Available online at www.sciencedirect.com
ScienceDirect
Parallel Computing 34 (2008) 392–410
PARALLEL COMPUTING
www.elsevier.com/locate/parco

A parallel algorithm for accurate dot product

N. Yamanaka [a,*], T. Ogita [b,c], S.M. Rump [d], S. Oishi [c]

[a] Graduate School of Science and Engineering, Waseda University, Tokyo 169-8555, Japan
[b] Department of Mathematics, Tokyo Woman's Christian University, Tokyo 167-8585, Japan
[c] Faculty of Science and Engineering, Waseda University, Tokyo 169-8555, Japan
[d] Institute for Reliable Computing, Hamburg University of Technology, Hamburg 21071, Germany

Received 15 December 2006; received in revised form 28 January 2008; accepted 25 February 2008
Available online 18 March 2008

Abstract

Parallel algorithms for accurate summation and dot product are proposed. They are parallelized versions of fast and accurate algorithms of calculating sum and dot product using error-free transformations which are recently proposed by Ogita et al. [T. Ogita, S.M. Rump, S. Oishi, Accurate sum and dot product, SIAM J. Sci. Comput. 26 (6) (2005) 1955–1988]. They have shown their algorithms are fast in terms of measured computing time. However, due to the strong data dependence in the process of their algorithms, it is difficult to parallelize them. Similarly to their algorithms, the proposed parallel algorithms in this paper are designed to achieve the results as if computed in K-fold working precision with keeping the fastness of their algorithms. Numerical results are presented showing the performance of the proposed parallel algorithm of calculating dot product.

Keywords: Parallel algorithm; Accurate dot product; Accurate summation; Higher precision

1. Introduction

Let $\mathbb{F}$ be a set of floating-point numbers. Let $x, y \in \mathbb{F}^n$. In this paper, we present a parallel algorithm to compute a dot product $x^{\mathrm{T}}y$ in high accuracy. Since dot product is a most basic task in numerical analysis, there are a number of algorithms for that. Accurate dot product algorithms have various applications in numerical analysis. Excellent overviews can be found in [6,7].

For parallel and efficient computations, we encounter the situations where the vectors spread across the processors, e.g. the computation of eigenvalues using the Arnoldi algorithm [1] and the computation of singular values using the Jacobi algorithm [3]. For these purposes, it is well known that highly accurate computations of dot product are needed to avoid loss of orthogonality.

Recently, the authors (Ogita, Rump and Oishi) presented an accurate dot product algorithm **DotK** [8] using so-called "error-free transformations". Their algorithm is fast, not only in terms of the number of floating-

* Corresponding author.
E-mail address: naoya_yamanaka@suou.waseda.jp (N. Yamanaka).

0167-8191/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2008.02.002

point operations but also in terms of measured computing time on a serial computer. However, it is difficult to parallelize **DotK** because of the strong data dependence.

To overcome this, we develop a parallelizing method for **DotK** and present a new algorithm **PDotK** of calculating a dot product whose result is aimed to be as accurate as that of **DotK**. It is shown that an error bound on the result by **PDotK** is less than or equal to that of **DotK**.

This paper is organized as follows: In Section 2, we briefly review the error-free transformations and the algorithms of accurate summation and dot product (**SumK** and **DotK**) proposed in [8]. In Section 3, we develop parallel algorithms **PSumK** and **PDotK**, which are the parallelized versions of **SumK** and **DotK**, respectively. We also analyze the proposed algorithms and present the theorems for them. By these, we can confirm that the proposed parallel algorithms achieve the results as if computed in K-fold working precision. In Section 4, we present results of numerical experiments showing the performance of the proposed algorithm **PDotK**. Finally, we conclude the paper.

Although the developed parallel algorithms can be implemented not only on a shared memory system but on a distributed one, it seems that it is not so efficient to use those algorithms on the distributed memory system at the level of sum and dot product. Therefore, we presume the shared memory system to be used in this paper.

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 sum and dot product

In this section, we briefly review some algorithms used in the accurate sum and dot product algorithms **SumK** and **DotK** [8].

Throughout this paper, we assume floating-point arithmetic adhering to IEEE standard 754. Let $\mathrm{fl}(\cdots)$ be the result of floating-point operations, where all operations inside parentheses are executed by ordinary floating-point arithmetic in rounding-to-nearest. We donote by $\mathbf{u}$ the machine epsilon; in IEEE standard 754 double precision $\mathbf{u} = 2^{-53}$. We assume that neither overflow nor underflow occur.

Following [6], we define γ_n as

$$\gamma_n := \frac{n\mathbf{u}}{1 - n\mathbf{u}} \quad \text{for } n \in \mathbb{N}.$$

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

Let $p = (p_1, \ldots, p_n)^{\mathrm{T}} \in \mathbb{F}^n$. Then it holds that [6]

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

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

The algorithms **SumK** and **DotK** are based on the error-free transformations of addition and/or multiplication of two floating-point numbers. Following [8], we call the algorithms **TwoSum** and **TwoProduct**, respectively.

First, we introduce the addition algorithm **TwoSum**. Knuth [2] presented the following algorithm which transforms a pair (x, y) with $x, y \in \mathbb{F}$ into a new pair (a, b) with $a, b \in \mathbb{F}$ satisfying $x + y = a + b$ with $a = \mathrm{fl}(x + y)$, $|b| \leqslant \mathbf{u}|a|$.

Algorithm 2.1 (*Knuth* [2]). Error-free transformation of the sum of two floating-point numbers.

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

The algorithm **TwoSum** can be extended to an error-free *vector* transformation with respect to the summation:

Algorithm 2.2 (*Ogita et al.* [8]). Error-free vector transformation with respect to the summation

function $q = \mathbf{VecSum}\ (p)$
$q_1 = p_1$
for $i = 2{:}n$
$[q_i, q_{i-1}] = \mathbf{TwoSum}\ (p_i, q_{i-1})$
end

Fig. 1 illustrates an outline of **VecSum** for $n = 5$. Namely, q_n is a floating-point approximation of $\sum_{i=1}^{n} p_i$, i.e. $\mathrm{fl}\left(\sum_{i=1}^{n} p_i\right)$, and $q_1, q_2, \ldots, q_{n-1}$ are the residuals of $\mathrm{fl}\left(\sum_{i=1}^{n} p_i\right)$. The algorithm **VecSum** transforms a vector $p = (p_1, p_2, \ldots, p_n)^{\mathrm{T}} \in \mathbb{F}^n$ into a new vector $q = (q_1, q_2, \ldots, q_n)^{\mathrm{T}} \in \mathbb{F}^n$ satisfying $\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} q_i$.
By (1), the algorithm **VecSum** satisfies [8, (4.4)]

$$\sum_{i=1}^{n-1} |q_i| \leqslant \gamma_{n-1} \sum_{i=1}^{n} |p_i|.. \tag{2}$$

In [8], the authors (Ogita, Rump and Oishi) developed an algorithm **SumK**, which calculates the summation using **VecSum** iteratively. By **SumK**, we can obtain the result as if computed in K-fold precision.

Algorithm 2.3 (*Ogita et al.* [8]). 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 res $= \mathbf{SumK}(p, K)$
for $i = 1 : K - 1$
$p = \mathbf{VecSum}(p)$
end
res $= \mathrm{fl}\left(\sum_{i=1}^{n} p_i\right)$

Fig. 2 illustrates an outline of **SumK** for $n = 5$ and $K = 4$.
Denote s and S by

$$s := \sum_{i=1}^{n} p_i, \quad S := \sum_{i=1}^{n} |p_i|. \tag{3}$$

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

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

Then error bounds of the result **res** by **SumK** are given as follows [8]:

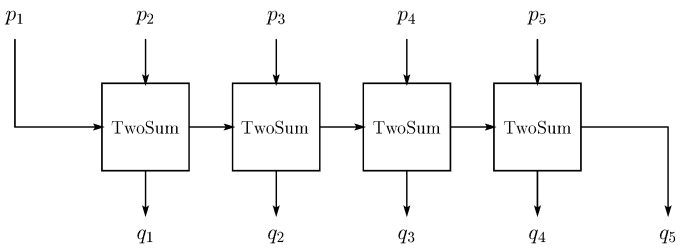

Fig. 1. Outline of **VecSum** for $n = 5$.

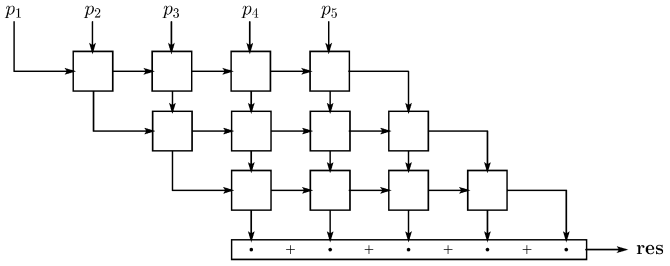

Fig. 2. Outline of **SumK** for $n = 5$ and $K = 4$.

