# A Fast Parallel High-Precision Summation Algorithm Based on AccSumK

Xiaojun Lei, Tongxiang Gu, Stef Graillat, Hao Jiang, Jin Qi

# A Fast Parallel High-Precision Summation Algorithm Based on AccSumK

Xiaojun Lei[a], Tongxiang Gu[b,*], Stef Graillat[c], Hao Jiang[d], Jin Qi[b]

[a]*Graduate School of Chinese Academy of Engineering Physics, Beijing 100088, China*
[b]*Institute of Applied Physics and Computational Mathematics, Beijing 100094, China*
[c]*Sorbonne Université, CNRS, LIP6, Paris F-75005, France*
[d]*National University of Defense Technology, Changsha, 410073, China*

**Abstract**

In this paper, we present a new parallel accurate algorithm called `PAccSumK` for computing summation of floating-point numbers. It is based on `AccSumK` algorithm. In the experiment, for the summation problems with large condition numbers, our algorithm outperforms in term of accuracy and computing time the `PSumK` algorithm. The reason is that our algorithm is based on a more accurate algorithm called `AccSumK` algorithm compared to the `SumL` algorithm used in `PSumK`. The proposed parallel algorithms in this paper is designed to compute a result as if computed internally in $K$-fold the working precision. Numerical results are presented showing the performance and the accuracy of our new parallel algorithm for calculating summation.

*Keywords:* Parallel algorithms, Accurate summation, Higher precision, Floating-point arithmetic

## 1. Introduction and Motivation

In this paper, we present a new parallel algorithm to compute a sum of floating-point numbers with high accuracy. Since summation is one of the most basic tasks in numerical analysis, getting fast and accurate algorithms

*Corresponding author

*Email addresses:* `leixiaojun19@gscaep.ac.cn` (Xiaojun Lei), `txgu@iapcm.ac.cn` (Tongxiang Gu), `stef.graillat@lip6.fr` (Stef Graillat), `haojiang@nudt.edu.cn` (Hao Jiang), `qi_jin@iapcm.ac.cn` (Jin Qi)

is an important problem. Moreover, computing accurate summation has numerous applications in numerical analysis and scientific computing.

The authors (Rump, Ogita and Oishi) presented an accurate sum algorithm called `AccSumK` [2, 3] using so-called "error-free transformations". Given a vector of floating-point numbers with exact sum $s$, their algorithm calculates a $K$-fold faithful rounding of $s$. The algorithm is fast for mildly conditioned sums with slowly increasing computing time proportional to the logarithm of the condition number. The algorithm is fast, not only in terms of the number of floating-point operations but also in terms of measured computing time on a serial computer. But it is not a parallel algorithm. The authors (Yamanaka, Ogita, Rump and Oishi) presented a parallel algorithm `PSumK` [5]. The `PSumK` algorithm is based on the `SumL` [5] and `SumK` [1] algorithms. However, the accuracy of `SumL` depends on the condition number of the sum whereas the accuracy of `AccSumK` is independent of the condition number. For ill-conditioned problems, the `AccSumK` algorithm is more accurate than the `SumL` algorithm.

Based on the above observations on `AccSumK` and `PSumK`, we develop a parallelizing method to used `AccSumK` into a new parallel algorithm called `PAccSumK`.

The rest of the paper is organized as follows: In Section 2, we briefly review the error-free transformations and the algorithm `AccSumK` for accurate summation proposed in [2, 3]. In Section 3, we develop parallel algorithm `PAccSumK`, which is based on `AccSumK` algorithm. We propose an analysis of the error bounds of the proposed algorithm. This makes it possible to confirm that the proposed parallel algorithm achieves a result as if computed in $K$-fold the working precision. In Section 4, we present results of numerical experiments showing the performance and the accuracy of the proposed algorithm `PAccSumK`. Finally, we conclude the paper and propose some future work.

Throughout the paper, floating-point addition, subtraction and multiplication are counted as one floating-point operation (flop). Moreover, we use MATLAB-style notation for describing algorithms.

## 2. Accurate summation algorithm AccSumK

In this section, we use the same notation as in [3, 5] and we recall some results and algorithms described in [1, 2, 3, 5]. The set of floating-point numbers is denoted by $\mathbb{F}$, and $\mathbb{U}$ denotes the set of subnormal floating-point

numbers together with zero and the two normalized floating-point numbers of smallest nonzero magnitude. Throughout this paper, we assume floating-point arithmetic adhering to IEEE standard 754 [7]. Let $p = (p_i) \in \mathbb{F}^n$ and let $\mathrm{fl}(\cdot)$ 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, and the underflow unit by $\eta$, that is the smallest positive (denormalized) floating-point number. In IEEE standard 754 double precision $\mathtt{u} = 2^{-53}$ and $\eta = 2^{-1074}$.

Let us denote $s$ and $S$ by

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

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

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

A computed result $\mathtt{Res}$ of the summation of $p = (p_i)$ is said to be computed in $K$-fold precision if it satisfies[1]

$$|\mathtt{Res} - \sum_{i=1}^{n} p_i| \leq \mathtt{u}|s| + (\varphi\mathtt{u})^K \sum_{i=1}^{n} |p_i|, \tag{1}$$

with a moderate constant $\varphi$. The second term in the right-hand side of (1) reflects computation $K$-fold precision, and the first one rounding back into working precision.

Because "unit in the last place" (ulp) depends on the floating-point format and needs extra care in the underflow range. It is useful to use the "unit in the first place" (ufp) defined in [3] by

$$0 \neq r \in \mathbb{R} \quad \Rightarrow \quad \mathrm{ufp}(r) := 2^{\lfloor \log_2 |r| \rfloor}, \tag{2}$$

where $\mathrm{ufp}(0) := 0$.

In order to explain faithful rounding, the definition of floating-point predecessor and successor is required.

$\mathrm{pred}(r) := \max\{f \in \mathbb{F} : f < r\}$ and $\mathrm{succ}(r) := \min\{f \in \mathbb{F} : f < r\}$.

**Definition 1.** [3] A floating-point number $f \in \mathbb{F}$ is called a faithful rounding of a real number $r \in \mathbb{R}$ if

$$\text{pred}(f) < r < \text{succ}(f). \tag{3}$$

We denote this by $f \in \square(r)$. For $r \in \mathbb{F}$ this implies $f = r$.

**Definition 2.** [3] A sequence $f_1, \ldots, f_k \in \mathbb{F}$ is called a ($K$-fold) faithful rounding of $s \in \mathbb{R}$ if

$$f_i \in \square(s - \sum_{\nu=1}^{i-1} f_\nu) \qquad for\ 1 \le i \le k. \tag{4}$$

Following [4], we define $\gamma_n$ as

$$\gamma_n := \frac{n\mathtt{u}}{1 - n\mathtt{u}}, \quad n \in \mathbb{N}.$$

When using $\gamma_n$, we implicitly assume that $n\mathtt{u} < 1$.

Let $p = (p_1, ..., p_n)^T \in \mathbb{F}^n$. Then it holds that [4]

$$\tilde{s} := \text{fl}(\sum_{i=1}^{n} p_i) \Rightarrow |\tilde{s} - \sum_{i=1}^{n} p_i| \le \gamma_{n-1} \sum_{i=1}^{n} |p_i|. \tag{5}$$

Note that (5) is valid for any order of addition in the summation. Recently, it was shown that $\gamma_n$ can be replaced by $n\mathtt{u}$, and the restriction on $n$ can be removed [12, 13, 14, 15].

The algorithm `AccSumK` is based on the error-free transformations of addition and/or multiplication of two floating-point numbers.

