RISC Microprocessors and Scientific Computing

David H. Bailey

RNR Technical Report RNR-93-004
March 26, 1993

Abstract

This paper discusses design features in currently available RISC microprocessors that result in less-than-optimal sustained performance on large-scale scientific calculations. Recommendations for future designs are suggested.

The author is with the Numerical Aerodynamic Simulation (NAS) Systems Division at NASA Ames Research Center, Moffett Field, CA 94035.

1. Introduction

Scientists accustomed to running large-scale, computationally intensive applications have traditionally utilized conventional vector supercomputers, such as those manufactured by Cray Research, Inc., Fujitsu or NEC. However, with the recent popularization of RISC workstations, many of these same scientists are using their workstations not just to edit their source codes and display their results, but to perform their computations as well. Another avenue from which scientific computer users have been introduced to RISC processors, an avenue which is potentially very significant for the future of scientific computing, is the recent incorporation of RISC processors into highly parallel supercomputers.

The reason for this interest is of course the recent sharp rise in the peak floating point performance of RISC processors, and the expectation of continued increases in the future. Indeed, on many scientific applications, particularly those that enjoy high levels of data locality, RISC-based systems already feature superior sustained performance per dollar, when compared with conventional vector computers. However, as we will see, not all applications run so well on these systems. This phenomenon is analogous to the situation in vector computing, where many scientific applications run quite well, but others do not.

RISC is an acronym for "reduced instructed set computer," a concept popularized a few years ago by researchers at Stanford University, among other places. In this paper, however, "RISC processor" will be used more loosely to designate any of the recently developed high-performance floating-point processors. Indeed, it will be seen that the issues raised herein do not have much to do with the basic concept of reduced instruction set computing, but instead deal with other features of these processors.

In the following, 64-bit data is assumed in all discussion of performance rates. Thus, for instance, Mflop/s denotes millions of 64-bit floating point operations per second, and Mword/s denotes millions of 64-bit words per second.

2. Divide and Conquer

Several features common to many current RISC processors result in less-than-optimal performance when systems incorporating these processors are used for scientific computations. One example of such a feature is the design of the divide operation.

The IBM RS6000 processor has an IEEE-compliant floating point divide operation, which employs a Newton iteration scheme and requires 20 clock periods. The Hewlett-Packard PA-RISC floating point divide also requires 20 clock periods. The MIPS R4400, which is used in the new Challenge series workstations from Silicon Graphics, Inc., requires 36 clock periods. Of currently popular RISC processors, it appears that only the Sun SuperSPARC can perform a floating point divide in under ten clock periods (see Table 1).

On the Intel i860, divisions are performed in software by means of a Newton iteration scheme. This sequence requires 38 clock periods. If a true IEEE-compliant quotient is required, an additional sequence of operations is required that increases this cost by a factor of five, to approximately 190 clock periods. The IEEE-compliant divide operation on the DEC Alpha processor requires 63 clock periods. It is not known if a reasonably accurate approximate result can be obtained in fewer clock periods on the Alpha.

It should be mentioned that the above figures are based on scalar, non-pipelined divide operations. Presumably the RISC processors could deliver improved divide performance if a series of divisions could be pipelined, in the same way that adds and multiplies are often pipelined. Regrettably, however, it appears that existing compilers on RISC systems do not exploit this possibility.

By contrast, the Cray-2, Cray Y-MP and C-90 processors can perform vectorized divisions at a rate of one result every four clock periods. One important disadvantage of the current Cray systems is that they do not perform IEEE arithmetic. Further, the results of the divide sequence are only guaranteed to 46 of 48 bits, and this fact results in some rather annoying anomalies (such as $7.0/7.0 \neq 1.0$). Most users of Cray systems have found these "features" acceptable, although there are some situations where Cray's arithmetic is problematic.

A processor of the NEC SX-3 can perform one division every two clock periods, in vector mode, and the results are fully accurate — they are not merely the product of the numerator with an approximation to the reciprocal of the denominator. On the other hand, one SX-3 CPU can perform eight floating point adds and eight floating point multiplies per clock period, so that its peak divide performance is 1/16 that of its peak add or peak multiply performance. While this ratio is significantly worse than on the Cray systems, it is still better than on most of the RISC systems.

