

Learning to live with errors: a fresh look at floating-point computation

Josh Milthorpe* and Alistair Rendell

Department of Computer Science
Australian National University
Canberra, ACT 0200

Submission for 3rd Australian Undergraduate Students' Computing Conference 2005

17 November 2005

Abstract

Interval analysis is an alternative to conventional floating-point computation that offers guaranteed error bounds. Despite this advantage, interval methods have rarely been applied in high performance scientific computing. In part, this is because of the additional cost associated with performing interval operations over the corresponding floating-point operations. The aim of this work is to take a fresh look at interval arithmetic and the viability of using intervals in large scale scientific computations. In this paper we will report on the performance of interval arithmetic on the UltraSPARC architecture, with a focus on the Sun Studio implementation.

Different methods of calculating interval results will be discussed, including one novel approach to interval multiplication. Based on the benchmark results for different interval implementations, changes to existing interval algorithms are suggested. The hardware modification of floating point units to provide additional architectural support for intervals is also considered.

Finally, brief consideration will be given to the application of interval techniques to a real-world problem in computational chemistry, namely the evaluation of pairwise interactions under the Coulomb potential.

1 Introduction

Much of science is concerned with physical phenomena that are continuous, such as time, distance or temperature. In contrast, digital computing is limited to discrete representations. Traditionally, this divide has been bridged through the use of floating-point arithmetic, with all numbers approximated to a specified precision using a finite set of machine-representable numbers. Inherent in this approximation is the concept of a rounding error. The behaviour of rounding errors in a given computation can be difficult to predict in advance; they may cancel out or compound, depending on both the algorithm and the particular input data.

Currently, virtually all scientific codes ignore rounding errors, instead using double-precision arithmetic and hoping for the best. However, large parallel computers in use today are capable of performing in excess of 10^{12} floating-point operations per second (over 1 Teraflop/s). Each of these operations is a potential source of rounding error, so even when using double-precision arithmetic which has rounding errors of the order of 10^{-15} , it is easy to see that this error may quickly grow to be as large as the computed quantities themselves.

In 1959 Moore[12] proposed interval arithmetic as an extension to ordinary arithmetic on real numbers. An interval is a continuum of real numbers, defined by a lower bound and an upper bound. By substituting intervals for floating-point numbers in automated computation, an interval result can always be found to bound the correct real result.

Implemented correctly, interval computation can be used to bound errors from all sources including input uncertainty, truncation and rounding errors. This has led interval researchers to label their field *Reliable*

Computing. In many fields verified results, such as those produced by interval computation, are obviously very valuable. Unsurprisingly, some researchers[8] have questioned why interval techniques are not more widely used.

However, the application of interval techniques is not always straightforward; in particular, it is not always possible to find an algorithm that produces *sharp*¹ interval bounds. A further barrier to widespread adoption of interval methods is the potentially large performance penalty of interval computation. Most interval implementations are entirely in software and suffer from overhead due to function calls, error checking, changing rounding mode and other sources. Recently, efforts have been made to provide fast interval libraries[3; 11] or compiler support[1; 2; 4].

This paper reports some measurements of the performance of codes compiled with a commercial interval implementation, the Sun Studio 10 Fortran compiler[2]. Improvements are suggested for this implementation and for hardware support for intervals on the UltraSPARC architecture. Finally, interval analysis is applied to an example problem from computational chemistry, to assist in choosing an algorithm to give more accurate results.

2 Interval arithmetic and performance

For a computer with hardware support for floating-point arithmetic, the most natural representation for an interval is a pair of floating-point numbers (the lower bound \underline{X} and upper bound \overline{X}). An interval has a width $\overline{X} - \underline{X}$ and a midpoint $(\underline{X} + \overline{X})/2$. The ubiquitous IEEE 754 standard[10] for floating-point arithmetic defines two rounding modes to support interval arithmetic: upwards (round towards $+\infty$) and downwards (round towards $-\infty$).

For two intervals X and Y , the binary arithmetic operations are defined by $X \circ Y \equiv \{x \circ y | x \in X; y \in Y\}$; where $\circ \in \{+, -, \times, /\}$. When interval endpoints are represented by floating-point numbers, endpoint results that cannot be represented exactly must be rounded outwards to a representable number.