We first recall a fast algorithm for adding two floating-point numbers. It requires only 3 flops. When $|a| \ge |b|$, Algorithm 2.1 is an error-free transformation that satisfies $x + y = a + b$ with $x = \text{fl}(a + b), |y| \le \mathtt{u} \cdot \text{ufp}(x)$.

Algorithm 2.1 [6]. *Compensated summation of two floating-point numbers.*

function $[x, y] =$ `FastTwoSum`$(a, b)$
$x = \text{fl}(a + b)$
$q = \text{fl}(x - a)$
$y = \text{fl}(b - q)$

Choose a split to extract a vector into a sum of higher order parts and a vector of lower order parts.

Algorithm 2.2 [3]. *Error-free vector transformation extracting high order part.*

function $[\tau, p'] = \texttt{ExtractVector}(\sigma, p)$
  $\tau = 0$
  for $i = 1{:}n$
    $q_i = \mathrm{fl}((\sigma + p_i) - \sigma)$
    $p'_i = \mathrm{fl}(p_i - q_i)$
    $\tau = \mathrm{fl}(\tau + q_i)$
  end for

Algorithm 2.3 [3]. *Computation of* $2^{\lceil log_2|p| \rceil}$ *for* $p \neq 0$.

function $L = \texttt{NextPowerTwo}(p)$
  $q = \mathtt{u}^{-1}p$
  $L = \mathrm{fl}(|(q + p) - q|)$
  if $L == 0$
    $L = |p|$
  end if

**Theorem 1.** [3] *Let $L$ be the result of Algorithm 2.3 (*`NextPowerTwo`*) applied to a nonzero floating-point number $p$. If no overflow occurs, then $L = 2^{\lceil log_2|p| \rceil}$.*

**Remark 1.** [3] $L = 2^{\lceil log_2|p| \rceil}$ is satisfied in the presence of underflow, and use Algorithm 2.3 (`NextPowerTwo`) so that only basic floating-point operations are necessary.

We let the input vector $p$ be $p^{(0)}$ and the output vector $p$ be $p^{(m)}$. The main part of the summation algorithm is Algorithm 2.4. It transforms the input vector $p^{(0)}$ into two floating-point numbers $\tau_1$, $\tau_2$ representing the high order part of the sum and a vector $p^{(m)}$ of low order parts. The transformation is error-free, i.e., $s := \sum p_i^{(0)} = \tau_1 + \tau_2 + \sum p_i^{(m)}$. Moreover, $\mathrm{fl}(\tau_1 + \tau_2 + \sum p_i^{(m)})$ is a faithfully rounded result of $s$ (see [2]).

Algorithm 2.4 [3]. *Transformation of vector $p$ plus $\varrho$ with check for zero.*

function $[\tau_1, \tau_2, p, \sigma] = \texttt{Transform}(p, \varrho)$
  $\mu = \max(|p_i|)$
  if $\mu = 0, \tau_1 = \varrho, \tau_2 = 0$, return, end if
  $M = \texttt{NextPowerTwo}(\mathrm{length}(p_0){+}2)$
  $\sigma' = 2^M \texttt{NextPowerTwo}(\mu)$
  $t' = \varrho$

repeat
$t = t^{'}; \sigma = \sigma^{'}$
$[\tau, p] = \texttt{ExtractVector}(\sigma, p)$
$t^{'} = \mathrm{fl}(t + \tau)$
if $t^{'} = 0$, $[\tau_1, \tau_2, p] = \texttt{Transform}(p, 0)$; return; end if
$\sigma^{'} = \mathrm{fl}(2^M \mathtt{u}\sigma)$
until $\sigma \leq \frac{1}{2}\mathtt{u}^{-1}\eta$ or $|t^{'}| \geq \mathrm{fl}(2^{2M+1}\mathtt{u}\sigma)$
$[\tau_1, \tau_2] = \texttt{FastTwoSum}(t, \tau)$

The following Algorithm 2.5 establishes an approximation of $K$-fold accuracy.

Algorithm 2.5 [3]. *Error-free vector transformation including faithful rounding.*

function $[\texttt{res}, R, p^{'}] = \texttt{TransformK}(p, \varrho)$
$[\tau_1, \tau_2, p^{'}, \sigma] = \texttt{Transform}(p, \varrho)$
$\texttt{res} = \mathrm{fl}(\tau_1 + (\tau_2 + (\sum_{i=1}^{n} p_i^{'})))$
$R = \mathrm{fl}(\tau_2 - (\texttt{res} - \tau_1))$

Algorithm 2.6 computes a result of $K$-fold accuracy, stored in a non-overlapping result vector Res of length $K$.

Algorithm 2.6 [3]. *Accurate summation with faithful rounding and result of $K$-fold accuracy.*

function $\texttt{Res} = \texttt{AccSumK}(p, K)$
$p^{(0)} = p$, $R_0 = 0$
for $k = 1 : K$
$[\texttt{Res}_k, R_k, p^{(k)}] = \texttt{TransformK}(p^{(k-1)}, R_{k-1})$
if $\texttt{Res}_k \in \mathbb{U}, \texttt{Res}_{k+1\ldots K} = 0$, return; end if
end for

Figure 1 illustrates an outline of AccSumK for $K = 3$.

**Theorem 2.** [3] *Let $p$ be a vector of $n$ floating-point numbers. Define $M := \lceil log_2(n+2) \rceil$, and assume $2^{2M}\mathtt{u} \leq 1$. Let $1 \leq K \in \mathbb{N}$ be given, and let* Res *be the results vector of Algorithm 2.6 (*AccSumK*) applied to $p$ and $K$.*

*Then,* $\texttt{Res}_1, \ldots, \texttt{Res}_K$ *is a faithful rounding of $s$. Furthermore,*

$$s = \sum_{\nu=1}^{k} \texttt{Res}_\nu + R_k + \sum_{i=1}^{n} p_i^{(k)} \quad for 1 \leq k \leq K, \tag{6}$$

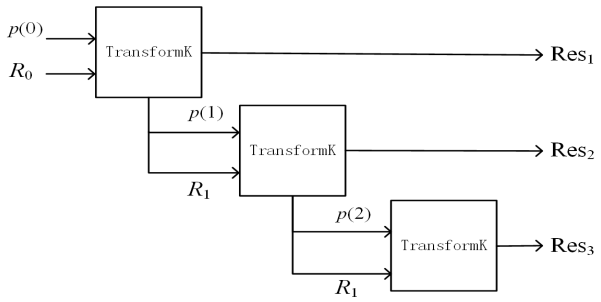


Figure 1: Outline of `AccSumK` for $K = 3$

*Abbreviate* $\hat{\mathtt{Res}} := \sum_{k=1}^{K} \mathtt{Res}_k$. *Then* $\mathtt{Res}_k \in \mathbb{U}$ *for some* $1 < k < K$ *implies* $\hat{\mathtt{Res}} = s$, *and* $\mathtt{Res}_k \notin \mathbb{U}$ *implies*

$$|s - \hat{\mathtt{Res}}| < 2\mathtt{u}^K \mathrm{ufp}(s) \leq 2\mathtt{u}^K |s|, \tag{7}$$

*and*

$$|s - \hat{\mathtt{Res}}| < 2\mathtt{u}^K \mathrm{ufp}(\mathtt{Res}_1) \leq 2\mathtt{u}^K |\mathtt{Res}_1|. \tag{8}$$

**Remark 2.** [3] `AccSumK` needs $(4m + 5K + 3)n + \mathcal{O}(m + K)$ flops if the "repeat-until" loop in the first extraction `TransformK` is executed $m$ times.

Next, we briefly review some algorithms used in the `SumK` and `SumL` algorithms. The following Algorithm 2.7 is proposed by Knuth. It needs 6 flops.

