Achieving Numerical Reproducibility in the Parallelized Floating Point Dot Product

Alyssa Anderson
College of Saint Benedict/Saint John's University

Achieving Numerical Reproducibility in the Parallelized Floating Point Dot Product

AN HONORS THESIS

College of St. Benedict/St. John's University

In Partial Fulfillment

of the Requirements for Distinction

In the Departments of *Computer Science* and *Mathematics*

by

Alyssa Anderson

April, 2014

PROJECT TITLE: *Achieving Numerical Reproducibility in the Parallelized Floating Point Dot Product*

Approved by:

Michael Heroux
Thesis Advisor; Scientist in Residence, Department of Computer Science

Imad Rahal
Reader; Associate Professor of Computer Science; Chair, Department of Computer Science

Bob Hesse
Reader; Associate Professor of Mathematics; Chair, Department of Mathematics

Anthony Cunningham
Director, Honors Thesis Program

Contents

1 Introduction **4**

1.1 Introduction to Parallel Computing . . . 5

2 Problem Statement **6**

2.1 The Parallel Dot Product . . . 6

3 Background **7**

3.1 Floating Point Numbers . . . 7

3.1.1 8-bit Floating point Representation . . . 9

3.2 Reproducibility vs. Accuracy . . . 11

4 Methods **11**

4.1 Code and Parallel Programming Model . . . 11

4.2 Architecture . . . 12

4.3 Kahan Summation Algorithm . . . 12

4.4 Knuth Summation Algorithm . . . 17

4.5 The Experiment . . . 24

5 Results **25**

5.1 Double-Precision Results . . . 25

5.2 Single-Precision Results . . . 42

5.3 Single-Precision with Double-Precision Accumulator Results . 49

6 Conclusions **54**

7 Contributions to this Topic **56**

8 Future Work **56**

9 Appendix **57**

9.1 Source Code . . . 57

9.1.1 TestDotProductOMPDouble.cpp . . . 57

9.1.2 TestDotProductKahanOMPDouble.cpp . . . 58

9.1.3 TestDotProductKnuthOMPDouble.cpp . . . 61

9.1.4 TestDotProductOMPFloat.cpp . . . 64

9.1.5 TestDotProductKahanOMPFloat.cpp . . . 65

9.1.6 TestDotProductKnuthOMPFloat.cpp . . . 68

9.1.7 TestDotProductOMPDoubleAccumulator.cpp 70
9.1.8 TestDotProductKahanOMPDoubleAccumulator.cpp . . 72
9.1.9 TestDotProductKnuthOMPDoubleAccumulator.cpp . . 75

1 Introduction

We are taught the importance of real numbers before we reach high school. The world depends on computers every day to do accurate real-world mathematics. However, most people do not know that the way the world understands real numbers is very different from how computers represent real numbers, and that this can cause computations to be incredibly inaccurate. Because real numbers like $\pi = 3.141592...$ can have an infinite number of digits and because they are not countably infinite, the computer must store real numbers in a finite representation that approximates real numbers, called floating point representation. Floating point numbers are often taken for granted by scientists and computer users alike, because we expect them to work like real numbers. However, simply by changing the order in which we add a list of floating point numbers can give us a different result. This result may even be less accurate than another ordering. This is because floating point addition is not associative. That is, $(a + b) + c$ is not necessarily equal to $a + (b + c)$. Correspondingly, one ordering of a sum may produce a more accurate result than another ordering. While we are not necessarily affected by this in serial computations, parallel techniques introduce the ability to order computations differently, thus producing a difference in results between runs.

Reproducibility means getting a bit-wise identical result every time an application is run. Because scientific applications have come to use parallel computing techniques in order to solve more and more complex problems, non-reproducibility is often introduced into scientific applications in the form of basic operations these applications rely on, such as the dot product. Techniques for achieving reproducibility in parallel computations include eliminating (or reducing) rounding error, using a deterministic computation order, pre-rounding, and using a super-accumulator. We will use methods that attempt to reduce rounding error. We begin with an explanation of the problem in section two, and then we cover a review of relevant background knowledge in section three. Next, we describe the methods chosen for addressing the problem and justify our choice of methods in section four. Section five presents reproducibility results for each reproducibility technique used. Section six draws conclusions about the reproducibility of each method. Section seven explains our contributions to this field. Section eight describes future work that may be done in this area, and section nine provides an Appendix that includes the source code for this thesis.

1.1 Introduction to Parallel Computing

We now provide a brief introduction to parallel computing in order to clarify concepts from this field that are relevant to this thesis. Parallel computing is an alternative to traditional serial computing, in which software is written to be run on a single processor and tasks (or units of work) are run one at a time. Parallel computing, however, involves separating the work of independent computing tasks among several processors in order to speed up the amount of time it takes to do all of the tasks from executing them in serial. In other words, parallel computing is a way of multi-tasking for more efficient use of computing resources. On a multicore or many- processor, a single independent software thread may be executed on each core. Independence, in this context, means that each thread does not depend on the results of any other threads to do a task. A software thread is a single software unit of parallel work that has an independent flow of control from the other software threads.[9] A hardware thread is any hardware unit (a single core in multicore and many-core architectures) that can execute a single software thread at a time.[9] Thus, in parallel computing, many software threads may be dispersed among a few hardware threads. There may also be dependencies among threads which dictate whether a thread can be executed at a given time. That is, if the computation of one software thread depends on the result of another software thread that has not yet finished execution, it must wait until that thread has finished to execute. This introduces problems such as "race conditions" in which two threads are mutually dependent one each other's result.[9] These conditions are often difficult to predict or only appear under specific circumstances and are one reason that parallel computing is much more difficult than serial computing.[9]

Parallel computing is meant to function in the same way as seqential (or serial) computing in that we obtain the same results as similar computations executed in serial (with only one hardware thread). The potential for decrease in runtime has made parallel computing a necessary tool for computing in the face of limitations in current CPU clock speeds (see Section 2: Problem Statement). However, as we will see, parallel computing may introduce problems such as non-reproducibility into floating-point computations.

2 Problem Statement

Parallel computing has become necessary for increasing performance since engineers have hit a power wall in computing architectures. This is due to the fact that increasing the clock speed causes a non-linear increase in the amount of power required. However, clock speeds have topped out at as high as 3 GHz, at which point a chip cannot handle higher clock speeds without overheating. Additionally, Instruction-Level Parallelism (ILP), or parallelism at the hardware level, has reached its limit since it has been implemented in computing architectures for decades and is now a natural part of hardware. [9] Thus, parallel computing, or sharing work among several processors, has become necessary for maintaining increases in performance. However, parallel computing introduces its own set of challenges that sequential computing does not have. A full discussion of these challenges is outside of the scope of this thesis. However, one of these challenges is that parallel computations may introduce non-reproducibility into computations that were previously reproducible. With advances geared toward Exascale computing, in which 10^{18} floating point operations per second (FLOPS) will be possible, reproducibility and accuracy will become increasingly difficult to maintain. [3]

We hope to achieve a bit-reproducible parallel dot product. The dot product is a useful operation for studying reproducibility because it is common to many parallel scientfic computing applications and is non-reproducible when computed in parallel. Reproducibility in parallel scientific applications is desirable for several reasons. If a result is not reproducible, it may be hard to validate that the result is correct. Additionally, non-reproducible results can mask bugs in the code that cause similar variation in results. [2] Having a reproducible scientific application makes any errors in or problems with the result distinguishable from those due to the non-associativity of floating point addition. That is, once the result is reproducible, we can no longer attribute non-reproducibility to the cause of erroneous or inconsistent results.

2.1 The Parallel Dot Product

The dot product of two vectors $\mathbf{x} = \{x_1, x_2, ..., x_n\}$ and $\mathbf{y} = \{y_1, y_2, ..., y_n\}$ is defined as

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + ... + x_n y_n \tag{1}$$

Each multiplication of corresponding entries of each vector is independent of the others, and so these computations may be done in parallel. (Ex. Each parallel thread may compute a local sum of the multiplications of corresponding elements of the two vectors.) However, the result requires adding these local sums together. Since these sums are all being added to a global result, they are not independent of each other. Thus, we have what is called a **reduction pattern**, where a commutative operator (addition, in this case) is used to combine the local results after the results of the first part of the computation were computed in parallel.[9] In our implementations, this means that each parallel thread may add its result to the global sum with the guarantee that no two threads will try to update the global sum at the same time. However, parallelizing the dot product means that the order in which the local results are combined changes between runs. Thus, each thread may finish its scheduled work at a different time, allowing for whichever thread is ready first to add its result to the global sum. This is how non-reproducibility is introduced into the computation.

3 Background

3.1 Floating Point Numbers

We will use the IEEE-754-2008 standard in our discussion of floating point numbers, since this is the current approved standard for almost all computer architectures. Floating point is a binary representation used to approximate real numbers in computers. Similar to scientific notation of real numbers (base 10), floating point numbers are represented as base 2. The precision of a floating point number depends on the number of bits used in its representation. A single-precision floating point number, or a **float** in C++, is 32 bits in length whereas a double-precision floating point number, or **double** in C++, is 64 bits in length.

We will discuss double-precision floating point numbers to understand the specifics of floating point numbers. The leftmost bit of a double-precision number is the sign bit, which is 0 if positive and 1 if negative. The remaining two parts of the 64-bit number (as we move from the left-most or most significant bits to the right-most or least significant bits) are split into the exponent and mantissa (or significand) bits. The exponent is composed of 11 bits, but represents the binary number represented by the 11 bits subtracted by a fixed offset of $2^{10} - 1$, or 1023. The mantissa is made up of 53 bits, but is represented by the remaining 52 bits. This is because the first bit of the mantissa is always 1, and thus is an implied bit that does not need to be physically represented. [7]

Floating point addition of positive numbers works as follows. [7] (See Example 1 for an example of floating point addition.) First, the mantissa of the number with the smaller exponent must be shifted to the right a number of places equal to the difference in exponents. (Both mantissas must now account for the leading 1 to the left of the mantissa at the start of addition.) The exponent of this number is then changed to the exponent of the larger number. The two mantissas are then added using binary addition. If the result requires that the leading 1 become 10 (due to overflow), then the result must be shifted to the right one place and 1 must be added to the exponent. To ensure correct rounding, floating point addition requires that there be two extra guard bits and a sticky bit (one that becomes the OR of all bits that pass through it as bits are shifted to the right during addition) to the right of the mantissa bits. There are several rounding modes maintained by the IEEE-754 standard, such as round-to-infinity, round-to-zero, round-to-negative-infinity, and round-to-nearest (in which the floating point number is rounded to the nearest even floating point number). Round-to-nearest is the most common and is the rounding mode that we will use. After floating point addition is computed, any bits retained in the two guard and sticky bits determine the rounding. We will call these bits GGS. If the first guard bit is 0, then no rounding is required. If the $GGS = 100$, then rounding up (by adding 1 to the least significant - rightmost - mantissa bit) is only required if the least significant mantissa bit is not 0. If it is 0, then nothing needs to be done to the result. If $GGS = 101, 110$, or 111, then rounding up is required. When rounding up, we must again make sure to shift the result right one place and add one to the exponent if rounding up requires 1 to be added to the leading 1. Floating point addition and subtraction work similarly, except that the mantissas are either added or subtracted depending on the sign bits. (If both sign bits are equal in addition, then the mantissas are added. If they are not equal, one is subtracted from the other). If subtraction is involved, there can be underflow (the leading 1 can become 0, causing the mantissa to be shifted left and the exponent to be reduced a number of times until there is a leading 1).

Although floating point numbers are meant to represent real numbers, they are only an approximation of real numbers. Floating point numbers cannot accurately represent all real numbers, since there is only ever a finite amount of precision available (usually 32 or 64 bits). Thus, exact arithmetic is not guaranteed. In fact, there are many special cases in which floating point arithmetic can fail. A thorough explanation of all of the intricacies of floating point arithmetic is beyond the scope of this thesis. However, of main importance is the fact that floating point addition is non-associative. That is, for floating point numbers a, b, and c, there is no guarantee that $(a + b) + c = a + (b + c)$. We will see this in Example 1. On the other hand, floating point addition is in fact commutative, meaning for floating point numbers a and b, $a + b = b + a$. Because floating point addition is non-associative, we know that changing the order in which the sums of the dot product are accumulated may produce different results. Parallelizing the dot product causes the order of accumulation of the local sums to be non-deterministic. Thus, for two vectors $\mathbf{x}$ and $\mathbf{y}$, the result may change from one parallel computation of their dot product to the next.

3.1.1 8-bit Floating point Representation

For simplicity, we will use our own 8-bit (or 1-byte) floating point representation in IEEE-754-2008 style to show how floating point addition works and to show how the Kahan and Knuth methods correct for non-associativity. Our exponent will be composed of 3 bits, consequently having an offset of $2^2 - 1 = 3$. Thus, for our purposes, a binary floating point number looks like
$s\ \ eee\ \ m_1m_2m_3m_4$
and represents the number $1.mmmm \times 2^{eee-3}$ in binary, or the decimal value $(1 + \sum_{n=1}^{4} m_n * 2^{-n}) * 2^{eee-3}$ $(* - 1$ if $s == 1)$.

Example 1. Foating Point Nonassociativity
We will show, using our 8-bit floating point number representation, that floating point numbers are not associative. Let a, b, and c be floating point numbers such that
$a = 0\ 110\ 0100$, or 10 in decimal, $b = 0\ 110\ 0101$, or 10.5 in decimal, and $c = 0\ 001\ 0001$, or 0.265625 in decimal. First, we will compute $(a + b) + c$.

Note that the bit(s) left of the | symbol are to the left of the radix point.

	0 110 1|0100	*a with leading 1*
+	0 110 1|0101	*b with leading 1*
	0 110 10|1001	*a + b before normalization*
	0 111 1|01001	*shifted one to the right and added one to exp*
	0 111 0100	*a + b (no rounding required, truncated last bit)*
+	0 001 0001	*c*
	0 111 1|0100	*a + b with leading 1*
+	0 111 0|0000010001	*normalized c with leading 1*
	0 111 1|0100010001	*(a + b) + c before rounding*
	0 111 0100	*(a + b) + c (no rounding required)*

So, $(a + b) + c =$ 0 111 0100, which equals 20 in decimal.
Next, we compute $a + (b + c)$.

	0 110 0101	*b*
+	0 001 0001	*c*
	0 110 1|0101	*b with leading 1*
	0 110 0|000010001	*normalized c with leading 1*
	0 110 1|010110001	*b + c before rounding*
	0 110 1|0101	
+	0 000 0|0001	*1 to be added for rounding*
	0 110 0110	*b + c*

	0 110 1|0100	*a with leading 1*
+	0 110 0110	*b + c with leading 1*
	0 110 10|1010	*a + (b + c) before normalization*
	0 111 1|01010	*a + (b + c) before rounding (no rounding required)*
	0 111 0101	*a + (b + c)*

So, $a + (b + c) =$ 0 111 0101, which equals 21 in decimal. $(a + b) + c = 20 \neq 21 = a + (b + c)$. Thus, floating point numbers are not associative.

Example 2. Floating Point Truncation Error
This example demonstrates how truncation error affects a numerical result. Because we only have a finite amount of precision available to us in floating point numbers, the significant digits of a number can be lost if their exponents are too dissimilar. When a large number is added to a small number in floating point, the result may not accurately account for the smaller number (the smaller number's significant digits are shifted to the right to make the exponents equal for adding) since the smaller number's significant digits will

be truncated if they are shifted more than 3 digits past the least significant bit. To demonstrate this, we wil add 01110100, or 20 in decimal to 00010001, or 0.265625.

$$\begin{array}{ll}
 & 0\,111\,0100 \\
+ & 0\,001\,0001 \\
\hline
 & 0\,111\,1|0100 \quad \textit{with leading 1} \\
+ & 0\,111\,0|0000010001 \quad \textit{with leading 1} \\
\hline
 & 0\,111\,1|0100010001 \quad \textit{addition before rounding} \\
\hline
 & 0\,111\,0100 \quad \textit{no rounding required: last bit of mantissa = 0 and GGS = 010}
\end{array}$$

As shown, the significant digits of the smaller number were truncated since they were shifted too far to the right to make the two exponents equal. Thus, the answer is 20 instead of the exact arithmetic 20.265625. Thus, if our global sum variable is 20 and we keep adding partial sums of 0.265625 to it, the sum will not get any bigger than 20, even though the after 4 additions of partial sums, the global sum should be at least 21, which can be represented by our 8-bit representation. These extreme cases can wreak havok on numerical computations, although they are unlikely.

3.2 Reproducibility vs. Accuracy

While reproducibility and accuracy are related, bit-accurate does not mean the same thing as bit-reproducible. Reproducibility in parallel programming is defined as "getting the same result independent of the number of processors," but in our case we define it to mean getting the same result on runs with identical settings.[10] Accuracy, on the other hand, is dependent on the amount of numerical error in the result.[10] Floating point addition is not associative due to truncation error. Thus, if a dot product were computed using infinite precision, then it would be 100% accurate and thus would be reproducible since every run would give the same, 100% accurate result. However, the converse does not hold. That is, reproducibility does not imply accuracy.

