Performance Degradation in the Presence of Subnormal Floating-Point Values

Orion Lawlor, Hari Govind, Isaac Dooley, Michael Breitenfeld, Laxmikant Kale*
Department of Computer Science
201 N Goodwin, Urbana, IL 61801
University of Illinois at Urbana-Champaign
olawlor@acm.org,{gpnthpnc, idooley2, brtnfld, kale}@uiuc.edu

Abstract

Operating system interference in parallel programs can cause tremendous performance degradation. This paper discusses the interference caused by the quiet generation of subnormal floating point values. We analyze the performance impact of subnormal values in a parallel simulation of a stress wave propagating through a three dimensional bar. The floating-point exception handling mechanisms of various parallel architectures and operating systems lead to widely differing performance for the same program. We show that a parallel program will exhibit greatly amplified performance degredation due to this interference. In addition we provide an simple example program that demonstrates underflow on a single processor. Finally we suggest a novel option for fixing these undesired slowdowns.

1. Introduction

It is a general belief that floating point underflows are not very common in real applications [7]. Large numbers of subnormal values which can greatly slow down the computation on a processor designed with the assumption that their occurrence would be rare. The reason for such a slow down may not be obvious to an application programmer who is not usually concerned with the nuances of floating point computation, including how each operating system and compiler handles subnormal values. Here we investigate the effect of subnormals in a real parallel simulation as well as a simple serial program. We show that the occurrence of subnormal values is not rare, and that their existance can significantly degrade performance.

The IEEE 754 Standard specifies a standard method for implementing floating-point values and operations. It does not, however, suggest any exact implementation details. Systems using the standard may implement it entirely in hardware or software or in a combination of both. Often the common cases are handled in hardware, but traps to software cause the operating system to handle rare cases. Performance penalties vary greatly across architectures and operating systems due to different ways the rare floating-point cases such as underflow are handled. Furthermore, even different processors in the same architecture family can exhibit large variations in their handling of subnormal floating point values.

Subnormal floating-point numbers are the class of smallest floating-point numbers, with magnitudes from approximately 2^{-149} to 2^{-126} for single-precision and 2^{-1074} to 2^{-1022} for double precision. The IEEE 754 standard for floating-point numbers defines a denormalized, also called a subnormal, value as "A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero" [6]. Since all subnormal values have the same exponent, the smallest subnormal value will have much lower accuracy than the larger subnormal values. Thus an unacceptable loss of precision may occur when using subnormal numbers. The processor must notify the program of the loss of precision in some manner via a software trap or other mechanism. Also modern processors may not be designed to accellerate the handling of these and other supposedly rare occurrences. Different processors may handle subnormal values directly in pipelined hardware, escape out to a microcode handler, or even issue a trap to the OS.

2. Parallel simulation

Our example parallel program simulates a *1D* wave propagating through a finite *3D* bar. The bar is initially at rest with zero displacements and zero stresses. A velocity is imposed at one end and this produces, as is evident from the solution of the displacement equation of motion,

*This material is based upon work supported by the NSF under Grants NGS 0103645 and DMR 0121695, and by the DOE under grant B341494.

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{C_d^2}\frac{\partial^2 u}{\partial t^2},$$

a one-dimensional stress wave propagating at the dilatational wave speed C_d with a magnitude of

$$\sigma_x = -\rho C_d V,$$

where ρ is the density via the displacement equation of motion. Note that the stress at a point in the bar is theoretically zero until the stress wave reaches it and has a value of σ_x thereafter. Numerically it is impossible to capture the step function perfectly which results in numerical artifacts before the wave reaches the point (round-off) and after the wave passes (overshoot). An initial stress can not simply be applied everywhere in the bar unless the full solution of the equilibrium state is given. This would involve a static solution of the stress state for which the dynamic code is not set-up to solve.

The unstructured tetrahedral mesh that spans the bar is initially partitioned into chunks using METIS [10] and these chunks are then mapped to 32 processors. Subnormal values are generated when the stress wave travels along the bar. The processors advance in lockstep with frequent synchronizing communication. This wave of subnormal values overloads the processor or processors currently hosting the wavefront, since traps to the operating system will occur for each subnormal value. The load imbalance caused by them further compounds the performance problem. The processors advance in lockstep with frequent synchronizing communication.

