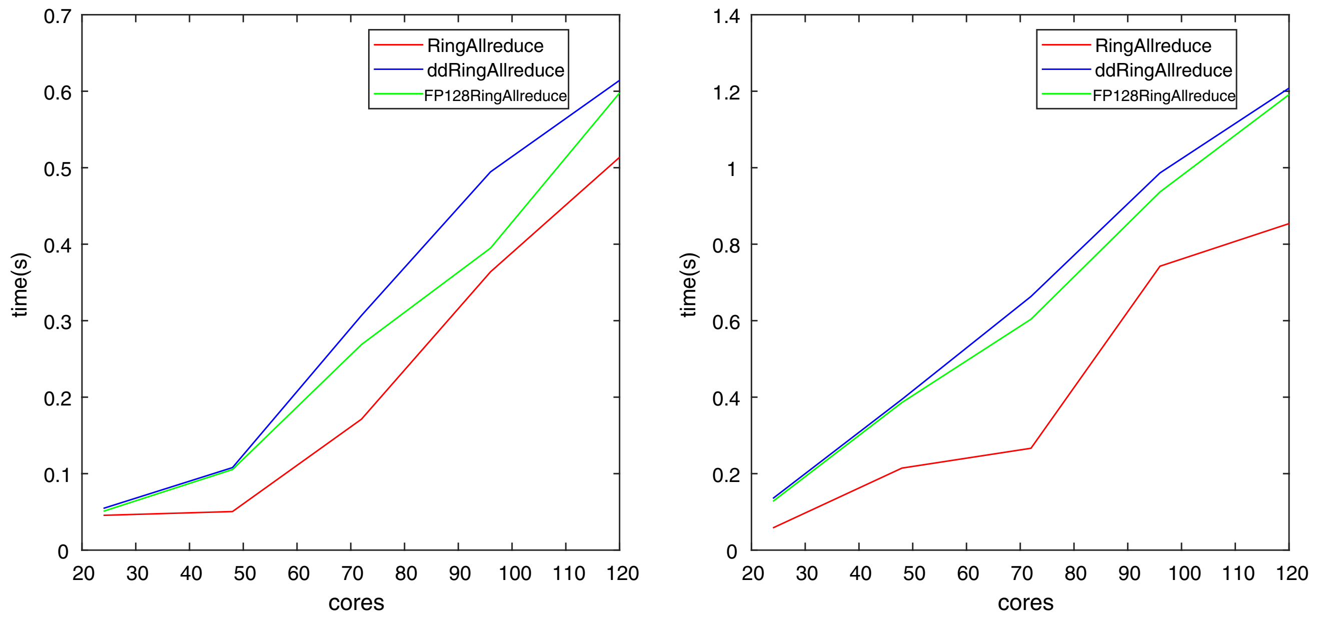


Fig. 6 Performance comparison for large scale problems left: **a** $K = 100,000$, right: **b** $K = 1,000,000$

Figure 2b shows the CPU time graph, with 5 runs averaged for each summation scale. In subsequent examples comparing CPU time, 5 runs are also performed and averaged. The vertical axis represents CPU time. When the data volume exceeds 100, `ddRingAllreduce` is slower than `RingAllreduce`. This is because when the data size is less than 100, the extra computational overhead brought by double-double precision in `ddRingAllreduce` is overlapped by communication time, so the additional floating-point operation overhead does not increase CPU time. In Fig. 2b, for the five summation scales, `ddRingAllreduce` algorithm is on average 1.0656 times slower than `RingAllreduce` algorithm.

4.2 Example 2

We use the same data generation method as the ReproBLAS library (Ahrens et al. 2015), i.e., $p_i = \sin(2.0 \times \pi \times (mpi\_rank \div mpi\_size - 0.5))$. Each process generates a number, where *mpi_rank* represents the process number and *mpi_size* represents the total number of processes.

Figure 3a shows the relative error image, with the number of summing data on the horizontal axis. `ddRingAllreduce` provides machine-precision-level results for all five different scales, while the results obtained by the `RingAllreduce` algorithm are incorrect for all five scales. From the image, it can be seen that the `ddRingAllreduce` algorithm is more accurate than the `RingAllreduce` algorithm. Figure 3b shows the CPU time image. Similar to Example 1, when the number of summing data is less than 100, the two algorithms have similar times. For the five different summing scales, the average time of `ddRingAllreduce` algorithm is 1.0906 times slower than that of the `RingAllreduce` algorithm.