Theorem 2.4 (Ogita et al. [8]). *Let* **res** *be the result obtained by* **SumK**, *then*

$$|\mathbf{res} - s| \leqslant (\mathbf{u} + 3\gamma_{n-1}^2)|s| + \gamma_{2(n-1)}^K S. \tag{4}$$

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

$$\left|\frac{\mathbf{res} - s}{s}\right| \leqslant \mathbf{u} + 3\gamma_{n-1}^2 + \gamma_{2(n-1)}^K \cdot \mathrm{cond}\left(\sum_{i=1}^n p_i\right). \tag{5}$$

Basically, the theorem says

$$\left|\frac{\mathbf{res} - s}{s}\right| \lesssim \mathbf{u} + \mathcal{O}(\mathbf{u}^K) \cdot \mathrm{cond}\left(\sum_{i=1}^n p_i\right),$$

which means **res** is obtained by **SumK** as if computed in internally K-fold working precision (corresponding to the second term of the right-hand side) and then rounded to the working precision (corresponding to the first term of the right-hand side). Thus, **res** can achieve almost the maximum accuracy in the working precision if $\mathrm{cond}(\sum_{i=1}^n p_i) < \gamma_{2(n-1)}^{-K}\mathbf{u} \approx (2n\mathbf{u})^{-K}\mathbf{u}$.

We present here a summation algorithm **SumL** whose result is represented by a sum of L floating-point numbers.

Algorithm 2.5. 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 $q = \mathbf{SumL}(p, L)$
  for $k = 1 : L - 1$
    $p(1 : n - k + 1) = \mathbf{VecSum}(p(1 : n - k + 1))$
    $q_k = p_{n-k+1}$
  end
  $q_L = \mathrm{fl}\left(\sum_{i=1}^{n-L+1} p_i\right)$

Fig. 3 illustrates an outline of **SumL** for $n = 5$ and $L = 3$.

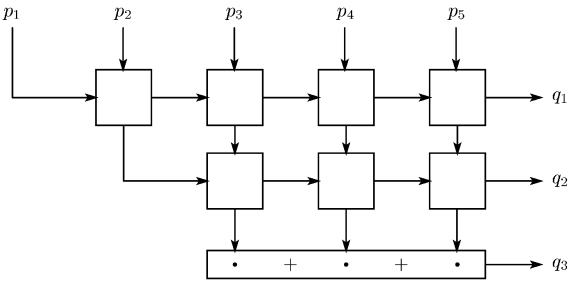

Fig. 3. Outline of **SumL** for $n = 5$ and $L = 3$.

For later use, we present the following theorem for **SumL**.

Theorem 2.6. *Algorithm* 2.5 (**SumL**) *satisfies the following inequalities*:

$$\left|\sum_{k=1}^{L} q_k - \sum_{i=1}^{n} p_i\right| \leqslant \gamma_{n-1}^{L} \sum_{i=1}^{n} |p_i|, \tag{6}$$

$$\sum_{k=1}^{L} |q_k| \leqslant (1+\gamma_{2(n-1)}) \sum_{i=1}^{n} |p_i|. \tag{7}$$

Proof. We first prove (6). Let $p^{(j)} = \left(p_1^{(j)}, p_2^{(j)}, \ldots, p_{n-j+1}^{(j)}\right)^{\mathrm{T}}$, $1 \leqslant j \leqslant L-1$, denote the intermediate vector as the result of **VecSum** of jth loop in **SumL**. Let $p^{(0)} := p$. Then **SumL** satisfies

$$\sum_{i=1}^{n} p_i = \sum_{k=1}^{j} q_k + \sum_{i=1}^{n-j} p_i^{(j)} \quad \text{for } 1 \leqslant j \leqslant L-1.$$

For $j = L-1$, it follows by (1) that

$$\begin{aligned}\left|\sum_{k=1}^{L} q_k - \sum_{i=1}^{n} p_i\right| &= \left|\sum_{k=1}^{L} q_k - \left(\sum_{k=1}^{L-1} q_k + \sum_{i=1}^{n-L+1} p_i^{(L-1)}\right)\right| = \left|q_L - \sum_{i=1}^{n-L+1} p_i^{(L-1)}\right| \\ &= \left|\mathrm{fl}\left(\sum_{i=1}^{n-L+1} p_i^{(L-1)}\right) - \sum_{i=1}^{n-L+1} p_i^{(L-1)}\right| \leqslant \gamma_{n-L} \sum_{i=1}^{n-L+1} |p_i^{(L-1)}|.\end{aligned}$$

Applying (2) to this inductively, we have

$$\left|\sum_{j=1}^{L} q_j - \sum_{i=1}^{n} p_i\right| \leqslant \gamma_{n-L}\gamma_{n-L+1} \sum_{i=1}^{n-L+2} |p_i^{(L-2)}| \leqslant \cdots \leqslant \gamma_{n-1}^{L} \sum_{i=1}^{n} |p_i|.$$

The inequality (6) is proved. □

We next prove (7). Using the fact that $q_k = \mathrm{fl}\left(\sum_{i=1}^{n-k+1} p_i^{(k-1)}\right)$ and (1) yield

$$|q_k| \leqslant (1+\gamma_{n-k}) \sum_{i=1}^{n-k+1} |p_i^{(k-1)}| \quad \text{for } 1 \leqslant k \leqslant L.$$

Again applying (2) to this inductively, we have

$$|q_k| \leqslant (1+\gamma_{n-k})\gamma_{n-k+1} \sum_{i=1}^{n-k+2} |p_i^{(k-2)}| \leqslant (1+\gamma_{n-k})\gamma_{n-k+1}\cdots\gamma_{n-1} \sum_{i=1}^{n} |p_i| \leqslant (1+\gamma_{n-k})\gamma_{n-1}^{k-1} \sum_{i=1}^{n} |p_i|.$$

Using $\gamma_n < 1$, we obtain

$$\sum_{k=1}^{L} |q_k| = \sum_{k=1}^{L} (1+\gamma_{n-k})\gamma_{n-1}^{k-1} \sum_{i=1}^{n} |p_i| \leqslant (1+\gamma_{n-1}) \sum_{k=1}^{L} \gamma_{n-1}^{k-1} \sum_{i=1}^{n} |p_i| \leqslant \frac{1+\gamma_{n-1}}{1-\gamma_{n-1}} \sum_{i=1}^{n} |p_i| = (1+\gamma_{2(n-1)}) \sum_{i=1}^{n} |p_i|,$$

which is the desired formula (7). □

We can see from (6) that the resultant vector q of **SumL** satisfies

$$\left|\frac{\sum_{k=1}^{L} q_k - s}{s}\right| \lesssim \mathcal{O}(\mathbf{u}^L) \cdot \mathrm{cond}\left(\sum_{i=1}^{n} p_i\right),$$

where $s = \sum_{i=1}^{n} p_i$. This means $\sum_{k=1}^{L} q_k$ approximates s as if using L-fold working precision.

Next, we proceed to the dot product. We know a useful multiplication algorithm **TwoProduct** [4], which transforms a pair (x, y) with $x, y \in \mathbb{F}$ into a new pair (a, b) with $a, b \in \mathbb{F}$ satisfying $x \cdot y = a + b$, $|b| \leqslant \mathbf{u}|a|$.

The multiplication routine needs to split the input arguments into two parts. For the number t given by $\mathbf{u} = 2^{-t}$, we define $s := \lceil t/2 \rceil$; in IEEE 754 double precision we have $t = 53$ and $s = 27$. The following algorithm by Dekker [4] splits a floating-point number $x \in \mathbb{F}$ into two parts x, y, where both parts have at most $(s-1)$ nonzero bits.

Algorithm 2.7 (*Dekker* [4]). The algorithm **Split** splits a t-bits floating-point number $x \in \mathbb{F}$ into $x_h, x_t \in \mathbb{F}$ such that $x = x_h + x_t$.

function $[x_h, x_t] = \mathbf{Split}(x)$
$c = \mathrm{fl}((2^{\lceil t/2 \rceil} + 1) \cdot x)$
$x_h = \mathrm{fl}(c - (c - x))$
$x_t = \mathrm{fl}(x - x_h)$

Using **Split**, the following multiplication routine by G.W. Veltkamp (see [4]) can be formulated.