We now give a general analysis of the impact of the slower operations due to subnormals in a parallel context. Let f be the fraction of operations which are impacted by subnormal operands throughout the computation. Assume each computation involving a subnormal takes k times longer than it would with a normalized number. Assume also that N operations are performed, each taking one unit of time. Then the serial program's runtime is N when no subnormals are present, and kN when all denormalized numbers are present. In general the serial case has runtime $N\left(kf + (1-f)\right)$. In practice k may greatly exceed 100 on some processors, since a software trap may take over 100 times as long as a floating point operation.

In parallel the effect of the slow operations on subnormals is greatly amplified. Assume there are p processors. The runtime for the parallel program without any subnormals is $\frac{N}{p}$ and it is $\frac{kN}{p}$ when any processor has all subnormals. Let f' be the maximum fraction of subnormals on any processor. The runtime is then $\frac{kf'N+N\left(1-f'\right)}{p}$. It is significant that the entire simulation can be held up by one processor, and if k is large, large amounts of idle time will exist on processors with few subnormals. Assume exactly one processor has all the subnormal numbers and that it does not have any normalized values. Then the average utilization over all processors for the run is $\frac{p+k-1}{kp}$. Suppose in some program $k = 50$ and $p = 32$, the average utilization of the processors is just over 4%. If $k = 250$ and $p = 128$, then the average processor utilization is about 1%. Ideally, a load balanced parallel application will have utilization slightly less than 100%.

3. Parallel simulation on AlphaEV6.8CB processor cluster

Initially the wave propagation simulation was carried out on an Alpha cluster without using any special compiler flags or architectural modifications. These computations were performed on Lemieux, a National Science Foundation Terascale Computing System at the Pittsburgh Supercomputing Center. The Alpha processor by default, flushes the subnormals to zero.

Figure 1 shows the overall utilization of 32 processors during the simulation. The *y*-axis denotes the utilization, while *x*-axis, the wall clock time. When the overall utilization is high it implies that most or all of the processors are being well utilized. On the other hand a lower utilization implies that many processors are idle for a portion of the time. The flat portion of low utilization at the beginning is due to the serial partitioning of the mesh on processor 0 while all other processors are idle.

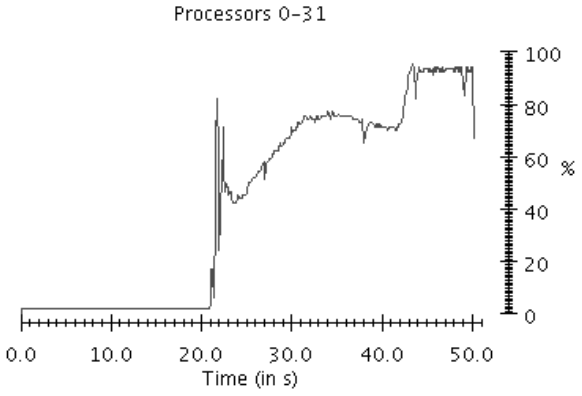

Figure 1. Overall processor utilization on the Alpha cluster which flushes all subnormals to zero in default mode.

Figure 2 gives an overview of the utilization for each processor throughout the entire simulation at any instant. Corresponding to each processor is a row of different colored bands indicating processor utilization over time. White represents 100% utilization while black represents 0% utilization. The *x* axis for the overview is wall-time, just as in

figure 1. The top row represents processor 0 while the bottom row represents processor 31. There are 32 horizontal bands, corresponding to the 32 processors used. Henceforth we shall refer to such figures succinctly as just ”overview”.