4.3 Example 3

We use Algorithm 4.2 in Yamanaka et al. (2008) to generate arbitrarily ill-conditioned sum data. Convert the ill-conditioned dot product data into ill-conditioned sum data in the same way as in Example 1.

In this example, we will incorporate the use of *__float*128 in the scatter-reduce phase of `RingAllreduce` as part of the experiment, we refer to it as `FP128RingAllreduce`.

Figure 4a shows the relative error image. The accurate result of the summing problem is $cond^{-1}$. From the image, it can be seen that the `ddRingAllreduce` algorithm can handle summing problems with larger condition numbers, and is more accurate than the `RingAllreduce` algorithm for summing problems with larger condition numbers. `FP128RingAllreduce` is the most accurate algorithm among these three, because `FP128RingAllreduce` is 128-bit while `ddRingAllreduce` is only 106-bit. Figure 4b shows the CPU time image. Similar to Examples 1 and 2, when the number of summing data is less than 100, the two algorithms have similar times. For the five different summing scales, the average time of the `ddRingAllreduce` algorithm is 1.1238 times slower than that of the `RingAllreduce` algorithm. `FP128RingAllreduce` is on average 1.0829 times faster than `ddRingAllreduce`, but 1.0378 times slower than `RingAllreduce`. *__float*128 is supported by some compilers such as GCC, MPIC++, while double-double is a software simulation that requires more floating-point operations. Therefore, `FP128RingAllreduce` is faster than `ddRingAllreduce`. However, not all compilers support *__float*128, for example, NVCC does not support it. Therefore, when high precision is required, if the compiler supports *__float*128, we recommend using *__float*128 supported by the compiler. If the compiler does not support it, then use double-double.

Next, we evaluate how tight the error bound for `ddRingAllreduce` in Theorem 2 is in practice. To do this, we set $n = 200$ and vary the condition number *cond* from 1 to 10^{100} in Algorithm 4.2 (Yamanaka et al. 2008), the exact result of sum is equal to $cond^{-1}$. The error bounds (3) and true relative errors of the results obtained by `RingAllreduce` are displayed in Fig. 5a. The error bounds (5) and true relative errors of the results obtained by `ddRingAllreduce` are displayed in Fig. 5b. In Fig. 5, the lines labeled '**exp.**' denote the experimental error, the lines labeled '**est.**' denote the error bounds. It can be seen from this experiment that the theoretical error bounds of `RingAllreduce` and `ddRingAllreduce` are tight.

4.4 Performance comparison for large-scale problems

In this section, we compare the performance of `RingAllreduce`, `FP128RingAllreduce`, and `ddRingAllreduce` on two large-scale problems. The array size of each process is $K = 100,000$ and $K = 1,000,000$, respectively. Each element value is set to 1.5, and the number of nodes varies from 1 to 5 (i.e., the number of processes ranges from 24 to 120).

Figure 6 shows the comparison results. Figure 6a shows the experiment where the array size of each process is $K = 100,000$. The average time of the `ddRingAllreduce` algorithm is 1.3786 times slower than that of the `RingAllreduce` algorithm. `FP128RingAllreduce` is on average 1.1130 times faster than `ddRingAllreduce`, but 1.2385 times slower than `RingAllreduce`. Figure 6b shows the experiment with $K = 1,000,000$. The average time of the `ddRingAllreduce` algorithm is 1.5863 times slower than that of the `RingAllreduce` algorithm. `FP128RingAllreduce` is on average 1.0438 times faster than `ddRingAllreduce`, but 1.5198 times slower than `RingAllreduce`. On larger-scale problems, since the `ddRingAllreduce` algorithm uses double-double arithmetic and adds more floating-point operations, it is slower than the `RingAllreduce` algorithm.