Algorithm 2.8 (*Veltkamp* (*see* [4])). Error-free transformation of the product of two floating-point numbers.

function $[a, b] = \mathbf{TwoProduct}(x, y)$
$a = \mathrm{fl}(x \cdot y)$
$[x_1, x_2] = \mathbf{Split}(x)$
$[y_1, y_2] = \mathbf{Split}(y)$
$b = \mathrm{fl}(x_2 \cdot y_2 - (((a - x_1 \cdot y_1) - x_2 \cdot y_1) - x_1 \cdot y_2))$

Let $x, y \in \mathbb{F}^n$. Now, we name the following algorithm **VecProduct**, which is used in **DotK** and transforms a dot product $x^T y$ into a summation $\sum_{i=1}^{2n} t_i$ such that $x^{\mathrm{T}} y = \sum_{i=1}^{2n} t_i$.

Algorithm 2.9. The algorithm **VecProduct** transforms two vectors $x, y \in \mathbb{F}^n$ into a new vector $t \in \mathbb{F}^{2n}$ satisfying $x^{\mathrm{T}} y = \sum_{i=1}^{2n} t_i$.

function $t = \mathbf{VecProduct}(x, y)$
$r = 0$
for $i = 1{:}n$
$[h_i, t_i] = \mathbf{TwoProduct}(x_i, y_i)$
$[r, t_{n+i-1}] = \mathbf{TwoSum}(r, h_i)$
end
$t_{2n} = r$

Fig. 4 illustrates an outline of **VecProduct** for $n = 4$.
For later use, we present the following theorem for **VecProduct**.

Theorem 2.10. *Algorithm* 2.9 (**VecProduct**) *satisfies the following inequalities*:

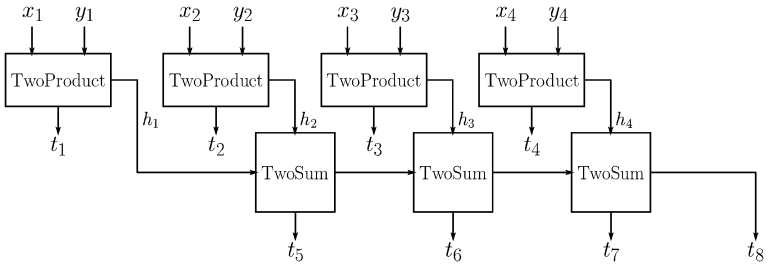

Fig. 4. Outline of **VecProduct** for $n = 4$.

$$\sum_{i=1}^{2n-1} |t_i| \leqslant \gamma_n \sum_{i=1}^{n} |x_i y_i|, \tag{8}$$

$$|t_{2n}| \leqslant (1+\gamma_n) \sum_{i=1}^{n} |x_i y_i|. \tag{9}$$

Proof. We first prove (8). The left-hand side of (8) can be separated as follows:

$$\sum_{i=1}^{2n-1} |t_i| = \sum_{i=1}^{n} |t_i| + \sum_{i=n+1}^{2n-1} |t_i|. \tag{10}$$

By the property of **TwoProduct**, it holds that $|t_i| \leqslant \mathbf{u}|h_i|$ for $1 \leqslant i \leqslant n$, so that

$$\sum_{i=1}^{n} |t_i| \leqslant \mathbf{u} \sum_{i=1}^{n} |h_i|. \tag{11}$$

An n-vector $(t_{n+1}, t_{n+2}, \ldots, t_{2n})^{\mathrm{T}}$ is obtained by the iterative use of **TwoSum**, which is identical to the result of **VecSum**(h) for $h = (h_1, h_2, \ldots, h_n)^{\mathrm{T}}$. Therefore, (2) yields

$$\sum_{i=n+1}^{2n-1} |t_i| \leqslant \gamma_{n-1} \sum_{i=1}^{n} |h_i|. \tag{12}$$

Moreover, it follows by $h_i = \mathrm{fl}(x_i y_i)$ that

$$|h_i| \leqslant (1+\mathbf{u})|x_i y_i|. \tag{13}$$

Inserting (11)–(13) into (10), we have

$$\sum_{i=1}^{2n-1} |t_i| \leqslant \mathbf{u} \sum_{i=1}^{n} |h_i| + \gamma_{n-1} \sum_{i=1}^{n} |h_i| = (\mathbf{u}+\gamma_{n-1}) \sum_{i=1}^{n} |h_i| \leqslant (1+\mathbf{u})(\mathbf{u}+\gamma_{n-1}) \sum_{i=1}^{n} |x_i y_i|. \tag{14}$$

Here, it holds that

$$(1+\mathbf{u})(\mathbf{u}+\gamma_{n-1}) = (1+\mathbf{u}) \frac{n\mathbf{u} - (n-1)\mathbf{u}^2}{1-(n-1)\mathbf{u}} \leqslant \frac{n\mathbf{u}}{1-n\mathbf{u}} = \gamma_n. \tag{15}$$

Inserting (15) into (14) proves (8).

We next prove (9). As mentioned before, the vector $(t_{n+1}, t_{n+2}, \ldots, t_{2n})^{\mathrm{T}}$ is identical to the result of **VecSum**(h), so that $t_{2n} = \mathrm{fl}(\sum_{i=1}^{n} h_i)$. By (1), we have

$$\left| t_{2n} - \sum_{i=1}^{n} h_i \right| \leqslant \gamma_{n-1} \sum_{i=1}^{n} |h_i|. \tag{16}$$

It follows by (16) and (13) that

$$\begin{aligned} |t_{2n}| &\leqslant \left| \sum_{i=1}^{n} h_i \right| + \gamma_{n-1} \sum_{i=1}^{n} |h_i| \leqslant (1+\gamma_{n-1}) \sum_{i=1}^{n} |h_i| \leqslant (1+\gamma_{n-1})(1+\mathbf{u}) \sum_{i=1}^{n} |x_i y_i| \leqslant \frac{1}{1-n\mathbf{u}} \sum_{i=1}^{n} |x_i y_i| \\ &= (1+\gamma_n) \sum_{i=1}^{n} |x_i y_i|, \end{aligned}$$

which proves (9). □

Utilizing **VecProduct** and **SumK**, the authors (Ogita, Rump and Oishi) developed the algorithm **DotK**, which calculates the dot product as in K-fold precision.

Algorithm 2.11 (*Ogita et al.* [8]). Dot product $x^{\mathrm{T}}y$ for $x, y \in \mathbb{F}^n$ as in K-fold precision.

function res $=$ **DotK**(x, y, K)
$t =$ **VecProduct** (x, y)
res $=$ **SumK** $(t,\ K-1)$

The condition number of the dot product $x^{\mathrm{T}}y$ is defined by

$$\operatorname{cond}(x^{\mathrm{T}}y) := 2\frac{|x^{\mathrm{T}}||y|}{|x^{\mathrm{T}}y|}, \quad x^{\mathrm{T}}y \neq 0.$$

Then error bounds of the result **res** by **DotK** are given as follows [8]:

Theorem 2.12 (Ogita et al. [8]). *Let* **res** *be the result obtained by* **DotK**, *then*

$$|\mathbf{res} - x^{\mathrm{T}}y| \leqslant (\mathbf{u} + 2\gamma_{4n-2}^2)|x^{\mathrm{T}}y| + \gamma_{4n-2}^K|x^{\mathrm{T}}||y|. \tag{17}$$

Moreover, if $x^{\mathrm{T}}y \neq 0$, then

$$\left|\frac{\mathbf{res} - x^{\mathrm{T}}y}{x^{\mathrm{T}}y}\right| \leqslant \mathbf{u} + 2\gamma_{4n-2}^2 + \frac{1}{2}\gamma_{4n-2}^K \cdot \operatorname{cond}(x^{\mathrm{T}}y). \tag{18}$$

Similarly to Theorems 2.4, 2.12 says

$$\left|\frac{\mathbf{res} - x^{\mathrm{T}}y}{x^{\mathrm{T}}y}\right| \lesssim \mathbf{u} + \mathcal{O}(\mathbf{u}^K) \cdot \operatorname{cond}(x^{\mathrm{T}}y),$$

which means the result **res** is obtained by **DotK** as if computed in internally K-fold working precision and rounded to the working precision, which is similar to the result by **SumK**.

3. Parallel algorithms

In this section, we will develop parallel algorithms for calculating sum and dot product in internally K-fold working precision. First, we will present an algorithm of parallelizing **SumK**, which is named **PSumK**. Next, we will present an algorithm of parallelizing **DotK**, which is named **PDotK**.

Suppose the number of CPUs to be $M \geqslant 2$ on a shared memory system. Then the CPUs can be numbered as id from 1 to M, so $1 \leqslant id \leqslant M$.