4 Methods

4.1 Code and Parallel Programming Model

All code for this thesis is written in C++. All source code may be found in the Appendix. OpenMP version 3.0 is the programming model used to implement parallelization of the dot product, because of its widespread use and simplicity.

4.2 Architecture

The target machine on which our tests were conducted is a machine named Melcior at Saint John's University in Collegeville, Minnesota. Melchior is an Intel Xeon MIC Processor with 8 nodes (labeled 0 through 7). All tests were run on node 1. Each node has 24 thread states running with a clock speed of 1.9 GHz, with the exception of node 2, which has a clock speed of 2.00 GHz. This particular machine also has two attached coprocessors: an NVIDIA Tesla K20 card and an Intel PHI 5110P Knights card, neither of which were used in this thesis. We chose to use the GCC version 4.4.7-4 compiler. The Linux kernel version of our platform was 2.6.32-431.5.1.el6.x86_64.

4.3 Kahan Summation Algorithm

The Kahan summation algorithm is a technique for enhancing the precision of a global sum (a sum of computations done on several parallel threads), such as that required by the parallel reduction pattern in the parallel dot product. The method was developed in 1965, and uses extra precision by means of carry digits that hold the truncated part of each intermediate sum in a correction term. [10] In other words, loss of precision due to truncation error (see Example 1) can be (at least mostly) recovered by means of a correction term. Although the Kahan sum can be used for individual summations, since we are using it for a running sum of multiplications (the dot product), we will explain the algorithm as a running summation. The Kahan Running Summation Algorithm is as follows:

1. Let `sum`, `correction`, `corrected_next_summand`, and `new_sum` be floating point numbers. (The precision of all floating point numbers, including the summands, should be the same).

2. Initialize `sum` and `correction` to 0.0.

3. for $i = 1$ to the number of summands

    - `corrected_next_summand` $\leftarrow$ next summand - `correction`
    - `new_sum` $\leftarrow$ `sum` $+$ `corrected_next_summand`

- `correction` $\leftarrow$ (`new_sum` - `sum`) - `corrected_next_summand`
- `sum` $\leftarrow$ `new_sum`

4. end for

At the start of the first iteration, the `correction` term will be equal to 0.0, and thus the first `corrected_next_summand` will be equal to the first summand. Thus, `sum` at the end of the first iteration will be equal to the first summand and the `correction` term will be zero since `new_sum` = `sum`. The idea is that as more iterations of the for-loop execute, `sum` will get much bigger in relation to the individual summands. Thus, the truncation error that is likely when adding each summand to the overall sum is what the correction term is recovering, as long as $|$`sum`$| > |$`corrected_next_summand`$|$. The `correction` term takes the truncated addition of `sum` to `corrected_next_summand`, which is stored in `new_sum`, and first subtracts `sum` (which does not yet have the next summand, `corrected_next_summand`, added to it). By subracting `sum`, we thus recover how much was added to `sum` by a normal addition of `sum` with the next summand (a less precise version of the next summand is left over). The next part of the correction term subtracts the `corrected_next_summand` (the most precise version of the next summand) from this less precise version of the next summand, thus recovering the bits of the mantissa that were lost in normal floating point addition since these two terms are of similar magnitude (if both are normalized to have the same exponent as the `new_sum` variable, their bit difference is beyond the least significant digit of the mantissa) and thus this will give an accurate difference that does not lose the significant bits of the next summand.

The Kahan summation method is ideal for running sums like the dot product, since only the first few sums are likely to be greater than the next summand and thus most of the truncated bits will be recovered.

Example 3. Reproducibility by Kahan Method
In this Kahan summation example, since we are using our 8-bit floating point representation, we will have 8-bit correction terms (all variables will be in our 8-bit floating point representation). Suppose we add floating point numbers a, b, and c from Example 1 again. We can now show that $(a+b)+c = a+(b+c)$ when we use the Kahan summation algorithm to compute each sum. As before,
$a = 0\ 110\ 0100 = 10$,
$b = 0\ 110\ 0101 = 10.5$, and

$c = 0\,001\,0001 = 0.265625$. First, we will compute $(a+b)+c$. The bit(s) left of the | symbol are to the left of the radix point.

```
sum = 0.0;
correction = 0.0;
corrected_next_term = a - correction;
    //corrected_next_term ← 0 110 0100
new_sum = sum + corrected_next_term;
    //new_sum ← 0 110 0100
correction = (new_sum - sum) - corrected_next_term;
    //correction ←
```

$$\begin{array}{rl} & 0\,110\,0100 \\ - & 0\,000\,0000 \\ \hline & 0\,110\,0100 \\ - & 0\,110\,0100 \\ \hline & 0\,000\,0000 \end{array}$$

```
sum = new_sum;
    //sum ← 0 110 0100
```

```
corrected_next_term = b - correction;
    //corrected_next_term ← $b - 0.0 = 0\,110\,0101$
new_sum = sum + corrected_next_term;
    //new_sum ←
```

$$\begin{array}{rll} & 0\,110\,0100 & a \\ + & 0\,110\,0101 & b \\ \hline & 0\,111\,0100 & \textit{as in Example 1} \end{array}$$

```
correction = (new_sum - sum) - corrected_next_term;
    //correction ←
```

$$\begin{array}{rl}
 & 0\ 111\ 0100 \\
- & 0\ 110\ 0100 \\ \hline
 & 0\ 111\ 1|0100 \\
- & 0\ 111\ 0|10100 \\ \hline
 & 0\ 111\ 0|10100 \quad \textit{new\_sum - sum before normalization} \\ \hline
 & 0\ 110\ 1|0100 \\ \hline
 & 0\ 110\ 0100 \quad \textit{new\_sum - sum = 8} \\
- & 0\ 110\ 0101 \quad \textit{corrected\_next\_term = 10.5} \\ \hline
 & 0\ 110\ 1|0100 \\
- & 0\ 110\ 1|0101 \\ \hline
 & 1\ 110\ 0|0001 \quad \textit{result before normalization} \\ \hline
 & 1\ 010\ 1|0000 \quad \textit{result = -0.5}
\end{array}$$

sum = new_sum;

//sum ← 0 111 0100

---

corrected_next_term = c - correction;

//corrected_next_term ←

$$\begin{array}{rl}
 & 0\ 001\ 0001 \\
- & 1\ 010\ 0000 \\ \hline
 & 0\ 010\ 0|10001 \\
+ & 0\ 010\ 1|0000 \\ \hline
 & 0\ 010\ 1|10001 \\ \hline
 & 0\ 010\ 1000
\end{array}$$

new_sum = sum + corrected_next_term;

//new_sum ←

$$\begin{array}{rl}
 & 0\ 111\ 0100 \\
+ & 0\ 010\ 1000 \\ \hline
 & 0\ 111\ 1|0100 \\
+ & 0\ 111\ 0|000011000 \\ \hline
 & 0\ 111\ 1|010011000 \\ \hline
 & 0\ 111\ 0101
\end{array}$$

correction = (new_sum - sum) - corrected_next_term;

//correction ← do not need to compute for (a + b) + c since there are no more iterations

sum = new_sum;

//sum ← 0 111 0101

Now we compute (b + c) + a = a + (b + c) by the commutative property of floating point addition.

```
sum = 0.0;
correction = 0.0;
corrected_next_term = b - correction;
    //corrected_next_term ← 0 110 0101
new_sum = sum + corrected_next_term;
    //new_sum ← 0 110 0101
correction = (new_sum - sum) - corrected_next_term;
    //correction ← 0.0
sum = new_sum;
    //sum ← 0 110 0101
```

```
corrected_next_term = c - correction;      //corrected_next_term ← 0 001 0001
new_sum = sum + corrected_next_term;
    //new_sum ←
                  0 110 0001
             +    0 001 0001
             -----------------
                  0 110 1|0001
             +    0 110 0|000010001
             -----------------
                  0 110 1|000110001
             -----------------
                  0 110 0010
correction = (new_sum - sum) - corrected_next_term;
    //correction ←
                  0 110 0010
             −    0 110 0101
             ------------------------------
                  1 110 0|0011  before normalization
             ------------------------------
                  1 011 1|1000
             ------------------------------
                  1 011 1000  new_sum - sum
             −    0 001 0001  corrected_next_term
             ------------------------------
                  1 011 1|1000
             −    0 011 0|010001
             ------------------------------
                  1 011 1100
sum = new_sum;
    //sum ← 0 110 0010
```

```
corrected_next_term = a - correction;
    //corrected_next_term ←
```

$$\begin{array}{cl} & 0\,110\,0100 \\ - & 1\,011\,1100 \\ \hline & 0\,110\,1|0100 \\ + & 0\,011\,1|1100 \\ \hline & 0\,110\,1|0100 \\ + & 0\,110\,0|0011100 \\ \hline & 0\,110\,1|0111100 \\ \hline & 0\,110\,1|1000 \\ \hline & 0\,110\,1000 \end{array}$$

new_sum = sum + corrected_next_term;

  //new_sum ←

$$\begin{array}{cl} & 0\,110\,1|0010 \\ + & 0\,110\,1|1000 \\ \hline & 0\,110\,10|1010 \\ \hline & 0\,111\,1|01010 \\ \hline & 0\,111\,0101 \end{array}$$

correction = (new_sum - sum) - corrected_next_term;

  //correction ← do not need to compute since there are no more iterations

sum = new_sum;

  //sum ← 0 111 0101

Thus, $(a + b) + c = a + (b + c) = 0\,111\,0101 = 21$ when computed using the Kahan summation algorithm.

4.4 Knuth Summation Algorithm

The Knuth summation algorithm, like the Kahan summation algorithm, is a technique for enhancing precision in computing a global sum. It is based on Donald Ervin Knuth's development of a formal way to analyze error in finite precision arithmetic. [10] The algorithm comes from Theorm B in Volume 2 of Knuth's Book, The Art of Computer Programming, given below. [8]

Theorem. *Suppose that + is reserved for infinite precision addition (elsewhere in this thesis + means floating point addition) and that $\oplus$ and $\ominus$ are the (finite) floating point addition and subtraction operators, respectively. Then for normalized floating point numbers u and v,*

$$u + v = (u \oplus v) + ((u \ominus u') \oplus (v \ominus v''))$$

where $u' = (u \oplus v) \ominus v$ *and* $v'' = (u \oplus v) \ominus u'$.

In other words, $((u \ominus u') \oplus (v \ominus v''))$ gives an explicit formula for the difference between $u+v$ and $u \oplus v$. Thus, it is the correction term in the algorithm, even though we cannot expect it to be exact since the theorem uses exact (infinite) addition to add the correction term to the trancated sum $u \oplus v$. However, since the correction term is exact, we expect improvement over the Kahan summation. [10] The Knuth summation is an improvement over the Kahan summation, because its correction term corrects both for truncated bits of v when $|u| > |v|$ and for truncated bits of u when $|v| > |u|$. In other words, the correction term is the same if u and v are reversed. Again, we provide the algorithm as a running summation. The Knuth Running Summation Algorithm is as follows:

1. Let `sum`, `correction`, `u`, `v`, `upt`, `up`, and `vpp` be floating point numbers. (The precision of all floating point numbers, including the summands, should be the same).

2. Initialize `sum` and `correction` to 0.0.

3. for $i = 1$ to the number of summands

    - `u` ← `sum`
    - `v` ← next_summand + `correction`
    - `upt` ← `u` + `v`
    - `up` ← `upt - v`
    - `vpp` ← `upt - up`
    - `sum` ← `upt`
    - `correction` ← (`u - up`) + (`v - vpp`)

4. end for

Example 4. Reproducibility by Knuth Method
In this Knuth summation example, all variables will again be in our 8-bit floating point representation. Suppose we add floating point numbers a, b, and c from Example 1 again. We can now show that $(a+b)+c = a+(b+c)$ when we use the Knutn summation algorithm to compute each sum. As before,

$a = 0\,110\,0100 = 10$,
$b = 0\,110\,0101 = 10.5$, and
$c = 0\,001\,0001 = 0.265625$. First, we will compute $(a + b) + c$. The bit(s) left of the | symbol are to the left of the radix point.

```
sum = 0.0;
correction = 0.0;
u = sum;
    //u ← 0.0
v = a + correction;
    //v ← 0 110 0100
upt = u + v;
    //upt ← 0 110 0100
up = upt - v;
    //up ← 0.0
vpp = upt - up;
    //vpp ← 0 110 0100
sum = upt;
    //sum ← 0 110 0100
correction = (u - up) + (v - vpp);
    //correction ← 0.0
```

```
u = sum;
    //u ← 0 100 0100
v = b + correction;
    //v ← 0 110 0101
upt = u + v;
    //upt ←
```

$$\begin{array}{rl} & 0\,110\,0010 \\ + & 0\,110\,0101 \\ \hline & 0\,110\,1|0100 \\ + & 0\,110\,1|0101 \\ \hline & 0\,110\,10|1001 \\ \hline & 0\,111\,1|01001 \\ \hline & 0\,111\,0100 \end{array}$$

```
up = upt - v;
    //up ←
```

$$\begin{array}{rl} & \text{0 111 0100} \\ - & \text{0 110 0101} \\ \hline & \text{0 111 1}|\text{01000} \\ - & \text{0 111 0}|\text{10101} \\ \hline & \text{0 111 0}|\text{10011} \\ \hline & \text{0 110 1}|\text{0011} \end{array}$$

vpp = upt - up;

    // ←

$$\begin{array}{rl} & \text{0 111 0100} \\ - & \text{0 110 0011} \\ \hline & \text{0 111 1}|\text{0100} \\ - & \text{0 111 0}|\text{10011} \\ \hline & \text{0 111 0}|\text{10101} \\ \hline & \text{0 110 1}|\text{0101} \end{array}$$

sum = upt;

    //sum ← 0 111 0100

correction = (u - up) + (v - vpp);

    //u - up ←

$$\begin{array}{rl} & \text{0 110 0100} \\ - & \text{0 110 0011} \\ \hline & \text{0 110 1}|\text{0100} \\ - & \text{0 110 1}|\text{0011} \\ \hline & \text{0 110 0}|\text{0001} \\ \hline & \text{0 010 1}|\text{0000} \\ \hline & \text{0 010 0000} \end{array}$$

    //v - vpp ←

$$\begin{array}{rl} & \text{0 110 0101} \\ - & \text{0 110 0101} \\ \hline & \text{0.0} \end{array}$$

    //correction ← 0 010 0000

---

u = sum;

    //u ← 0 111 0100

v = c + correction;

    //v ←

$$\begin{array}{rl} & 0\;001\;0001 \\ + & 0\;010\;0000 \\ \hline & 0\;010\;0|10001 \\ + & 0\;010\;1|0000 \\ \hline & 0\;010\;1|10001 \\ \hline & 0\;010\;1000 \end{array}$$

```
upt = u + v;
    //upt ←
```

$$\begin{array}{rl} & 0\;111\;0100 \\ + & 0\;010\;1000 \\ \hline & 0\;111\;1|0100 \\ + & 0\;111\;0|000011 \\ \hline & 0\;111\;1|0101 \end{array}$$

```
up = upt - v;
    //up ←
```

$$\begin{array}{rl} & 0\;111\;0101 \\ - & 0\;010\;1000 \\ \hline & 0\;111\;1|0101 \\ - & 0\;111\;0|000011 \\ \hline & 0\;111\;0|010001 \\ \hline & 0\;101\;1|0001 \end{array}$$

```
vpp = upt - up;
    //vpp ←
```

$$\begin{array}{rl} & 0\;111\;1|0101 \\ - & 0\;101\;1|0001 \\ \hline & 0\;111\;1|0101 \\ - & 0\;111\;0|010001 \\ \hline & 0\;111\;1|000011 \\ \hline & 0\;111\;1|0001 \end{array}$$

```
sum = upt;
    //sum ← 0 111 0101
correction = (u - up) + (v - vpp);
    //correction ← do not need to compute since there are no more iterations
```

Now we compute (b + c) + a = a + (b + c) by the commutative property of floating point addition.

```
sum = 0.0;
```

```
correction = 0.0;
u = sum;
    //u ← 0.0
v = b + correction;
    //v ← 0 110 0101
upt = u + v;
    //upt ← 0 110 0101
up = upt - v;
    //up ← 0.0
vpp = upt - up;
    //vpp ← 0 110 0101
sum = upt;      //sum ← 0 110 0101
correction = (u - up) + (v - vpp);
    //correction ← 0.0
```

```
u = sum;
    //u ← 0 110 0101
v = c + correction;
    //v ← 0 001 0001
upt = u + v;
    //upt ←
                  0 110 1|0101
             +    0 001 1|0001
             ---------------------
                  0 110 1|0101
             +    0 110 0|000010001
             ---------------------
                  0 110 1|010110001
             ---------------------
                  0 110 1|0110
up = upt - v;
    //up ←
                  0 110 1|0110
             −    0 001 1|0001
             ---------------------
                  0 110 1|0110
             −    0 110 0|000010001
             ---------------------
                  0 110 1|010101111
             ---------------------
                  0 110 1|0101
             ---------------------
                  0 110 0101
vpp = upt - up;
    //vpp ←
```

$$\begin{array}{cl}
 & 0\ 110\ 1|0110 \\
- & 0\ 110\ 1|0101 \\
\hline
 & 0\ 110\ 0|0001 \\
\hline
 & 0\ 010\ 1|0000 \\
\hline
 & 0\ 010\ 0000
\end{array}$$

sum = upt;