5 Conclusions and future work

In this paper, we address the problem of inaccuracies in the `RingAllreduce` algorithm, a specific algorithm for global reduction operations. We propose a high-precision version of the `RingAllreduce` algorithm called `ddRingAllreduce`, which we analyze and verify to achieve higher accuracy than `RingAllreduce` algorithm. For large condition number summation problems, the `ddRingAllreduce` algorithm performs better in terms of accuracy than that of `RingAllreduce` algorithm, and in practice, the proposed algorithm often yields more accurate results than the theoretical error bounds. The `ddRingAllreduce` algorithm incurs less time overhead for small-scale problems, but it requires some overhead for large-scale problems.

For the future work, one can implement other high-precision reduction operations based on the `RingAllreduce` algorithm, such as multiplication. Or you can implement high-precision versions of other MPI_Allreduce algorithms, such as the butterfly algorithm. Although high-precision algorithms offer higher accuracy, they require more floating-point calculations and communication, so balancing computation speed and accuracy is always a research direction.

Acknowledgements The second author was supported by the foundation of key laboratory of computational physics, China. The third author is financially supported by the National Natural Science Foundation of China(62032023).

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

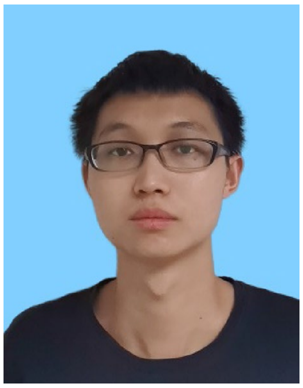
Ahrens, P., Nguyen, H., Demmel, J.: Efficient reproducible floating point summation and BLAS. ACM Trans. Math. Softw. **46**(3), 1–49 (2015)

Ahrens, P., Demmel, J., Nguyen, H.D.: Algorithms for efficient reproducible floating point summation. ACM Trans. Math. Softw. **46**(3), 1–49 (2020)

ANSI/IEEE.: IEEE Standard for Binary Floating Point Arithmetic, Std 754–2019. IEEE, New York (2019)

Blanchard, P., Higham, N., Mary, T.: A class of fast and accurate summation algorithms. SIAM J. Sci. Comput. **42**(3), 1541–1557 (2020)

Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. **18**, 224–242 (1971)
Demmel, J., Hida, Y.: Fast and accurate floating point summation with application to computational geometry. Numer. Algorithms **37**, 101–112 (2004)
Demmel, J., Nguyen, H.D.: Fast reproducible floating-point summation. In: Prof of the 21th IEEE Symposium on Computer Arithmetic, pp. 163–172 (2013)
Demmel, J., Nguyen, H.D.: Parallel reproducible summation. IEEE Trans. Comput. **64**(7), 2060–2070 (2015)
Dogru, A.H., Fung, L.S., Middya, U., Al-Shaalan, T.M., Tom B., Hahn H., Werner A.H., Al-Zamel, N., Pita, J., Hemanthkumar, K., et al.: Newfrontiers in large scale reservoir simulation. SPE (2011)
Fousse, L., Hanrot, G., Lefevre, V., Pelissier, P., Zimmermann, P.: MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33**, 13-es (2007)
Hida, Y., Li, X.S., Bailey, D.H.: Algorithms for quad-double precision floating point arithmetic. In: ARITH01, pp. 55–162 (2001)
Higham, N.: Accuracy and Stability of Numerical Algorithms, 2nd edn. SIAM Publications, Philadelphia (2002)
Iakymchuk, R., Collange, S., Defour, D., Graillat, S.: ExBLAS: reproducible and accurate BLAS library. NRE2015 (SC15) (2015)
Jiang, H.: Study on reliable computing and rounding error analysis in floating-point arithmetic (in Chinese). PhD Thesis, Changsha, National University of Defense Technology (2013)
Kimura, R.: Numerical weather prediction. J. Wind. Eng. Ind. Aerodyn. **90**, 1403–1414 (2002)
Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms, vol. 2. Addison-Wesley, Reading (1969)
Lei, X., Tongxiang, G., Graillat, S., et al.: A fast parallel high-precision summation algorithm based on AccSumK. J. Comput. Appl. Math. **406**, 0377–0427 (2021)
Lei, X., Gu, T., Graillat, S., Xu, X., Meng, J.: Comparison of reproducible parallel preconditioned BiCGSTAB algorithm based on ExBLAS and ReproBLAS. In: HPC Asia'23, Association for Computing Machinery, New York, pp 46–54 (2023)
Li, X.S., Demmel, J., Bailey, D.H., et al.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. **28**(2), 152–205 (2002)
Muller, J.M., Brisebarre, N., Dinechin, F.D.: Handbook of Floating-Point Arithmetic. Birkhäuser (2010)
Ogita, T., Rump, S., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. **26**(6), 1955–1988 (2005)
Patarasuk, P., Xin, Y.: Bandwidth optimal all-reduce algorithms for clusters of workstations. J. Parallel Distrib. Comput. **69**(2), 117–124 (2009)
Rabenseifner, R.: Optimization of collective reduction operations. In: LNCS 3036: International Conference on Computational Science, pp. 1–9 (2004)
Rabenseifner, R., Traff, J.L.: More efficient reduction algorithms for nonpower-of-two number of processors in message-passing parallel systems. In: LNCS 3241: EuroPVM/MPI, pp. 36–46 (2004)
Rump, S.: Ultimately fast accurate summation. SIAM J. Sci. Comput. **31**(5), 3466–3502 (2009)
Rump, S., Ogita, T., Oishi, S.: Accurate floating-point summation I: faithful rounding. SIAM J. Sci. Comput. **31**(1), 189–224 (2008)
Rump, S., Ogita, T., Oishi, S.: Accurate floating-point summation part II: sign K-Fold faithful and rounding to nearest. SIAM J. Sci. Comput. **31**(2), 1269–1302 (2008)
The MPI forum.: MPI: A Message-Passing Interface Standard, version 1.3 (2008). https://www.mpi-forum.org/docs/mpi-1.3/mpi-report-1.3-2008-05-30.pdf
van de Geijn, R.: On global combine operations. J. Parallel Distrib. Comput. **22**(2), 324–328 (1994)
Xiaowen, X., Zeyao, M., Hengbin, A.: Algebraic two-level iterative method for 2-D 3-T radiation diffusion equations. Chin. J. Comput. Phys. **26**(1), 1 (2009)
Yamanaka, N., Ogita, T., Rump, S., Oishi, S.: A parallel algorithm for accurate dot product. Parallel Comput. **34**(6–8), 392–410 (2008)
Zhou, Y.: A discussion on the matching relations among the word length, speed and memory space of digital electronic computer for the use of scientific calculation (in Chinese). J. Numer. Method Comput. Appl. **1**(3), 181–192 (1980)



Xiaojun Lei is a second-year doctoral student at the Institute of Applied Physics and Computational Mathematics. His research interests are parallel computing and machine learning. His research direction is the theory and method of high precision floating-point computing, and the reproducible Krylov subspace iteration method and machine learning method Solve PDEs.

Tongxiang Gu was born in April 21, 1964. He is a professor in computational mathematics and computer software. He got his doctor degree of computational mathematics from Chinese academy of engineering physics (CEAP) in 2001, and completed his postdoctoral studies in the institute of software of Chinese academy of sciences. Now he works in laboratory of computational physics in institute of applied physics and computational mathematics. He is a member of Beijing computational mathematics society. His major research interest is parallel computing and machine learning, which includes parallel algorithm, numerical algebra, preconditioning, numerical and ML simulation of PDEs, and development of parallel software. He has presided or participated in several projects in China, such as National Natural Science Foundation, National High Technology Research and Development Program (863 Program), National Basic Research Program (973 Program), defense basic research project, major pre-research projects of CAEP and key laboratory foundation of CAEP. The related work has won the second prize in scientific and technological progress of ministerial-level. So far, he has published three monographs, an undergraduate textbook, and more than 100 papers about large sparse algebraic equations, parallel iterative algorithms, preconditioning technologies, and machine learning for PDEs.

Xiaowen Xu is a Professor of Institute of Applied Physics and Computational Mathematics (IAPCM), China. He is the deputy director of IAPCM. He got his B.S degree from Xiangtan University in 2002, and his PhD degree in computational mathematics from Chinese Academy of Engineering Physics in 2007. His research interests include high performance numerical algorithm & software in scientific and engineering fields, parallel programming framework for large-scale numerical simulations. He is member of CCF and member of SIAM and CSIAM.