3.1. Parallelizing method of *SumK* (*PSumK*)

We first present an outline of **PSumK** which is a parallelized version of **SumK**. Let $p \in \mathbb{F}^n$.

Step 1: Distribute sub-vectors of p which are divided into M pieces to all CPUs. Every CPU with $2 \leqslant id \leqslant M$ has $c := \lceil n/M \rceil$ components. The CPU with $id = 1$ has $n - c(M-1)$ components. Let $p^{(id)}$ denote the sub-vector on the idth CPU. Let c_{id} denote the number of components of $p^{(id)}$, then it holds that

$$c_1 \leqslant c_2 = c_3 = \cdots = c_M = c.$$

Step 2: Use **SumL**$(p^{(id)}, K)$, whose output is denoted by $q^{(id)}$. Then $q^{(id)}$ with K components is obtained on every CPU.

Step 3: Gather $q^{(id)}$ for all id into a vector q on the CPU with $id = 1$. Then the length of the gathered vector q becomes MK.

Step 4: Use **SumK**(q, K) on CPU with $id = 1$.

An important point is that the output $q^{(id)}$ with K components is necessary using **SumL** in Step 2. Otherwise, it is not possible to achieve the result with K-fold precision.

Next, we present here the concrete algorithm of **PSumK**.

Algorithm 3.1 (***PSumK***). A parallelized version of **SumK**.

function res $= \mathbf{PSumK}(p, K, M)$
$c = \lceil \frac{n}{M} \rceil, \quad c_1 = n - c(M-1)$
%*parallel private* (`indexl` , `index2`)
if $id == 1$
`indexl` $= 1 : c_1$
else
`indexl` $= c_1 + c(id-2) + 1 : c_1 + c(id-1)$
end
`index2` $= K(id-1) + 1 : K \cdot id$
$q($`index2`$) = \mathbf{SumL}(p($`indexl`$), K)$
% *end parallel*
res $= \mathbf{SumK}(q, K)$

Fig. 5 illustrates an outline of **PSumK** for $n = 15$, $K = 3$ and $M = 3$.

In **PSumK**, computational cost for the parallelized part is $(6K-5)c$ flops and the rest requires $(6K-5)(MK-1)$ flops. Therefore, the theoretical parallel efficiency in terms of the flops count becomes as follows:

$$r_1 = \frac{(6K-5)c}{(6K-5)c + (6K-5)(MK-1)} \approx \frac{1}{1 + \frac{MK}{c}}.$$

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

As in (3), s and S are defined by

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

Then we present the following theorem for **PSumK**.

Theorem 3.2. *Let* **res** *be the result obtained by Algorithm* 3.1 (**PSumK**). *Define* $c := \lceil \frac{n}{M} \rceil$. *Suppose* $2\max\{c, MK\}\mathbf{u} < 1$. *Then the following inequality holds*:

$$|\mathbf{res} - s| \leqslant (\mathbf{u} + 3\gamma^2_{MK-1})|s| + \phi_1 S, \tag{19}$$

where

$$\phi_1 := (1 + \mathbf{u} + 3\gamma^2_{MK-1})\gamma^K_{c-1} + (1 + \gamma_{2(c-1)})\gamma^K_{2(MK-1)}.$$

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

$$\left|\frac{\mathbf{res} - s}{s}\right| \leqslant \mathbf{u} + 3\gamma^2_{MK-1} + \phi_1 \mathrm{cond}\left(\sum p_i\right). \tag{20}$$

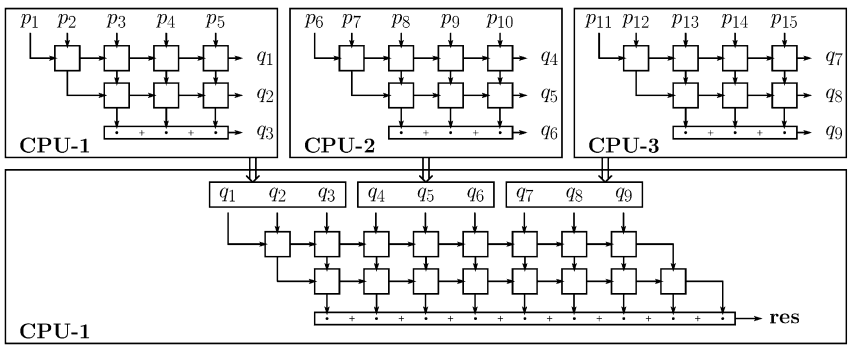

Fig. 5. Outline of **PSumK** for $n = 15$, $K = 3$ and $M = 3$.

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

$$\sum_{i=1}^{n} p_i = \sum_{id=1}^{M} \sum_{i=1}^{c_{id}} p_i^{(id)} = s, \tag{21}$$

$$\sum_{i=1}^{n} |p_i| = \sum_{id=1}^{M} \sum_{i=1}^{c_{id}} |p_i^{(id)}| = S, \tag{22}$$

$$\sum_{j=1}^{MK} q_j = \sum_{id=1}^{M} \sum_{k=1}^{K} q_k^{(id)}, \tag{23}$$

$$\sum_{j=1}^{MK} |q_j| = \sum_{id=1}^{M} \sum_{k=1}^{K} |q_k^{(id)}|. \tag{24}$$

Then

$$|\mathbf{res} - s| = |\mathbf{res} - \sum_{j=1}^{MK} q_j + \sum_{j=1}^{MK} q_j - s| \leqslant \left|\mathbf{res} - \sum_{j=1}^{MK} q_j\right| + \left|\sum_{j=1}^{MK} q_j - s\right|. \tag{25}$$

Here, using $\mathbf{res} = \mathbf{SumK}(q, K)$ and (4) yields

$$\left|\mathbf{res} - \sum_{j=1}^{MK} q_j\right| \leqslant \mathbf{u}'\left|\sum_{i=1}^{MK} q_i\right| + \gamma_{2(MK-1)}^{K} \sum_{i=1}^{MK} |q_i| \leqslant \mathbf{u}'\left(\left|\sum_{i=1}^{MK} q_i - s\right| + |s|\right) + \gamma_{2(MK-1)}^{K} \sum_{i=1}^{MK} |q_i|, \tag{26}$$

where $\mathbf{u}' := \mathbf{u} + 3\gamma_{MK-1}^2$. Inserting (26) into (25), we have

$$|\mathbf{res} - s| \leqslant \mathbf{u}'|s| + (1 + \mathbf{u}')\left|\sum_{i=1}^{MK} q_i - s\right| + \gamma_{2(MK-1)}^{K} \sum_{i=1}^{MK} |q_i|. \tag{27}$$

It follows by (21) and (23) that

$$\left|\sum_{j=1}^{MK} q_j - s\right| = \left|\sum_{id=1}^{M} \sum_{k=1}^{K} q_k^{(id)} - \sum_{id=1}^{M} \sum_{i=1}^{c_{id}} p_i^{(id)}\right| \leqslant \sum_{id=1}^{M} \left|\sum_{k=1}^{K} q_k^{(id)} - \sum_{i=1}^{c_{id}} p_i^{(id)}\right|.$$

Here, recalling that $q^{(id)} = \mathbf{SumL}(p^{(id)}, K)$ and using (6), we have

$$\left|\sum_{j=1}^{MK} q_j - s\right| \leqslant \sum_{id=1}^{M} \left(\gamma_{c_{id}-1}^{K} \sum_{i=1}^{c_{id}} |p_i^{(id)}|\right) \leqslant \gamma_{c-1}^{K} S. \tag{28}$$

Furthermore, applying (7) to the right-hand side of (24) and using (22) yield

$$\sum_{j=1}^{MK} |q_j| = \sum\nolimits_{id=1}^{M} \left(\sum_{j=1}^{K} |q_j^{(id)}|\right) \leqslant \sum\nolimits_{id=1}^{M} \left((1 + \gamma_{2(c_{id}-1)}) \sum_{i=1}^{c_{id}} |p_i^{(id)}|\right) \leqslant (1 + \gamma_{2(c-1)})S. \tag{29}$$

Inserting (28) and (29) into (27), we finally have

$$|\mathbf{res} - s| \leqslant \mathbf{u}'|s| + \{(1 + \mathbf{u}')\gamma_{c-1}^{K} + (1 + \gamma_{2(c-1)})\gamma_{2(MK-1)}^{K}\}S,$$

which proves (19) and divided by $|s|$ also proves (20). □

Similarly to the error bound for **SumK** in Theorems 2.4, 3.2 says

$$\left|\frac{\mathbf{res} - s}{s}\right| \lesssim \mathbf{u} + \mathcal{O}(\mathbf{u}^K) \cdot \mathrm{cond}\left(\sum p_i\right),$$

which means the result **res** is obtained by **PSumK** as if computed in internally K-fold working precision and then rounding the result to the working precision, which is similar to the result by **SumK**.