    //sum ← 0 110 0110

correction = (u - up) + (v - vpp);

    //u - up ←

$$\begin{array}{cl}
 & 0\ 110\ 0101 \\
- & 0\ 110\ 0101 \\
\hline
 & 0.0
\end{array}$$

    //v - vpp ←

$$\begin{array}{cl}
 & 0\ 001\ 1|0001 \\
- & 0\ 010\ 1|0000 \\
\hline
 & 0\ 010\ 0|10001 \\
- & 0\ 010\ 1|0000 \\
\hline
 & 1\ 010\ 0|01111 \\
\hline
 & 1\ 000\ 1|1110
\end{array}$$

    //correction ← 1 000 1110

---

u = sum;

    //u ← 0 110 0110

v = a + correction;

    //v ←

$$\begin{array}{cl}
 & 0\ 110\ 0100 \\
+ & 1\ 000\ 1110 \\
\hline
 & 0\ 110\ 1|0100 \\
- & 0\ 000\ 1|1110 \\
\hline
 & 0\ 110\ 1|0100 \\
- & 0\ 110\ 0|000001111 \\
\hline
 & 0\ 110\ 1|001110001 \\
\hline
 & 0\ 110\ 1|0100
\end{array}$$

upt = u + v;

    //upt ←

$$\begin{array}{cl} & 0\ 110\ 1|0110 \\ + & 0\ 110\ 1|0100 \\ \hline & 0\ 110\ 10|1010 \\ \hline & 0\ 111\ 1|0101 \\ \hline & 0\ 111\ 0101 \end{array}$$

up = upt - v;

    //up ←

$$\begin{array}{cl} & 0\ 111\ 1|0101 \\ - & 1\ 110\ 1|0100 \\ \hline & 0\ 111\ 1|0101 \\ - & 0\ 111\ 0|10100 \\ \hline & 0\ 111\ 0|1011 \\ \hline & 0\ 110\ 1|0110 \end{array}$$

vpp = upt - up;

    //vpp ←

$$\begin{array}{cl} & 0\ 111\ 0101 \\ - & 0\ 110\ 0110 \\ \hline & 0\ 111\ 1|0101 \\ - & 0\ 111\ 0|10110 \\ \hline & 0\ 111\ 0|10100 \\ \hline & 0\ 110\ 1|0100 \end{array}$$

sum = upt;

    //sum ← 0 111 0101

correction = (u - up) + (v - vpp);

    //correction ← do not need to compute since there are no more iterations

Thus, $(a + b) + c = a + (b + c) = 0\ 111\ 0101 = 21$ when computed using the Knuth summation algorithm.

4.5 The Experiment

To understand how well each of these methods improves the reproducibility of the dot product, we wrote a test program, called TestDotProductOMP-Double.cpp, that assigns random floating point values in the range $[0.0, 1.0)$ to two arrays of a user-specified length, and then computes both a serial dot product and a parallel dot product. Since the serial result is reproducible, we only need to compute it once. To see variation in the parallel dot product, however, the user may specify a number of times to compute the dot product in parallel. TestDotProductOMP.cpp then computes the parallel dot product this number of times, and prints the difference between the serial result and the parallel result for each parallel computation.

This test program was then modified so that the parallel results were computed using the Kahan and Knuth methods, respectively. To implement each method in parallel, the algorithms for addition were applied both for accumulating local sums and as the combiner function for combining the local sums into the global sum. Our first implementations of the Kahan and Knuth dot products were done using double-precision floating point numbers. However, we decided to do a single-precision version of each method, found in TestDotProductOMPFloat.cpp, TestDotProductKahanOMPFloat.cpp, and TestDotProductKnuthOMPFloat.cpp. We then implemented versions of each method in which the values in the two dot product vectors x and y are single-precision floating point numbers and the accumulators (the variables used for summation) are double-precision floating point numbers. The idea is that having a larger accumulator provides extra precision to the summation for a more accurate, and thus more reproducible result. These implementations are found in source code files TestDotProductOMPDoubleAccumulator.cpp, TestDotProductKahanOMPDoubleAccumulator.cpp, and TestDotProductKnuthOMPDoubleAccumulator.cpp.

We decided to do 100 parallel dot product computations for each test, and to run our tests on a variety of problem sizes (10000, 100000, and 1000000) and numbers of parallel threads (1, 2, 4, 6, and 12). Fifteen tests were run for each method, one for each combination of our three problem sizes and five thread sizes.

5 Results

5.1 Double-Precision Results

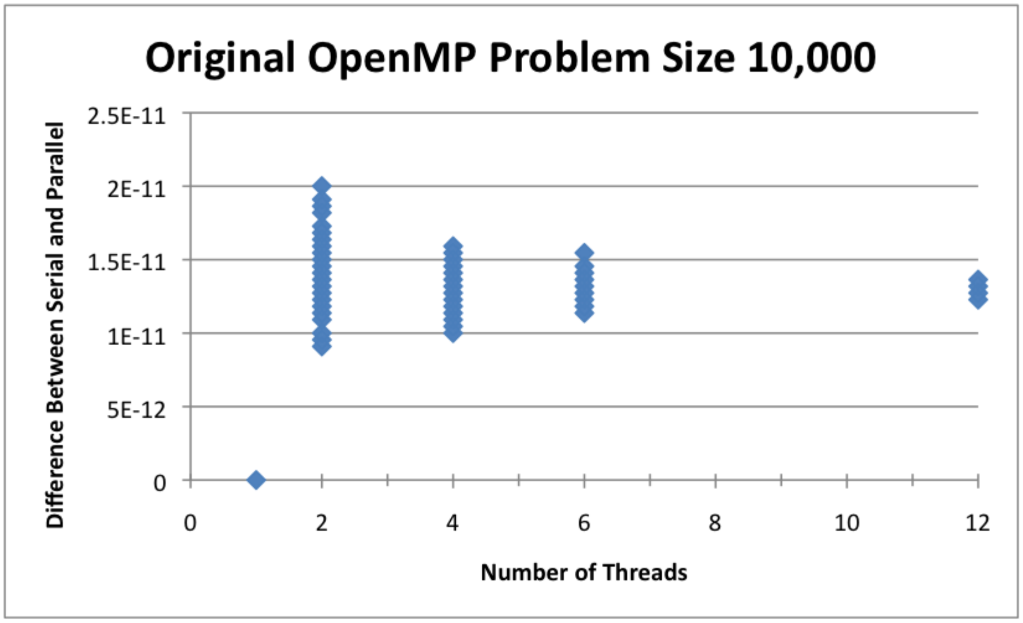

Figure 1: A scatterplot of 100 runs of the original double-precision test program, TestDotProductOMPDouble.cpp, with problem size 10,000 for each number of threads (1, 2, 4, 6, and 12). This plot shows the range of variation in results for each number of threads before using any technique for achieving reproducibility.

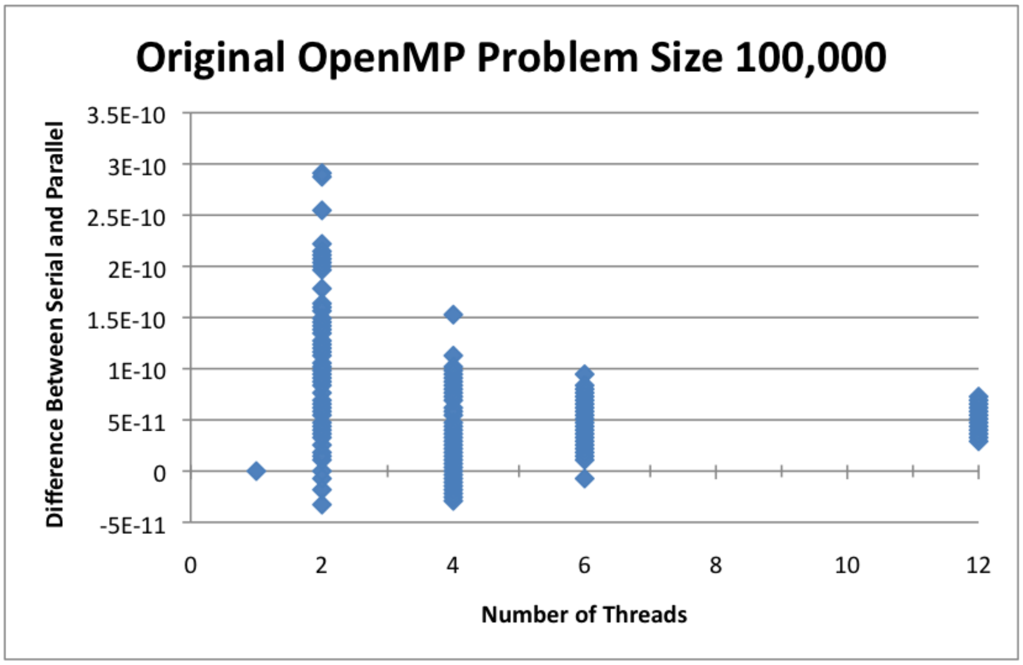

Figure 2: A scatterplot of 100 runs of the original double-precision test program, TestDotProductOMPDouble.cpp, with problem size 100,000 for each number of threads (1, 2, 4, 6, and 12). This plot shows the range of variation in results for each number of threads before using any technique for achieving reproducibility.

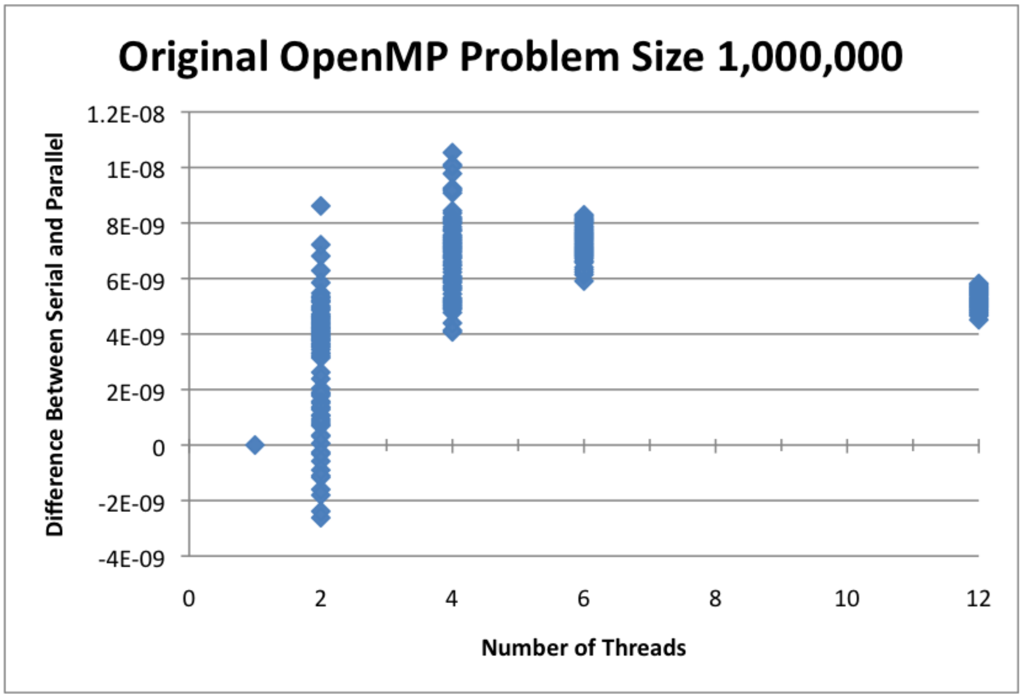

Figure 3: A scatterplot of 100 runs of the original double-precision program, TestDotProductOMPDouble.cpp, with problem size 1,000,000 for each number of threads (1, 2, 4, 6, and 12). This plot shows the range of variation in results for each number of threads before using any technique for achieving reproducibility.

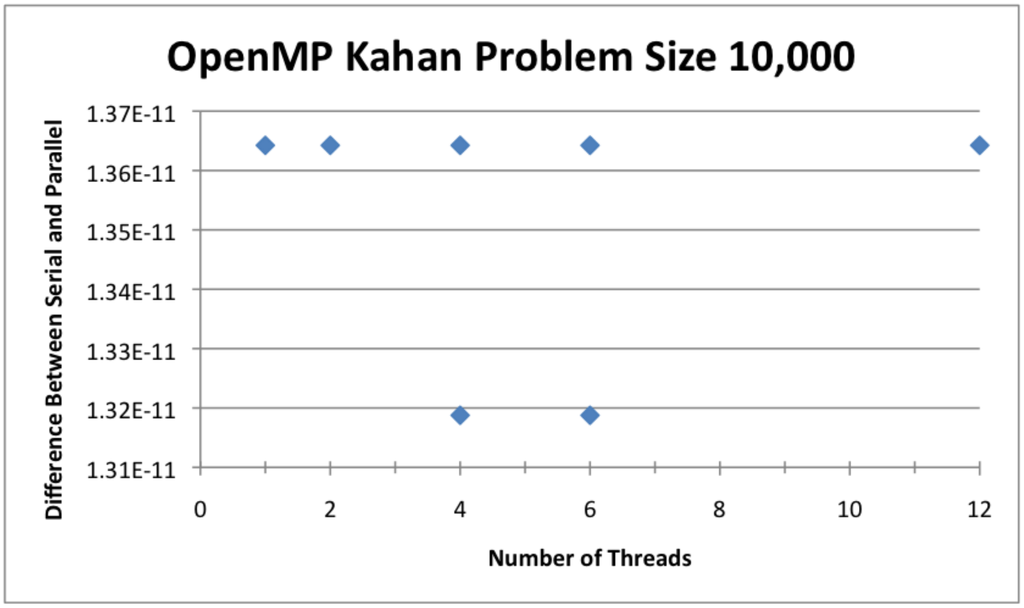

Figure 4: A scatterplot of 100 runs of TestDotProductKahanOMPDouble.cpp, the double-precision implementation using the Kahan method, with problem size 10,000 for each number of threads (1, 2, 4, 6, and 12). This plot shows the range of variation in results for each number of threads.

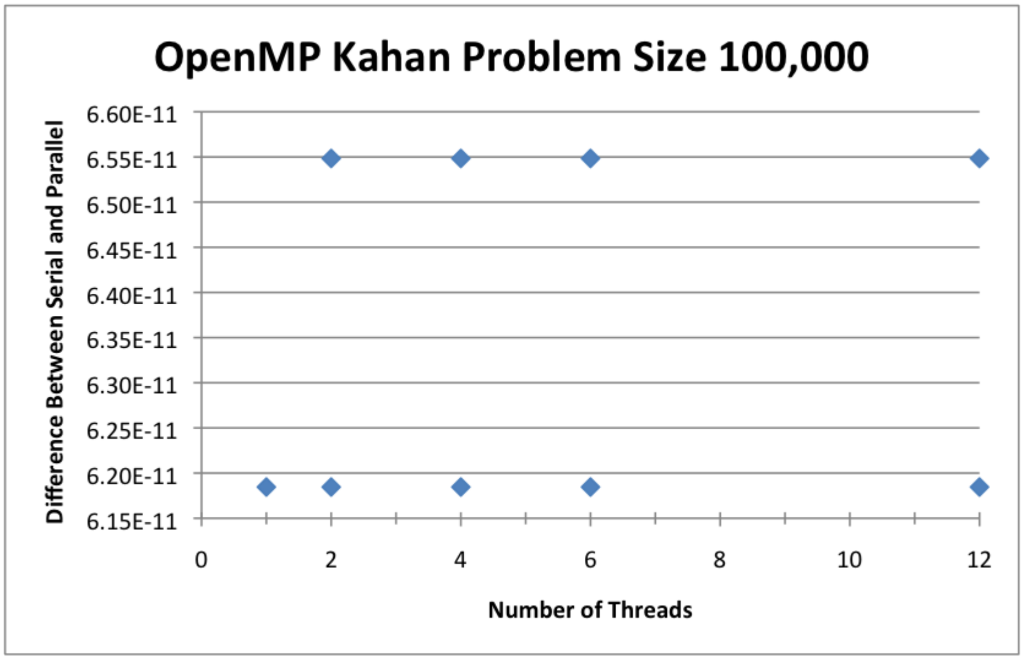

Figure 5: A scatterplot of 100 runs of TestDotProductKahanOMPDouble.cpp, the double-precision implementation using the Kahan method, with problem size 100,000 for each number of threads (1, 2, 4, 6, and 12). This plot shows the range of variation in results for each number of threads.