How important is the performance of the divide operation in scientific computation? It is true that many important inner computational loops have no divide operations whatsoever. In most other cases, divisions can be moved out of inner loops, either by the programmer or by optimizing compilers. But what about loops where the divisors are not constant, and therefore cannot be moved out of the loop? And even in those cases where divide operations can be moved out of inner loops, unless the inner loop length is fairly large the cost of the single divide operation can still dominate the cost of the loop if the processor requires scores of clock periods to complete it.

An example of an important divide-intensive numerical algorithm is the tridiagonal solver of ADI schemes, which are used for certain partial differential equation problems. Of $9n$ floating point operations in the inner loop, n of these are floating-point divisions. Such considerations are not entirely academic. Colleagues of the author at NASA Ames Research Center, who developed and implemented the NAS Parallel Benchmarks [6, 7], have found that the slow performance of the divide operation on the Intel i860 processor significantly reduces the sustained performance of the iPSC/860 system on the block tridiagonal and scalar pentadiagonal benchmarks.

The lack of fast hardware for integer division may also become a serious issue when RISC processors are used in distributed memory parallel computers, a path now taken by Intel, Thinking Machines, Cray and others. Consider a one-dimensional array of length $7168 = 7 \times 1024$ mapped to a 1024 processor machine. Element A_j is located at offset $(j \bmod 7)$ on processor $(j \div 7)$. Hence the need for a fast integer divide operation. Partly for this reason, Cray has proposed a programming model for an upcoming highly parallel system that places somewhat artificial limits on array distributions [12].

3. The Cache Flow Problem

All of the current selection of high-performance RISC microprocessors employ a cache memory system and some scheme for virtual memory management. The objective is to provide the illusion of a very large memory, possibly extending to disk, all of which is readily accessible by the processor for high throughput performance. Bandwidth between this cache memory and the processing units is generally quite high. Bandwidth between main memory and the processing units typically is significantly lower.

The Intel i860, for example, has an eight Kbyte internal data cache. It can, in a single clock period, transfer 16 bytes, or two operands, between the internal cache and two adjacent registers. Concurrently with this operation it can perform floating point operations on data in other registers. With a clock rate of 40 Mhertz, the theoretical peak performance is 60 Mflop/s, and its cache memory bandwidth is 80 Mword/s. This ratio of roughly one operand per operation has been found to be acceptable for scientific computing. This is the ratio for the MasPar MP-2 and the vector processors of the TMC CM-5. Cray systems can fetch two operands and store one operand every clock period, during which time two floating point operations can be performed. Thus its ratio is 1.5.

The performance of the i860 on data in main memory is quite another matter. From main memory it can only load one 64-bit word every two clocks, or store one word every three clocks. These figures assume a long vector operation using a particular machine instruction. Conventional loads and stores from main memory require even more time. Thus there is a ratio of at least six between achievable cache and main memory bandwidth figures. This is a limitation of the processor and cannot be corrected with a secondary cache.

This limited main memory bandwidth is reflected in less-than-stellar performance of the i860 on real scientific codes. Figure 1 (solid line) gives performance rates of Fortran-coded DAXPY loops of different lengths (assuming the start of the data vectors are in cache). This code was compiled with the latest version of the Intel-supported Portland Group compiler, using O4 optimization. The performance is roughly 12 Mflop/s until the vector length is sufficiently large that the two vectors cannot be held entirely in cache, at which time it drops sharply to only about four Mflop/s. The dashed line in Figure 1 give performance rates with the Mvect option, which utilizes an instruction that bypasses cache. As can be seen, this results in improved long vector performance, but poorer short vector performance. Even with this option, long vector performance is only about 60% of the short vector, cached data performance.

For other types of Fortran loops typical of scientific application codes, the results are similar. For instance, on the NAS Parallel Benchmarks [7], the 128-node Intel iPSC/860 system typically achieves only about five percent of its peak multiprocessor performance. This is partly due to contention in the network, but the other principal factor is low single node performance, typically only four to six Mflop/s from Fortran or C. While further improvements in the compiler can be expected to marginally improve these rates, there seems little hope that sustained rates of more than eight to ten Mflop/s will ever be achieved on the main body of real scientific codes [11].