3.2. Parallelizing method of ***DotK*** (***PDotK***)

Next, we present an outline of **PDotK**, which is a parallelized version of **DotK**. Let $x, y \in \mathbb{F}^n$.

Step 1: Distribute sub-vectors of both x and y which are divided into M pieces to all CPUs. Every CPU with $2 \leqslant id \leqslant M$ has $c := \lceil n/M \rceil$ components corresponding to x and y, respectively. The CPU with $id = 1$ has $n - c(M-1)$ components as well. Let $x^{(id)}$ and $y^{(id)}$ denote the sub-vector on the idth CPU. Let c_{id} denote the number of components of $p^{(id)}$, then it holds that

$$c_1 \leqslant c_2 = c_3 = \cdots = c_M = c.$$

Step 2: Use $t^{(id)} = \mathbf{VecProduct}(x^{(id)}, y^{(id)})$ on all CPUs, i.e. transform dot product $(x^{(id)})^{\mathrm{T}} y^{(id)}$ into summation $\sum_{j=1}^{2c_{id}} t_j^{(id)}$.

Step 3: Use $\mathbf{SumL}(t^{(id)}, K-1)$, whose output is denoted by $q^{(id)}$. Then $q^{(id)}$ with K components is obtained on every CPU.

Step 4: Gather $q^{(id)}$ for all id into a vector q on the CPU with $id = 1$. Then the length of the gathered vector q becomes MK.

Step 5: Use $\mathbf{SumK}(q, K)$ on CPU with $id = 1$.

Now, we present here the concrete algorithm of **PDotK**.

Algorithm 3.3 (***PDotK***). A parallelized version of **DotK**.

function res $= \mathbf{PDotK}(x, y, K, M)$
$c = \lceil \frac{n}{M} \rceil, \quad c_1 = n - c(M-1)$
% *parallel private* (`idxl` , `idx2` , `idx2s`, `idx3` , `stl` , `st2` , c_{id})
if $id == 1$
$\mathtt{stl} = 1, \quad \mathtt{st2} = 1$
$c_{id} = c_1$
else
$\mathtt{stl} = c_1 + c(id-2) + 1, \quad \mathtt{st2} = 2\{c_1 + c(id-2)\} + 1$
$c_{id} = c$
end
$\mathtt{idxl} = \mathtt{stl} : \mathtt{stl} + c_{id} - 1$
$\mathtt{idx2} = \mathtt{st2} : \mathtt{st2} + 2c_{id} - 1, \quad \mathtt{idx2s} = \mathtt{st2} : \mathtt{st2} + 2c_{id} - 2$
$\mathtt{idx3} = K(id-1) + 1 : K \cdot id - 1$
$t(\mathtt{idx2}) = \mathbf{VecProduct}(x(\mathtt{idxl}), y(\mathtt{idxl}))$
$q(\mathtt{idx3}) = \mathbf{SumL}(t(\mathtt{idx2s}), K-1)$
$q(K \cdot id) = t(\mathtt{st2} + 2c_{id} - 1)$
% *end parallel*
res $= \mathbf{SumK}(q, K)$

In this algorithm, the variables `stl` and `st2` are the starting indeces for each CPU; `stl` is for the input vectors x and y, and `st2` is for the intermediate array t, respectively. In **PDotK**, computational cost for the parallelized part is $(12K+1)c$ flops and the rest requires $(6K-5)(MK-1)$ flops. Therefore, the theoretical parallel efficiency in terms of the flops count becomes as follows:

$$r_2 = \frac{(12K+1)c}{(12K+1)c + (6K-5)(MK-1)} \approx \frac{1}{1 + \frac{MK}{2c}}.$$

If $2c \gg MK$, then r_2 approaches to one.

Then we present the following theorem for **PDotK**.

Theorem 3.4. *Let* **res** *be the result obtained by Algorithm 3.3* (**PDotK**). *Define* $c := \lceil \frac{n}{M} \rceil$. *Suppose* $2\max\{c, MK\}\mathbf{u} < 1$. *Then the following inequality holds*:

$$|\mathbf{res} - x^{\mathrm{T}}y| \leqslant (\mathbf{u} + 3\gamma_{MK-1}^2)|x^{\mathrm{T}}y| + \phi_2|x^{\mathrm{T}}||y|, \tag{30}$$

where

$$\phi_2 := (1 + \mathbf{u} + 3\gamma_{MK-1}^2)\gamma_{2c-1}^K + (1 + 3\gamma_c)\gamma_{2(MK-1)}^K.$$

Moreover, if $x^T y \neq 0$, *then*

$$\left|\frac{\mathbf{res} - x^{\mathrm{T}}y}{x^{\mathrm{T}}y}\right| \leqslant \mathbf{u} + 3\gamma_{MK-1}^2 + \frac{1}{2}\phi_2 \mathrm{cond}(x^{\mathrm{T}}y). \tag{31}$$

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

$$q_K^{(id)} = t_{2c_{id}}^{(id)}, \tag{32}$$

$$\sum_{j=1}^{2c_{id}} t_j^{(id)} = (x^{(id)})^{\mathrm{T}} y^{(id)}, \tag{33}$$

$$\sum_{i=1}^{MK} q_i = \sum_{id=1}^{M}\sum_{j=1}^{K} q_j^{(id)}, \tag{34}$$

$$\sum_{i=1}^{MK} |q_i| = \sum_{id=1}^{M}\sum_{j=1}^{K} |q_j^{(id)}|. \tag{35}$$

Then

$$|\mathbf{res} - x^{\mathrm{T}}y| = \left|\mathbf{res} - \sum_{j=1}^{MK} q_j + \sum_{j=1}^{MK} q_i - x^{\mathrm{T}}y\right| \leqslant |r_1| + |r_2|, \tag{36}$$

where

$$r_1 := \mathbf{res} - \sum_{j=1}^{MK} q_j \quad \text{and} \quad r_2 := \sum_{j=1}^{MK} q_j - x^{\mathrm{T}}y.$$

Here, using $\mathbf{res} = \mathbf{SumK}(q, K)$ and (4) yields

$$|r_1| \leqslant \mathbf{u}'\left|\sum_{i=1}^{MK} q_i\right| + \gamma_{2(MK-1)}^K \sum_{i=1}^{MK} |q_i| \leqslant \mathbf{u}'(|r_2| + |x^{\mathrm{T}}y|) + \gamma_{2(MK-1)}^K \sum_{j=1}^{MK} |q_j|, \tag{37}$$

where $\mathbf{u}' := \mathbf{u} + 3\gamma_{MK-1}^2$. Inserting (37) into (36), we have

$$|\mathbf{res} - x^{\mathrm{T}}y| \leqslant \mathbf{u}'|x^{\mathrm{T}}y| + (1 + \mathbf{u}')|r_2| + \gamma_{2(MK-1)}^K \sum_{j=1}^{MK} |q_j|. \tag{38}$$

By (32)–(34), it holds that

$$\begin{aligned} |r_2| &= \left|\sum_{id=1}^{M}\sum_{k=1}^{K} q_k^{(id)} - \sum_{id=1}^{M}(x^{(id)})^{\mathrm{T}} y^{(id)}\right| \leqslant \sum_{id=1}^{M}\left|\sum_{k=1}^{K} q_k^{(id)} - (x^{(id)})^{\mathrm{T}} y^{(id)}\right| = \sum_{id=1}^{M}\left|\sum_{k=1}^{K} q_k^{(id)} - \sum_{j=1}^{2c_{id}} t_j^{(id)}\right| \\ &= \sum_{id=1}^{M}\left|(q_K^{(id)} - t_{2c_{id}}^{(id)}) + \sum_{k=1}^{K-1} q_k^{(id)} - \sum_{j=1}^{2c_{id}-1} t_j^{(id)}\right| = \sum_{id=1}^{M}\left|\sum_{k=1}^{K-1} q_k^{(id)} - \sum_{j=1}^{2c_{id}-1} t_j^{(id)}\right|, \end{aligned} \tag{39}$$

and using (6) and (8) yields

$$|r_2| \leqslant \sum_{id=1}^{M} \left(\gamma_{2c_{id}-2}^{K-1} \sum_{j=1}^{2c_{id}-1} |t_j^{(id)}| \right) \leqslant \sum_{id=1}^{M} \gamma_{2c_{id}-2}^{K-1} \gamma_{c_{id}} |(x^{(id)})^{\mathrm{T}}||y^{(id)}| \leqslant \gamma_{2c-1}^{K} |x^{\mathrm{T}}||y|. \tag{40}$$

