Speeding up N-body Calculations on Machines without Hardware Square Root

ALAN H. KARP
IBM Scientific Center, Palo Alto, CA 94304

ABSTRACT

The most time consuming part of an N-body simulation is computing the components of the accelerations of the particles. On most machines the slowest part of computing the acceleration is in evaluating $r^{-3/2}$, which is especially true on machines that do the square root in software. This note shows how to cut the time for this part of the calculation by a factor of 3 or more using standard Fortran.

1 INTRODUCTION

Many phenomena in astrophysics and chemistry are being simulated using N-body methods [1, 2]. The most time consuming part of such simulations is computing the accelerations on each particle due to all the others. This is true for the simple N^2 methods, tree-based methods [3], or those using neighbor lists [4].

If the potential being used has an odd power of the particle separation in it, computing the orthogonal components of the acceleration will involve taking a square root. Although some machines do square root in hardware, many do not. It is not unusual to find that half the run time of an N-body calculation is spent in the square root subroutine.

In our case, we want to evaluate the acceleration on each particle in a system of self-gravitating bodies. For example, the x-component of the acceleration for particle j under the gravitational influence of particle k is

$$\hat{\mathbf{i}} \cdot \mathbf{a}_{jk} = \frac{Gm_k(x_j - x_k)}{r^3}$$

where $\hat{\mathbf{i}}$ is the unit vector in the x direction, G is the gravitational constant, m_k is the mass of particle k, and r is the separation between the particles.

$$r = \sqrt{(x_j - x_k)^2 + (y_j - y_k)^2 + (z_j - z_k)^2}$$

For efficiency, we usually code r^3 as $r^2\sqrt{r^2}$.

The system square root routine, not knowing how its result will be used, computes the square root with a divide-free Newton iteration to compute the inverse of the square root followed by a multiplication by the input value to get the final result. The compiler then multiplies by r^2 and divides the result into the numerator. Because both divisions and square roots are usually slow, this operation takes a long time.

Table 1 shows the time needed for some common operations—an empty loop, a simple assignment, division, square root, and $x^{-3/2}$. All times are in machine cycles per element measured on an IBM RISC System/6000 Model 540. Because RISC machines of this kind usually improve per-

Received October 1992
Accepted November 1992
Alan Karp's present address: Hewlett-Packard Labs 3U-7, 1501 Page Mill Road, Palo Alto, CA 94304. karp@hpl.hp.com

Scientific Programming, Vol. 1, pp. 133–140 (1992)
CCC 1058-9244/93/020133-08

Table 1. Time for Some Common Operations in Machine Cycles Per Element Run on an IBM RS/6000-540

m	Rolled	Unrolled*
Empty	2	2
assignment	4	4
x^{-1}	20	20
$\sqrt{x}$	56	56
$(x\sqrt{x})^{-1}$	77	77

* Unrolled refers to loops unrolled eight ways. Note that loop unrolling has no effect.

formance by pipelining arithmetic operations, unrolling loops frequently speeds things up. Clearly, the operations measured do not benefit.

It is possible to do considerably better than the direct computation. This note shows how to use standard Fortran to evaluate the acceleration in about one third the time taken by the direct evaluation.

2 ALGORITHM

The only way we can beat the efficiency of the system routines is to use our extra knowledge of the problem. In this case, we will not compute $\sqrt{r^2}$. Instead, we will compute r^{-3} directly from r^2.

The simplest approach is to approximate this function with a polynomial. Chebychev polynomials are frequently used because they minimize the maximum error of the approximation on some interval [5]. One difficulty is that the approximation is accurate only if the arguments are limited to a relatively small range. With a *range reduction* the procedure for an input argument r^2 becomes

1. Find a u such that $\alpha \le ur^2 < \beta$, where α and β are numbers of order unity.
2. Approximate $(ur^2)^{-3/2}$.
3. Get the correct result by multiplying the approximation by $t = u^{3/2}$.

Because I want to keep my algorithm entirely in Fortran, I decided to use two tables for u and t. The entries in u are simply the power of 2 such that $1 \le ur^2 < 2$. The entries in t are $u^{3/2}$. The only problem is to figure out which table entries to use for a given input value.