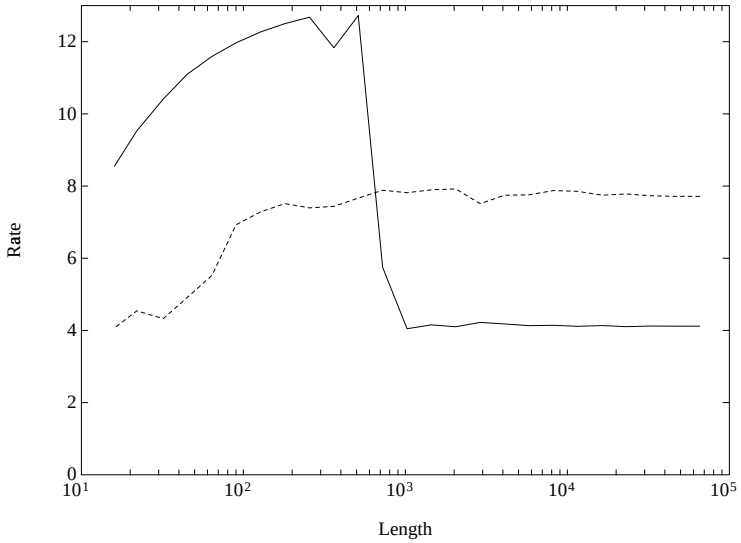


Figure 1: i860 Performance on a Fortran DAXPY Loop (Mflop/s)

The next edition of the i860, known as the i860XP, improves the main memory bandwidth by means of burst load and store operations. It remains to be seen how much actual improvement will be realized by the user, since it appears that the initial i860XP Fortran and C compilers are not able to exploit these special instructions.

The situation is somewhat better with other RISC processors. The IBM RS6000 series features a larger cache (64 Kbytes on the model 370). Here the ratio between internal cache bandwidth and main memory is unity (one operand per clock period), and both of these figures are one-half the floating point computation. Partly as a result of this excellent main memory bandwidth, its resulting Linpack 100×100 performance rate is a respectable 26 Mflop/s.

However, the four-way associative design of the IBM RS6000 cache produces some rather odd behavior with respect to strides. Programmers of cache-based systems are prepared for slowdown anytime the strides of memory accesses are not unity, since these result in poor cache utilization. But few programmers are aware that additional slowdowns result for certain strides such as 73 or 102. It turns out that such strides result in poor performance because they are very nearly simple fractions of 512 [5].

The Hewlett Packard PA-RISC processor runs at 99 Mhertz. It has a large cache, typically one Mbyte, and it can fetch or store one operand every clock period (after the first). However, data in main memory data can be fetched or stored at a rate of only one word every three clock periods. As with several of the other processors, the PA-RISC can perform up to two floating point operations every clock period.

The R4400 processor, which is used in some new Silicon Graphics workstations, runs at 100 Mhertz. It can fetch or store a operand from the internal cache every clock period. It has an external cache, typically one Mbyte, from which it can access a word every other

Processor	Clock (Mhz)	Divide (CP)	Cache BW (MW/s)	Memory BW (MW/s)	Peak Perf. (MF/s)	Linpack (MF/s)
Cray C-90/1	240	4		1440	960	387
Cray Y-MP/1	166	4		500	333	161
IBM RS/6000	62	20	62	62	125	26
DEC Alpha	150	63	150	37	150	30
HP PA-RISC	99	20	99	33	198	41
Intel i860	40	190	80	16	60	10
SGI (R4400)	100	36	100	50	50	17
SuperSPARC	40	7	40	40	40	

Table 1: Performance Characteristics of RISC Processors

clock period, which is the same rate for data in main memory. It can perform a floating point operation every other clock period, for a peak rate of 50 Mflop/s.

The 40 Mhertz Sun SuperSPARC processor has a 16 Kbyte internal primary data cache (four-way set-associative). The transfer rate from the primary cache is one word per clock cycle. Since three instructions may be issued simultaneously (floating point, integer, load/store), the theoretical peak performance is 40 Mflop/s, and the bandwidth between the cache and the CPU is 40 Mword/s. To improve memory performance, a direct mapped secondary cache of one Mbyte has been added.

Recently the "Alpha" RISC processor was announced by Digital Equipment Corporation. It features a full 64-bit design and a 150 to 200 Mhertz clock, depending on model. Since it can perform a floating point operation in every clock period, its theoretical peak performance is cited as 150 to 200 Mflop/s. However, in spite of its much faster peak speed, the size of the Alpha's primary cache is the same as that of the i860: only eight Kbytes.