For addition, this reduces to

$$X + Y = [\lfloor \underline{X} + \underline{Y} \rfloor, \lceil \overline{X} + \overline{Y} \rceil]$$

Here the floor and ceiling symbols indicate that inexact results are to be rounded downward and upward respectively to a representable number. The simplest scheme for interval multiplication is

$$X \times Y = [a, b]$$

$$a = \min(\lfloor \underline{X} \times \underline{Y} \rfloor, \lfloor \underline{X} \times \overline{Y} \rfloor, \lfloor \overline{X} \times \underline{Y} \rfloor, \lfloor \overline{X} \times \overline{Y} \rfloor)$$

$$b = \max(\lceil \underline{X} \times \underline{Y} \rceil, \lceil \underline{X} \times \overline{Y} \rceil, \lceil \overline{X} \times \underline{Y} \rceil, \lceil \overline{X} \times \overline{Y} \rceil)$$

However, Moore[13] notes that the signs of the endpoints will determine which endpoints should be multiplied together to form the interval product (see Table 1). Thus in most cases, only two floating-point multiplications are required to form an interval product. In the case where both intervals contain zero, it is necessary to perform two multiplications for each end point and take the minimum or maximum.

Sun provides support for intervals in the Sun Studio suite. This takes the form of interval types that are recognised by the compiler with the operations defined on those types, and a set of optimized intrinsic interval subroutines. Sun's Fortran compiler can be used to quickly and easily develop interval codes, or convert existing floating-point codes to use interval methods.

However there are some aspects of Sun's interval support that lead to suboptimal performance. The basic interval arithmetic operations compile to procedure calls, and on the UltraSPARC, an inefficient method of interval multiplication is used.

¹An interval is said to be *sharp* if it is as narrow as possible.

Signs				Result	
\underline{a}	\bar{a}	\underline{b}	\bar{b}	\underline{c}	\bar{c}
+	+	+	+	$\underline{a} \times \underline{b}$	$\bar{a} \times \bar{b}$
+	+	-	-	$\bar{a} \times \underline{b}$	$\underline{a} \times \bar{b}$
+	+	-	+	$\bar{a} \times \bar{b}$	$\bar{a} \times \bar{b}$
-	-	+	+	$\underline{a} \times \bar{b}$	$\bar{a} \times \underline{b}$
-	-	-	-	$\bar{a} \times \bar{b}$	$\underline{a} \times \underline{b}$
-	-	-	+	$\underline{a} \times \bar{b}$	$\underline{a} \times \bar{b}$
-	+	+	+	$\underline{a} \times \bar{b}$	$\bar{a} \times \bar{b}$
-	+	-	-	$\bar{a} \times \underline{b}$	$\underline{a} \times \underline{b}$
-	+	-	+	$\min(\underline{a} \times \bar{b}, \bar{a} \times \underline{b})$	$\max(\underline{a} \times \underline{b}, \bar{a} \times \bar{b})$

Table 1: Interval multiplication table for $c = a \times b$

Performing interval arithmetic through procedure calls is far slower than inline arithmetic. Data must be passed as arguments to the called routine, requiring manipulation of the stack, and there will be a jump in the instruction stream. More importantly, the compiler may be unable to generate sequences of operations that can be efficiently pipelined, which will reduce the throughput on a deeply pipelined processor like the UltraSPARC. Finally, without information about the behaviour of a procedure call (e.g. side-effects), an optimising compiler is limited in the types of optimisations that it is able to perform.

Inspection of the assembly code for Sun's interval subroutines reveals that the method of interval multiplication based on signs of endpoints (discussed above) is used. This is implemented through a series of comparison and branch statements that determine which endpoints are used. Branch statements can be problematic where it is difficult to predict in advance which branch the code will take. The UltraSPARC III Cu executes statements in order, has a 14-stage pipeline, and performs branch prediction. This means that during execution of code that contains branch statements, it may have many instructions underway in the pipeline that depend on the results of a branch prediction that has not yet been resolved. If the prediction is incorrect, the pipeline must be flushed and reloaded with the instructions corresponding to the correct branch, which results in wasted cycles.