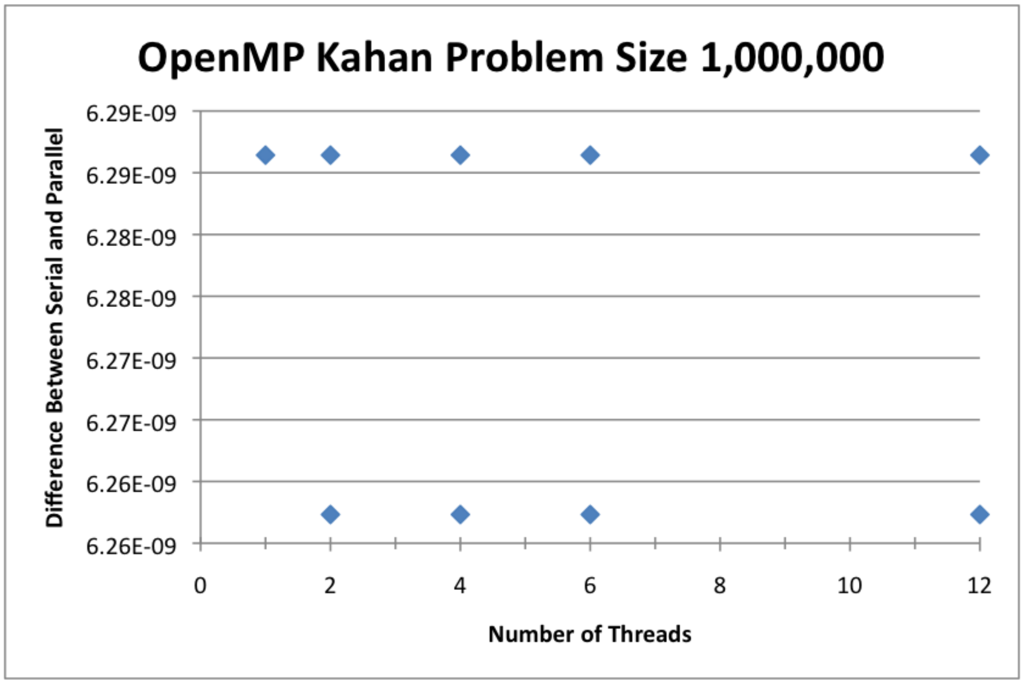

Figure 6: A scatterplot of 100 runs of TestDotProductKahanOMPDouble.cpp, the double-precision implementation using the Kahan method, with problem size 1,000,000 for each number of threads (1, 2, 4, 6, and 12). This plot shows the range of variation in results for each number of threads.

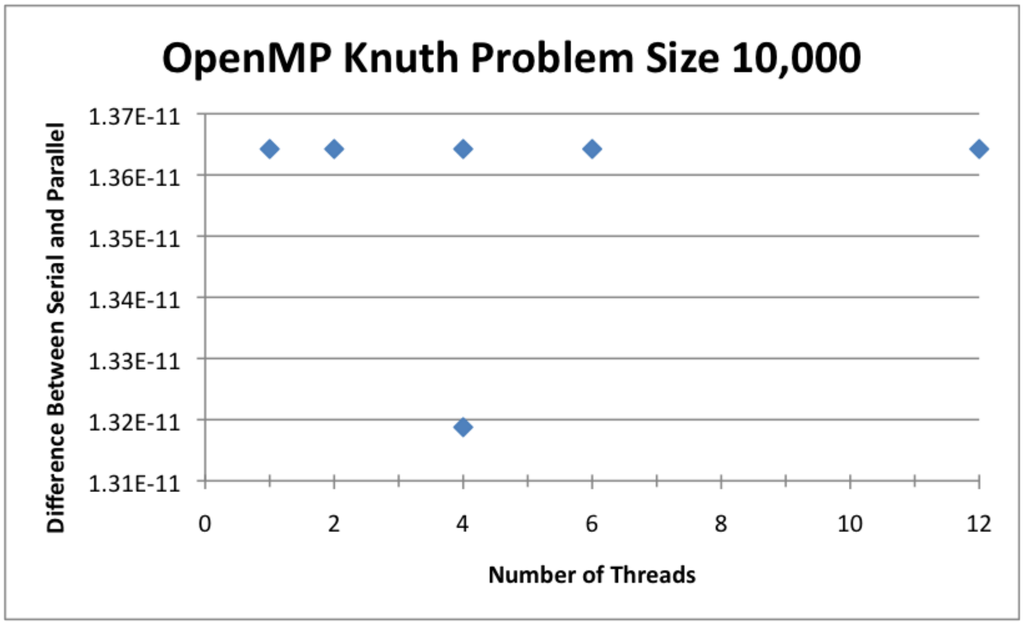

Figure 7: A scatterplot of 100 runs of TestDotProductKnuthOMPDouble.cpp, the double-precision implementation using the Knuth method, with problem size 10,000 for each number of threads (1, 2, 4, 6, and 12). This plot shows the range of variation in results for each number of threads.

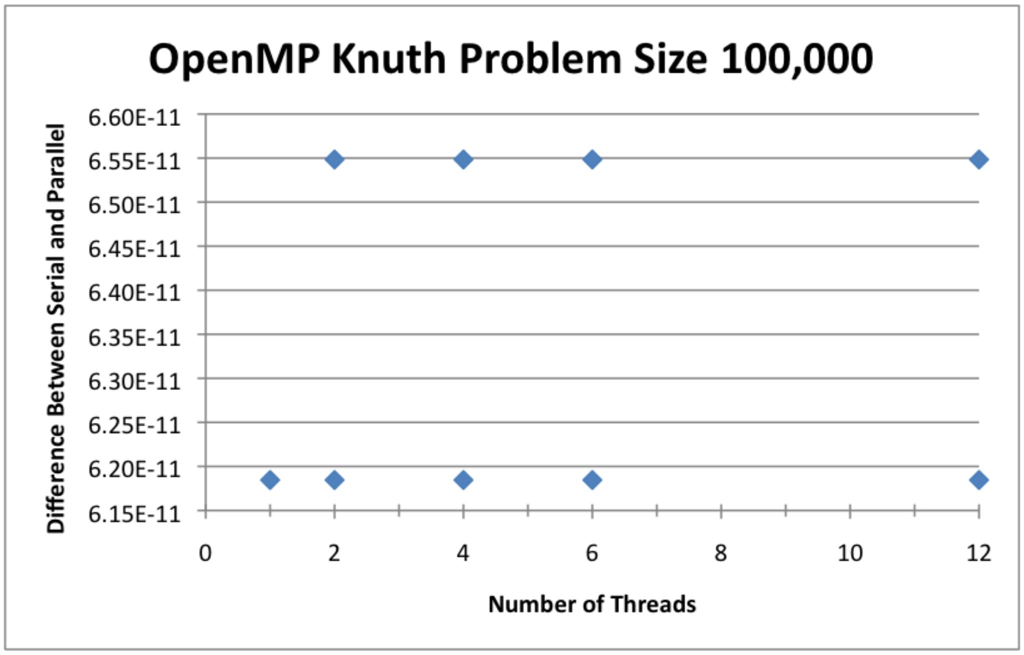

Figure 8: A scatterplot of 100 runs of TestDotProductKnuthOMPDouble.cpp, the double-precision implementation using the Knuth method, with problem size 100,000 for each number of threads (1, 2, 4, 6, and 12). This plot shows the range of variation in results for each number of threads.

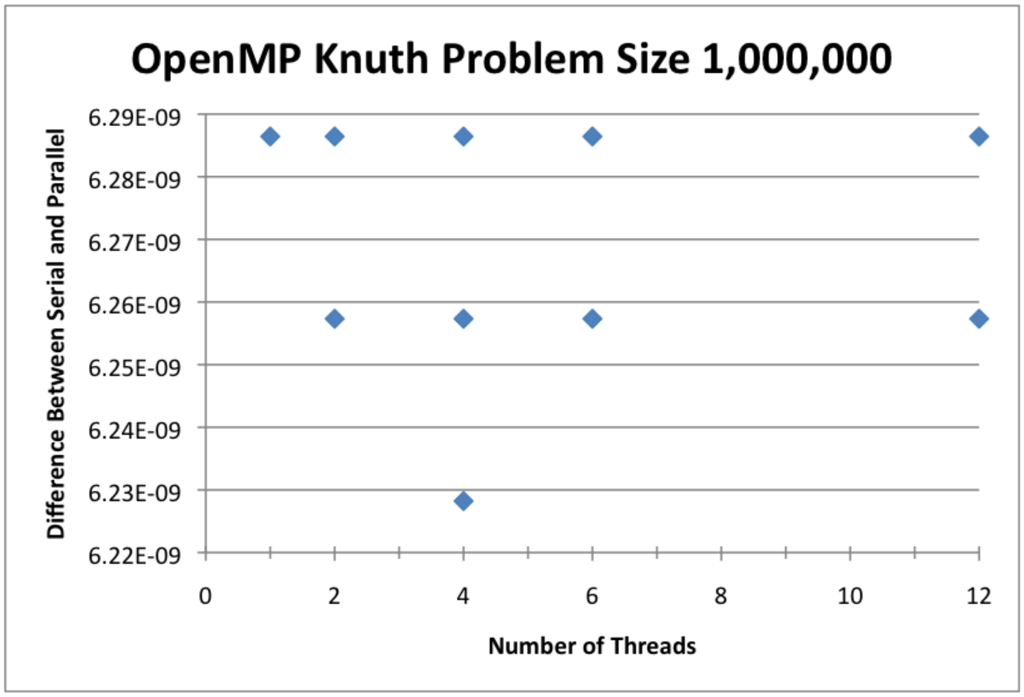

Figure 9: A scatterplot of 100 runs of TestDotProductKnuthOMPDouble.cpp, the double-precision implementation using the Knuth method, with problem size 1,000,000 for each number of threads (1, 2, 4, 6, and 12). This plot shows the range of variation in results for each number of threads.

Table 1: Double-Precision Non-Reproducible Number of Unique Values Over 100 Runs

Number of Threads	Problem Size 10,000	Problem Size 100,000	Problem Size 1,000,000
1	1	1	1
2	22	50	77
4	14	27	75
6	9	23	51
12	4	13	37

Table 2: Double-Precision Kahan Number of Unique Values Over 100 Runs

Number of Threads	Problem Size 10,000	Problem Size 100,000	Problem Size 1,000,000
1	1	1	1
2	1	2	2
4	2	2	2
6	2	2	2
12	1	2	2

Table 3: Double-Precision Knuth Number of Unique Values Over 100 Runs

Number of Threads	Problem Size 10,000	Problem Size 100,000	Problem Size 1,000,000
1	1	1	1
2	1	2	2
4	2	2	3
6	1	2	2
12	1	2	2

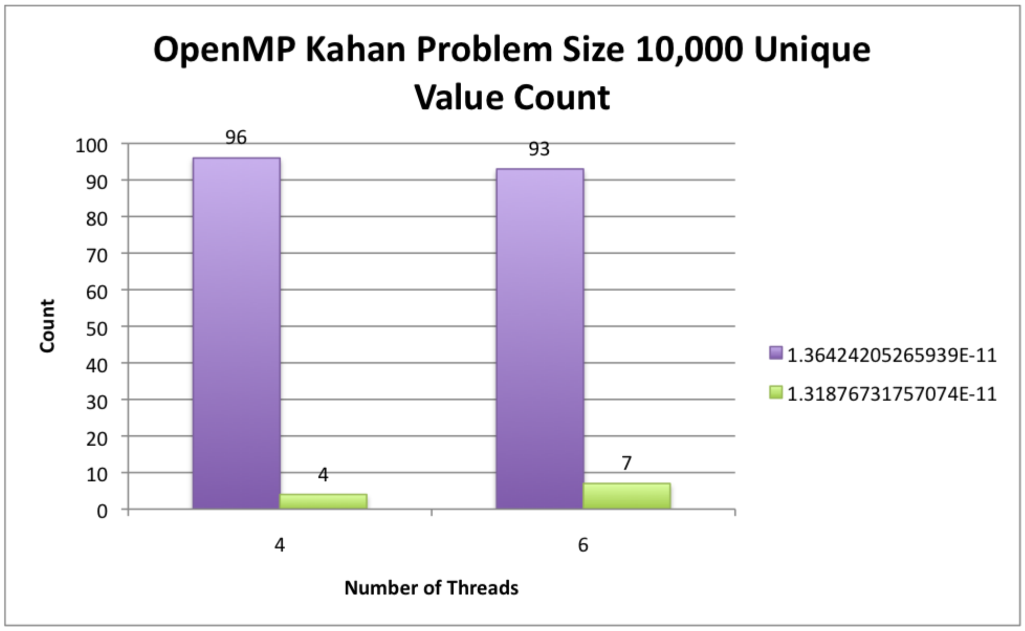

Figure 10: The number of times each unique value occurred over 100 runs for each number of threads for the double-precision implementation of the Kahan method given problem size 10,000. Tests for 1, 2, and 12 threads did not show any variation over all 100 runs, so these results are not shown.

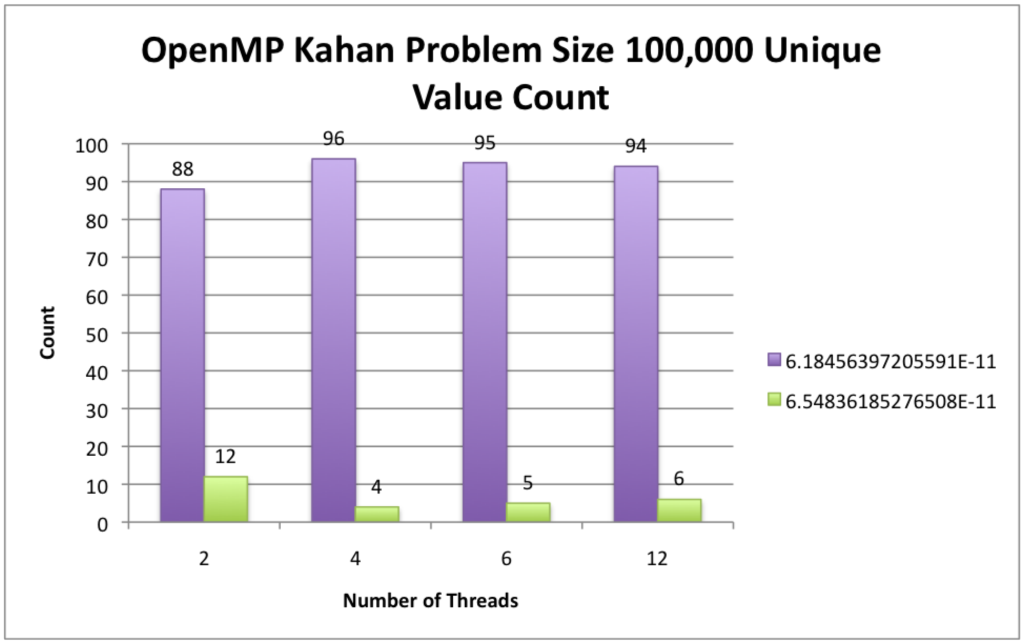

Figure 11: The number of times each unique value occurred over 100 runs for each number of threads for the double-precision implementation of the Kahan method given problem size 100,000. Test for 1 thread will not show any variation over all 100 runs since it is serial, so results for 1 thread are not shown.

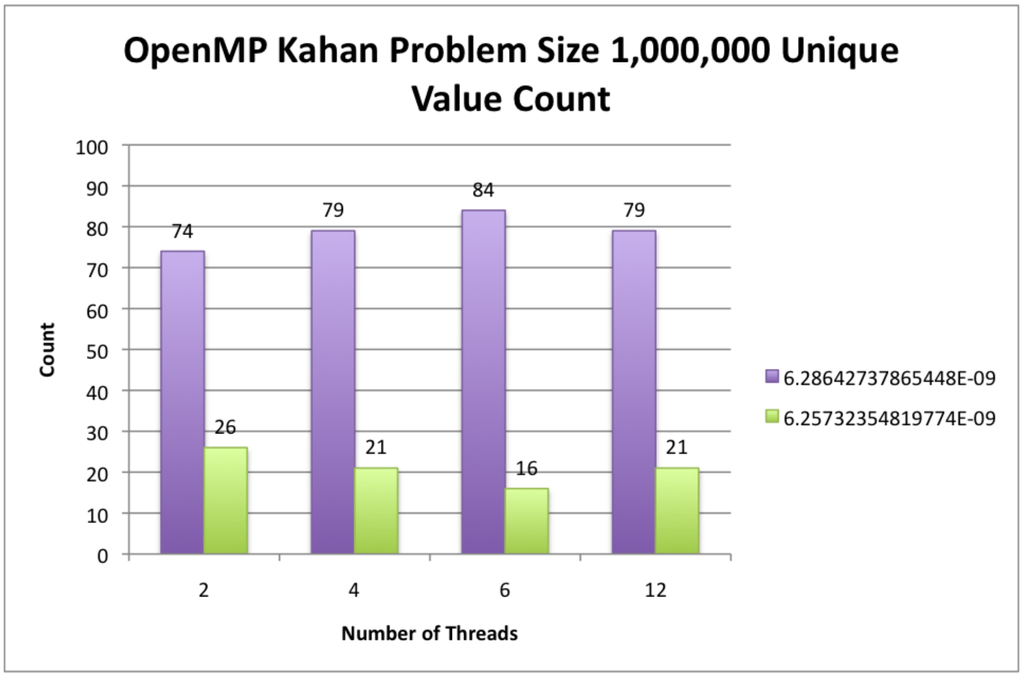

Figure 12: The number of times each unique value occurred over 100 runs for each number of threads for the double-precision implementation of the Kahan method given problem size 1,000,000. Test for 1 thread will not show any variation over all 100 runs since it is serial, so results for 1 thread are not shown.