Algorithm 2.7 [8]. *Error-free transformation of the sum of two floating-point numbers.*

function $[x, y] = \mathtt{TwoSum}(a, b)$
$x = \mathrm{fl}(a + b)$
$q = \mathrm{fl}(x - a)$
$y = \mathrm{fl}((a - (x - q)) + (b - q)))$

Using the `TwoSum` algorithm, it can be extended to the error-free transformation of vector summation.

Algorithm 2.8 [1]. *Error-free vector transformation with respect to the summation.*

function $q = \mathtt{VecSum}(p)$
$q_1 = p_1$

for $i = 2 : n$
  $[q_i, q_{i-1}] = \texttt{TwoSum}(p_i, q_{i-1})$
end for

In [1], the authors (Ogita, Rump and Oishi) developed an algorithm `SumK`, which calculates the summation using `VecSum` iteratively.

Algorithm 2.9 [1]. *Summation $\sum_{i=1}^{n} p_i$ for $p \in \mathbb{F}^n$ as in $K$-fold precision by $(K-1)$-fold error-free vector transformation.*

function $\texttt{res} = \texttt{SumK}(p, K)$
  for $i = 1 : K-1$
    $p = \texttt{VecSum}(p)$
  end for
  $\texttt{res} = \mathrm{fl}(\sum_{i=1}^{n} p_i)$

The error bounds of the result `res` by `SumK` are given as follows [1]:

**Theorem 3.** [1] *Let* `res` *be the result obtained by* `SumK`*, then*

$$|\texttt{res} - s| \le (\mathtt{u} + 3\gamma_{n-1}^2)|s| + \gamma_{2(n-1)}^K S. \tag{9}$$

*Moreover, if $s \neq 0$, then*

$$\left|\frac{\texttt{res} - s}{s}\right| \le \mathtt{u} + 3\gamma_{n-1}^2 + \gamma_{2(n-1)}^K \mathrm{cond}(\sum p_i). \tag{10}$$

In [5], the authors developed a summation algorithm `SumL` whose result is represented by an array of $L$ floating-point numbers.

Algorithm 2.10 [5]. *Summation $\sum_{i=1}^{n} p_i$ for $p \in \mathbb{F}^n$ as in $L$-fold precision whose result is represented by a sum of $L$ floating-point numbers.*

function $q = \texttt{SumL}(p, L)$
  for $k = 1 : L-1$
    $p(1 : n-k+1) = \texttt{VecSum}(p(1 : n-k+1))$
    $q_k = p_{n-k+1}$
  end for
  $q_L = \mathrm{fl}(\sum_{i=1}^{n-L+1} p_i)$

For later use, we present an error bound and inequality for `SumL`.

**Theorem 4.** [5] *Let $q$ be the result obtained by Algorithm 2.10 (*`SumL`*), abbreviate $\hat{q} := \sum_{k=1}^{L} q_k$, then*

$$|\hat{q} - s| \le \gamma_{n-1}^L S, \tag{11}$$

*Moreover, if* $s \neq 0$*, then*

$$|\frac{\hat{q}-s}{s}| \leq \gamma_{n-1}^{L}\mathrm{cond}(\sum p_i). \tag{12}$$

*Algorithm 2.10 also satisfies the following inequality:*

$$\sum_{k=1}^{L}|q_k| \leq (1+\gamma_{2(n-1)})S. \tag{13}$$

**Remark 3.** Let $L = K$, divide the right term of (11) by the right term of (7), we can get

$$\frac{\gamma_{n-1}^{K}S}{2\mathrm{u}^{K}|s|} \approx \frac{(n-1)^{K}}{2}\mathrm{cond}(\sum p_i). \tag{14}$$

From (14) we know that the error bounds of `AccSumK` is smaller than that of `SumL`.

## 3. Parallel algorithm

In this section, we will develop parallel algorithm for calculating sum internally in $K$-fold working precision. It is based on the `AccSumK` algorithm, which is named `PAccSumK`.

Suppose the number of CPUs to be $P \geq 2$ on a distributed memory system. Then the CPUs can be numbered $id$ from 1 to $P$, so $1 \leq id \leq P$.

`PAccSumK` is the same algorithm as `PSumK` [5], but the difference is that we use the `AccSumK` algorithm instead of `SumL` and `SumK`.

Next, we present here the concrete algorithm of `PAccSumK`.

Algorithm 3.1 (`PAccSumK`). A parallel algorithm based on the `AccSumK` algorithm.

function `Res` = `PAccSumK`$(p, K, P)$
  $c = \lceil n/P \rceil$, $c_1 = n - c(P-1)$
  % *parallel private* (index1, index2)
  *if* $id == 1$
    index1 $= 1 : c_1$
  *else*
    index1 $= c_1 + c(id-2) + 1 : c_1 + c(id-1)$
  *end*
  index2 $= K(id\text{-}1)+1 : K \cdot id$

$q$(index2) = `AccSumK`($p$(index1),$K$)
% *end parallel*
`Res` = `AccSumK`($q, K$)

**Remark 4.** When the `AccSumK` algorithm in the parallel part of Algorithm 3.1 is replaced with the `SumL` algorithm, and the `AccSumK` algorithm in the serial part is replaced with the `SumK` algorithm, it is the `PSumK` algorithm. In `PSumK`, computational cost for the parallelized part is $(6K-5)c$ flops and the rest requires $(6K-5)(PK-1)$ flops [5]. When the `AccSumK` algorithm in Algorithm 3.1 is replaced with the `SumL` algorithm, we call it `PSumL`. In `PSumL`, computational cost for the parallelized part is $(6K-5)c$ flops and the rest requires $(6K-5)PK$ flops. This is a unified framework, the parallel part and the serial part can use different summation algorithms, we focus on the `PAccSumK` algorithm.

In `PAccSumK`, computational cost for the parallelized part is $(4m+5K+3)c+\mathcal{O}(m+K)$ flops and the rest requires $(4m+5K+3)(PK)+\mathcal{O}(m+K)$ flops. Therefore, the theoretical parallel efficiency in terms of the flops count becomes as follows:

$$\begin{aligned} r_1 &= \frac{(4m+5K+3)c+\mathcal{O}(m+K)}{(4m+5K+3)c+\mathcal{O}(m+K)+(4m+5K+3)(PK)+\mathcal{O}(m+K)} \\ &\approx \frac{1}{1+\frac{PK}{c}}. \end{aligned}$$

If $c \gg PK$, then $r_1$ approaches to one.

We present the following theorem for `PAccSumK`.

**Theorem 5.** *Let* `Res` *be the result obtained by Algorithm 3.1 (*`PAccSumK`*), abbreviate* $\hat{\texttt{Res}} := \sum_{k=1}^{K} \texttt{Res}_k$. *Define* $c := \lceil \frac{n}{P} \rceil$, $M' := \lceil log_2(c+2) \rceil$, *and assume* $2^{2M'}\mathtt{u} \leq 1$. *Let* $1 \leq K \in \mathbb{N}$ *be given. Then the following inequality holds:*

$$|\hat{\texttt{Res}} - s| < 2\mathtt{u}^K|s| + (1+2\mathtt{u}^K)2\mathtt{u}^K S. \tag{15}$$

*Moreover, if* $s \neq 0$, *then*

$$|\frac{\hat{\texttt{Res}} - s}{s}| < 2\mathtt{u}^K + (1+2\mathtt{u}^K)2\mathtt{u}^K \mathrm{cond}(\sum p_i). \tag{16}$$

Figure 3 illustrates an outline of `PAccSumK` for $K=3$ and $P=3$.

