



Whitepaper

Parallelization and Floating Point Numbers

By Tim Mattson and Ken Strandberg

Sometimes we forget that not all numbers are the same. This becomes very apparent in dealing with floating point numbers in parallel computing. This article addresses a common challenge parallel programmers often face and must resolve to ensure their parallel programs deliver safe and expected results.

Introduction

The more cores programmers run their parallelized code on, the more ways operations can be interleaved and the more challenges programmers face. Parallel programmers must deal with a host of issues peculiar to parallel programs such as synchronization, protecting shared variables, and finding thread safe versions of common math routines (such as random number generation). One of the most subtle problems faced by the parallel programmer, however, arises from the properties of floating point numbers. Floating point numbers are the same in serial and parallel computations, of course, but when a program executes in parallel, special features of these numbers are more likely to impact your results.

Floating point numbers aren't new. They've been around for a long time; they were standardized by IEEE in the 1980s in IEEE-754. The Intel® 8087 processor was the first chip to support the standard at the time. IEEE-754 defines the floating point standard that virtually every CPU vendor uses.

IEEE-754 defines normalized floating point numbers with the following binary format

$$\pm 1.d\dots d \times 2^{exp}$$

The binary digits "d...d" are the fractional part of the number (also known as the mantissa) and "exp" is the exponent. The leading "1" is implied; you know it will always be there so you don't need to actually store the bit. So, if you have 32 bits total to hold a single precision number, then use 1 bit for the sign and 8 bits for the exponent, which leaves you 23 bits for the fractional part of the number. Add back the implied leading "1" and the total number of significant bits is 23+1. The following table provides details about the types of floating point numbers defined in the IEEE 754 standard. This includes single, double, and double-extended floating point numbers, tracking the evolution of processor data widths over the decades.

Format	# Bits	# Sig. Bits	# Exp. Bits	Exp. Range
Single	32	23+1	8	$2^{-126} - 2^{127}$ ($\sim 10^{\pm 38}$)
Double	64	52+1	11	$2^{-1022} - 2^{1023}$ ($\sim 10^{\pm 308}$)
Double Extended	≥ 80	≥ 64	≥ 15	$2^{-16382} - 2^{16383}$ ($\sim 10^{\pm 4932}$)

Floating Point Numbers Aren't Real!

Most of us think in terms of real numbers. These are the numbers you encounter in math courses in school. Both real and floating point numbers have a whole part, a fractional part, and exponents. Here are some examples (expressed as base 10 rather than binary formats):

- 42.0×10^{23}
- 2.34564
- 35405004999604.60409954

Real numbers are infinitely precise. Between any two real numbers, there are an infinite number of real numbers. They have many nice properties that we often take for granted. The most important property is that real numbers define a closed set with respect to the basic mathematical operations (+, -, *, /). In

A op B = C

any real numbers A and B will generate a real number C, whenever A op B is mathematically well defined. Since real numbers are a closed set and infinitely precise, they have a number of very nice properties:

- They're commutative: $A \text{ op } B = B \text{ op } A$ (for * and +).

- They're associative: $A \text{ op } (C \text{ op } B) = (A \text{ op } C) \text{ op } B$ (for + and *).
- They're distributive: $A * (B + C) = A * B + A * C$.

Unfortunately, floating point numbers are not real. Floating point numbers use a fixed number of bits. And that's where the trouble starts. As you write any program using these numbers, you need to keep that fact in mind.

Floating Point Math

With floating point math, you can easily generate results that don't fit into a floating point format. For example, consider the following code:

```
A = 0.01;
if (100 * A != 1.0) printf("oops");
```

The output of this snippet would be

"oops"

Why? 0.01 does not have an exact binary representation in floating point format. So, the value is rounded to the nearest floating point number, and hence $100 * 0.01$ is not precisely equal to 1.0. Here's another one.

```
b = 1000.2;
c = b - 1000.0;
printf("%.f", c);
```

The output of this snippet would be

0.200012

Again, 0.2 does not have a binary representation, so the computer rounds it to the nearest floating point value. You can easily generate numbers that don't fit into the floating point format, and thus you produce answers from the basic arithmetic operations that don't fit into a floating point format. In other words, the floating point numbers when operated on by the basic arithmetic operations do not constitute a closed set.

The impact of this is significant. Floating numbers are not associative or distributive. So,

- $A * (C * B) \neq (A * C) * B$ and
- $A * (B + C) \neq A * B + A * C$

This means that as you change the order of a long sequence of arithmetic operations, you can generate different answers. Mathematically with real numbers, the answers can't depend on the order of the operations (for commutative operations) or the way they are grouped together (associatively). But with floating point numbers, if you interleave the operations in different ways, you get different results.

Here's a good test to demonstrate the implications of this behavior by floating point numbers:

1. Fill 2 arrays each with 10000 random values between 0.0 and 1.0.
2. Shift one up by 100 and shift the other down by 0.001.
3. Mix the arrays together, sum them, and subtract a large number (500000).

Here are the results run on 1, 2, and 4 threads.