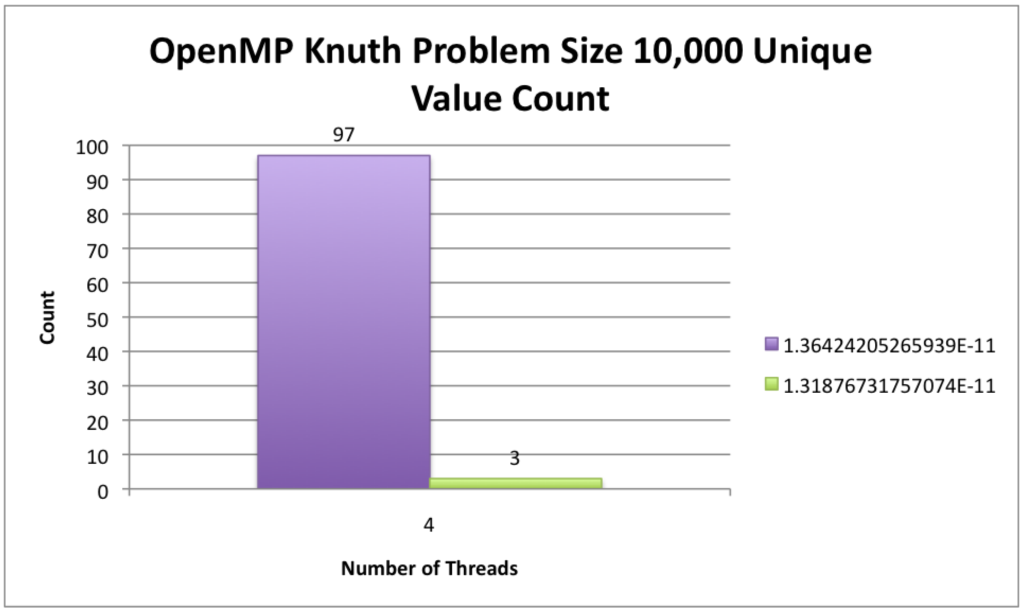

Figure 13: The number of times each unique value occurred over 100 runs for each number of threads for the double-precision implementation of the Knuth method given problem size 10,000. Tests for 1, 2, 6, and 12 threads did not show any variation over all 100 runs, so these results are not shown.

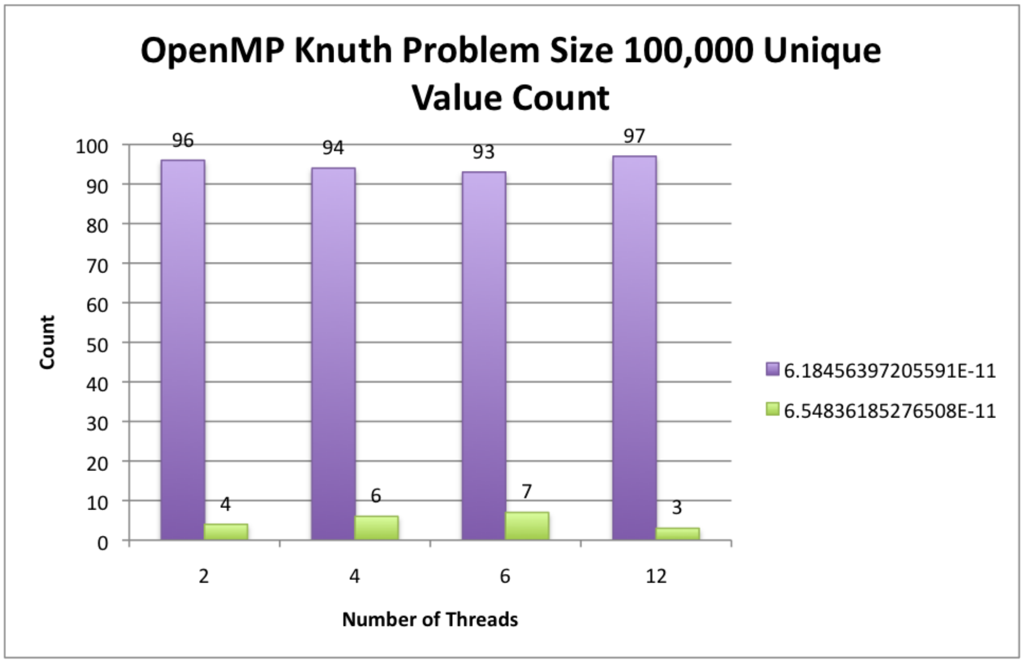

Figure 14: The number of times each unique value occurred over 100 runs for each number of threads for the double-precision implementation of the Knuth method given problem size 100,000. Test for 1 thread will not show any variation over all 100 runs since it is serial, so results for 1 thread are not shown.

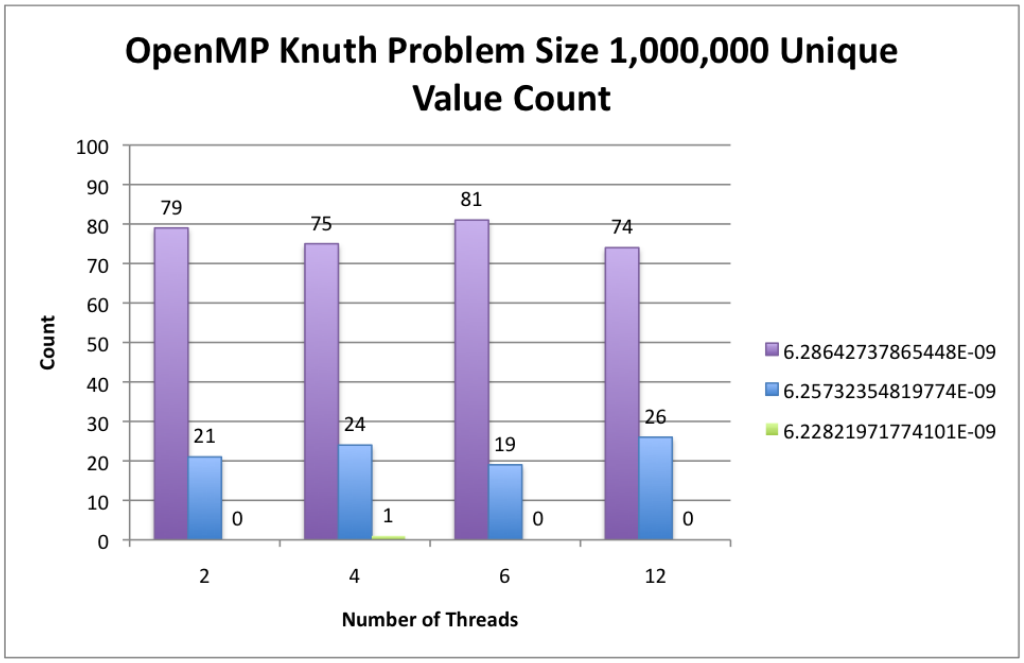

Figure 15: The number of times each unique value occurred over 100 runs for each number of threads for the double-precision implementation of the Knuth method given problem size 1,000,000. Test for 1 thread will not show any variation over all 100 runs since it is serial, so results for 1 thread are not shown.

5.2 Single-Precision Results

Table 4: Single-Precision Non-Reproducible Number of Unique Values Over 100 Runs

Number of Threads	Problem Size 10,000	Problem Size 100,000	Problem Size 1,000,000
1	1	1	1
2	26	47	74
4	16	45	79
6	10	19	55
12	6	14	34

Table 5: Single-Precision Kahan Number of Unique Values Over 100 Runs

Number of Threads	Problem Size 10,000	Problem Size 100,000	Problem Size 1,000,000
1	1	1	1
2	2	2	3
4	2	2	3
6	2	1	3
12	2	1	2

Table 6: Single-Precision Knuth Number of Unique Values Over 100 Runs

Number of Threads	Problem Size 10,000	Problem Size 100,000	Problem Size 1,000,000
1	1	1	1
2	2	1	3
4	2	1	2
6	2	2	2
12	2	1	1

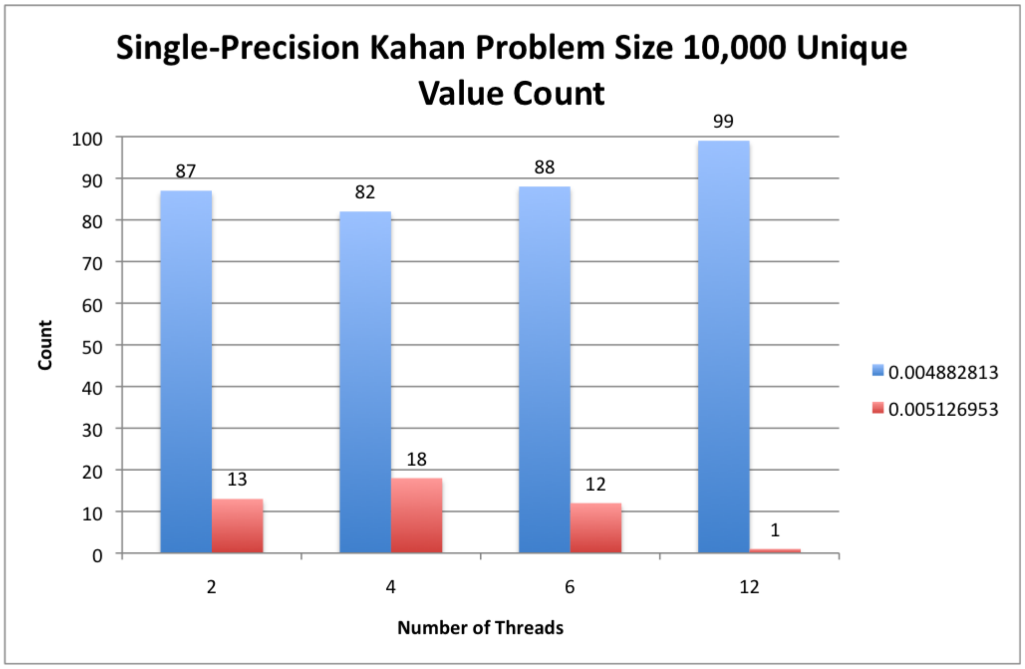

Figure 16: The number of times each unique value occurred over 100 runs for each number of threads for the single-precision implementation of the Kahan method given problem size 10,000. Test for 1 thread will not show any variation over all 100 runs since it is serial, so results for 1 thread are not shown.

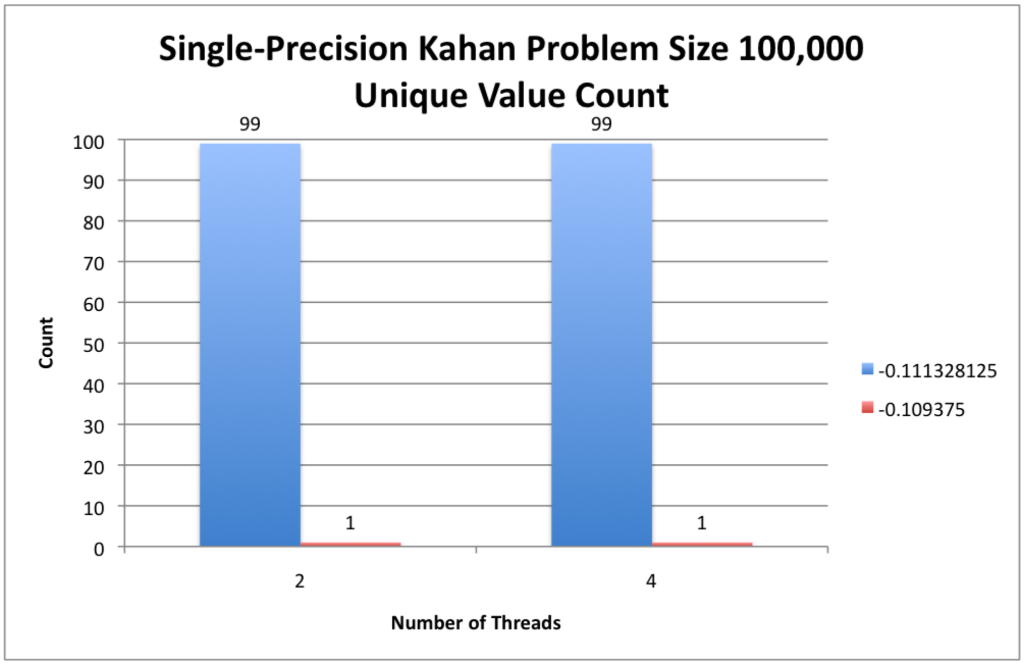

Figure 17: The number of times each unique value occurred over 100 runs for each number of threads for the single-precision implementation of the Kahan method given problem size 100,000. Tests for 1, 6, and 12 threads did not show any variation over all 100 runs, so these results are not shown.

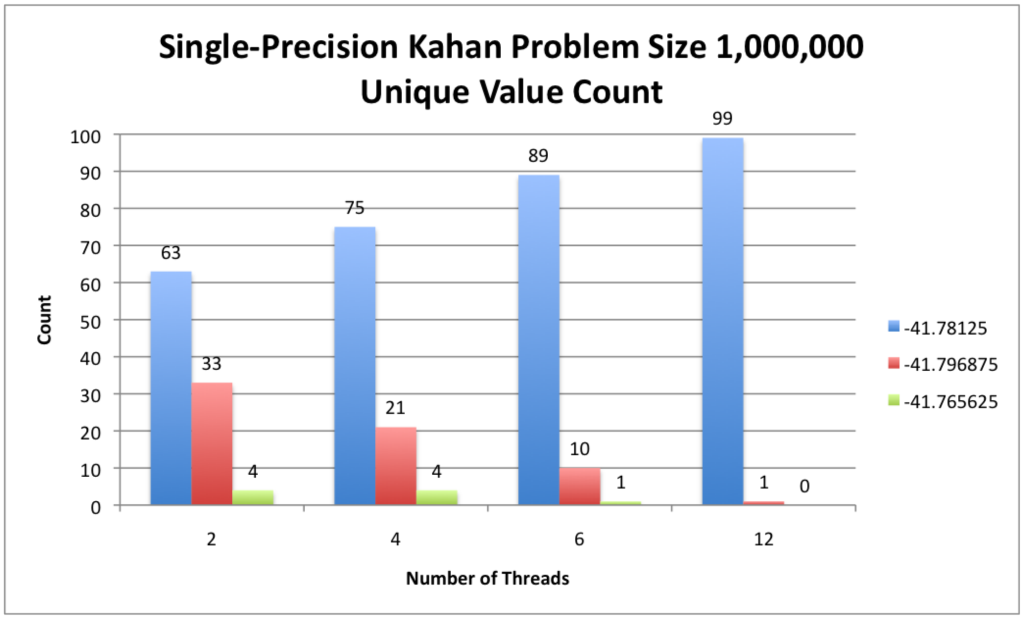

Figure 18: The number of times each unique value occurred over 100 runs for each number of threads for the single-precision implementation of the Kahan method given problem size 1,000,000. Test for 1 thread will not show any variation over all 100 runs since it is serial, so results for 1 thread are not shown.