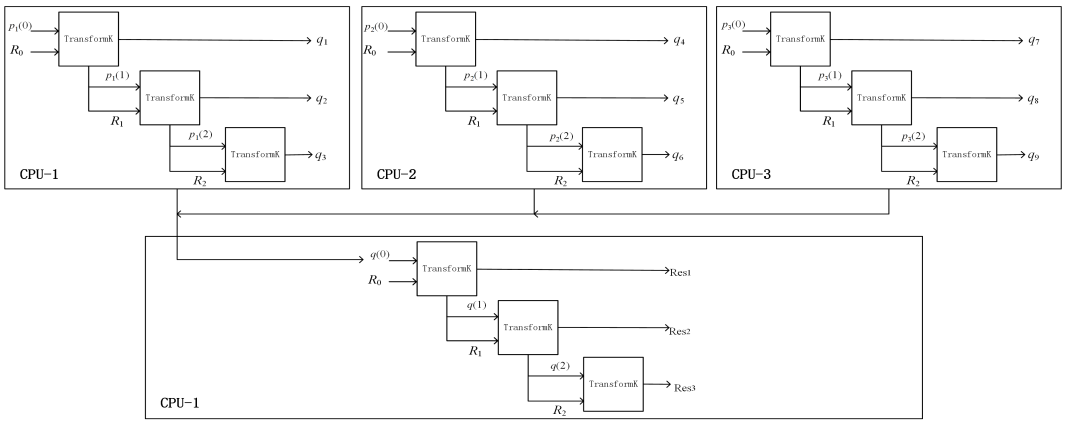


Figure 2: Outline of `PAccSumK` for $K = 3$ and $P = 3$

**Proof**. From the definition of vectors $p$, $p^{(id)}$, $q$ and $q^{(id)}$, we collect the following relations:

$$\sum_{i=1}^{n} p_i = \sum_{id=1}^{P} \sum_{i=1}^{c_{id}} p_i^{(id)} = s, \tag{17}$$

$$\sum_{i=1}^{n} |p_i| = \sum_{id=1}^{P} \sum_{i=1}^{c_{id}} |p_i^{(id)}| = S, \tag{18}$$

$$\sum_{j=1}^{PK} q_j = \sum_{id=1}^{P} \sum_{k=1}^{K} q_k^{(id)}, \tag{19}$$

$$\sum_{j=1}^{PK} |q_j| = \sum_{id=1}^{P} \sum_{k=1}^{K} |q_k^{(id)}|. \tag{20}$$

Then

$$|\hat{\texttt{Res}} - s| = |\hat{\texttt{Res}} - \sum_{j=1}^{PK} q_j + \sum_{j=1}^{PK} q_j - s| \leq |\hat{\texttt{Res}} - \sum_{j=1}^{PK} q_j| + |\sum_{j=1}^{PK} q_j - s|. \tag{21}$$

Here, using $\texttt{Res}$=$\texttt{AccSumK}(q, K)$ and (7) yields

$$|\hat{\texttt{Res}} - \sum_{j=1}^{PK} q_j| < 2\texttt{u}^K |\sum_{i=1}^{PK} q_i| \leq 2\texttt{u}^K (|\sum_{i=1}^{PK} q_i - s| + |s|), \tag{22}$$

Inserting (22) into (21), we have

$$|\hat{\texttt{Res}} - s| < 2\mathtt{u}^K|s| + (1+2\mathtt{u}^K)|\sum_{i=1}^{PK} q_i - s|. \tag{23}$$

It follows by (17) and (19) that

$$|\sum_{j=1}^{PK} q_j - s| = |\sum_{id=1}^{P}\sum_{k=1}^{K} q_k^{(id)} - \sum_{id=1}^{P}\sum_{i=1}^{c_{id}} p_i^{(id)}| \le \sum_{id=1}^{P}|\sum_{k=1}^{K} q_k^{(id)} - \sum_{i=1}^{c_{id}} p_i^{(id)}|. \tag{24}$$

Here, recalling that $q^{(id)}$=$\texttt{AccSumK}(p^{(id)}, K)$ and using (7), we have

$$|\sum_{j=1}^{PK} q_j - s| < \sum_{id=1}^{P}(2\mathtt{u}^K|\sum_{i=1}^{c_{id}} p_i^{(id)}|) \le 2\mathtt{u}^K S. \tag{25}$$

Inserting (25) into (23), we finally have

$$|\hat{\texttt{Res}} - s| < 2\mathtt{u}^K|s| + (1+2\mathtt{u}^K)2\mathtt{u}^K S,$$

which proves (15) and divided by $|s|$ also proves (16). □

Theorem 5 and (1) mean the result Res is obtained by PAccSumK as if computed internally in $K$-fold working precision. In fact, we can construct $p$ such that the right-hand side equation of (25) holds.

**Remark 5.** The PAccSumK algorithm does not share the same accuracy as AccSumK. Indeed, the accuracy of PAccSumK depends on the condition number of the problem and so is not a $K$-fold faithful rounding. Because Res is the $K$-fold faithful rounding of $q$, which will produce an error, $q^{(id)}$ is the $K$-fold faithful rounding of $p^{(id)}$, and each process will also produce an error, Res is not the $K$-fold faithful rounding of $p$.

In order to compare PAccSumK and PSumK, we quote the following theorem for PSumK.

**Theorem 6.** [5] *Let* res *be the result obtained by* PSumK. *Define* $c := \lceil \frac{n}{P} \rceil$. *Suppose*
$2\max\{c, PK\}\mathtt{u} < 1$. *Then the following inequality holds:*

$$|\texttt{res} - s| \le (\mathtt{u} + 3\gamma_{PK-1}^2)|s| + \phi_1 S, \tag{26}$$

*where* $\phi_1 := (1+\mathtt{u}+3\gamma_{PK-1}^2)\gamma_{c-1}^K + (1+\gamma_{2(c-1)})\gamma_{2(PK-1)}^K$.
*Moreover, if* $s \neq 0$, *then*

$$|\frac{\mathtt{res}-s}{s}| \leq \mathtt{u} + 3\gamma_{PK-1}^2 + \phi_1 \mathrm{cond}(\sum p_i). \tag{27}$$

The second term at the right end (15) is smaller than the second term at the right end (26), therefore, when $K$ is the same, `PAccSumK` can solve problems with larger condition numbers than `PSumK`. The first term at the right end (15) is also smaller than the first term at the right end (26). The error of the algorithm is dominated by the second term.

In order to compare `PAccSumK` and `PSumL`, we present the following theorem for `PSumL`.

**Theorem 7.** *Let* `Res` *be the result obtained by* `PSumL`, *abbreviate* $\hat{\mathtt{Res}} := \sum_{k=1}^{L} \mathtt{Res}_k$. *Define* $c := \lceil \frac{n}{P} \rceil$. *Suppose* $2\max\{c, PK\}\mathtt{u} < 1$. *Then the following inequality holds:*

$$|\hat{\mathtt{Res}} - s| \leq \{\gamma_{c-1}^L + (1+\gamma_{2(c-1)})\gamma_{n-1}^L\}S. \tag{28}$$

*Moreover, if* $s \neq 0$, *then*

$$|\frac{\hat{\mathtt{Res}}-s}{s}| \leq \{\gamma_{c-1}^L + (1+\gamma_{2(c-1)})\gamma_{n-1}^L\}\mathrm{cond}(\sum p_i). \tag{29}$$

**Proof**. We have

$$|\hat{\mathtt{Res}} - s| = |\hat{\mathtt{Res}} - \sum_{j=1}^{PL} q_j + \sum_{j=1}^{PL} q_j - s| \leq |\hat{\mathtt{Res}} - \sum_{j=1}^{PL} q_j| + |\sum_{j=1}^{PL} q_j - s|. \tag{30}$$

Here, using `Res`=`SumL`$(q, L)$ and (11) yields

$$|\hat{\mathtt{Res}} - \sum_{j=1}^{PL} q_j| \leq \gamma_{n-1}^L \sum_{i=1}^{PL} |q_i|, \tag{31}$$