Furthermore, it follows by (35) and (32) that

$$\sum_{j=1}^{MK} |q_j| = \sum_{id=1}^{M} \left(\sum_{k=1}^{K} |q_k^{(id)}| \right) = \sum_{id=1}^{M} \left(|q_K^{(id)}| + \sum_{k=1}^{K-1} |q_k^{(id)}| \right) = \sum_{id=1}^{M} \left(|t_{2c_{id}}^{(id)}| + \sum_{k=1}^{K-1} |q_k^{(id)}| \right).$$

Using (7)–(9) yields

$$\begin{aligned} \sum_{j=1}^{MK} |q_j| &\leqslant \sum_{id=1}^{M} \left((1+\gamma_{c_{id}}) |(x^{(id)})^{\mathrm{T}}||y^{(id)}| + (1+\gamma_{4(c_{id}-1)}) \sum_{i=1}^{2c_{id}-1} |t_i^{(id)}| \right) \\ &\leqslant \sum_{id=1}^{M} \{1 + \gamma_{c_{id}} + \gamma_{c_{id}}(1+\gamma_{4(c_{id}-1)})\} |(x^{(id)})^{\mathrm{T}}||y^{(id)}| \leqslant (1+3\gamma_c)|x^{\mathrm{T}}||y|. \end{aligned} \tag{41}$$

Here, $\gamma_c + \gamma_c(1+\gamma_{4c}) \leqslant 3\gamma_c$ is proved as follows: Since $2n\mathbf{u} < 1$ and $M \geqslant 2$, $4c\mathbf{u} < 1$ and $\gamma_{4c} < \frac{1}{3}$, so that $\gamma_c + \gamma_c(1+\gamma_{4c}) \leqslant \frac{7}{3}\gamma_c < 3\gamma_c$.

Inserting (40) and (41) into (38), we finally have

$$|\mathbf{res} - x^{\mathrm{T}}y| \leqslant \mathbf{u}'|x^{\mathrm{T}}y| + \{(1+\mathbf{u}')\gamma_{2c-1}^{K} + (1+3\gamma_c)\gamma_{2(MK-1)}^{K}\}|x^{\mathrm{T}}||y|,$$

which proves (30), and divided by $|x^{\mathrm{T}}y|$ also proves (31). □

Again similarly to the error bound for **DotK** in Theorems 2.12, 3.4 says

$$\left| \frac{\mathbf{res} - x^{\mathrm{T}}y}{x^{\mathrm{T}}y} \right| \lesssim \mathbf{u} + \mathcal{O}(\mathbf{u}^K) \cdot \mathrm{cond}(x^{\mathrm{T}}y),$$

which means the result **res** is obtained by **PDotK** as if computed in internally K-fold working precision and then rounded to the working precision, which is similar to the result by **DotK**.

If $c \ll n$ and $MK \ll n$, then the error bounds (19) and (31) become less than (4) and (18), respectively. Moreover, note that **PSumK** and **PDotK** for $K = 2$ can be specialized according to **Sum2** and **Dot2** in [8], respectively. Then the error bounds (19) and (31) change slightly.

Thus, we have not lost any accuracy by parallelizing **SumK** and **DotK** as **PSumK** and **PDotK**, respectively. In the next section, we will confirm the performance of **PDotK** by numerical experiments.

4. Numerical results

In this section, we present some results of numerical experiments showing the performance of **PDotK** on our shared memory system. We use a PC with 4 processors of Intel Dual-Core Xeon 2.80GHz (1024KB cache) and Intel C++ Compiler 9.0. Therefore, there are 8 CPUs in the shared memory system.

All floating-point operations are done in IEEE standard 754 double precision. We use a compile option `-O3` as usual and the special ones `-axW -xW` for the Intel processor. Moreover, to avoid overdoing the compiler optimizations for **TwoSum** and **TwoProduct**, an extra compile option `-mp` has to be used, which ensures that floating-point arithmetic conforms more closely to the IEEE standard. Otherwise, these algorithms do not work correctly. The parallelization is done using OpenMP with a compile option `-openmp` supported by the Intel compiler.

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

$$R_1 := \frac{T_1}{T_M} \quad \text{and} \quad R := \frac{R_1}{M},$$

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

$$T_M = C + \frac{T_1}{M},$$

where C denotes the elapsed time for the parallelization overhead.

Before testing **PDotK**, we evaluate the parallelization overhead by OpenMP in use. To do this, we measure the scalability and the parallel efficiency of **PSum** with $n = 10^4$, 10^6 and 10^8 which is a parallelized version of recursive summation algorithm:

Algorithm 4.1 (***PSum***). A parallelized version of recursive summation algorithm for calculating $\sum_{i=1}^{n} p_i$ with $p_i \in \mathbb{F}$.

function $s = \mathbf{PSum}(p)$
$s = 0$
% *parallel for*
for $i = 1 : n$
$s = \mathrm{fl}(s + p_i)$
end

We display the results in Fig. 6.

From Fig. 6, it can be seen that increasing the number of processors tends to lower the parallel efficiency. The reason is that the parallelization causes the overhead for initialization, reduction handling and so forth (cf. for example, see [5] for more details). Therefore, for smaller n, the cost for the parallelization overhead becomes relatively larger compared with that for pure floating-point operations.

To generate arbitrarily ill-conditioned data for testing dot product as examples, we develop the following Matlab code.

Algorithm 4.2 (*GenDot2*). Generator of arbitrarily ill-conditioned vectors x, y for testing dot product $x^{\mathrm{T}}y$.

```
function [x,y] = GenDot2(n,cnd)
%{
  Generate arbitrarily ill-conditioned data for dot product
  input
      n: n = length(x)
    cnd: anticipated condition number of dot product
  output
```

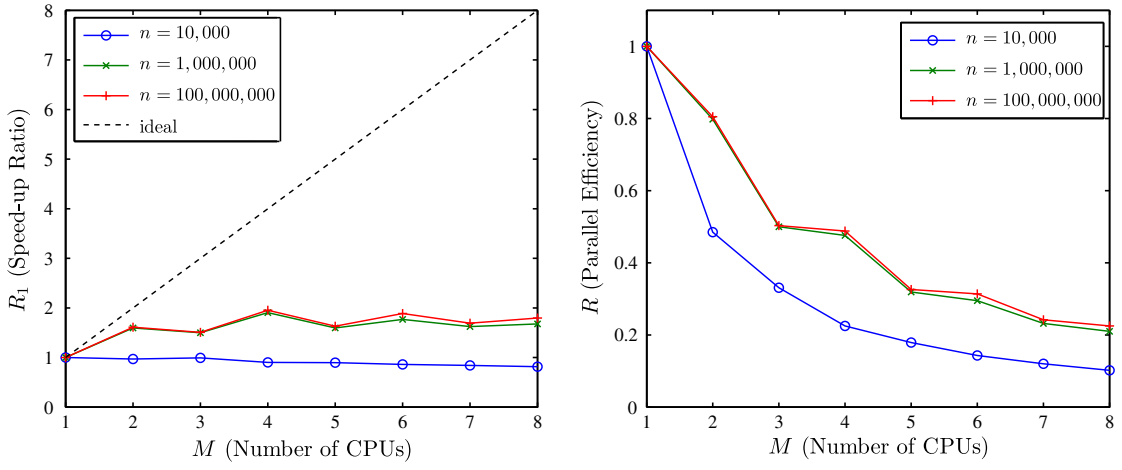

Fig. 6. Scalability (left) and parallel efficiency (right) of recursive summation **PSum**.

```
    x, y: generated vectors with length n
   Note that the exact result of the dot product is cnd ∧-1.
%}
  m=floor(n/2);
  Eps=2∧-24;
  L = floor(log(cnd)/-log(Eps));
  if mod(n,2) == 0
    r = mod(1:m-2,L);
    c=randn(1,m-2).*Eps.∧r;
    x = [1 c 0.5*cnd∧-1 -1 -c 0.5*cnd∧-1]';
    b = randn(1,m-2);
    y=[1 b 1 1 b 1]';
  else
    r = mod(1:m-1,L);
    c=randn(1,m-1).*Eps.∧r;
    x=[1 c cnd∧-1 -1 -c]';
    b=randn(1,m-1);
    y=[1 b 1 1 b]';
  end
```

Using Algorithm 4.2, we can generate two n-vectors x and y as follows: For example, suppose we treat the case where n is an odd number. Set $\texttt{cnd} = 2^{400} \approx 2.58 \cdot 10^{120}$, which is an anticipated condition number of the dot product. Then $\texttt{L} = 16$. Generate pseudo-random vectors $a, b \in \mathbb{F}^{m-1}$ whose components are uniformly distributed in $[-1, 1]$. Define $r_i := \mathrm{mod}(i, 16)$ for $1 \leqslant i \leqslant m-1$, i.e. $r_i = i - \lfloor i/16 \rfloor \cdot 16$, and $c_i := a_i \cdot \texttt{Eps}^{\,r_i}$. Then, we obtain

$$x = (1, c_1, c_2, \ldots, c_{m-1}, \texttt{cnd}^{\,-1}, -1, -c_1, -c_2, \ldots, -c_{m-1})^{\mathrm{T}} \in \mathbb{F}^n,$$
$$y = (1, b_1, b_2, \ldots, b_{m-1}, 1, 1, b_1, b_2, \ldots, b_{m-1})^{\mathrm{T}} \in \mathbb{F}^n,$$

where $n = 2m + 1$. Clearly, the exact result of dot product $x^{\mathrm{T}}y$ is equal to $\texttt{cnd}^{\,-1}$. In the case where n is an even number, x and y are generated similarly.