The range reduction I use is based on the IEEE double precision number format [6]. Each number consists of 64 bits—1 sign bit, 11 exponent bits, and 52 fraction bits. In addition, there is an implicit 1 bit for normalized numbers.

If I know the argument is positive, as it must be for the function I am interested in, I can extract the exponent by shifting the high order 32 bits of the floating-point number 20 bits to the right. In Fortran, this procedure requires that I EQUIVALENCE the double precision number to an integer or pass a double precision argument to a subroutine that uses it as an integer. Shifting the integer gives the index in the tables. Because there are only 11 bits to represent the exponent, I know my tables need only 2,048 entries.

I could have coded my tables as DATA statements in the program, but I decided to ask the user to make a single call to set them up as is frequently done with Fourier transform routines. The code to build the tables is contained in the program in the Appendix.

Now that I have scaled the input to a modest range, I can do the approximation. The coefficients of the fit are easy to compute [5]. If I write the approximation of $(ur^2)^{-3/2}$ as

$$f(x) \approx \frac{1}{2} c_0 + \sum_{k=1}^{m} c_k T_k(x),$$

where $x = 2(ur^2) - 3$, the coefficients c_k can be calculated from

$$c_k = \frac{2}{N} \sum_{j=1}^{N} f(x_j) T_k(x_j),$$

where the x_j are the zeros of $T_N(x)$, $x_j = \cos[\pi(j - 1/2)/N]$. The change of variable from (ur^2) to x is needed because the Chebychev polynomials are orthogonal on the interval $[-1, 1]$. If we choose $N \gg m$, the approximation will be very close to the minimax polynomial.

It is important to use knowledge of the hardware in writing the code. I made my runs on an IBM RISC System 6000 Model 540. RISC machines nominally do all operations in one machine cycle, but in practice complicated operations are pipelined. On this machine, all floating-point additions and multiplications are treated as compound multiply/add operations [7]. An isolated operation takes two cycles, but a sequence of operations produces one result per cycle after a delay of two cycles. Thus, our goal is to produce compound operations that can be pipelined.

Figure 1 shows part of a function that the user invokes to do the Chebychev fit. The input value is

```
equivalence (r2,ir2)
it = ishft(ir2,-20)
x = r2 * u(it)
x = 4.d0*x - 6.d0
t00 = 1.d0
s = c(1)
t01 = 0.5d0*x
s = s + c( 2)*t01
t02 = x*t01 - t00
s = s + c( 3)*t02
t03 = x*t02 - t01
s = s + c( 4)*t03
result = s*t(it)
```

FIGURE 1 Code to evaluate a four-term Chebychev fit. The input is `r2`. The tables `u`, `t`, and `c` were calculated in the setup routine.

`r2`. The statement EQUIVALENCE (`r2`, `ir2`) is needed because the shift function will only work on an integer argument.* Only four terms are shown, but the extension to more is obvious.

After some set-up code to do the range reduction and shift the arguments into the range of the Chebychev polynomials, all operations but the last are multiply/adds. Unfortunately, these operations are dependent on each other, so we are not making optimal use of the arithmetic pipeline. For example, the multiply/add that updates `s` cannot be started until the previous `t` is ready. Also, computation of the next `t` cannot be started until the previous one is done. However, we can overlap these two calculations so we expect each order of approximation to take an additional 3 cycles.

Table 2 summarizes the timing and accuracy results. The relative errors are measured using the direct calculation as the correct value. These errors are identical to the bounds computed by summing the absolute values of the dropped coefficients [5].

The times are given in machine cycles per element. In each case I measured the elapsed time with a clock accurate to a few nanoseconds. The times reported are the smallest of 20 runs of 10,000 random inputs. Although there are some anomalies, most of the time it takes 3 cycles to add one more order to the approximation. The anomalies are caused by running out of registers and the performance of the memory when loading the coefficients.