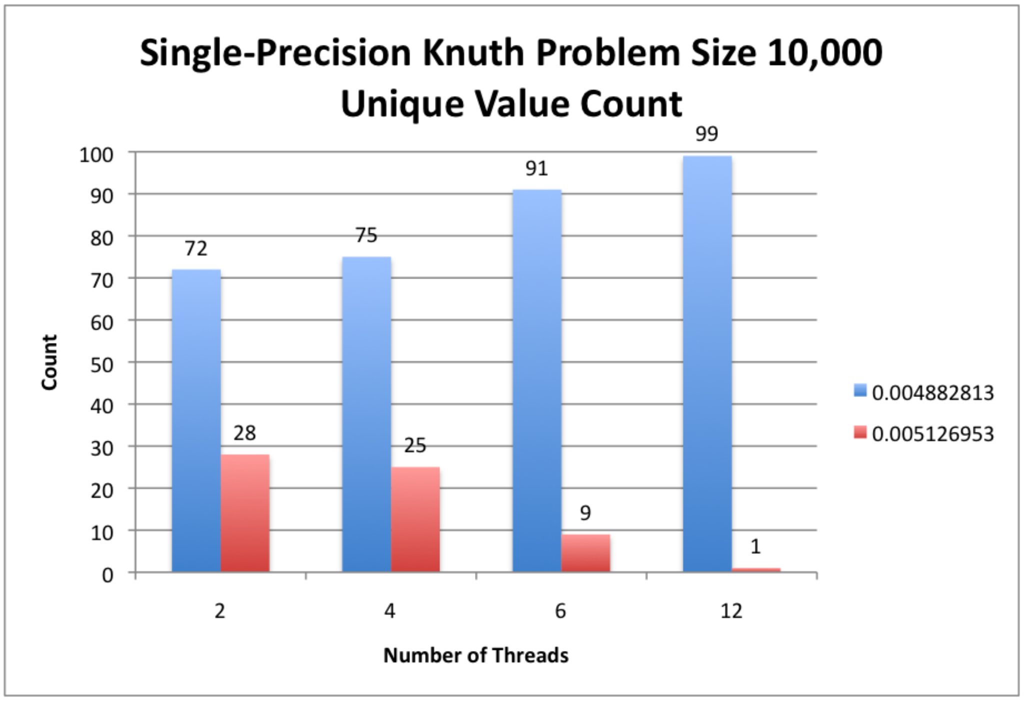

Figure 19: The number of times each unique value occurred over 100 runs for each number of threads for the single-precision implementation of the Knuth method given problem size 10,000. Test for 1 thread will not show any variation over all 100 runs since it is serial, so results for 1 thread are not shown.

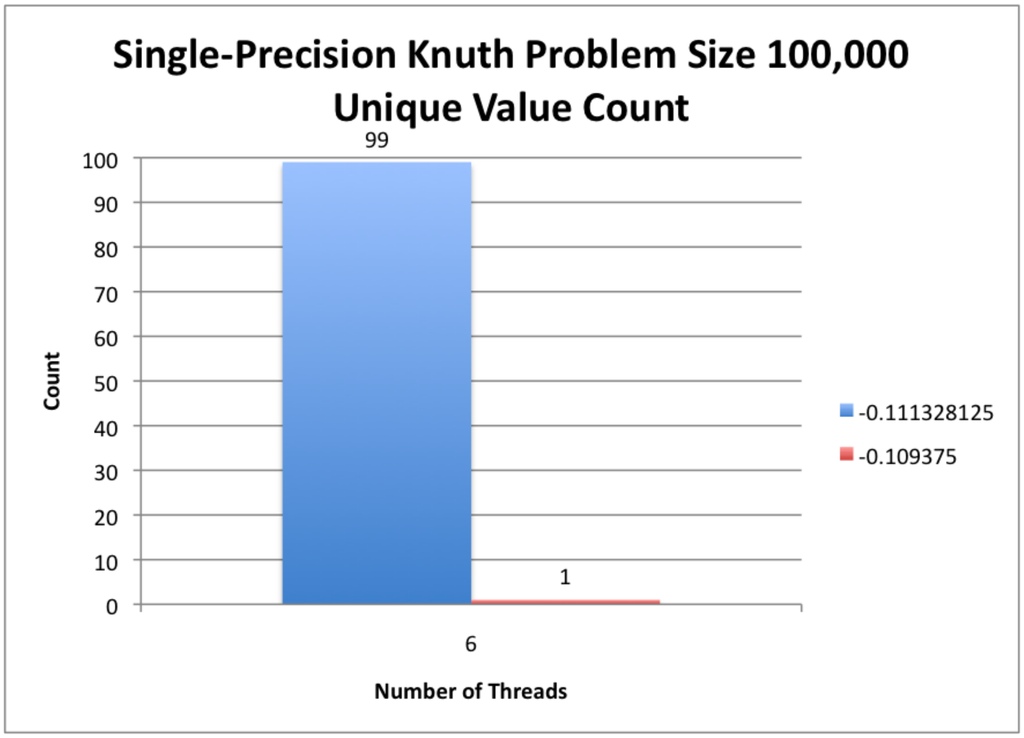

Figure 20: The number of times each unique value occurred over 100 runs for each number of threads for the single-precision implementation of the Knuth method given problem size 100,000. Tests for 1, 2, 4, and 12 threads did not show any variation over all 100 runs, so these results are not shown.

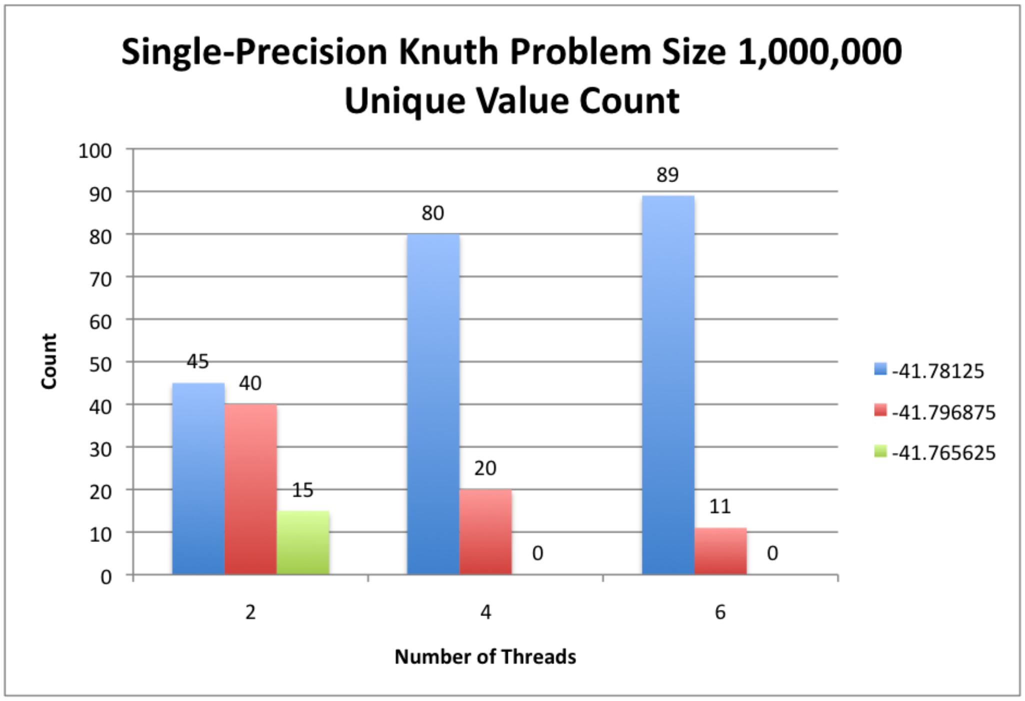

Figure 21: The number of times each unique value occurred over 100 runs for each number of threads for the single-precision implementation of the Knuth method given problem size 1,000,000. Tests for 1 and 12 threads did not show any variation over all 100 runs, so these results are not shown.

5.3 Single-Precision with Double-Precision Accumulator Results

Table 7: Single-Precision with Double Accumulator Non-Reproducible Number of Unique Values Over 100 Runs

Number of Threads	Problem Size 10,000	Problem Size 100,000	Problem Size 1,000,000
1	1	1	1
2	1	4	27
4	1	3	11
6	1	3	6
12	1	4	5

Table 8: Single-Precision with Double Accumulator Kahan Number of Unique Values Over 100 Runs

Number of Threads	Problem Size 10,000	Problem Size 100,000	Problem Size 1,000,000
1	1	1	1
2	1	2	1
4	1	2	3
6	1	2	1
12	1	1	1

Table 9: Single-Precision with Double Accumulator Knuth Number of Unique Values Over 100 Runs

Number of Threads	Problem Size 10,000	Problem Size 100,000	Problem Size 1,000,000
1	1	1	1
2	1	2	1
4	1	2	2
6	1	2	1
12	1	1	1

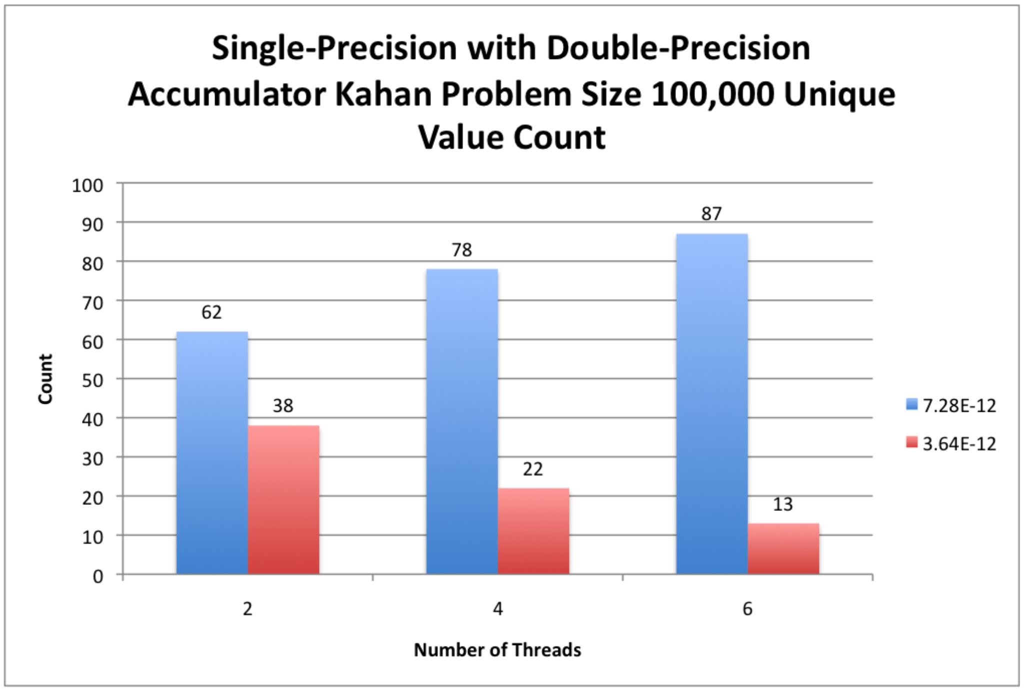

Figure 22: The number of times each unique value occurred over 100 runs for each number of threads for the single-precision with double precision accumulator implementation of the Kahan method given problem size 100,000. Tests for 1 and 12 threads did not show any variation over all 100 runs, so these results are not shown.

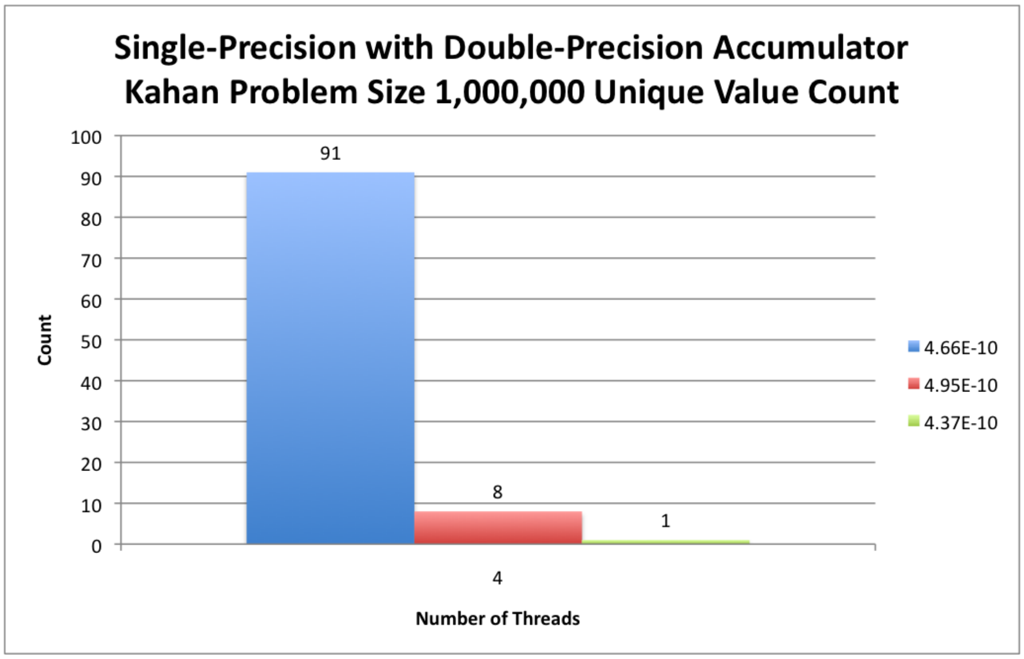

Figure 23: The number of times each unique value occurred over 100 runs for each number of threads for the single-precision with double precision accumulator implementation of the Kahan method given problem size 1,000,000. Tests for 1, 2, 6, and 12 threads did not show any variation over all 100 runs, so these results are not shown.

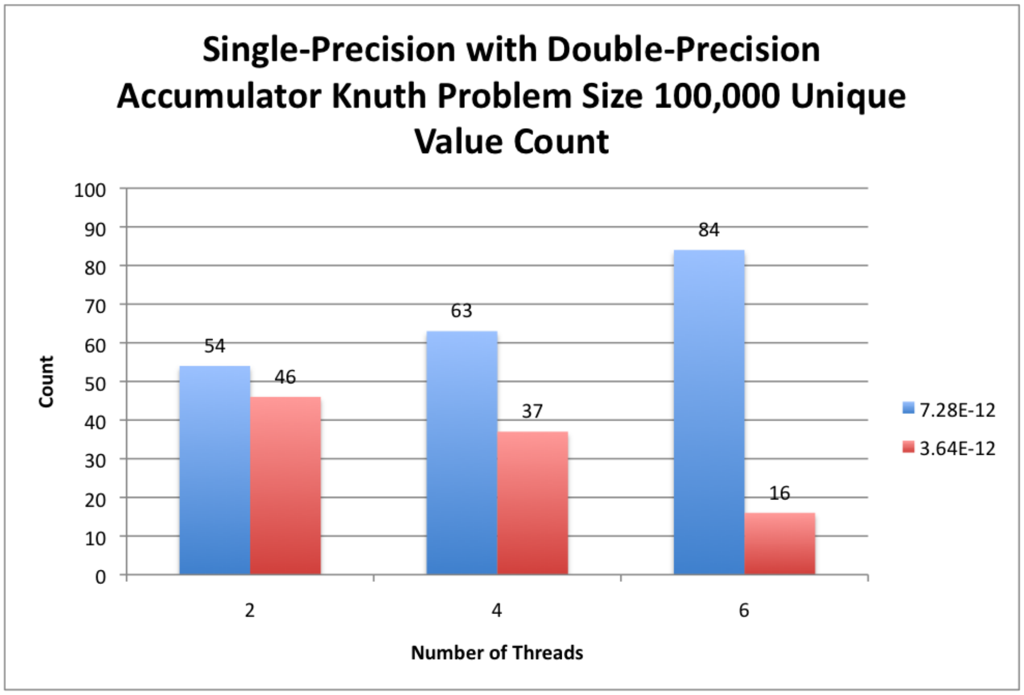

Figure 24: The number of times each unique value occurred over 100 runs for each number of threads for the single-precision with double precision accumulator implementation of the Knuth method given problem size 100,000. Tests for 1 and 12 threads did not show any variation over all 100 runs, so these results are not shown.

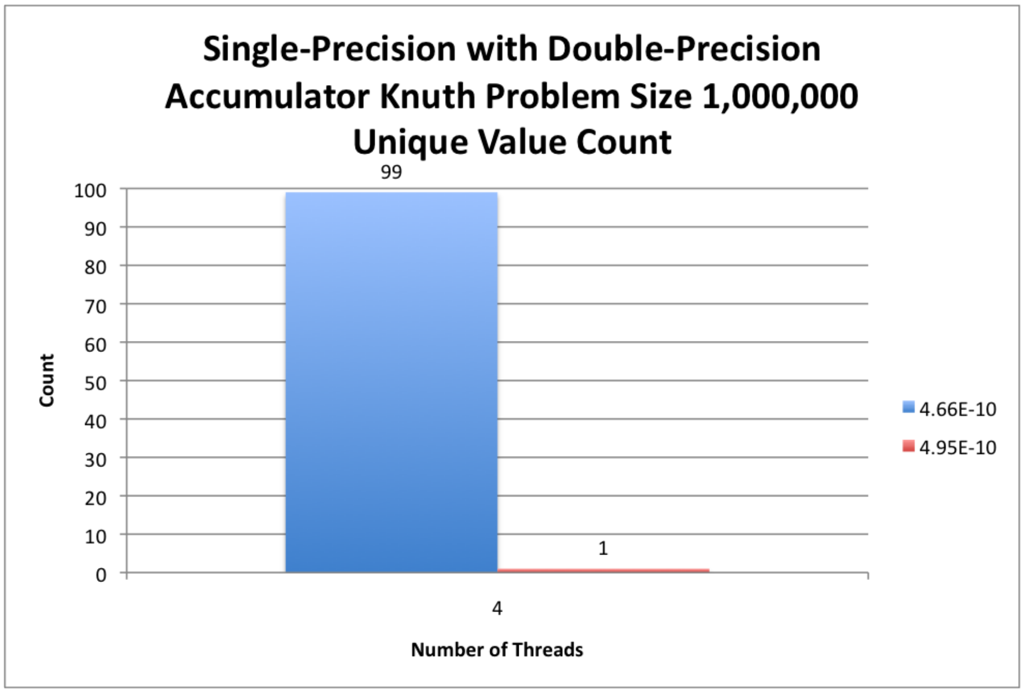

Figure 25: The number of times each unique value occurred over 100 runs for each number of threads for the single-precision with double precision accumulator implementation of the Knuth method given problem size 1,000,000. Tests for 1, 2, 6, and 12 threads did not show any variation over all 100 runs, so these results are not shown.