Inserting (31) into (30), we have

$$|\hat{\mathtt{Res}} - s| \leq |\sum_{i=1}^{PL} q_i - s| + \gamma_{n-1}^L \sum_{i=1}^{PL} |q_i|. \tag{32}$$

It follows by (17) and (19) that

$$|\sum_{j=1}^{PL} q_j - s| = |\sum_{id=1}^{P}\sum_{k=1}^{L} q_k^{(id)} - \sum_{id=1}^{P}\sum_{i=1}^{c_{id}} p_i^{(id)}| \le \sum_{id=1}^{P} |\sum_{k=1}^{L} q_k^{(id)} - \sum_{i=1}^{c_{id}} p_i^{(id)}|. \quad (33)$$

Here, recalling that $q^{(id)}$=$\texttt{SumL}(p^{(id)}, L)$ and using (11), we have

$$|\sum_{j=1}^{PL} q_j - s| \le \sum_{id=1}^{P} (\gamma_{c_{id}-1}^{L} \sum_{i=1}^{c_{id}} |p_i^{(id)}|) \le \gamma_{c-1}^{L} S. \quad (34)$$

Furthermore, applying (13) to the right-hand side of (20) and using (18) yield

$$\sum_{j=1}^{PL} |q_j| = \sum_{id=1}^{P} (\sum_{j=1}^{L} |q_j^{(id)}|) \le \sum_{id=1}^{P} ((1+\gamma_{2(c_{id}-1)}) \sum_{i=1}^{c_{id}} |p_i^{(id)}|) \le (1+\gamma_{2(c-1)}) S. \quad (35)$$

Inserting (35) and (34) into (32), we finally have

$$|\hat{\texttt{Res}} - s| \le \{\gamma_{c-1}^{L} + (1+\gamma_{2(c-1)})\gamma_{n-1}^{L}\} S,$$

which proves (28) and divided by $|s|$ also proves (29). □

Let $L = K$, divide the right term in (29) by the second term on the right in (16).

$$\frac{\gamma_{c-1}^{K} + (1+\gamma_{2(c-1)})\gamma_{n-1}^{K}}{(1+2\texttt{u}^{K})2\texttt{u}^{K}} \approx \frac{(c-1)^K + (n-1)^K}{2}. \quad (36)$$

From (36) we know that the error bounds of `PAccSumK` is smaller than that of `PSumL`. The error bounds of `PAccSumK` is less than `PSumL` because the error bounds of `AccSumK` is less than `SumL`. We can also see that the error bounds of `PAccSumK` has nothing to do with the number of sums $n$, while the error bounds of `PSumL` is related to $n$. In the example 4 of our numerical results section, we take $n = 2002, K = 7, P = 6$, therefore, the second term in the error bounds of `PAccSumK` is approximately $6.42 \times 10^{22}$ times smaller than the error bounds of `PSumL`.

## 4. Numerical results

The following parallel experiment is performed on Sugon HPC cluster with 424 compute nodes, consisting of two 14-core Intel(R) Xeon(R) Gold

6162 2.60GHz processors each (28 cores per node). Nodes are connected by Intel Omni-Path high-speed computing network. The peak performance of the whole system is 980Tflops/s, and the memory is 96GB. The OS used by the cluster is Redhat7.2. The MPI library used for this experiment is Intel MPI. Accuracy is evaluated by relative error $e = |\texttt{res} - s|/|s|$, where `res` is an estimate of $s$. For `PAccSumK`, the terms $\texttt{Res}_k$ are non-overlapping, then `res` is equal to the sum of the array elements calculated by the ordinary summation algorithm. For `PSumL`, we sort $\texttt{Res}_k$ by magnitude and sum in decreasing order [11]. If the output of the algorithm is a floating-point number, then `res` is equal to the output of the algorithm, $s$ is the exact value calculated by the MPFR library [9] or known. When not stated separately, `PAccSumK` always takes $K = 3$.

### *4.1. Example 1*

We use Algorithm 4.2 in Yamanaka et al. [5] 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 `TwoProduct` 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. Figure 3 is the relative error image of `AccSumK` and `SumK`. `SumK` is the serial version of `PSumK`. In this example, we use $K = 2$ to complete on the MATLAB plateform. From Figure 3 we can see that the `AccSumK` algorithm is more accurate than `SumK` algorithm.

### *4.2. Example 2*

We use Algorithm 6.1 in Ogita et al. [1] 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.

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

Figure 4 is the relative error image of different condition numbers when 4 processes and 10 processes are used, and the summation scale is 200 and 2000 respectively. `PSum` is the result of parallel recursive summation algorithm [5]. The test results show that `PAccSumK` has higher accuracy than `PSum`. For using different processes, the relative error results of `PAccSumK` are almost

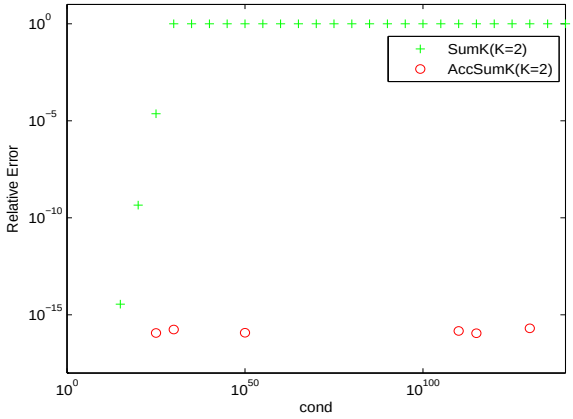


Figure 3: Relative error image of serial algorithm

equal so that the images overlap, and `PSum` is quite different for the results of using different processes.

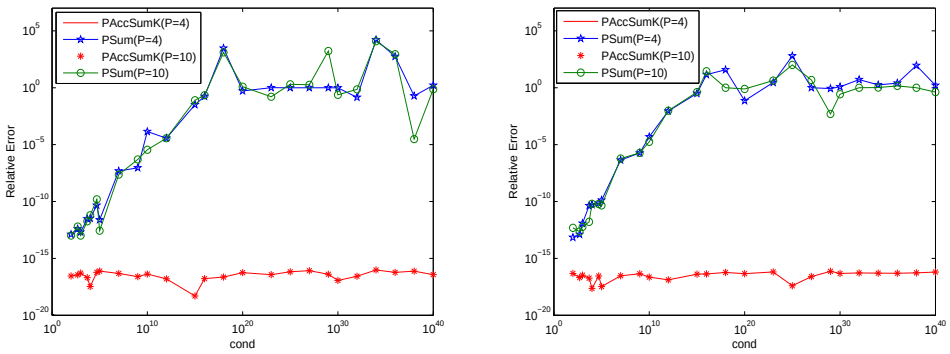


Figure 4: relative error images (left $n = 200$) (right $n = 2000$)

Figure 5 is the CPU time image of the algorithm under different condition numbers when using 4 processes and 10 processes, and the summation scale is 200 and 2000 respectively. The `PAccSumK` algorithm uses more CPU time than the `PSum` algorithm.

Taking the `PSum` algorithm as a comparison benchmark, the calculation time of the algorithm is averaged, and then the ratio is calculated. The calculation time ratio of the `PAccSumK` is shown in table 1.