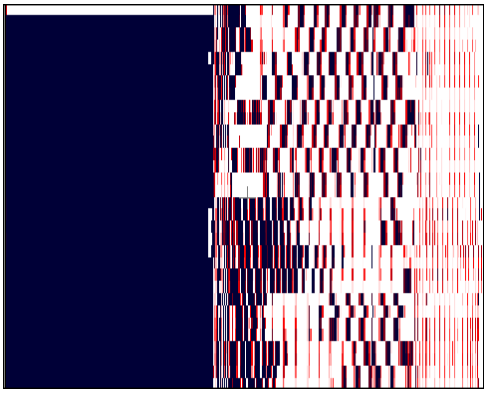

Figure 2. Processor utilization overview on Alpha processors when flushing subnorms to zero.

Processor zero is heavily loaded in the beginning(indicated by the white band on the top), since the initial mesh is partitioned on that processor. After partitioning each processor is assigned a chunk.

The simulation was repeated again enabling denormal numbers instead of flushing them to zero. This results in a considerable slowdown by a factor of nine. This is due to the fact that we used the nondefault compiler option “-fpe1”, causes traps to software when subnormal numbers are generated. This compiler flag is not very obvious to users who are not aware of the Alpha’s default flush to zero behavior. Figure 3 shows the much slower simulation with denormalized numbers enabled. The overall utilization is significantly lower, and only a few processors have high utilization at any time. To accurately detect the existence or impact of subnormal values on this platform, the different ”-fpe*” compiler flags can be used.

To more clearly demonstrate the phenomenon that causes the problem, we partition the bar linearly along the length into rectangular slabs. Under this partitioning, the wave originates at processor 31 which hosts chunk number 31 and travels in decreasing order through all chunks and processors until it reaches processor zero. The figures 4 and 5 clearly illustrate the overloading caused by subnormals generated when the wave front is on a particular processor. The subnormal and stress waves traverse together, the stress wave generating the subnormal numbers. Observe that the execution time is even longer than with the original METIS partitioning. The new linear partitioner is faster than the more complicated topological METIS based partitioner, but results in a poor decomposition.

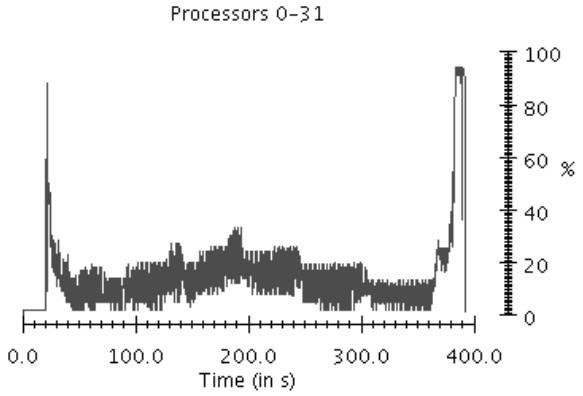

Figure 3. Utilization graph when flush to zero is disabled leading to software traps.

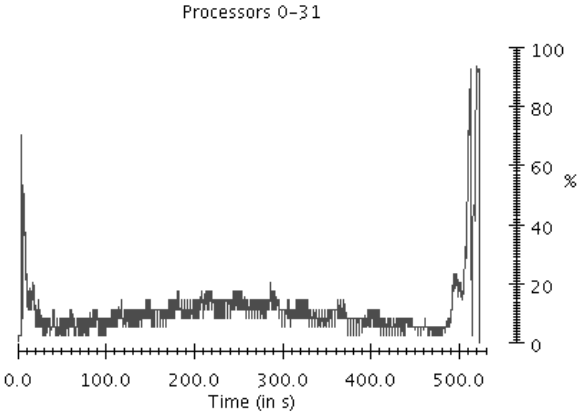

Figure 4. Overall processor utilization with linear partitioning along the bar.

The overview in figure 5 clearly illustrates the overloading caused by subnormals generated when the wave front is on a particular processor. Since each processor can only progress in timestep with the others, one processor is busy while all the rest are mostly idle. Clearly this is an undesired performance hindrance. In contrast, in sequential computation, the impact of the subnormal values will not hinder the progress of any additional processors. Table 1 gives a summary of the results.