One way to improve the overlap is to do more than one evaluation on each pass through the loop, that is, unroll the loop. I experimentally determined that unrolling the loop eight ways gave me as much speed-up as I was going to get. The last column in Table 2 shows that adding one more term to the approximation costs less when the loop is unrolled, about 2 cycles per term.

Is it worth using this approximation? It depends on the accuracy needed. The time stepping scheme will have some truncation error. Clearly, there is no point making the function evaluation more than an order of magnitude more accurate than this value.

Table 2 shows that single precision (about 8 digit) accuracy with around 10 terms can be obtained at a cost of 28 cycles, a third the cost of the direct computation. If more accuracy is needed, almost 16-digit accuracy can be obtained if 20 terms are used, but the speed-up over the direct calculation is small.

Another approach is to use Newton's method. It is based on finding the roots of some function, in this case

$$f(y) = \frac{1}{y^2} - (r^2)^3.$$

The iteration is then

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} = \frac{1}{2}\, y_n(3 - (r^2)^3 y_n^2),$$

where n is the iteration index.

Newton's method is quadratically convergent when applied to a convex function such as the one in which we are interested [8]. This means that each iteration doubles the number of correct bits

Table 2. Summary of Measurements of Accuracy and Time for the Chebychev Approximation

m	Error	Rolled	Unrolled*
0	7.2×10^{-1}	8	6
2	3.1×10^{-2}	15	13
4	1.1×10^{-3}	21	17
6	3.8×10^{-1}	28	21
8	1.3×10^{-6}	33	25
10	4.2×10^{-8}	37	28
12	1.3×10^{-9}	42	33
14	4.2×10^{-11}	59	38
16	1.3×10^{-12}	60	43
18	4.1×10^{-14}	50	48
20	3.2×10^{-15}	53	50

* Unrolled refers to loops unrolled eight ways. Times are in machine cycles per element.

* "Strong typing is nice, but it shouldn't be invincible," N. L. Karp, private communication, 1985.

```
equivalence (r2,ir2)
it = ishft(ir2,-20)
x = r2 * u(it)
x3 = 0.5d0*x*x*x
s = d0
s = s*(1.5d0-x3*s*s)
s = s*(1.5d0-x3*s*s)
s = s*(1.5d0-x3*s*s)
s = s*(1.5d0-x3*s*s)
result = s*t(it)
```

FIGURE 2 Code for four Newton iterations. The arrays u and t were calculated in the setup routine.

in the result. We only need to get a reasonably accurate first guess. If we use the same range reduction as before, a reasonable first guess would be the function evaluated near the midpoint of the range. In fact, I chose to use a zero'th order Chebychev fit for the first guess.

Figure 2 shows the key part of the function invoked by the user. We see that this code will not use the hardware as effectively as the Chebychev code. Each Newton iteration has a multiplication, a multiply/add, and a final multiplication for a total of 6 cycles.

Table 3 shows the convergence and time for rolled and unrolled loops. If the loop is not unrolled, Newton's method takes 6 cycles per iteration as predicted: it takes only about 3 if the loop is unrolled. Single precision accuracy is achieved in about 29 cycles per element and double precision in 34 cycles per element.

Is it worth using Newton's method? Yes it is unless you need the last few bits correct. Without doing arithmetic in a higher precision the loss of a few bits of accuracy is inevitable. However, the simplicity of the code and its speed are in its favor.

Table 3. Summary of Measurements of Accuracy and Time for the Newton Method

n	Error	Rolled	Unrolled*
1	5.2×10^{-1}	17	14
2	3.3×10^{-1}	23	16
3	1.4×10^{-1}	30	20
4	3.0×10^{-2}	36	24
5	1.3×10^{-3}	42	26
6	2.7×10^{-6}	48	29
7	1.1×10^{-11}	54	32
8	4.3×10^{-16}	60	34

* Unrolled refers to loops unrolled eight ways. Times are in machine cycles per element.

We can do considerably better by making two changes to the Newton's method code. First of all, note the small improvement in the first few iterations. A better first guess would reduce the number of Newton iterations dramatically. I chose a six-term Chebychev fit that results in single precision accuracy with one Newton iteration and double precision with two.