1 thread	170.968750
2 threads	171.968750
4 threads	172.750000

Which one of these numbers is correct, the 1-thread, 2-thread, or 4-thread value? Are any of these the true value? Would you consider that with 4 threads, the answer is correct and the others wrong? Or with 1 thread?

This is not a trick question, nor is its goal to make programmers look silly. Developers are smart people. But, many programmers steeped in sequential programming for so many years make the assumption that there is only one right answer for their algorithm. After all, their code has always delivered the same answer every time it was run. When you consider the above example, however, all the answers are equally correct. To pick one arbitrarily and call it right and the others wrong is completely unjustified.

Dealing with FLOPs

The fact is that all the results of the above example are “correct.”

By mixing the numbers as this example does, it creates a pathological situation designed to maximize problems due to round off error. The test mixes very large and very small numbers together. The arithmetic unit aligns the numbers before adding them, which, given the large difference in their absolute magnitudes, all but guarantees that we’ll lose bits of precision in the process.

As the number of threads changes, the combinations of numbers being added also changes. With all the roundoff errors, as the way these numbers are combined changes, the way roundoff error is accumulated also keeps changing. Thus, the answers change.

So which answer is correct? The algorithm for adding them together is unstable. If you carefully add the numbers together so large numbers add with large numbers and small numbers add with small numbers, and then add the “large_set_sum” to the “small_set_sum”, you get a numerically stable result. The answer in this case is 177.750. Note that the test answers in every case considerably vary from the stable method of obtaining the answer.

Note also that, with a serial algorithm, you’d never know there was a problem. Only as the thread count grows and the answers change, does the instability of the algorithm become obvious. It’s apparent the problem is not in the compiler or even the program. The problem is with the numerical instability of the algorithm. And it’s only revealed by going to multiple threads.

Now if we simply require that we exactly reproduce the single thread result, we’re just denying there’s a problem with our algorithm. Interestingly, this is a common request: asking to conduct reductions in the serial order (which would all but eliminate the possibility of seeing any speedup from my parallel program). Instead, we should accept that this variation is telling us about the fundamental numerical error in our computation.

In your parallel code development, if the variation as the number of threads changes is small compared to the precision you require in your final answer, then the instability is minor enough to be safely ignored. Your code is done; you can move on to the next challenge.

But if the variation in answers as the thread count changes is significant to you, then you need to rethink your problem with an understanding of floating point numbers and design a better solution.

Safely Parallelize

Numerical instabilities that become apparent as the number of threads changes are but one example of the challenges we face when working with floating point numbers on a parallel system. Another type of problem appears when using numerical libraries that use floating point numbers. First, you must assure that the libraries in question are thread-safe. A thread-safe library is designed such that calls into the library can be safely made by multiple threads at the same time.

Even for a thread safe library, you must assure that the library routines are used correctly as the thread counts grow. The best example of this appears when working with random number generators. A random number generator creates a sequence of numbers that are statistically random (called pseudo-random numbers since they are not truly random if generated by a fixed algorithm). If each thread generates its own sequence, then you must assure that the sequences don't overlap. There are ways to give each thread a unique generator and hence assure that the pseudo-random sequences are not statistically correlated. In this case, however, the actual sequence of pseudo-random numbers changes as the number of threads vary. This can lead to algorithms that generate different results as the number of threads change. As with other problems we've mentioned with floating point numbers, each answer may be equally correct. The variation isn't telling you that there is an error in your program. Instead this variation may be telling you about the fundamental errors in your problem, an error present with one thread or many; it's just that with one thread you didn't know you had any problems.

The take-home lesson is that bitwise consistency as the number of threads change may not be possible or even desirable. If your program is free of data-races, the variation in results may be telling you something important. If the amount the results changes as you vary the number of threads matter in your final result, then you may very well have an unstable numerical algorithm. In other words, your problem may be giving you bad results regardless of the number of threads; a problem you may never had known had you not moved to a multi-threaded version of your program.

Conclusion

Floating point numbers aren't real. They don't follow some mathematical rules for real numbers, resulting in different answers based on the order in which the algorithm executes. In a well-posed problem with a thread-safe algorithm, these differences don't matter. They can't matter, because if they did, then all answers would be equally wrong as well as equally right. When troubleshooting your code, resist the urge to consider the order, processor, or compiler is to blame. Re-examine your requirements and your algorithm. As you parallelize your code, be sure to use libraries that are thread-safe, and

test your algorithms to deliver results acceptable to your application regardless of how the threads execute.

About the Authors



Dr. Timothy Mattson is a principle engineer at Intel working in the Microprocessor Technology laboratory. He is also a parallel programmer. Over the last 20 years, he has used parallel computers to make chemicals react, shake up proteins, find oil, understand genes, and solve many other scientific problems. Tim's long term research goal is to make sequential software rare by bringing today's sequential programmers into the realm of parallel development.



Ken Strandberg writes technical articles, white papers, seminars, web-based training, marketing collateral, and interactive content for emerging technology companies, Fortune 100 enterprises, and multi-national corporations. Mr. Strandberg's technology areas include Software, High-performance Computing and Clusters, Industrial Technologies, Design Automation, Networking, Medical Technologies, Semiconductor, and Telecom. Mr. Strandberg can be reached at ken@kenstrandberg.com.