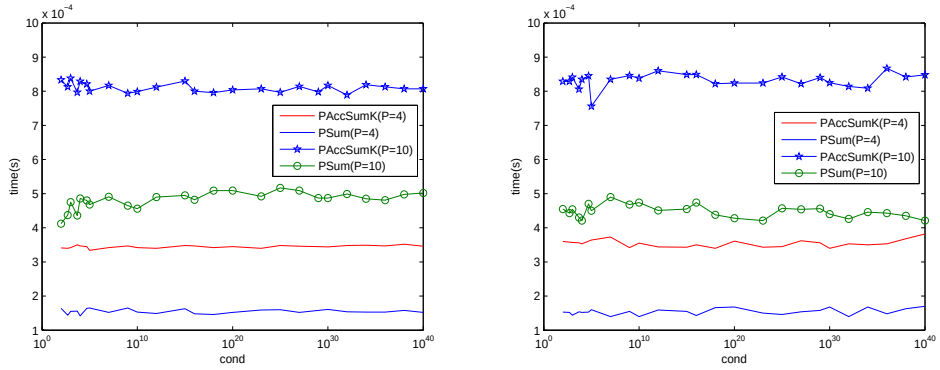


Figure 5: CPU time (left $n = 200$) (right $n = 2000$)

Table 1: 4 processes calculation time ratio

| | $n = 200$ | | $n = 2000$ | |
|---|---|---|---|---|
| Algorithm | `PSum` | `PAccSumK` | `PSum` | `PAccSumK` |
| time ratio | 1 | 2.222 | 1 | 2.298 |

### *4.3. Example 3*

We use the same data generation method as the ReproBLAS library [10], i.e. $p_i = \sin(2.0 * \pi * (i/n - 0.5))$, compare the relative error and CPU time of different processes, and compare the relative error and CPU time in different input data sizes.

Figure 6 (a) uses the process number of 2, 4, 8, 10, 16 and 20 to calculate the relative error when $n = 1 \times 10^i, i = 3, \ldots, 7$, the purpose is to compare the relative errors when using different number of processes. It can be seen from the image that the `PAccSumK` algorithm uses different processes to obtain almost the same summation result on the same set of summation data, so the relative error curve overlap. `PSum` got inaccurate results, `PAccSumK` obtains high-precision results. Figures 6 (b) and 6 (c) use process numbers of 4, 8, 10, and 20 when $n = 2 \times 10^i, i = 3, \ldots, 7$ and $n = 3 \times 10^i, i = 3, \ldots, 7$ calculate relative error, the purpose is to compare the relative error of the same process number on different $n$, the same process number has slightly different curves on the sum data of different $n$. `PSum` got inaccurate results, `PAccSumK` obtains high-precision results.

In terms of CPU time, from figure 7 we can observe `PAccSumK` algorithm takes more time than the `PSum` algorithm.

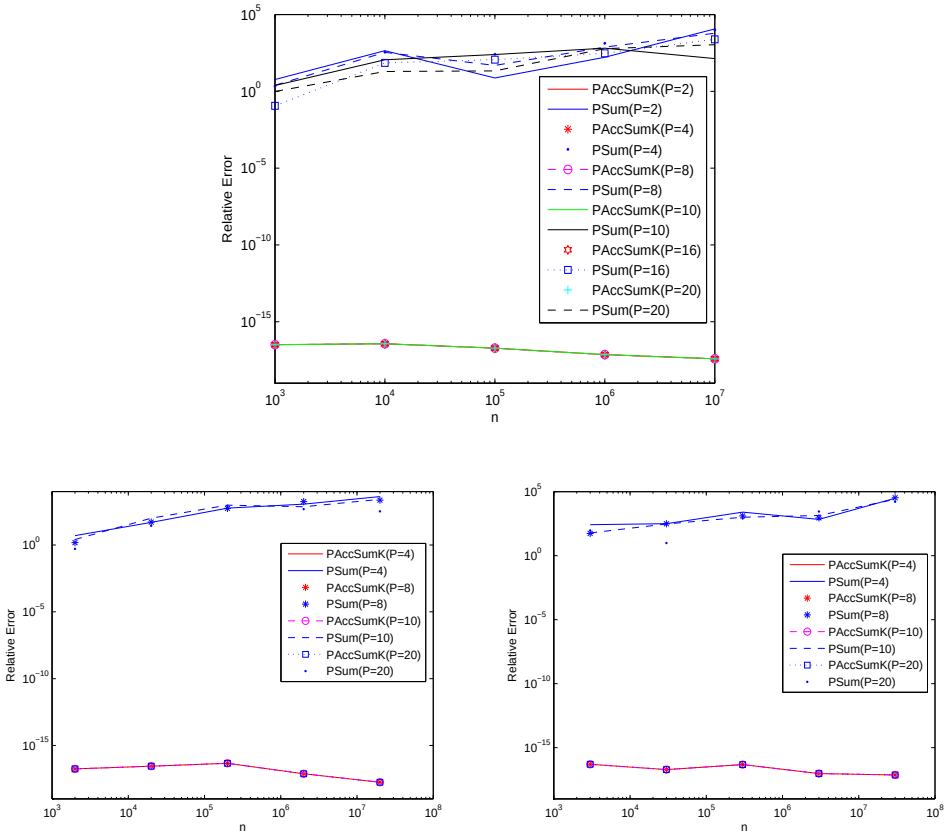


Figure 6: Relative error image (a) $n = 1 \times 10^i, i = 3, \ldots, 7$, (b) $n = 2 \times 10^i, i = 3, \ldots, 7$ and (c) $n = 3 \times 10^i, i = 3, \ldots, 7$.

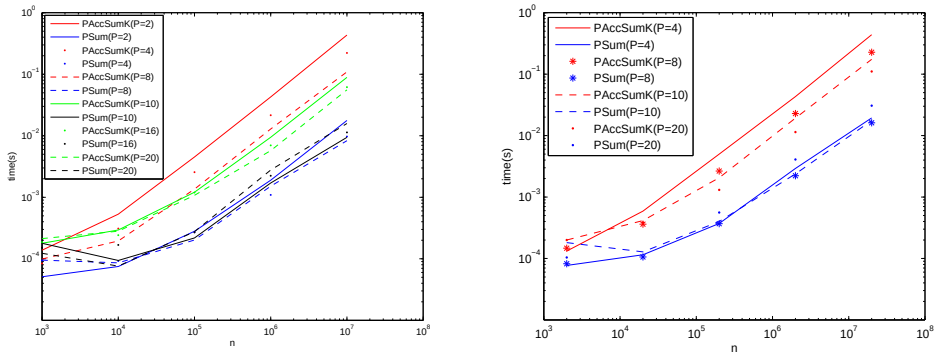


Figure 7: CPU time (left $n = 1 \times 10^i, i = 3, \ldots, 7$) (right $n = 2 \times 10^i, i = 3, \ldots, 7$)

Table 2: 4 processes' calculation time ratio

| | $n = 1 \times 10^i, i = 3, \ldots, 7$ | | $n = 2 \times 10^i, i = 3, \ldots, 7$ | |
|---|---|---|---|---|
| Algorithm | `PSum` | `PAccSumK` | `PSum` | `PAccSumK` |
| time ratio | 1 | 21.834 | 1 | 21.352 |

Table 3: calculation time ratio

| `PAccSumK` | $n = 1 \times 10^i, i = 3, \cdots, 7$ | $n = 2 \times 10^i, i = 3, \cdots, 7$ |
|---|---|---|
| $p = 2$ | 1 | - |
| $p = 4$ | 0.510 | 1 |
| $p = 8$ | 0.256 | 0.518 |
| $p = 10$ | 0.207 | 0.403 |
| $p = 16$ | 0.146 | - |
| $p = 20$ | 0.130 | 0.254 |

Taking the `PSum` algorithm as a comparison benchmark, the calculation time of the algorithm is averaged, and then the ratio is calculated. The calculation time ratio of the `PAccSumK` is shown in table 2.

Table 3 shows the calculation speed of `PAccSumK` using different processes on $n = 1 \times 10^i, i = 3, \ldots, 7$ and $n = 2 \times 10^i, i = 3, \ldots, 7$ scale sum data, $n = 1 \times 10^i, i = 3, \ldots, 7$ sum data is based on the number of processes as 2, $n = 2 \times 10^i, i = 3, \ldots, 7$ sum data is based on the number of processes 4 as a benchmark to calculate the acceleration when using other processes.