To focus our mind on the parallel efficiency of **PDotK**, we compare the elapsed time for **PDotK** only with that for **DotK**. More comparisons with other dot product algorithms can be found in [8].

First, we evaluate how tight the error bound for **PDotK** in Theorem 3.4 is in practice. To do this, we set $n = 1,000$ and vary cnd from 2 to 10^{140} in Algorithm 4.2. The error bounds (31) and true relative errors of the results obtained by **PDotK** for $K = 2, 4, 8$ in case of $M = 4$ are displayed in Fig. 7. The lines labeled 'est' for each K denote the error bounds (31) corresponding to K.

From Fig. 7, it can be seen that **PDotK** gives much accurate results than the theoretical error bounds (31), especially for larger K. This is due to the fact that the error bound depends on the constant $\gamma_m^K \approx m^K \mathbf{u}^K$ for $m = 2\lceil n/M \rceil - 1$, while the true error usually does not depend on γ_m^K but $\mathbf{u}^K$ in practice because γ_m^K is the worst case estimation of rounding errors. However, it is attainable in some constructed examples so that it can not be improved. Thus, the more dimension n increases, the larger the difference between the theoretical error bound and the true error becomes. Note that the "bold line" parallel to the 'cond'-axis in Fig. 7 consists of "many markers" which indicate that the relative errors of computed results are all 1's. In case of using examples generated by Algorithm 4.2 (**GenDot2**), if K is not sufficiently large against the condition number, then **PDotK** frequently produces the wrong result **res** $= 0$, and hence the relative error of **res** frequently becomes $|(x^{\mathrm{T}}y - \mathbf{res})/x^{\mathrm{T}}y| = 1$.

Next, we evaluate the elapsed time and parallel efficiency of **PDotK** for $n = 10^6$ and 10^8. For both cases, we set $\texttt{cnd} = 2^{400} \approx 2.58 \cdot 10^{120}$ in Algorithm 4.2 and vary $K = 2, 4, 8$ in **PDotK**. The elapsed times of **PDotK** for both cases are displayed in Table 1. Moreover, Figs. 8 and 9 illustrate both the scalability and the parallel efficiency of **PDotK** compared with **DotK** for $n = 10^6$ and 10^8, respectively.

From Table 1, Figs. 8 and 9, we can observe the following facts:

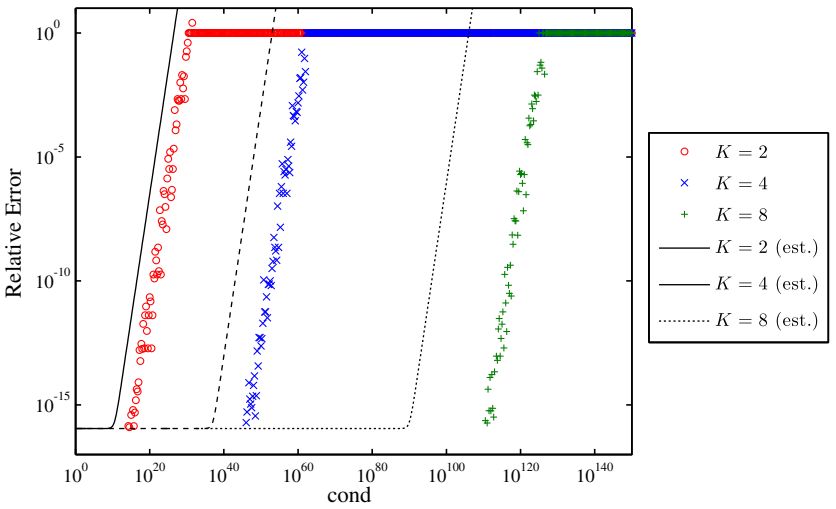

Fig. 7. Error bounds and true relative errors of the results by **PDotK** for $K = 2, 4, 8$ $(n = 1,000, M = 4)$.

Table 1
Elapsed time (s) of **PDotK** for $n = 10^6$ and 10^8

M	$n = 10^6$			$n = 10^8$		
	$K = 2$	$K = 4$	$K = 8$	$K = 2$	$K = 4$	$K = 8$
1	3.43E−02	7.86E−02	1.68E−01	3.38E+00	7.83E+00	1.67E+01
2	1.71E−02	3.93E−02	8.36E−02	1.70E+00	3.91E+00	8.32E+00
3	1.21E−02	2.64E−02	5.57E−02	1.20E+00	2.65E+00	5.60E+00
4	1.00E−02	2.14E−02	4.43E−02	9.68E−01	2.11E+00	4.38E+00
5	1.07E−02	2.43E−02	5.14E−02	1.05E+00	2.40E+00	5.09E+00
6	1.00E−02	2.36E−02	5.57E−02	9.76E−01	2.35E+00	5.12E+00
7	1.07E−02	2.43E−02	5.07E−02	1.05E+00	2.43E+00	5.21E+00
8	1.00E−02	2.21E−02	4.57E−02	9.80E−01	2.26E+00	4.81E+00

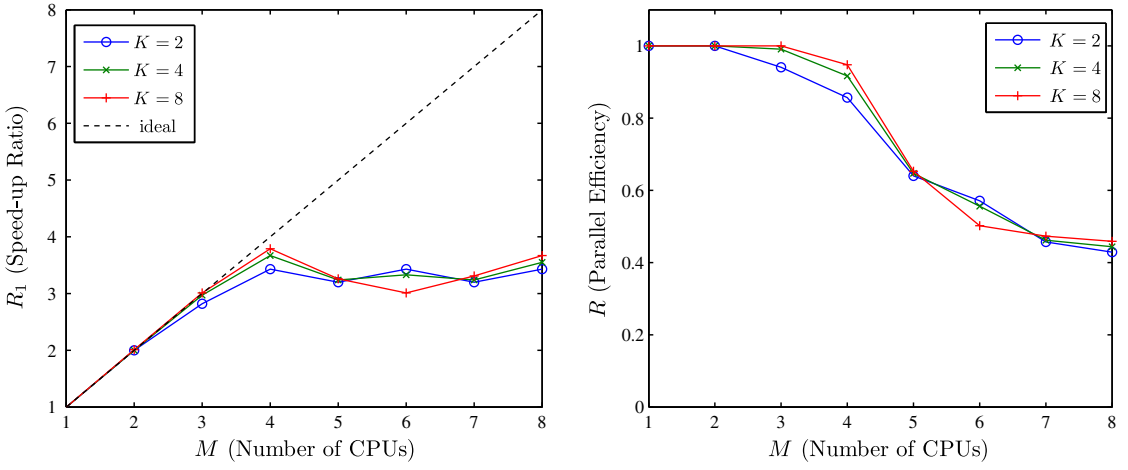

Fig. 8. Scalability (left) and parallel efficiency (right) of **PDotK** for $n = 10^6$.