3 Performance results

To determine whether it is possible to improve on the performance of Sun's interval implementation on the UltraSPARC platform, hand optimised subroutines were written to perform equivalent interval operations. Here, results are presented for a vector sum and product, a dot product, and a general matrix multiplication. These operations were chosen because they form the building blocks of many more complex numerical routines used in scientific and engineering codes.

For the vector sum and product, codes were constructed to perform these operations using the basic arithmetic operations $+$ and \times . For example, the sum was implemented as

```
do i = 1, size(input)
  result = result + input(i)
enddo
```

The codes were compiled for both floating-point and interval values with the compiler options `-fast -xarch=v8plusb`. The subroutines were run for a large number of iterations, for vectors of various sizes. The objective here is to assess the likely throughput performance that might be obtained when executing this code as the inner loop in a more complex real-world application. Thus it was assumed that the data required

is serviced from level 2 cache and the range of the loop is restricted accordingly (the UltraSPARC III Cu used here has an 8MB L2 cache). The total time was divided by the number of iterations, the size of the vectors, and the CPU cycle time (900 MHz, or 1.11... ns, on the machine used), to give the number of cycles to perform a single basic 'operation' (addition or multiplication). Sun also provides optimized intrinsic array functions for sum and product; the timings for these are given for comparison.

Handwritten interval versions of these operations were constructed. The product was based on the first definition of interval multiplication given above (using min/max of all endpoint products). It does not contain branches, but does use the *conditional move* instruction available on the UltraSPARC III Cu, which sets an integer or floating-point register based on the result of a comparison. The results for all sum and product benchmarks are shown in Table 2.

Floating Point		Interval					
Basic		Basic		Intrinsics		Handwritten	
Sum	Product	Sum	Product	Sum	Product	Sum	Product
1.5	1.5	81	124	15	70	7.2	23

Table 2: Sum and product benchmarks on UltraSPARC III Cu: cycles per operation

Both floating-point sum and product benchmarks took 1.5 cycles per operation. The basic (compiler-generated) interval sum is 54 times slower than the floating-point sum and the interval product is 82 times slower. The interval intrinsics are significantly faster than the basic interval results; the sum is around five times faster, but the product is only twice as fast.

The performance of the handwritten interval codes is much better than the array intrinsics. Partly, this is because the Sun code implements an extended interval arithmetic [16] that guarantees containment for operations on infinite intervals such as $[-\infty, 0]$. Alterations are required to the handwritten code if it is to guarantee containment for infinite intervals; however, these intervals do not occur in the example application (see §4). For the sum, analysis of the assembly code suggests that much of the performance improvement is because the handwritten code prefetches data, which masks the latency of loads. For the product, the improvement is largely due to the use of min/max as opposed to branching for interval multiplication.

3.1 Improved hardware support for interval multiplication

While the min/max product runs faster on the UltraSPARC than the compiler's branching implementation of interval multiplication, it is still not optimal. Currently eight floating-point multiplications are required, corresponding to the four combinations of operand endpoints, rounded upwards and downwards. However, it should only be necessary to perform four full floating-point multiplications, where the architecture provides a means to check whether the previous operation was rounded.

An 'outward-rounded multiplication' machine instruction could be developed to accept two floating-point operands, and return *two* floating-point results rounded in opposite directions. For example, first calculate the upward-rounded result. If the floating-point status indicates that rounding occurred, then the downward rounded result is one ulp (unit in the last place) less than the upward-rounded result, i.e. subtract binary one from the last place in the significand, and carry as required. Otherwise, if the floating-point status indicates that no rounding occurred, then the multiplication was exact and the upward- and downward-rounded results are the same.

While such an instruction would reduce the number of multiplication instructions required to implement interval multiplication, further changes are required to significantly improve the throughput of interval multiplication on the UltraSPARC and similar architectures. This is an area of current research.

Stine and Schulte[14] have proposed an interval multiplier that would avoid branches in most interval multiplications, thereby greatly increasing the speed. Their *combined interval and floating-point multiplier*

would incorporate additional control logic for switching rounding modes, and multiplexors to determine which endpoints to multiply together. However, because this scheme requires significant changes to existing hardware, it is unlikely to be implemented for widely-used architectures in the near future.