Like other recently announced RISC processors, the bandwidth (and latency) to and from main memory for the DEC Alpha is significantly lower than that of the cache. The Alpha can load or store a word from main memory only about every four clock periods (peak rates). Thus the ratio between achievable internal cache and main memory bandwidths is at least four. The latency for main memory accesses on the Alpha is about 20 clock periods, which is more than six times that of cache accesses (three clock periods). In an attempt to deal with this disparity, the DEC Alpha, like some others, features support for a second-level external cache, typically consisting of about one Mbyte of SRAM. The peak rate for loads and stores from second-level cache is one word every two to three clock periods, and the latency is roughly eight clock periods.

This information is summarized in Table 1. It is interesting to note that the Linpack 100×100 performance of an individual system appears to be well estimated as one-half of the minimum of the memory bandwidth and peak performance figures. The only systems that achieve significantly greater rates than this estimate are the DEC Alpha and HP PA-

RISC processors. But these systems benefit from an external cache, which is sufficient to completely contain the Linpack 100×100 benchmark. Indeed, their Linpack performance figures are roughly half of the secondary cache bandwidth figures (60 Mword/s and 99 Mword/s, respectively). On larger benchmarks that exceed the size of the external cache, and which better reflect modern large-scale computing, the actual performance rates of these two processors are lower. For example, the performance of the Alpha on the PERFECT suite is 18 Mflop/s. Note that this figure is much closer to half of its main memory bandwidth figure (37.5 Mword/s). No Linpack 100×100 figure was available for the SuperSPARC; however, its Linpack 1000×1000 figure of 22 Mflop/s is in general agreement with this rule.

4. Dollars and Sense

One advantage of RISC processors over vector processors is that they obtain respectable performance on loops with apparent or real recursions, or with short loop lengths. It is well known that the presence of these features in a code significantly reduces the performance that a vector processor can achieve on it. However, the experience of most vector programmers is that these difficulties can usually be remedied without major revision to the code. On large two or three-dimensional applications, for example, it is often possible to avoid these difficulties by vectorizing in another dimension. Once this tuning has been done, the vector systems generally achieve rather high performance. On the NAS Parallel Benchmarks, for example, a Cray Y-MP processor achieves (with minor tuning) over 65 Mflop/s on all eight tests, and it exceeds 175 Mflop/s on all but two tests [7]. 175 Mflop/s is 53 percent of the theoretical peak rate of a Y-MP processor.

The principal reason that the Cray Y-MP generally sustains such a high percentage of its peak performance is that its main memory bandwidth is well matched to its peak computation rate. The total main memory bandwidth of the eight-processor Y-MP is 4 Gword/s, which is actually a greater figure than its peak computing rate, which is 2.6 Gflop/s. By comparison, the main memory bandwidth of the 128-processor Intel iPSC/860 system is 2 Gword/s, which figure is only about one fourth of its peak computation rate (7.6 Gflop/s). When the Intel main memory bandwidth figure is reduced by roughly half to account for interprocessor communication, the resulting ratio of four is roughly what is observed in overall system performance between these systems on the NAS Parallel Benchmarks [7].

On the other hand, when the price of systems is considered, the RISC systems do reasonably well, even when comparing main memory (i.e. not cache resident) performance statistics. The Intel iPSC/860 system, for instance, costs only about one-sixth that of an eight-processor Y-MP system. Thus the total main memory bandwidth per dollar of the Intel iPSC/860 is three times that of the Cray Y-MP. On the NAS Parallel Benchmarks, the sustained performance per dollar of the Intel is on a par with the Y-MP. Were it not for for other factors, such as the relatively immature state of compilers for the Intel system, and its limited interprocessor network bandwidth, it is likely that the Intel would surpass the Cray in this measure also. NAS Parallel Benchmark figures are not yet available for

parallel systems based on the DEC Alpha. But if we consider a workstation system, and take 18 Mflop/s as typical of sustained performance, it is clear that the Alpha workstation, which can be purchased for as little as $25,000, is considerably less than a tenth as costly as a Y-MP processor that sustains 180 Mflop/s

In short, it appears that even with the shortcomings addressed in this paper, RISC processor systems still deliver sustained performance per dollar that is now competitive with the vector systems on most applications, and much better on some. Nonetheless, one can ask whether their sustained performance rates (as well as their sustained performance per dollar figures) might have been even better had the designers placed greater emphasis on main memory bandwidth, as opposed to increasing the performance on data in the primary cache.