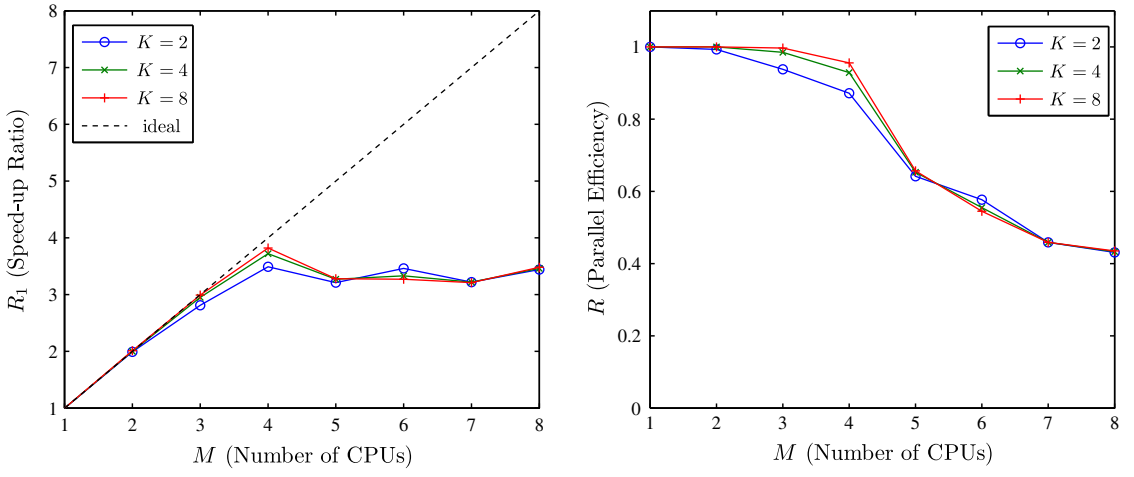

Fig. 9. Scalability (left) and parallel efficiency (right) of **PDotK** for $n = 10^8$.

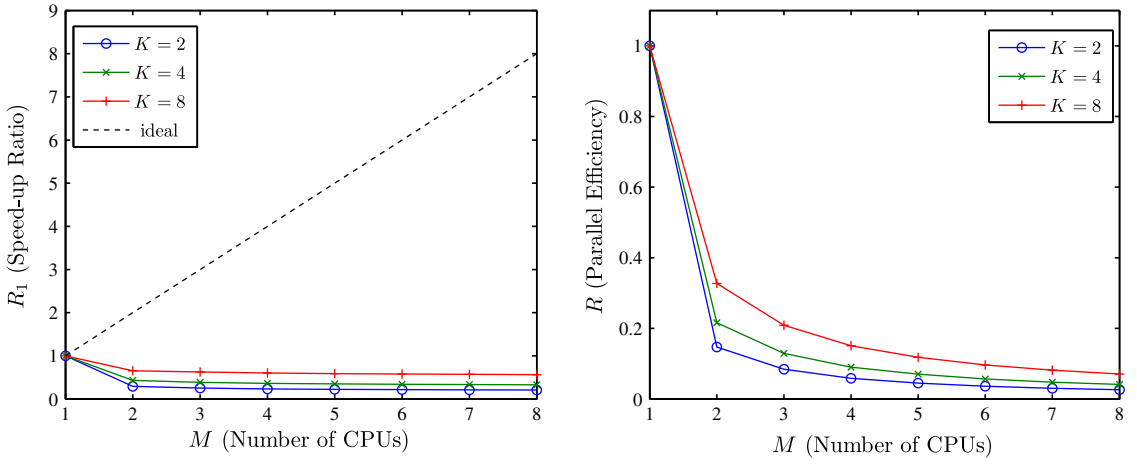

Fig. 10. Scalability (left) and parallel efficiency (right) of **PDotK** on distributed memory environment including data transfer time ($n = 10^6$).

- When the number of floating-point operations increases according to an increase of K for each n, the parallel efficiency is basically improved because the parallelization overhead becomes relatively small.
- In the case of $M = 2$, **PDotK** achieves an almost ideal parallel efficiency for $n \geqslant 10^6$.
- In the cases of $M = 3, 4$, the parallel efficiency is acceptable, although it gradually drops down as M increases.
- In the cases of $M \geqslant 5$, the parallel efficiency significantly drops down and it seems to be meaningless in terms of the elapsed time to increase M in this computer environment.

It turns out that the proposed algorithm **PDotK** is very effective for $M = 2$, and meaningful for $M \leqslant 4$, especially for larger K. The results with K-fold precision are ensured by Theorem 3.4.

So far we conducted the numerical experiments on multi-core system. Therefore, the cores on the same processor conflict with each other for the shared cache, and hence for the bandwidth to access the shared memory. To see this effect clearly, we also conduct the numerical experiments on distributed memory system, on which each processing unit does not share the cache with others. We use 8 PCs with Intel Xeon 3.0 GHz (4096KB cache), GNU C Compiler 3.4.6 and MPICH 1.2.7p1 with compile options `-O3 -funroll-loops -mcpu=nocona`. These PCs are connected with Gigabit Ethernet LAN.

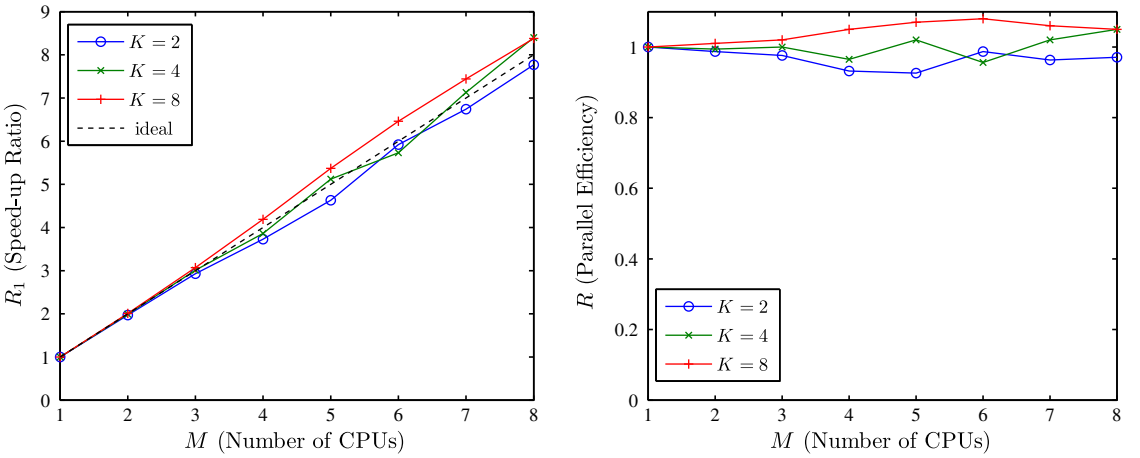

Fig. 11. Scalability (left) and parallel efficiency (right) of **PDotK** on distributed memory environment excluding data transfer time ($n = 10^6$).

We implement **PDotK** for this distributed system and use the same examples as the above ones. Figs. 10 and 11 illustrate both the scalability and the parallel efficiency of **PDotK** compared with **DotK** for $n = 10^6$. The difference between Figs. 10 and 11 is that the computing time for **PDotK** in Fig. 10 includes the data transfer time via MPI for sending the necessary data in input vectors x and y to each PC, and that in Fig. 11 does not.

From Figs. 10 and 11, we can observe the following facts:

- In Fig. 10, the data transfer time is dominant and it is meaningless to use **PDotK** instead of **DotK** in terms of the elapsed time.
- In Fig. 11, **PDotK** achieves almost ideal parallel efficiency.

Thus, if the necessary data in input vectors are already distributed to all CPUs which do not conflict with each other for the shared cache and the shared memory, then **PDotK** can achieve a high scalability.

5. Conclusion

In this paper, we proposed the parallel algorithms for accurate sum and dot product. The proposed algorithms achieved the results as if computed in K-fold working precision similarly to the algorithms in [8] on parallel computing environments. By the numerical experiments, we confirmed that the proposed parallel algorithm works effectively up to 4 CPUs on our computer environment. In practice, the proposed algorithms frequently give more accurate results than the theoretical error bounds.

Acknowledgements

The authors heartily wish to thank the two anonymous referees for their thorough reading and most valuable comments. This research was partially supported by Grant-in-Aid for Young Scientists (B) (No. 19700013) from the Ministry of Education, Science, Sports and Culture of Japan.

References

[1] W.E. Arnoldi, The principle of minimized iteration in the solution of the matrix eigenvalue problem, Quart. Appl. Math. 9 (1951) 17–25.
[2] D.E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, vol. 2, Addison-Wesley, Reading, Massachusetts, 1969.

[3] Z. Drmač, K. Veselić, New fast and accurate Jacobi SVD algorithm: I, LAPACK Working Note, 169 (2005).
[4] T.J. Dekker, A floating-point technique for extending the available precision, Numer. Math. 18 (1971) 224–242.
[5] M. Gerndt, B. Mohr, J.L. Träff, Evaluating OpenMP performance analysis tools with the APART test suite, Lect. Notes Comput. Sci. 3149 (2004) 155–162.
[6] N.J. Higham, Accuracy and Stability of Numerical Algorithms, second ed., SIAM, Philadelphia, 2002.
[7] X. Li et al., Design, implementation and testing of extended and mixed precision BLAS, ACM Trans. Math. Softw. 28 (2002) 152–205.
[8] T. Ogita, S.M. Rump, S. Oishi, Accurate sum and dot product, SIAM J. Sci. Comput. 26 (6) (2005) 1955–1988.