3.2 Dot product and GEMM

Sun provides interval versions of more complex subroutines, for example DOT_PRODUCT and GEMM (general matrix multiply). Based on the methods described above, handwritten versions of these operations were constructed, and the performance compared against the Sun interval routines.

Table 3 shows the times for Sun's floating-point and interval intrinsic DOT_PRODUCT function, and for the handwritten interval function.

Floating Point Intrinsic	Interval	
	Intrinsic	Handwritten
3.0	97	16

Table 3: Dot product benchmarks on UltraSPARC III Cu: cycles per operation

It is noteworthy that the handwritten dot product performs better than the simple product; this is because the product carries a dependency between each multiplication that limits loop unrolling (the partial product Π_n depends on the value of Π_{n-1}) whereas the dot product does not.

The product C of two matrices A and B is a matrix defined by:

$$C_{ik} = \sum_j A_{ij} B_{jk}$$

It appears from this definition that, for the common case where $\dim(i) = \dim(j) = \dim(k)$, matrix multiplication is an $O(N^3)$ operation. In fact, optimizations exist that reduce the complexity to $O(N^{2.8})$ [15] or as low as $O(N^{2.376})$ [5]. As well, optimizations exist for sparse, symmetric, triangular and other special classes of matrices[6]. However, for the purposes of benchmarking interval matrix multiplication, the $O(N^3)$ general matrix multiply was chosen.

Several equivalent formulations of the matrix product exist[9] the dot product formulation was chosen for the handwritten code. Table 4 gives the times for the handwritten routine as well as the DGEMM (floating point) and GEMM.I (interval) optimized interval subroutine from the Sun Performance Library. The definition of an 'operation' changes here: because this formulation of matrix multiplication is an $O(N^3)$ operation, the total runtime was divided by N^3 for a $N \times N$ matrix, thereby giving the time for each pair of add and multiply operations.

Floating Point SunPerf Library	Interval	
	SunPerf Library	Handwritten
1.2	62	18

Table 4: GEMM benchmarks on UltraSPARC III Cu: cycles per operation

The handwritten interval code takes 15 times longer than the Sun Performance Library floating-point subroutine to multiply matrices of the same size. The SunPerf interval routine is three times slower again.

4 Pairwise electrostatic interactions

The remainder of this paper will demonstrate the utility of interval methods in scientific computation, through application to an example problem that arises in computational chemistry. The problem is that of calculating the potential energy of a system of charged particles under electrostatic forces. Given N particles with charge q_n at locations \mathbf{R}_n , and Coulomb force constant k , the total potential energy of the system is calculated as

$$PE = \sum_{i=1}^N \sum_{j=i+1}^N \frac{kq_i q_j}{|\mathbf{R}_i - \mathbf{R}_j|}$$

From this definition it is apparent that exact calculation of the potential energy is of $O(N^2)$ complexity, making direct pairwise evaluation impractical for large numbers of particles. Approximate methods exist that will calculate potentials for large systems to an arbitrary level of accuracy, most notably Greengard & Rokhlin's Fast Multipole Method[7] which scales as $O(N)$. However pairwise evaluation may still be used where the potential calculation is part of a larger computation, other parts of which scale as $O(N^2)$ or worse.

Because the potential varies inversely with the distance between particles, some particle interactions are liable to be orders of magnitude greater or smaller than others. This suggests that rounding errors are likely to occur in summing the individual potential energies, where the terms differ in magnitude[9]. Interval analysis was used to determine to what extent the accuracy of the final result is affected by the summation method used. Systems of particles were constructed where the particles were assigned random locations within a cube of side length 1. To simplify the analysis, each particle was assigned an equal charge of 1 (i.e. $q_i = q_j = 1$).

Results for two codes will be discussed. For the first, simple recursive summation was used, i.e. the interactions were simply summed in the order of evaluation. In the second, a method of summation similar to that proposed by Wolfe[17] was used, in which terms of similar size are accumulated into partial sums, which are then summed from smallest to largest. A floating-point implementation of both methods was constructed: using Sun's interval support, this was converted to use interval arithmetic by changing the type of physical quantities from double-precision real to double-precision interval. Both interval implementations were developed by replacing floating-point operations with equivalent interval operations, therefore the interval result is guaranteed to contain the true floating-point result.