5. The World of Large-Scale Scientific Computing

Some large-scale scientific programs can be expected to run quite well on the current RISC systems. Clearly those that have been written from scratch on a cache-based workstation by expert programmers can be expected to run at somewhat higher performance rates than those ported from other systems without significant alteration. Another class of applications that will perform well on RISC-based systems, relative to vector systems, are those with loops that are not readily vectorizable, or those with short inner loops. Others that can be expected to do well are codes that utilize libraries, such as LAPACK [2], which have been highly tuned for cache-based systems, among others.

Some scientific codes will achieve excellent performance on RISC computer systems simply because they have the good fortune to consist of very well localized, and therefore cache-efficient, computations. For example, a vortex roll-up simulation code developed by the author [8] behaves very well in cache memories, because the computations involve high precision arithmetic operations that are naturally very well localized. By comparison, this code does not run efficiently on a Cray Y-MP, due principally to unavoidably short inner loop lengths. For these reasons, a multiprocessor implementation of this program on the Intel iPSC/860 system out-performs one processor of a Cray Y-MP when only eight i860 nodes are utilized. Thus on this application the Intel system achieves a much higher performance rate per dollar than the Cray.

Another fortunate class of applications are those based on dense matrix computations, which can be structured to run efficiently in cache systems [13, 14, 15]. Indeed, the impressive performance achieved by the LAPACK library on a variety of systems is due in large part to the utilization of these blocking techniques.

Many other important scientific applications are not so fortunate. For one thing, large-scale scientific codes are not often based on efficient, assembly-coded library routines. This is regrettable, since a significant number of them could utilize such libraries. Hopefully the current movement to standardize the calling sequences to such libraries will induce greater numbers of programmers to adopt them. Second, many large applications have previously run on vector systems, where inner loop vector lengths and strides are the principal performance issues. Data locality issues were simply not considered when these

codes were written. Third, quite a few large scientific codes employ advanced numerical algorithms that do not feature high levels of data locality, and no amount of superficial code manipulation will change this.

We have already discussed the impact of the divide operation on large-scale scientific applications. Now let us investigate the impact of the cache design on these codes. To that end, four types of numerical algorithms will be examined to see whether they can be expected to run well on cache-based RISC processor systems (1) as is, (2) after processing by intelligent cache blocking software tools that may be available in the future, (3) after significant revision by expert programmers, and (4) after the wholesale substitution of advanced, cache-efficient algorithms in important compute-intensive routines. In the following, by an "expert programmer", we will mean a programmer who is highly skilled in tuning scientific codes on cache-based computers and is generally knowledgeable in numeric techniques, but who does not necessarily have specialized expertise in state-of-the-art numerical algorithms.

6. Ordering Loops

The order in which loops are ordered can significantly affect the sustained performance of this code on a cache-based system. Consider, for example, the following code fragment:

```
      DIMENSION A(15,1000), C(15)
. . .
      DO 120 I = 1, 15
        S = 0.D0
C
        DO 110 J = 1, 1000
          S = S + J * A(I,J)
 110    CONTINUE
C
        C(I) = S
 120  CONTINUE
```

This code design makes good sense on a vector system, since the inner loop vector length is large and the stride of accesses in the inner loop is an odd number.

However, on a cache system this might not be the best approach, because of the nonunit stride accesses in the A array. It might be better to write this code as follows:

```
      DIMENSION A(15,1000), C(15)
. . .
      DO 100 I = 1, 15
        C(I) = 0.D0
 100  CONTINUE
C
      DO 120 J = 1, 1000
```

```
        DO 110 I = 1, 15
          C(I) = C(I) + J * A(I,J)
110     CONTINUE
120   CONTINUE
```

This code design features unit stride inner loops, which are definitely better for cache systems. The fact that the inner loop vector length is only 15 has no adverse consequence on a RISC system. On the contrary, this limited vector length is an advantage on the RISC system — the `C` vector is cache-resident and has a very high level of data re-use. Indeed, while the first variant runs nearly twice as fast than the second on a Cray Y-MP, the opposite is true on a node of the Intel iPSC/860.