It can be seen from table 3 that the CPU time used by the `PAccSumK` algorithm is approximately inversely proportional to the number of processes used.

### *4.4. Example 4*

We use the same ill-conditioned sum data as in Example 1.

The speed-up ratio $R_1$ and the parallel efficiency $R$ are defined by

$R_1 := \frac{T_1}{T_P}$ and $R := \frac{R_1}{P}$.

where $T_P$ means the elapsed time in case of using $P$ CPUs ($1 \leq P \leq 8$). From $R_1$, we can see the scalability of the algorithms. If the floating-point operations are perfectly parallelized, then it holds that

$T_P := C + \frac{T_1}{T_P}$,

where $C$ denotes the elapsed time for the parallelization overhead.

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

Figure 8 (a) is the relative error image under different condition numbers when using 4 processes and $K = 7$, figure 8 (b) is the relative error image under different condition numbers when using 6 processes and $K = 7$. This experiment uses the $mpi\_intelmpi - 2017.4.239$. The test results show that `PAccSumK` is more accurate than the other three parallel summation algorithms.

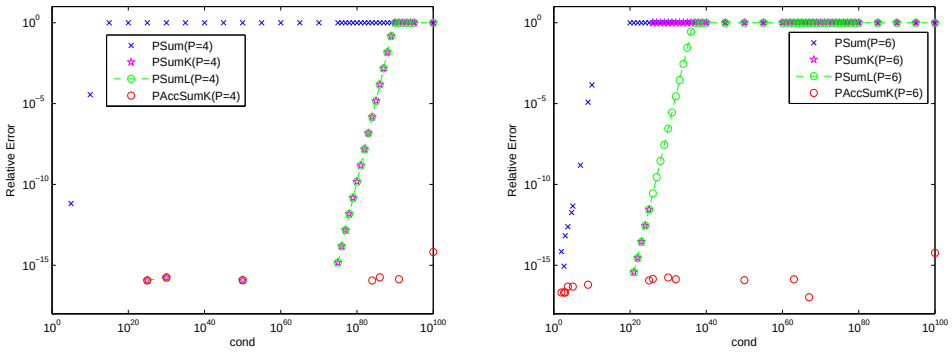


Figure 8: Relative error image (a) $P = 4$, (b) $P = 6$

Figure 9 (a) and (b) are the relative error images of `PSumK` when using $mpi\_mpich2 - gnu - 1.5$ and $mpi\_mpich - gnu - 3.3.1$, respectively. Using $mpi\_openmpi - gnu - 2.0.4$ will get a result similar to figure 9 (a).

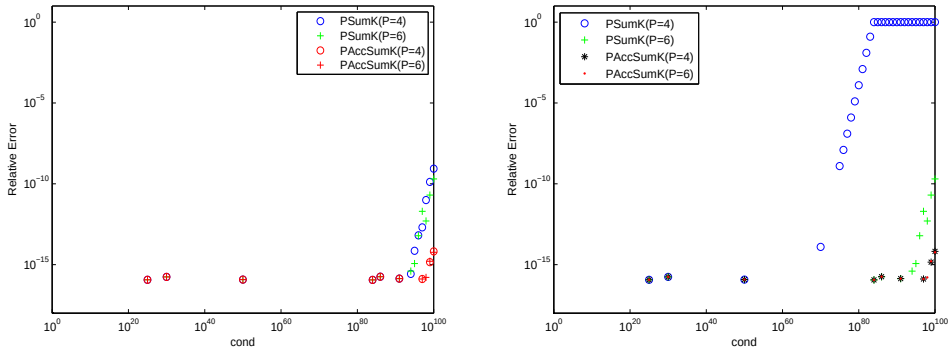


Figure 9: Relative error image (a) $mpi\_mpich2 - gnu - 1.5$, (b) $mpi\_mpich - gnu - 3.3.1$

Figure 10 is the CPU time image when $n = 10^6$, $P = 4$, and $K = 7$. `PAccSumK` algorithm is 1.7574 times slower than `PSum` algorithm. `PAccSumK`

algorithm is 1.4140 times faster than `PSumK` algorithm. `PAccSumK` algorithm is 1.4263 times faster than `PSumL` algorithm.

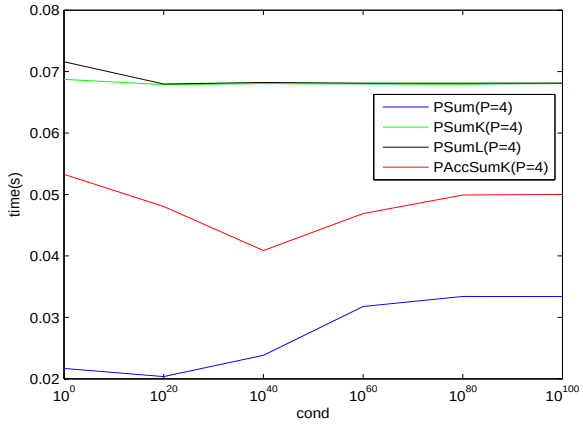


Figure 10: CPU time image

Next, we evaluate how tight the error bound for `PAccSumK` in Theorem 5 is in practice. To do this, we set $n = 2002$ and vary the condition number cond from 1 to $10^{140}$ in Algorithm 4.2 [5], the exact result of sum is equal to $cond^{-1}$. The error bounds (16) and true relative errors of the results obtained by `PAccSumK` for $K = 2, 4, 8$ in case of $P = 6$ are displayed in figure 11. In figure 11, the lines labeled '**exp.**' for each $K$ denote the experimental error corresponding to $K$, the lines labeled '**est.**' for each $K$ denote the error bounds (16) corresponding to $K$.

Finally, we evaluate the elapsed time and parallel efficiency of `PAccSumK` for $n = 10^6$ and $n = 10^7$. For both cases we set $cond = 1 \cdot 10^{100}$ in Algorithm 4.2 and vary $K = 7, 8, 9$ in `PAccSumK`. The general rule to choose $K$ is based on (16) if the condition number and the expected accuracy are known. Here, we give an example to choose $K$. Suppose we calculate in double precision, we need the relative error between the approximate solution and the exact solution to meet the machine accuracy, in this example, $cond = 1 \cdot 10^{100}$, we can calculate $K \approx 7.29$. Therefore, in this example, we choose $K = 7$. The elapsed times of `PAccSumK` for both cases are displayed in table 4. Moreover, figure 12 and 13 illustrate the scalability and the parallel efficiency of `PAccSumK` compared with `AccSumK` for $n = 10^6$ and $n = 10^7$, respectively.