6 Conclusions

We want to achieve a bit-wise reproducible result over all 100 runs of a single implementation of the parallel dot product. For a very conservative probability bound, we can assume that we only need to guarantee that the last bit of a floating point number is reproducible. Then, as a worst-case probability we have a 50% chance that the last bit is correct on a given run. Over 100 runs, the probability of this bit being correct every run is then $(.5)^{100} = 7.88860905221 * 10^{-31}$ by the multiplicative property of the probability of independent events occurring.

We first consider the double-precision results. As expected, both methods significantly improved the reproducibility of the dot product. However, neither method was able to guarantee a reproducible result for most problem sizes and number of threads. The original results (before a reproducibility technique is applied) show a large amount of variation in the results, with as many as 77 different unique values over 100 runs for problem size 1,000,000 (see Tables 1-3). Thus, in the best worst case, the amount of reproducibilty in the dot product is as low as 33% for a problem size of 1,000,000. Note that the amount of non-reproducibility also grows as the problem size grows. However, as the number of threads increase, the amount of non-reproducibility often decreases or does not change.

After applying the Kahan and Knuth methods, we begin to see a bi- and tri-modal tendency for the results to vary between two or three distinct values. Figures 10-15 show us the exact number of times each distinct value appears over 100 runs for each thread size. All results for Kahan and Knuth for thread amounts that are not shown were 100% reproducible. For both the Kahan method, a problem size of 10,000 was at least 93% reproducible for each number of threads. In the Knuth method, a problem size of 10,000 was at least 97% reproducible for each number of threads. Problem size 100,000 was at least 88% reproducible for the Kahan method and at least 93% reproducible for the Knuth method. Problem size 1,000,000 was at least 74% reproducible for both methods. As we can see, the reproducibility of these methods decreases as problem size increases. While both sums are effective for small problem sizes, the Knuth sum is an improvement over the Kahan sum for small problem sizes. It is not surprising that the amount reproducibility gained in each method converges to a similar value for large problem sizes, however, because both methods are nearly identical for running summations, and thus the first few terms that are added are the only ones in which the Knuth sum has a more accurate correction term.

Although the Kahan and Knuth methods were effective in increasing reproducibility in the parallel dot product, we can conclude that they are not good methods if we desire a 100% reproducible result. Additionally, while we expect the Knuth to be an improvement over the Kahan method for its improved correction term, we do not see much difference in the reproducibility of the result. This is due to the fact that for running sums, the Knuth method is nearly identical to the Kahan method except for the first few terms in which the next summand may be larger than the accumulated sum.

We now move on to the single-precision and single-precision with double-precision accumulator results. We hoped that by doubling the size of the accumulator variables, we could show improvement in the reproducibility of the Kahan and Knuth methods from the single-precision implementations. The single-precision results show similar numbers of unique values to that of the double-precision results (see Tables 4-6). In fact, there is even more variation in results from double-precision, with the most variation before adding reproducibility methods to be 79 distinct values for a problem size of 1,000,000. Additionally, there are more tri-modal results when implementing the Kahan and Knuth methods than for the double-precision versions. Figures 16-21 show that for single-precision results using the Kahan and Knuth methods, the amount of reproducibility guaranteed was much lower on average, and increased with the number of threads. However, for a problem size of 100,000 both methods were at least 99% reproducible.

When the accumulator was doubled, the number of unique values decreased significantly overall, especially in Table 7, before the Kahan and Knuth methods were applied (see Tables 7-9). Additionally, problem sizes of 10,000 were 100% reproducible for every number of threads for both the Kahan and Knuth implementations. Problem size 1,000,000 was at least 91% reproducible using the Kahan method, and at least 99% reproducible using the Knuth method. However, the reproducibility decreased significantly for a problem size of 100,000 for numbers of threads less than 6 (see Figures 22 and 24). Thus, overall, doubling the size of the accumulator variables improved reproducibility only for smaller problem sizes.

7 Contributions to this Topic

This thesis has made several contributions to this field that have not previously been done. First, a thorough explanation of the Kahan and Knuth algorithms was provided. This is useful from a teaching perspective since the examples in our 8-bit floating point representation show hardware-level behavior of floating point numbers (and their non-associability) in a way that is easy to understand. Additionally, these examples show how the Kahan and Knuth methods attempt to retain correction terms in order to provide a more reproducible floating point summation. Another contribution from this thesis is in the results section scatterplots showing variation over 100 runs of each implementation of the floating point dot product. Previous research has shown error bound values for implementations of the Kahan and Knuth methods in scientific applications by showing a table of error values deduced from many runs. Our plots expose total non-reproducibility behavior over many runs by plotting values from each individual run. Thus, by viewing these plots, we can see how many distinct values are apparent over 100 runs of a single implementation (this gives us an idea of how reproducible each implementation is).

8 Future Work

Future topics for exploration include researching the affect of using different methods for achieving reproducibility on application performance. Another useful avenue is comparing the accuracy obtained by using Kahan and Knuth methods to that achieved by other reproducibility methods, such as pre-rounding or the use of a superaccumulator to get 100% accurate and reproducible results.[3] [1] One may also explore several more methods for reproducibility in an attempt to find the method with the most reproducible result while minimizing inaccuracy and run-time. Demmel and Nguyen have written several papers on numerical reproducibility and methods for achieving reproducible floating point summation while achieving good performance. [5] [6] [4] An interesting application to explore is the non-reproducibility of parallel Monte Carlo particle simulations due to non-reproducibility of communication between processors, introduced by Cleveland et al.[2]

9 Appendix

9.1 Source Code

9.1.1 TestDotProductOMPDouble.cpp

```
// TestDotProductKahanOMPDouble.cpp
//
// Routine to show reproducibility of the parallel dot
    product
// using summation for double-precision floating point
// numbers.
//
//g++ -fopenmp TestDotProductOMPDouble.cpp -o testDotProd
#include <fstream>
#include <iostream>
#include <cstdlib>
using std::endl;
using std::cout;
#include <omp.h>

#include <vector>

/*
 * Method to compute the dot product in parallel using
 * dynamic scheduling of threads.
 *
 * @param[in]  n the number of vector elements
 * @param[in]  x, y the input vectors
 * @param[out] result a float value, on exit will contain the
      result
 *
 * @return returns parallel dot product result
 */
double ComputeDotProductOMP(int n, double * x, double * y ) {
  double result = 0.0;
#pragma omp parallel for reduction (+:result) schedule(
    dynamic,1)
  for (int i=0; i<n; i++) result += x[i]*y[i];
  return result;
}

int main(int argc, char * argv[]) {

  if (argc!=3) {
    cout << "Usage: " << argv[0] << " length_of_vector" << "
       number_of_trials" << endl;
    exit(1);
  }

  int n = atoi(argv[1]);  //problem size
  int numtrials = atoi(argv[2]); // number of times to
     compute the parallel dot product

  cout << "Problem size = " << n << endl;
  cout << "Number of trials = " << numtrials << endl;

  int nthreads;
#pragma omp parallel
  nthreads = omp_get_num_threads();
  cout << "Number of threads = " << nthreads << endl;

  // Generate random values
  std::vector< double > x(n), y(n);
  for (int i=0; i<n; ++i) {
    x[i] = drand48();
    y[i] = drand48();
  }

  //Compute serial dot product
  double result = 0.0;
  for (int i=0; i<n; i++) result += x[i]*y[i];

  //Print difference between serial dot product and parallel
     dot products
  cout.precision(17);
  for (int i=0; i<numtrials; ++i)
    cout << result - ComputeDotProductOMP(n, &x[0], &y[0]) <<
        endl;

  return 0 ;
}
```

9.1.2 TestDotProductKahanOMPDouble.cpp

```
// TestDotProductKahanOMPDouble.cpp
//
// Routine to show reproducibility of the parallel dot
   product
```

```
// using Kahan summation for double-precision floating point
// numbers.
// g++ -fopenmp TestDotProductKahanOMPDouble.cpp -o
   testDotProd
#include <fstream>
#include <iostream>
#include <cstdlib>
using std::endl;
using std::cout;
#include <omp.h>

#include <vector>

//a type to hold accumulator sum and correction terms for
   local and global sums
struct esum_type{
  double sum;
  double correction;
};

/*
 * Method to compute the dot product in parallel using
 * Kahan summation and dynamic scheduling of threads.
 *
 * @param[in]  n the number of vector elements
 * @param[in]  x, y the input vectors
 * @param[out] result a float value, on exit will contain the
     result
 *
 * @return returns parallel dot product result
 */
double ComputeDotProductKahanOMP(int n, double * x, double *
   y ) {
  esum_type sum; //global accumulator
  sum.sum = 0.0;
  sum.correction = 0.0;
  double corrected_next_term, new_sum;
  #pragma omp parallel shared(sum)
  {
    esum_type priv_sum; //local accumulator
    priv_sum.sum = 0.0;
    priv_sum.correction = 0.0;
    double priv_corrected_next_term, priv_new_sum;
    #pragma omp for schedule(dynamic,1)
    for (int i=0; i<n; i++) {
      priv_corrected_next_term = x[i]*y[i] + priv_sum.correction;
      priv_new_sum = priv_sum.sum + priv_corrected_next_term;
      priv_sum.correction = priv_corrected_next_term - (priv_new_sum - priv_sum.sum);
      priv_sum.sum = priv_new_sum;
    }
    #pragma omp critical //ensures that no two threads will access global sum at the same time
    {
      corrected_next_term = priv_sum.sum + sum.correction;
      new_sum = sum.sum + corrected_next_term;
      sum.correction = corrected_next_term - (new_sum - sum.sum);
      sum.sum = new_sum;
    }
  }
  return sum.sum;
}

int main(int argc, char * argv[]) {

  if (argc!=3) {
    cout << "Usage: " << argv[0] << " length_of_vector" << " number_of_trials" << endl;
    exit(1);
  }

  int n = atoi(argv[1]);  //problem size
  int numtrials = atoi(argv[2]); // number of times to compute the parallel dot product

  cout << "Problem size = " << n << endl;
  cout << "Number of trials = " << numtrials << endl;

  int nthreads;
#pragma omp parallel
  nthreads = omp_get_num_threads();
  cout << "Number of threads = " << nthreads << endl;

  // Generate random values
  std::vector< double > x(n), y(n);
  for (int i=0; i<n; ++i) {
    x[i] = drand48();
    y[i] = drand48();
  }

  //Compute serial dot product
  double result = 0.0;
  for (int i=0; i<n; i++) result += x[i]*y[i];

  //Print difference between serial dot product and parallel
     dot products
  cout.precision(17);
  for (int i=0; i<numtrials; ++i)
    cout << result - ComputeDotProductKahanOMP(n, &x[0], &y
        [0]) << endl;

  return 0 ;
}
```

9.1.3 TestDotProductKnuthOMPDouble.cpp

```
// TestDotProductKnuthOMPDouble.cpp
//
// Routine to show reproducibility of the parallel dot
   product
// using Knuth summation for double-precision floating point
// numbers.
// g++ -fopenmp TestDotProductKnuthOMPDouble.cpp -o
   testDotProd
#include <fstream>
#include <iostream>
#include <cstdlib>
using std::endl;
using std::cout;
#include <omp.h>

#include <vector>

//a type to hold accumulator sum and correction terms for
   local and global sums
struct esum_type{
  double sum;
  double correction;
};

/*
 * Method to compute the dot product in parallel using
 * Knuth summation and dynamic scheduling of threads.
 *
 * @param[in]  n the number of vector elements
 * @param[in]  x, y the input vectors
 * @param[out] result a float value, on exit will contain the
     result
 *
 * @return returns parallel dot product result
 */
double ComputeDotProductKnuthOMP(int n, double * x, double *
   y ) {
  esum_type sum; //global accumulator
  sum.sum = 0.0;
  sum.correction = 0.0;
  double u, v, upt, up, vpp;
  #pragma omp parallel shared(sum)
  {
    esum_type priv_sum; //local accumulator
    priv_sum.sum = 0.0;
    priv_sum.correction = 0.0;
    double priv_u, priv_v, priv_upt, priv_up, priv_vpp;
    #pragma omp for schedule(dynamic,1)
    for (int i=0; i<n; i++) {
      priv_u = priv_sum.sum;
      priv_v = x[i]*y[i] + priv_sum.correction;
      priv_upt = priv_u + priv_v;
      priv_up = priv_upt - priv_v;
      priv_vpp = priv_upt - priv_up;
      priv_sum.sum = priv_upt;
      priv_sum.correction = (priv_u - priv_up) + (priv_v -
         priv_vpp);
    }
    #pragma omp critical //ensures that no two threads will
       access global sum at the same time
    {
      u = sum.sum;
      v = priv_sum.sum + sum.correction;
      upt = u + v;
      up = upt - v;
      vpp = upt - up;
      sum.sum = upt;
      sum.correction = (u - up) + (v - vpp);
    }
  }
  return sum.sum;
}

int main(int argc, char * argv[]) {

  if (argc!=3) {
    cout << "Usage: " << argv[0] << " length_of_vector" << " 
       number_of_trials" << endl;
    exit(1);
  }

  int n = atoi(argv[1]);  // problem size
  int numtrials = atoi(argv[2]); // number of times to
      compute the parallel dot product

  cout << "Problem size = " << n << endl;
  cout << "Number of trials = " << numtrials << endl;

  int nthreads;
#pragma omp parallel
  nthreads = omp_get_num_threads();
  cout << "Number of threads = " << nthreads << endl;

  // Generate random values
  std::vector< double > x(n), y(n);
  for (int i=0; i<n; ++i) {
    x[i] = drand48();
    y[i] = drand48();
  }

  //Compute serial dot product
  double result = 0.0;
  for (int i=0; i<n; i++) result += x[i]*y[i];

  //Print difference between serial dot product and parallel
      dot products
  cout.precision(17);
  for (int i=0; i<numtrials; ++i)
    cout << result - ComputeDotProductKnuthOMP(n, &x[0], &y
        [0]) << endl;

  return 0 ;
}
```

9.1.4 TestDotProductOMPFloat.cpp

```
// TestDotProductKahanOMPFloat.cpp
//
// Routine to show reproducibility of the parallel dot
   product
// using summation for single-precision floating point
// numbers.
//
//g++ -fopenmp TestDotProductOMPFloat.cpp -o testDotProd
#include <fstream>
#include <iostream>
#include <cstdlib>
using std::endl;
using std::cout;
#include <omp.h>

#include <vector>

/*
 * Method to compute the dot product in parallel using
 * dynamic scheduling of threads.
 *
 * @param[in]  n the number of vector elements
 * @param[in]  x, y the input vectors
 * @param[out] result a float value, on exit will contain the
     result
 *
 * @return returns parallel dot product result
 */
float ComputeDotProductOMP(int n, float * x, float * y ) {
  float result = 0.0;
#pragma omp parallel for reduction (+:result) schedule(
   dynamic,1)
  for (int i=0; i<n; i++) result += x[i]*y[i];
  return result;
}

int main(int argc, char * argv[]) {

  if (argc!=3) {
    cout << "Usage: " << argv[0] << " length_of_vector" << "
       number_of_trials" << endl;
    exit(1);
  }

  int n = atoi(argv[1]);  // problem size
  int numtrials = atoi(argv[2]); // number of times to
     compute the parallel dot product

  cout << "Problem size = " << n << endl;
  cout << "Number of trials = " << numtrials << endl;

  int nthreads;
#pragma omp parallel
  nthreads = omp_get_num_threads();
  cout << "Number of threads = " << nthreads << endl;

  // Generate random values
  std::vector< float > x(n), y(n);
  for (int i=0; i<n; ++i) {
    x[i] = drand48();
    y[i] = drand48();
  }

  //Compute serial dot product
  float result = 0.0;
  for (int i=0; i<n; i++) result += x[i]*y[i];

  //Print difference between serial dot product and parallel
     dot products
  cout.precision(17);
  for (int i=0; i<numtrials; ++i)
    cout << result - ComputeDotProductOMP(n, &x[0], &y[0]) <<
        endl;

  return 0 ;
}
```