4. Parallel simulation on Intel Xeon processors

The same simulation was also performed on Tungsten, an NCSA Xeon Linux Cluster. The Intel Xeon processor, unlike the Alpha processor, enables underflow to subnormal numbers by default. This could lead to poor perfor-

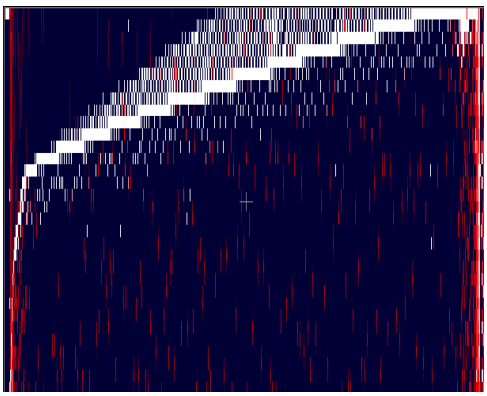

Figure 5. Overview with linear partitioning shows a wave of denormals that overloads individual processors.

Table 1. Summary of execution times on Alpha cluster.

Processor mode	Execution time
Default(flush to zero)	50.3s
Subnormals enabled (-fpe1)	392.4s
Linear partitioning	522.6s

mance compared to the cases where they can be flushed to zero without harm, but the default conforms to the IEEE 754 standard.

Intel Pentium 4 and Xeon processors internally store double precision values in their register stack in a higher precision format called double extended-precision [9]. Thus a subnormal value may exist in a processor without causing a trap until the value is written out to memory, at which point it is converted to a standard double precision 64-bit value. One problem with this method is that a user cannot test for the presence of denormalized numbers without them being converted to the 64-bit equivalents. Thus intermediary subnormal values may arise, although they will never hinder the performance of a program until they are written out from a floating point register. Simple methods for detecting the denormalized numbers may lead to false positive decisions that the denormalized numbers are the cause of other performance problems. To accurately detect the impact of these values, a compiler based method must be used.

Figure 6 illustrates the default behavior of the parallel simulation on the Xeon cluster. The Intel Pentium 4 and Xeon processors have a specific mode to enable flushing subnormals to zero. After enabling the FTZ mode, the simulation is much faster. Figure 7 shows the corresponding utilization. The simulation is nearly twice as fast with the FTZ mode as it was with the gradual underflow mode, as observed by the corresponding execution times in the first two rows of Table 2.

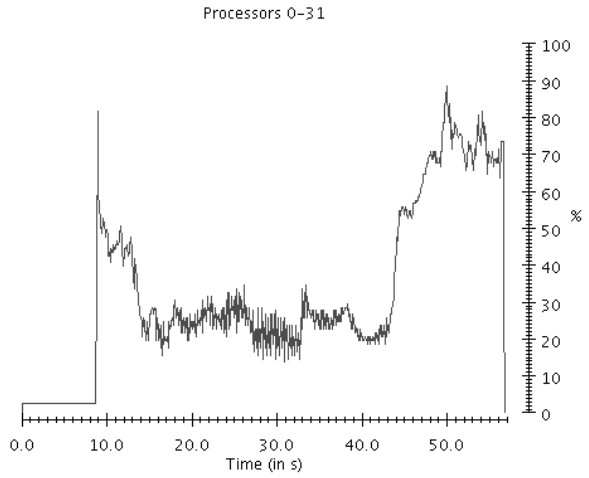

Figure 6. Overall utilization graph from the Xeon cluster.

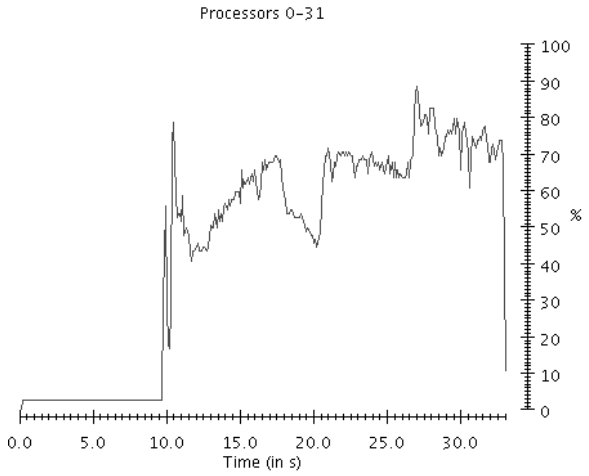

Figure 7. Utilization on Xeon cluster with the flush to zero enabled.