From table 4, figure 12 and 13, we can observe the following facts:

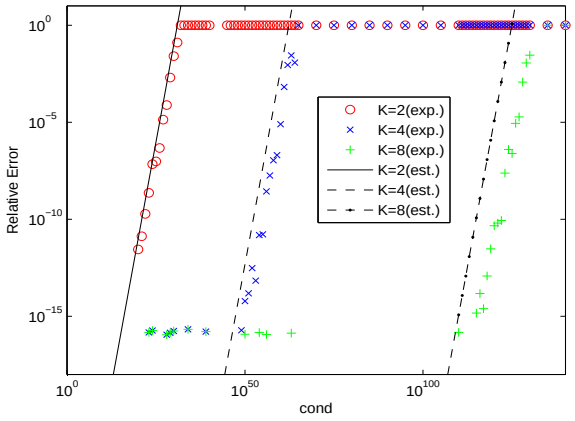


Figure 11: Error bounds and true relative errors of the results by `PAccSumK` for $K = 2, 4, 8$ ($n = 2002, P = 6$)

Table 4: Elapsed time (s) of `PAccSumK` for $n = 10^6$ and $n = 10^7$

| $P$ | $n = 10^6$ | | | $n = 10^7$ | | |
|---|---|---|---|---|---|---|
| | $K = 7$ | $K = 8$ | $K = 9$ | $K = 7$ | $K = 8$ | $K = 9$ |
| 1 | 0.110874 | 0.111125 | 0.110715 | 1.205098 | 1.204358 | 1.205449 |
| 2 | 0.099828 | 0.114573 | 0.123293 | 1.012866 | 1.156458 | 1.241574 |
| 3 | 0.069137 | 0.078671 | 0.084988 | 0.662176 | 0.785140 | 0.813300 |
| 4 | 0.053373 | 0.061192 | 0.065804 | 0.511626 | 0.572070 | 0.613447 |
| 5 | 0.044033 | 0.049782 | 0.053883 | 0.402653 | 0.460468 | 0.494046 |
| 6 | 0.038121 | 0.043568 | 0.045704 | 0.339094 | 0.386915 | 0.421356 |
| 7 | 0.032498 | 0.037115 | 0.040046 | 0.290361 | 0.333394 | 0.356951 |
| 8 | 0.028750 | 0.032369 | 0.034621 | 0.258213 | 0.293973 | 0.315831 |

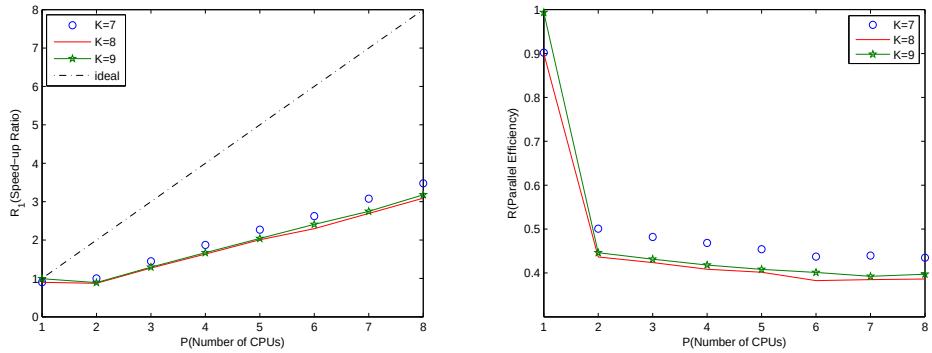


Figure 12: Scalability (left) and parallel efficiency (right) of `PAccSumK` for $n = 10^6$

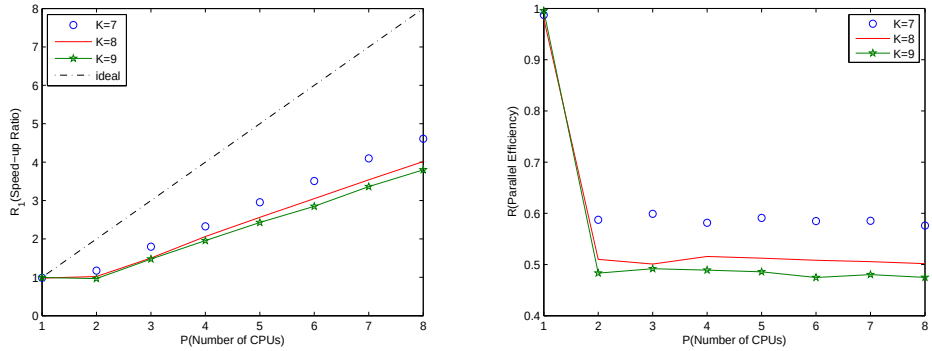


Figure 13: Scalability (left) and parallel efficiency (right) of `PAccSumK` for $n = 10^7$

- As the number of processes increases, the speed-up ratio increases approximately linearly, indicating that the `PAccSumK` algorithm has better scalability;
- In both scales, $K = 7$ has the highest speed-up ratio and parallel efficiency. Under the premise that the results can be obtained with high accuracy, it is best to take $K = 7$ in this example;

## 5. Conclusions and Future Work

In this paper, we proposed the parallel algorithm for accurate sum. We proved that our proposed algorithm achieved a result as if computed with $K$-fold the working precision. By the numerical experiments, we confirmed that the proposed parallel algorithm works effectively on our computer environment. For the summation problem with large condition number, `PAccSumK` performs better than `PSumK` and `PSumL` in terms of accuracy and time. In practical applications, large condition number sequences often appear in the case of massive cancellation, for example, in computational geometry [16]. In practice, the proposed algorithm frequently gives more accurate results than the theoretical error bounds.

In the future, we plan to replace the ordinary summation and dot product of linear equations solver with high-precision summation and dot product in order to obtain a high-precision linear equations solver. We also plan to use `AccSumK` algorithm to obtain a parallel high-precision reproducible summation algorithm.

## Acknowledgment

To begin with, we would like to thank the reviewers for their thorough reading of the article as well as their valuable comments and suggestions. The second and fourth authors were supported in part by Science Challenge Project (TZ2016002), NSF of China (61472462, 11671049, 11601033) and the foundation of key laboratory of computational physics. The third author was supported by the NuSCAP (ANR-20-CE48-0014) project of the French National Agency for Research (ANR).

## References

[1] T. Ogita, S. Rump, S. Oishi, Accurate sum and dot product, SIAM J. Sci. Comput. 26(6) (2005) 1955-1988.

[2] S. Rump, T. Ogita, S. Oishi, Accurate floating-point summation I: faithful rounding, SIAM J. Sci. Comput. 31(1) (2008) 189-224.

[3] S. Rump, T. Ogita, S. Oishi, Accurate floating-point summation part II: sign, $K$-Fold faithful and rounding to nearest, SIAM J. Sci. Comput. 31(2) (2008) 1269-1302.

[4] N. Higham, Accuracy and Stability of Numerical Algorithms, 2nd ed, SIAM Publications, Philadelphia, 2002.

[5] N. Yamanaka, T. Ogita, S. Rump, S. Oishi, A parallel algorithm for accurate dot product, Parallel Computing. 34(6-8) (2008) 392-410.

[6] T. J. Dekker, A floating-point technique for extending the available precision, Numer. Math. 18 (1971) 224-242.

[7] ANSI/IEEE, IEEE Standard for Binary Floating Point Arithmetic, Std 754-2019, IEEE, New York, 2019.

[8] D. E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, vol. 2, Addison-Wesley, Reading, Massachusetts, 1969.

[9] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: A multiple-precision binary floating-point library with correct rounding, ACM Trans. Math. Software. 33(2) (2007) 13.

[10] P. Ahrens, H. Nguyen, J. Demmel, Efficient reproducible floating point summation and BLAS, EECS Department, UC Berkeley, Technical Report No. UCB/EECS-2016-121, 2016.

[11] S. Rump, Inversion of extremely ill-conditioned matrices in floating-point, Japan J. Indust. Appl. Math. 26 (2009) 249-277.

[12] M. Lange, S. Rump, Sharp estimates for perturbation errors in summations, Math. Comp. 88 (2019) 349-368.

[13] S. Rump, Error bounds for computer arithmetics, in: 26th IEEE Symposium on Computer Arithmetic, IEEE, 2019.

[14] C.-P. Jeannerod, S. Rump, On relative errors of floating-point operations: Optimal bounds and applications, Math. Comp. 87 (2018) 803-819.

[15] M. Lange, S. Rump, Error estimates for the summation of real numbers with application to floating-point summation, BIT. 57 (2017) 927-941.

[16] J. Demmel, Y. Hida, Fast and accurate floating point summation with application to computational geometry, Numer. Algorithms. 37 (2004) 101-112.