There is another trick that can be used if only six terms are to be used in the polynomial approximation—compute the coefficients of the powers of x. This approach is not recommended in general because of the potential round-off errors when the coefficients are combined. Here there is no need to worry because the Newton iteration will tolerate such errors. The monomial coefficients are

$$\begin{aligned} d_0 &= \tfrac{1}{2}c_0 - c_2 + c_4 \\ d_1 &= c_1 - 3c_3 + 5c_5 \\ d_2 &= 2c_2 - 8c_4 \\ d_3 &= 4c_3 - 20c_5 \\ d_4 &= 8c_4 \\ d_5 &= 16c_5 \end{aligned}$$

If we use Horner's rule to evaluate the approximation,

$$s = d_0 + x(d_1 + x(d_2 + x(d_3 + x(d_4 + xd_5))))$$

the polynomial evaluation is all multiply/adds. The key part of the code is shown in Figure 3.

Table 4 summarizes the performance results. We see that we get single precision accuracy in only 19 cycles and double precision in 23 cycles. Because this algorithm outperforms the direct evaluation by a factor of nearly 4, an N-body code

```
equivalence (r2,ir2)
it = ishft(ir2,-20)
x = r2 * u(it)
xx = 2.d0*x - 3.d0
x3 = 0.5d0*x*x*x
s = d0+xx*(d1+xx*(d2+xx*d3)))
s = s*(1.5d0-x3*s*s)
result = s*t(it)
```

FIGURE 3 Code for the hybrid method. A third order Chebychev fit and one Newton iteration are shown. The coefficients d and the arrays u and t were calculated in the setup routine.

Table 4. Summary of Measurements of Accuracy and Time for the Hybrid Method*

n	Error	Rolled	Unrolled
0	7.2×10^{-1}	9	7
1	1.6×10^{-1}	10	10
2	3.1×10^{-2}	12	11
3	6.0×10^{-3}	13	12
4	1.1×10^{-3}	16	13
5	2.1×10^{-4}	17	14
N1	6.6×10^{-8}	23	19
N2	6.6×10^{-15}	30	23

* The first six rows are the order of the Chebychev fit; the last two are the Newton iterations. Unrolled refers to loops unrolled eight ways. Times are in machine cycles per element.

using this approach should run considerably faster.

The referee pointed out one more trick that reduces the times in Table 4 by one cycle per element. The Chebychev polynomials are evaluated on the interval $-1 \le x < 1$ while we have done a range reduction to $1 \le ur^2 < 2$. The variable xx is used to do the required change of variables. Examination of the code shows that xx can be substituted into the polynomial approximation for s, the terms can be arranged, and the new coefficients precomputed. These new coefficients, call them g_k are related to the d_k by

$$\begin{aligned}
g_0 &= d_0 - 3d_1 + 9d_2 - 27d_3 + 81d_4 - 243d_5 \\
g_1 &= 2d_1 - 12d_2 + 54d_3 - 216d_4 + 810d_5 \\
g_2 &= 4d_2 - 36d_3 + 216d_4 - 1080d_5 \\
g_3 &= 8d_3 - 96d_4 + 720d_5 \\
g_4 &= 16d_4 - 240d_5 \\
g_5 &= 32d_5
\end{aligned}$$

Table 5. Coefficients of the Minimax Fit*

k	c	d	g
0	0.60800336	0.54441741	7.05452470
1	−0.30951280	−0.27227564	−14.85088557
2	0.06613347	0.11188678	14.70832310
3	−0.01321311	−0.04324377	−7.83703555
4	0.00254752	0.02038017	2.17094576
5	−0.00048043	−0.00768693	−0.24598174

* Three forms are given: c are the coefficients of the orthogonal polynomials; d are the coefficients of the monomials on $[-1, 1]$; g are the coefficients of the monomials on $[1, 2]$.

We now evaluate the polynomial

$$s = g_0 + (ur^2)(g_1 + (ur^2)(g_2 + (ur^2)(g_3 + (ur^2)(g_4 + (ur^2)g_5)))).$$

The complete subroutine, including the set-up code, is shown in the Appendix; the coefficients are shown in Table 5.