The uncertainty in the result obtained by each method of summation (recursive and accumulator) is plotted in Figure 1. The terms used in this graph demand some explanation.

- *Interval error* is half of the width of the interval result, or alternatively the distance from the midpoint of the interval to either endpoint.
- *Nominal FP error* is the distance between the floating-point result and the furthest endpoint of the sharpest interval result that was obtained. This is the maximum possible error of the floating point result, given that the true result falls somewhere within the interval.² The actual error – the difference between the floating-point result and the exact result – is not known, because to ascertain the exact result requires infinite precision arithmetic.

All values plotted are relative: the nominal floating-point errors are divided by the absolute value of the interval midpoint to give a relative nominal error, and the interval results are divided by the lower bound of the absolute value of the interval to give an upper bound on the relative interval error.

The graph shows two important results. Firstly, the interval obtained by the code using the accumulator method of summation is sharper than that obtained by the code using the recursive summation method. This is equivalent to a proof that a floating-point code using the accumulator method has a smaller *possible*

²It may be argued that any floating point result that falls outside the interval bounds is just simply wrong, and therefore any metric of the error is misleading.

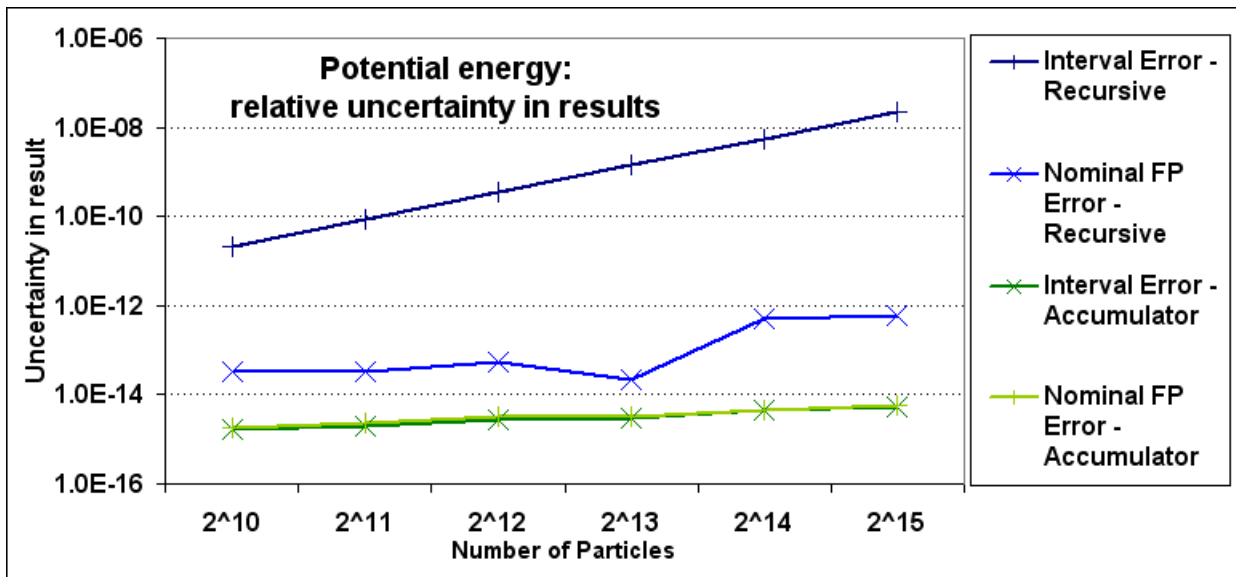


Figure 1: Accuracy of different methods of calculation of potential energy

error range than using recursive summation. Secondly, the nominal floating-point error for the recursive code is greater than the interval error for the accumulator method; the floating-point result obtained by recursive summation does not fall within the interval obtained by the accumulator method. This means that the floating-point result obtained through recursive summation is in error by *at least* the difference between this result and the nearest endpoint of the 'accumulator' interval.