9.1.5 TestDotProductKahanOMPFloat.cpp

```
// TestDotProductKahanOMPFloat.cpp
//
// Routine to show reproducibility of the parallel dot
   product
// using Kahan summation for single-precision floating point
// numbers.
// g++ -fopenmp TestDotProductKahanOMPFloat.cpp -o
   testDotProd
```

```
#include <fstream>
#include <iostream>
#include <cstdlib>
using std::endl;
using std::cout;
#include <omp.h>

#include <vector>

//a type to hold accumulator sum and correction terms for local and global sums
struct esum_type{
  float sum;
  float correction;
};

/*
 * Method to compute the dot product in parallel using
 * Kahan summation and dynamic scheduling of threads.
 *
 * @param[in]  n the number of vector elements
 * @param[in]  x, y the input vectors
 * @param[out] result a float value, on exit will contain the result
 *
 * @return returns parallel dot product result
 */
float ComputeDotProductKahanOMP(int n, float * x, float * y ) {
  esum_type sum; //global accumulator
  sum.sum = 0.0;
  sum.correction = 0.0;
  float corrected_next_term, new_sum;
  #pragma omp parallel shared(sum)
  {
    esum_type priv_sum; //local accumulator
    priv_sum.sum = 0.0;
    priv_sum.correction = 0.0;
    float priv_corrected_next_term, priv_new_sum;
    #pragma omp for schedule(dynamic,1)
    for (int i=0; i<n; i++) {
      priv_corrected_next_term = x[i]*y[i] + priv_sum.correction;
      priv_new_sum = priv_sum.sum + priv_corrected_next_term;
      priv_sum.correction = priv_corrected_next_term - (priv_new_sum - priv_sum.sum);
      priv_sum.sum = priv_new_sum;
    }
    #pragma omp critical  //ensures that no two threads will access global sum at the same time
    {
      corrected_next_term = priv_sum.sum + sum.correction;
      new_sum = sum.sum + corrected_next_term;
      sum.correction = corrected_next_term - (new_sum - sum.sum);
      sum.sum = new_sum;
    }
  }
  return sum.sum;
}

int main(int argc, char * argv[]) {

  if (argc!=3) {
    cout << "Usage: " << argv[0] << " length_of_vector" << " number_of_trials" << endl;
    exit(1);
  }

  int n = atoi(argv[1]);          // problem size
  int numtrials = atoi(argv[2]); // number of times to compute the parallel dot product

  cout << "Problem size = " << n << endl;
  cout << "Number of trials = " << numtrials << endl;

  int nthreads;
#pragma omp parallel
  nthreads = omp_get_num_threads();
  cout << "Number of threads = " << nthreads << endl;

  // Generate random values
  std::vector< float > x(n), y(n);
  for (int i=0; i<n; ++i) {
    x[i] = drand48();
    y[i] = drand48();
  }

  //Compute serial dot product
  float result = 0.0;
  for (int i=0; i<n; i++) result += x[i]*y[i];

  //Print difference between serial dot product and parallel
     dot products
  cout.precision(17);
  for (int i=0; i<numtrials; ++i)
    cout << result - ComputeDotProductKahanOMP(n, &x[0], &y
       [0]) << endl;

  return 0 ;
}
```

9.1.6 TestDotProductKnuthOMPFloat.cpp

```
// TestDotProductKnuthOMPFloat.cpp
//
// Routine to show reproducibility of the parallel dot
   product
// using Knuth summation for single-precision floating point
// numbers.
// g++ -fopenmp TestDotProductKnuthOMPFloat.cpp -o
   testDotProd

#include <fstream>
#include <iostream>
#include <cstdlib>
using std::endl;
using std::cout;
#include <omp.h>

#include <vector>

//a type to hold accumulator sum and correction terms for
   local and global sums
struct esum_type{
  float sum;
  float correction;
};


/*
 * Method to compute the dot product in parallel using
 * Knuth summation and dynamic scheduling of threads.
 *
 * @param[in]  n the number of vector elements
 * @param[in]  x, y the input vectors
 * @param[out] result a float value, on exit will contain the
     result
 *
 * @return returns parallel dot product result
 */
float ComputeDotProductKnuthOMP(int n, float * x, float * y )
    {
  esum_type sum; //global accumulator
  sum.sum = 0.0;
  sum.correction = 0.0;
  float u, v, upt, up, vpp;
  #pragma omp parallel shared(sum)
  {
    esum_type priv_sum; //local accumulator
    priv_sum.sum = 0.0;
    priv_sum.correction = 0.0;
    float priv_u, priv_v, priv_upt, priv_up, priv_vpp;
    #pragma omp for schedule(dynamic,1)
    for (int i=0; i<n; i++) {
      priv_u = priv_sum.sum;
      priv_v = x[i]*y[i] + priv_sum.correction;
      priv_upt = priv_u + priv_v;
      priv_up = priv_upt - priv_v;
      priv_vpp = priv_upt - priv_up;
      priv_sum.sum = priv_upt;
      priv_sum.correction = (priv_u - priv_up) + (priv_v -
         priv_vpp);
    }
    #pragma omp critical //ensures that no two threads will
       access global sum at the same time
    {
      u = sum.sum;
      v = priv_sum.sum + sum.correction;
      upt = u + v;
      up = upt - v;
      vpp = upt - up;
      sum.sum = upt;
      sum.correction = (u - up) + (v - vpp);
    }
  }
  return sum.sum;
}
```

```

int main(int argc, char * argv[]) {

  if (argc!=3) {
    cout << "Usage: " << argv[0] << " length_of_vector" << " 
       number_of_trials" << endl;
    exit(1);
  }

  int n = atoi(argv[1]);  // problem size
  int numtrials = atoi(argv[2]); // number of times to
     compute the parallel dot product

  cout << "Problem size = " << n << endl;
  cout << "Number of trials = " << numtrials << endl;

  int nthreads;
#pragma omp parallel
  nthreads = omp_get_num_threads();
  cout << "Number of threads = " << nthreads << endl;

  // Generate random values
  std::vector< float > x(n), y(n);
  for (int i=0; i<n; ++i) {
    x[i] = drand48();
    y[i] = drand48();
  }

  //Compute serial dot product
  float result = 0.0;
  for (int i=0; i<n; i++) result += x[i]*y[i];

  //Print difference between serial dot product and parallel
     dot products
  cout.precision(17);
  for (int i=0; i<numtrials; ++i)
    cout << result - ComputeDotProductKnuthOMP(n, &x[0], &y
       [0]) << endl;

  return 0 ;
}
```

9.1.7 TestDotProductOMPDoubleAccumulator.cpp

```
// TestDotProductKahanOMPFloat.cpp
//
// Routine to show reproducibility of the parallel dot
   product
// using summation for single-precision floating point
// numbers.
//
//g++ -fopenmp TestDotProductOMPFloat.cpp -o testDotProd
#include <fstream>
#include <iostream>
#include <cstdlib>
using std::endl;
using std::cout;
#include <omp.h>

#include <vector>

/*
 * Method to compute the dot product in parallel using
 * dynamic scheduling of threads.
 *
 * @param[in]  n the number of vector elements
 * @param[in]  x, y the input vectors
 * @param[out] result a float value, on exit will contain the
     result
 *
 * @return returns parallel dot product result
 */
double ComputeDotProductOMP(int n, float * x, float * y ) {
  double result = 0.0;
#pragma omp parallel for reduction (+:result) schedule(
   dynamic,1)
  for (int i=0; i<n; i++) result += x[i]*y[i];
  return result;
}

int main(int argc, char * argv[]) {

  if (argc!=3) {
    cout << "Usage: " << argv[0] << " length_of_vector" << "
       number_of_trials" << endl;
    exit(1);
  }

  int n = atoi(argv[1]);  // problem size
  int numtrials = atoi(argv[2]); // number of times to
     compute the parallel dot product

  cout << "Problem size = " << n << endl;
  cout << "Number of trials = " << numtrials << endl;

  int nthreads;
#pragma omp parallel
  nthreads = omp_get_num_threads();
  cout << "Number of threads = " << nthreads << endl;

  // Generate random values
  std::vector< float > x(n), y(n);
  for (int i=0; i<n; ++i) {
    x[i] = drand48();
    y[i] = drand48();
  }

  //Compute serial dot product
  double result = 0.0;
  for (int i=0; i<n; i++) result += x[i]*y[i];

  //Print difference between serial dot product and parallel
     dot products
  cout.precision(17);
  for (int i=0; i<numtrials; ++i)
    cout << result - ComputeDotProductOMP(n, &x[0], &y[0]) <<
        endl;

  return 0 ;
}
```

9.1.8 TestDotProductKahanOMPDoubleAccumulator.cpp

```
// TestDotProductKahanOMPFloat.cpp
//
// Routine to show reproducibility of the parallel dot
   product
// using Kahan summation for single-precision floating point
// numbers.
// g++ -fopenmp TestDotProductKahanOMPFloat.cpp -o
   testDotProd
#include <fstream>
#include <iostream>
```

```
#include <cstdlib>
using std::endl;
using std::cout;
#include <omp.h>

#include <vector>

//a type to hold accumulator sum and correction terms for
    local and global sums
struct esum_type{
  double sum;
  double correction;
};

/*
 * Method to compute the dot product in parallel using
 * Kahan summation and dynamic scheduling of threads.
 *
 * @param[in]  n the number of vector elements
 * @param[in]  x, y the input vectors
 * @param[out] result a float value, on exit will contain the
     result
 *
 * @return returns parallel dot product result
 */
double ComputeDotProductKahanOMP(int n, float * x, float * y
    ) {
  esum_type sum; //global accumulator
  sum.sum = 0.0;
  sum.correction = 0.0;
  double corrected_next_term, new_sum;
  #pragma omp parallel shared(sum)
  {
    esum_type priv_sum; //local accumulator
    priv_sum.sum = 0.0;
    priv_sum.correction = 0.0;
    double priv_corrected_next_term, priv_new_sum;
    #pragma omp for schedule(dynamic,1)
    for (int i=0; i<n; i++) {
      priv_corrected_next_term = x[i]*y[i] + priv_sum.
          correction;
      priv_new_sum = priv_sum.sum + priv_corrected_next_term;
      priv_sum.correction = priv_corrected_next_term - (
          priv_new_sum - priv_sum.sum);
      priv_sum.sum = priv_new_sum;
```

```cpp
    }
    #pragma omp critical  //ensures that no two threads will access global sum at the same time
    {
      corrected_next_term = priv_sum.sum + sum.correction;
      new_sum = sum.sum + corrected_next_term;
      sum.correction = corrected_next_term - (new_sum - sum.sum);
      sum.sum = new_sum;
    }
  }
  return sum.sum;
}

int main(int argc, char * argv[]) {

  if (argc!=3) {
    cout << "Usage: " << argv[0] << " length_of_vector" << " number_of_trials" << endl;
    exit(1);
  }

  int n = atoi(argv[1]);          // problem size
  int numtrials = atoi(argv[2]); // number of times to compute the parallel dot product

  cout << "Problem size = " << n << endl;
  cout << "Number of trials = " << numtrials << endl;

  int nthreads;
#pragma omp parallel
  nthreads = omp_get_num_threads();
  cout << "Number of threads = " << nthreads << endl;

  // Generate random values
  std::vector< float > x(n), y(n);
  for (int i=0; i<n; ++i) {
    x[i] = drand48();
    y[i] = drand48();
  }

  //Compute serial dot product
  double result = 0.0;
  for (int i=0; i<n; i++) result += x[i]*y[i];

  //Print difference between serial dot product and parallel dot products
  cout.precision(17);
  for (int i=0; i<numtrials; ++i)
    cout << result - ComputeDotProductKahanOMP(n, &x[0], &y[0]) << endl;

  return 0 ;
}
```

9.1.9 TestDotProductKnuthOMPDoubleAccumulator.cpp

```
// TestDotProductKnuthOMPDoubleAccumulator.cpp
//
// Routine to show reproducibility of the parallel dot product
// using Knuth summation for single-precision floating point
// numbers, but with a double-precision accumulator.
// g++ -fopenmp TestDotProductKnuthOMPDoubleAccumulator.cpp -o testDotProd

#include <fstream>
#include <iostream>
#include <cstdlib>
using std::endl;
using std::cout;
#include <omp.h>

#include <vector>

//a type to hold accumulator sum and correction terms for local and global sums
struct esum_type{
  double sum;
  double correction;
};


/*
 * Method to compute the dot product in parallel using
 * Knuth summation and dynamic scheduling of threads.
 *
 * @param[in]  n the number of vector elements
 * @param[in]  x, y the input vectors
 * @param[out] result a float value, on exit will contain the
     result
 *
 * @return returns parallel dot product result
 */
double ComputeDotProductKnuthOMP(int n, float * x, float * y
   ) {
  esum_type sum; //global accumulator
  sum.sum = 0.0;
  sum.correction = 0.0;
  double u, v, upt, up, vpp;
  #pragma omp parallel shared(sum)
  {
    esum_type priv_sum; //local accumulator
    priv_sum.sum = 0.0;
    priv_sum.correction = 0.0;
    double priv_u, priv_v, priv_upt, priv_up, priv_vpp;
    #pragma omp for schedule(dynamic,1)
    for (int i=0; i<n; i++) {
      priv_u = priv_sum.sum;
      priv_v = x[i]*y[i] + priv_sum.correction;
      priv_upt = priv_u + priv_v;
      priv_up = priv_upt - priv_v;
      priv_vpp = priv_upt - priv_up;
      priv_sum.sum = priv_upt;
      priv_sum.correction = (priv_u - priv_up) + (priv_v -
         priv_vpp);
    }
    #pragma omp critical //ensures that no two threads will
        access global sum at the same time
    {
      u = sum.sum;
      v = priv_sum.sum + sum.correction;
      upt = u + v;
      up = upt - v;
      vpp = upt - up;
      sum.sum = upt;
      sum.correction = (u - up) + (v - vpp);
    }
  }
  return sum.sum;
}

int main(int argc, char * argv[]) {

  if (argc!=3) {
    cout << "Usage: " << argv[0] << " length_of_vector" << "
       number_of_trials" << endl;
    exit(1);
  }

  int n = atoi(argv[1]);  // problem size
  int numtrials = atoi(argv[2]); // number of times to
      compute the parallel dot product

  cout << "Problem size = " << n << endl;
  cout << "Number of trials = " << numtrials << endl;

  int nthreads;
#pragma omp parallel
  nthreads = omp_get_num_threads();
  cout << "Number of threads = " << nthreads << endl;

  // Generate random values
  std::vector< float > x(n), y(n);
  for (int i=0; i<n; ++i) {
    x[i] = drand48();
    y[i] = drand48();
  }

  //Compute serial dot product
  double result = 0.0;
  for (int i=0; i<n; i++) result += x[i]*y[i];

  //Print difference between serial dot product and parallel
      dot products
  cout.precision(17);
  for (int i=0; i<numtrials; ++i)
    cout << result - ComputeDotProductKnuthOMP(n, &x[0], &y
       [0]) << endl;

  return 0 ;
}
```

References

[1] A. Arteaga, O.Fuhrer, and T. Hoefler, *Designing bit-reproducible portable high-performance applications*, Proceedings of the 28th IEEE

International Parallel And Distributed Processing Symposium (IPDPS), IEEE Computer Society, Apr. 2014.

[2] Mathew A. Cleveland, Thomas A. Brunner, Nicholas A. Gentile, and Jeffrey A. Keasler, *Obtaining identical results with double precision global accuracy on different numbers of processors in parallel particle monte carlo simulations*, Journal of Computational Physics **251** (2013), no. 0, 223 – 236.

[3] Sylvain Collange, David Defour, Stef Graillat, and Roman Iakymchuk, *Full-speed deterministic bit-accurate parallel floating-point summation on multi- and many-core architectures*, Tech. report, Feb 2014.

[4] J. Demmel and Hong Diep Nguyen, *Fast reproducible floating-point summation*, Computer Arithmetic (ARITH), 2013 21st IEEE Symposium on, April 2013, pp. 163–172.

[5] James Demmel and Yozo Hida, *Accurate and efficient floating point summation*, SIAM J. Sci. Comput. **25** (2003), no. 4, 1214–1248.

[6] James Demmel and Hong Diep Nguyen, *Numerical reproducibility and accuracy at exascale*, Proceedings of the 2013 IEEE 21st Symposium on Computer Arithmetic (Washington, DC, USA), ARITH '13, IEEE Computer Society, 2013, pp. 235–237.

[7] V. Carl Hamacher, *Computer organization and embedded systems*, McGraw-Hill, New York, NY, 2012.

[8] Donald E. Knuth, *The art of computer programming, volume 2 (3rd ed.): Seminumerical algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[9] M. McCool, J. Reinders, and A. Robison, *Structured parallel programming: Patterns for efficient computation*, Elsevier Science, 2012.

[10] Robert W. Robey, Jonathan M. Robey, and Rob Aulwes, *In search of numerical consistency in parallel programming*, Parallel Comput. **37** (2011), no. 4-5, 217–229.