5. Effects on a single processor

We analyzed the impact of subnormal values in a serial program to assess the slowdowns which might become amplified in a parallel program. Tables 3 and 4 show the summary of a large number of exhaustive tests for commonly

Table 2. Execution times on Xeon cluster.

Processor mode	Execution time
Default(Subnormals enabled)	56.8s
Flush to zero mode	33.1s
Load Balancing(Subnormals enabled)	45.6s

available processors, compilers, and compiler flags. When available, the compilers included gcc, icc, icpc, and xlc, and its variants gxlc and xlc_r. The compiler flags "-O", "-O2", "-O3", and "-ffast-math" were each tested. Our aim was to analyze how badly subnormal values can impact a simple program on some common modern processors.

The test program is a simple Gauss-Seidel type relaxation, which exhibits performance degredation similar to standard Jacobi or Stencil examples. These types of applications are common. Our test program repeatedly iterates over an array of double precision values, replacing them with an average of its previous value and its neighboring values. Our simple serial program illustrated by the code below is a simple example that portrays characteristics of other common real-world examples including our parallel wave propagation code. The full code is available at our website [4]. This code will cause many subnormals to be created. As a comparison, we run the same code with different initial values, namely 1.0 and 2.0 instead of 0.0 and 1.0 for the array a[i].

```
for (i=1; i<ARRAYLEN-1; i++)
    a[i] = 0.0;
a[0] = a[ARRAYLEN-1] = 1.0;
for (j=0; j<ITER; j++)
   for (i = 1; i<ARRAYLEN-1; i++)
      a[i]=(a[i-1]+a[i]+a[i+1])/3.0;
```

Tables 3 and 4 do not give the relative performance between the different compiler options, but rather only looks at the slowdown that occurs when compared to the same program with initial values that do not cause underflow. What can be seen here is that compiler choice is important for applications that demand high performance, especially when subnormal numbers appear.

The results shown in Tables 3 and 4 are significant because they demonstrate that applications on common modern processors may be drastically impacted by floating-point underflow to subnormal values. It is worth noting that the gcc flag "-ffast-math" does not eliminate the slowdown caused by subnormal numbers. It was interesting that greater slowdowns occurred when higher levels of optimization such as "-O3" were applied to the program. The slowdown ratio is significantly higher with certain optimization flags, because the optimized version without the subnormal numbers is significantly faster, while the subnormals tend to hamper the performance more with the optimization flags. Complete results are available on our website [4].

Table 3. Slowdown when subnormal values occur on a single processor, Worst case scenario.

Processor	Worst Case Compiler	slowdown
PPC 970(Apple G5)	xlc -O3	2.24
AMD Athlon-32	gcc-ffast-math	5.29
AMD Athlon-64	gcc-ffast-math	23.22
Pentium 4	gcc-ffast-math	125.04

Table 4. Slowdown when subnormal values occur on a single processor, Best case scenario.

Processor	Best Case Compiler	slowdown
PPC 970(Apple G5)	xlc -O2	1.57
AMD Athlon-32	icpc	0.94
AMD Athlon-64	gcc -O3	14.03
Pentium 4	icpc -O3	1.17

6. Suggestions for performance improvement

We now discuss some options for avoiding the huge performance hits that occur when subnormal numbers arise. Some of our suggestions are similar to those already noted in [2], [5], [3].

The easiest solution is to use compiler flags or architectures which cause all subnormal values to be flushed to zero, provided it does not alter the program behavior. Thus the future operations on these numbers will not trap. However, numerical solutions may lose some precision, which may or may not be acceptable.