3 CONCLUSIONS

How can I beat the performance of a highly tuned system routine with Fortran code? Simple—I cheat.

I cheat in a number of ways. First of all, I evaluate the function directly rather than in pieces. Second, I cheat by not getting the last few bits right. Finally, I cheat by not doing any error checking. (I could take the absolute value of the input at the cost of one additional cycle.) However, the output value is accurate for any floating-point input. Very large input values produce denormalized results; very small input values produce floating point infinity as they should.

If your machine supports the IEEE double extended format [6], a format with at least 64 bits in the fraction that is usually reserved for data kept in the registers, you can get the last few bits right using a simple trick. Compute the array t in extended precision, but store it as two double precision numbers, `t(1,i)` and `t(2,i)`. Then the final scaling becomes `s*(t(1,i)+t(2,i))`. I could not check the accuracy because the RISC System/6000 does not support the double extended format, but the change added only 1 cycle per result to the time of the unrolled loop.

Is it worth the effort and worry to use this new approach? If your calculation is typical and spends 3/4 of its time evaluating the acceleration, speeding up this one line of code by a factor of 3 will cut your total run time in half.

ACKNOWLEDGMENTS

I would like to thank Vivek Sarkar for helping me understand the RS/6000 instruction scheduling, Rad Olson and Bill Swope for trying to convince me I could not beat the system functions (I love a challenge), and the referee for several good ideas.

APPENDIX 1

Sample Code

This Appendix contains the complete version of the code as measured. For the sake of space, I changed the loop unrolling from eight-way to two-way. Some points are worthy of note.

If working with single precision data, change the shift to `ishft(x,23)` because IEEE single precision only uses 8 bits for the characteristic. Also, the center for the arrays `t` and `u` should be at 127.