In this simulation, the location and charge of each particle was taken to be exact, i.e. these values were represented by real numbers rather than intervals. In a practical application it may be necessary to represent uncertainty in physical values by representing them as intervals.

5 Conclusions

Sun's compiler support for interval computing aids ease of use, but fails to achieve the best possible performance on UltraSPARC because it compiles interval operations to function calls and uses branching multiplication. With little effort it is possible to write optimized interval subroutines to close the gap between the performance of interval and floating-point computation. Minimal change to existing hardware could greatly improve performance of interval multiplication on UltraSPARC and similar in-order machine architectures.

Through the use of interval arithmetic in the simple example of potential energy calculation, knowledge is obtained about the behaviour of rounding errors that is not available when using conventional floating-point arithmetic. Error bounds are computed concurrently with the target quantities, without the need for specialised knowledge of error analysis other than the inherent properties of intervals. This suggests that interval analysis provides a powerful method by which specialists in other fields, such as the physical sciences, can more easily analyse the error behaviour of their calculations. As the performance of interval implementations improves, this will make the use of interval methods for high performance scientific computation increasingly attractive.

6 Acknowledgements

The authors wish to acknowledge the contributions of Bill Clarke in many helpful discussions. Alistair Rendell acknowledges support from Australian Research Council grant DP0558228.

Bibliography

- [1] C++ interval arithmetic programming reference, Sun Studio 10. Tech. Rep. 819-0505-10, Sun Microsystems, Inc., Jan 2005.
- [2] Fortran 95 interval arithmetic programming reference, Sun Studio 10. Tech. Rep. 819-0503-10, Sun Microsystems, Inc., Jan 2005.
- [3] Sun Performance Library reference manual, Sun Studio 10. Tech. Rep. 819-0497-10, Sun Microsystems, Inc., Jan 2005.
- [4] AKKAŞ, A., SCHULTE, M. J., AND STINE, J. E. Intrinsic compiler support for interval arithmetic. *Numerical Algorithms* 37 (2004), 13–20.
- [5] COPPERSMITH, D., AND WINOGRAD, S. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing* (New York, NY, 1987), ACM Press, pp. 1–6.
- [6] DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., AND HANSON, R. J. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software* 14, 1 (Mar 1988), 1–17.
- [7] GREENGARD, L., AND ROKHLIN, V. A fast algorithm for particle simulations. *J. Computational Physics* 73, 2 (1987), 325–348.
- [8] GUSTAFSON, J. ASCI should require intervals. Email correspondence, May 1999.
- [9] HIGHAM, N. J. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2002.
- [10] IEEE COMPUTER SOCIETY STANDARDS COMMITTEE. WORKING GROUP OF THE MICROPROCESSOR STANDARDS SUBCOMMITTEE, AND AMERICAN NATIONAL STANDARDS INSTITUTE. *IEEE standard for binary floating-point arithmetic*. ANSI/IEEE Std 754-1985. IEEE Computer Society Press, Silver Spring, MD, 1985.
- [11] LERCH, M., TISCHLER, G., AND VON GUDENBERG, J. W. filib++ - interval library specification and reference manual. Tech. Rep. 279, Lehrstuhl für Informatik II, Universität Würzburg, 2001.
- [12] MOORE, R. E. Automatic error analysis in digital computation. Tech. Rep. LMSD-48421, Lockheed Missiles and Space Company, Jan 1959.
- [13] MOORE, R. E. *Interval Analysis*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1966.
- [14] STINE, J. E., AND SCHULTE, M. J. A combined interval and floating point multiplier. In *Proceedings of the 8th Great Lakes Symposium on VLSI* (Los Alamitos, CA, 1998), no. 98TB100222, IEEE Computing Society, pp. 208–215.
- [15] STRASSEN, V. Gaussian elimination is not optimal. *Numerische Mathematik* 13 (1969), 354–356.
- [16] WALSTER, G. W. Sun Studio white paper: Implementing the “simple” closed interval system. Tech. rep., Sun Microsystems, Inc., 2002.
- [17] WOLFE, J. M. Reducing truncation errors by programming. *Communications of the ACM* 7, 6 (1964), 355–356.