In situations where the numerical solution relies upon the accuracy subnormal numbers provide, the simple solution provided above will not work. In these cases, conventional wisdom is to use a better set of initial values. This may not solve the problem, especially in cases where a difference between two numbers is calculated, perhaps as an approximation to a derivative. The difference itself may become a subnormal value. Thus shifting or mapping the range of initial values to larger values may not suffice. In some cases rewriting the numerical algorithms may eliminate the occurrences of subnormal values, and is outside of the scope of this paper, but is addressed in [2].

In parallel programs, the impact of the phenomenon described is amplified into a load imbalance problem. Such problems can be solved by using the load balancing features in a migratable object based parallel framework like Charm++ [8]. In order to apply load balancing [12] we use the technique of virtualization which creates larger number of objects than the number of processors.

When a processor gets overloaded it migrates some of its objects to an under-loaded processor, thus improving the overall utilization and running time. Figure 8 along with the last row of Table 2 illustrate the effect of applying load balancing to Xeon cluster, showing that load balancing can significantly improve processor utilization, without flushing to zero, when underflow arises. Similar results are observed on the Alpha cluster.

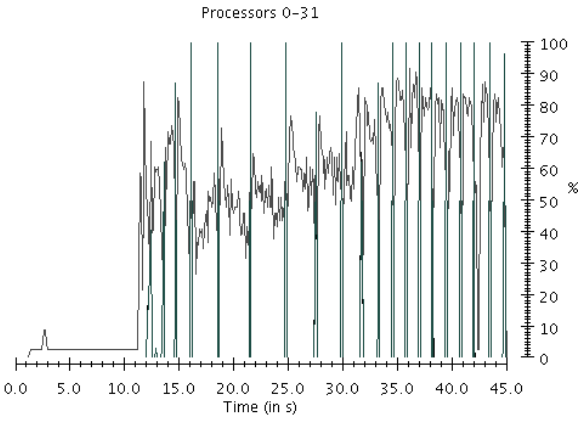

Figure 8. Utilization on Xeon cluster with load balancing. The vertical spikes correspond to load balancing steps.

7. Related work

Schwarz [11] discussed how the subnormal numbers can be implemented using small amount of additional hardware instead of handling them in software, which is usually done due to the complexity of the floating-point units required to handle them. In conclusion they state that using tagging and prenormalization, the new PowerFPU processor can execute instructions on subnormal operands with an additional overhead of very few cycles. However, our proposed solutions use only software based solutions. For the single processor case their solution can be more robust if our previous suggestions are not appropriate. Blackford [1] deals with challenges that exist when writing portable numerical libraries. They encounter some of the same issues and their suggestions vary from not using problematic software on particular architectures to additional programming.

8. Conclusion

In this paper we suggest a new technique that reduces the performance degradation associated with subnormal numbers in parallel programs, by load-balancing to remove load-imbalances. We have also shown that some real world applications exhibit behaviors that were previously thought to be rare or unavoidable. Our solutions require no hardware modifications. Further analysis and information may be found on our website [4].

References

[1] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. ScaLAPACK: A portable linear algebra library for distributed memory computers — design issues and performance. In *ACM Transactions on Mathematical Software*, volume 23, pages 133–147, 1996.

[2] J. Demmel. Underflow and the reliability of numerical software. *SIAM J. Sci. Stat. Comput.*, 5(4):887–919, 1984.

[3] J. W. Demmel and X. Li. Faster numerical algorithms via exception handling. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 234–241, Windsor, Canada, 1993. IEEE Computer Society Press, Los Alamitos, CA.

[4] I. Dooley. Subnormals in parallel programs. http://charm.cs.uiuc.edu/subnormal.

[5] J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems*, 18(2):139–174, March 1996.

[6] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, 1985.

[7] W. Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. 1996.

[8] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[9] *Intel Pentium 4 and Intel Xeon Processor Optimization*, 1999-2002.

[10] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high-performance scientific simulations. *Sourcebook of parallel computing*, pages 491–541, 2003.

[11] E. M. Schwarz, M. Schmookler, and S. D. Trong. Hardware implementations of denormalized numbers. In *ARITH '03: Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, page 70, Washington, DC, USA, 2003. IEEE Computer Society.

[12] G. Zheng. *Achieving High Performance on Extremely Large Parallel Machines*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.