In this case, it is possible that a "smart" compiler could perform this loop interchange. In other cases this is not likely, but a skilled programmer may well be able to make such changes to existing codes for improved performance on cache-based systems. However, it must be acknowledged that such revision requires significant effort and nontrivial expertise, and it may not be practical for very large "dusty deck" codes. Software tools, such as those now being developed at Rice University [9] and Stanford [16], may eventually be able to assist in this effort. But effective changes may require changes over multiple subroutine boundaries, rather than merely within the context of a single nested loop as above (consider the case where the loop lengths are not constants as above but instead are subroutine arguments).

Indeed, the current state of the art in this field is indicated by the fact that these software tools are not yet able to successfully optimize Gaussian elimination with partial pivoting (GEPP). This operation can, however, be manually blocked for improved cache performance. In fact, a hand-tuned, cache-efficient implementation of GEPP has been included in the LAPACK library [2]. Thus it appears unlikely that truly effective and highly automatic software tools of this sort will be available for several years.

Referring to the questions at the end of section five, we conclude that while effective loop interchanges can be made with some effort by an expert programmer, and it may be possible that compilers of the future could make such changes in some cases, it is not likely that compilers or other automatic tools could make these changes in all cases. Without such changes, the performance of many codes is likely to be suboptimal.

7. Blocking Transposes

At the heart of many two and three-dimensional scientific applications, similar operations must be performed in each dimension. On vector computers these are often implemented as a sequence of calls to subroutines that are virtually identical, except for the dimension in which multidimensional arrays are accessed.

It is clear that a Fortran routine which accesses arrays by other than the first dimension will not perform well on a cache-based system, since such accesses have nonunit stride. One solution is to revise the design of such programs to perform array transpositions between the computational steps, so that computations can always be done with unit stride data accesses. The resulting code may even be simpler than before, since often the same unit

stride computational routine can be employed in each dimension. It should be pointed out, however, that in some cases the performance of the resulting code is not much better than the original, since there is insufficient computation to offset the cost of the transpose operation. But in other cases it is profitable to make such a change.

In any event, even if the code is substantially revised to a design that interleaves computation and array transposition steps, one must still utilize an array transposition routine that has been optimized for a cache-based system. Consider the following straightforward Fortran routine to transpose a matrix:

```
      SUBROUTINE TRANS (M, N, A, B)
      DIMENSION A(M,N), B(N,M)
C
      DO 110 J = 1, N
        DO 100 I = 1, M
          B(J,I) = A(I,J)
 100    CONTINUE
 110  CONTINUE
C
      RETURN
      END
```

This loop will not run well on a cache-based system (unless the entire array fits in cache), due to the nonunit stride accesses in the `B` array. Nor does it do any good to interchange loops, since then the `A` array has nonunit stride accesses.

It is possible to perform transpositions in a cache-efficient manner by "blocking" the transpose: one fetches two opposing blocks from main memory to cache (note that each column of these blocks can be fetched with unit stride), transposes each block in cache, and then returns each of the transposed blocks to the opposite location. This is best described as follows, where A_{ij} and B_{ij} denote blocks small enough that both can simultaneously fit in cache:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}^t = \begin{bmatrix} A_{11}^t & A_{21}^t & A_{31}^t & A_{41}^t \\ A_{12}^t & A_{22}^t & A_{32}^t & A_{42}^t \\ A_{13}^t & A_{23}^t & A_{33}^t & A_{43}^t \\ A_{14}^t & A_{24}^t & A_{34}^t & A_{44}^t \end{bmatrix}$$

Another efficient scheme for transposing arrays in a hierarchical memory system is based on Fraser's algorithm. This scheme is described in [3].

Let us consider again the questions listed at the end of section five. First of all, it is clear that multidimensional array codes, particularly those that have heretofore run on vector computers, will not run well unmodified on cache-based systems. Second, while it is conceivable that at some time in the future, compilers and other software tools may be available to automatically transform such codes into more cache-efficient designs, that time

is at least several years away. Third, while it is possible, even now, for expert programmers to make such transformations, it is laborious to do so and may not be practical for a very large program. Fourth, it is indeed algorithmically possible to perform this type of computation in a mostly unit stride, cache-efficient manner; however, it is not entirely clear that all such codes would benefit from this change.