Some systems do not allow continuation of execution following an overflow. Because $r^{-3/2}$ will underflow and overflow at the boundaries of the floating-point arithmetic, adjust loop limits on these machines. However, make sure that the table is filled properly. Underflow should produce a true zero and overflow should produce floating point infinity.

```
c Approximate r**(-3/2) using Newton with Chebychev first guess
c
c Initialize arrays by calling with n = 0 on first call
c
      subroutine r32i ( a, r2, ir2, n )
      implicit double precision ( a-h, o-z)
      parameter ( ncheb = 200 )
      save g0, g1, g2, g3, g4, g5, c, t, u
c
c Calling sequence
c
c   a = output array
c  r2 = input array
c ir2 = input array to be used as an integer
c   n = length of input and output arrays
c
      dimension a(n), r2(n), ir2(2,n)
c
c Local variables
c
c f = temporary array needed to compute c
c z = temporary array needed to compute c
c c = coefficients of Chebychev approximation
c d = coefficients of monomial fit on -1 < x < 1
c g = coefficients of monomial fit on  1 < x < 2
c t = table of (2**k)**(-3/2) for -1024 < k < 1024
c u = table of 1/2**k for -1024 < k < 1024
c
      dimension t(0:2046),u(0:2046),c(0:ncheb-1),f(ncheb),z(ncheb)
c
c Function shifted from 1 to 2 to -1 to 1
c
      func(r) = (1.5d0 + 0.5d0*r)**(-1.5)
c
c If not first call, then compute function
c
     if (n .gt. 0 ) then
c
c Approximate results - loop unrolled 2 ways
c A greater degree of loop unrolling will probably perform better
```

```
c
         do i = 1, n, 2
            it = ishft(ir2(1,i ),-20)
            x = r2 (i) * u(it)
            x3 = 0.5d0*x*x*x
            s = g0 + x*(g1 + x*(g2 + x*(g3 + x*(g4 + x*g5 ) )))
            s = s*(1.5d0-x3*s*s)
c           s = s*(1.5d0-x3*s*s)           ! Use for double precision
            a(i) = s*t(it)
            it = ishft(ir2(1,i+1),-20)
            x = r2(i+1) * u(it)
            x3 = 0.5d0*x*x*x
            s = g0 + x*(g1 + x*(g2 + x*(g3 + x* (g4 + x*g5 ) )))
            s = s*(1.5d0-x3*s*s)
c           s = s*(1.5d0-x3*s*s)           ! Use for double precision
            a(i+1) = s*t(it)
         enddo
c
c Finish up unrolled loop
c
         do j = i, n
            it = ishft(ir2(1,j),-20)
            x = r2(j) * u(it)
            x3 = 0.5d0*x*x*x
            s = g0 + x*(g1 + x*(g2 + x*(g3 + x*(g4 + x*g5 ) )))
            s = s*(1.5d0-x3*s*s)
c           s = s*(1.5d0-x3*s*s)           ! Use for double precision
            a(j) = s*t(it)
         enddo
      else
c
c If the first call, build table of results for powers of 2
c
         xi = 1.d0
         t(1023) = 1.d0
         u(1023) = 1.d0
         do i = 1, 1023
            xi = 0.5d0*xi
            t(1023+i) = xi*sqrt(xi)
            t(1023-i) = 1.d0/t(1023+i)
            u(1023+i) = xi
            u(1023-i) = 1.d0/xi
         enddo
c
c Precompute zeros of Chebychev polynomials and function
c
         pi = 4.d0 * atan ( 1.d0 )
         do k = 1, ncheb
            z(k) = pi * ( k - 0.5d0 ) / ncheb
            zero = cos(z(k))
            f(k) = func(zero)
         enddo
c
```

```
c Get coefficients of Chebychev fit
c
         factor = 2.d0/ncheb
         do j = 0, ncheb-1
            sum = 0.d0
            do k = 1, ncheb
               arg = z(k) * j
               sum = sum + f(k)*cos(arg)
            enddo
            c(j) = factor*sum
         enddo
c
c Get coefficients of powers of x on -1 < x < 1
c
         do =  0.5d0*c(0) -       c(2) +      c(4)
         d1 =       c(1) -  3.d0*c(3) + 5.d0*c(5)
         d2 =  2.0d0*c(2) -  8.d0*c(4)
         d3 =  4.0d0*c(3) - 20.30*c(5)
         d4 =  8.0d0*c(4)
         d5 = 16.0d0*c(5)
c
c Get coefficients of powers of x on 1 < x < 2
c
         g0=d0-3.0d0*d1+ 9.0d0*d2-27.0d0*d3+ 81.0d0*d4- 243.0d0*d5
         g1=   2.0d0*d1-12.0d0*d2+54.0d0*d3-216.0d0*d4+ 810.0d0*d5
         g2=              4.0d0*d2-36.0d0*d3+216.0d0*d4-1080.0d0*d5
         g3=                        8.0d0*d3- 96.0d0*d4+ 720.0d0*d5
         g4=                                 16.0d0*d4- 240.0d0*d5
         g5=                                             32.0d0*d5
      endif
c
      end
```

REFERENCES

[1] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. New York: McGraw-Hill, 1981.

[2] J. A. Sellwod. *Annual Reviews of Astronomy and Astrophysics*, vol. 25. Palo Alto, CA: Annual Reviews, Inc., 1987, pp. 151–186.

[3] J. Barnes and P. Hut. "A hierarchical O(NlogN) force-calculation algorithm." *Nature*, vol. 324, pp. 446–449, 1986.

[4] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford, UK: Oxford University Press, 1987.

[5] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes*. New York: Cambridge University Press, 1986, pp. 147–151.

[6] American National Standards Institute, Inc. IEEE Standard for Binary Floating-Point Arithmetic. Technical Report ANSI/IEEE Std 754-1985. IEEE, 345 East 47th Street, New York, NY 10017, 1985.

[7] B. Olsson, R. Montoye, P. Markstein, and M. Nguyen Phu, *RISC System/6000 Floating-point Unit*. IBM, NY: RISC System/6000 Technology, 1990, pp. 34–42.

[8] J. M. Ortega, *Numerical Analysis*. New York: Academic Press, 1972, pp. 155–158.