8. Slow Fourier Transforms

The next example to be examined is the computation of the one-dimensional fast Fourier transform (FFT). Unfortunately, many FFT algorithms in the literature, if implemented in a straightforward manner, include such undesirable features as large, power-of-two memory strides. These strides also hamper vector computer performance. Fortunately, there are variant FFT algorithms that do not involve power-of-two memory strides, and which in fact can be implemented with exclusively stride one memory accesses [4]. These algorithms have been used on vector computers for some while.

However, even these algorithms typically have one feature that is undesirable for cache systems: the principal main memory arrays `X` and `Y` are both accessed roughly t times to perform a 2^t-point FFT. Is it possible to reduce the number of times these data arrays are accessed? Yes, as it turns out. However, no amount of loop restructuring or other superficial manipulation of the code will accomplish this. Instead, a completely different technique for performing the FFT must be employed. This algorithm can be sketched as follows, where the size of the input complex vector is $n = n_1 n_2$ words, and where the cache is assumed to hold at least $4b \max(n_1, n_2)$ words. See [3] for details.

1. Consider the data in main memory as a $n_1 \times n_2$ complex matrix in column-major (Fortran) order. Fetch the data b rows at a time into the cache. For each batch of b rows, perform b individual n_2-point FFTs on the $b \times n_2$ complex array in cache.

2. Multiply the resulting data in each batch by certain roots of unity: the element fetched from location (j, k) in the original complex matrix is multiplied by $e^{-2\pi ijk/n}$.

3. Transpose each of the resulting $b \times n_2$ complex matrices into a $n_2 \times b$ matrix, using the cache, and store the resulting data in main memory. Store successive batches of data in successive contiguous sections of main memory.

4. Consider the resulting data in main memory as a $n_2 \times n_1$ complex matrix. Fetch the data b rows at a time into the cache. For each batch of b rows, perform b individual n_1-point FFTs on the $b \times n_1$ complex array in cache, and return the resulting b rows to the same locations in main memory from which they were fetched. The result is a correctly ordered discrete Fourier transform.

With this algorithm, the principal main memory data arrays need only be accessed two times, no matter what the size of the FFT. Note that all of the individual n_1-point and n_2-point FFTs are performed entirely in the cache.

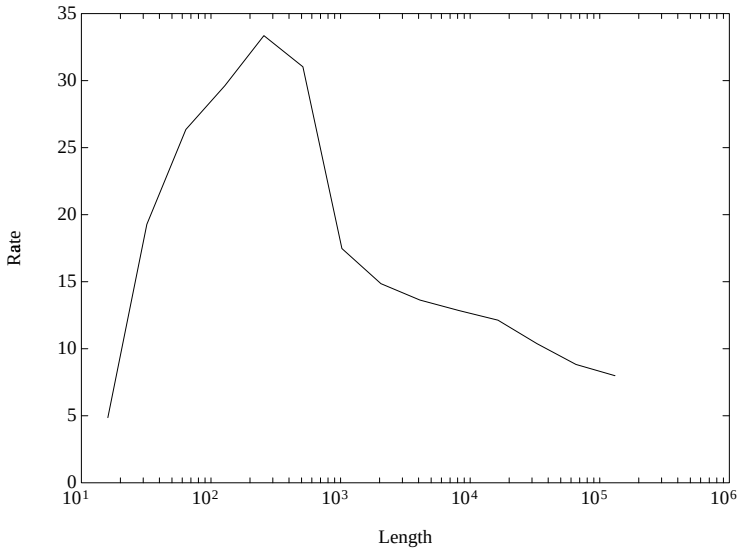


Figure 2: i860 Performance on a Library FFT (Mflop/s)

To summarize this discussion, let us again return to the four questions at the end of section five. First of all, it is clear that the FFT as often implemented is clearly inappropriate for cache-based computer systems. Second, it appears unlikely that foreseeable software tools will be able to make more than superficial changes to most codes implementing FFTs. Third, even expert programmers are unlikely to be able to make more than modest improvements in such codes. Fourth, a moderately cache-efficient algorithm is known for one-dimensional FFTs; however it is not well known outside the FFT field.

It is true that the FFT is an operation that is often available in vendor-supplied libraries, and thus one could argue that individual scientists do not need to be concerned these issues — they can merely use library routines. However, many scientists regrettably have incorporated their own "home-grown" FFT routines into their program files, in order to make their codes more easily portable between systems. Also, it is important to note that vendor-supplied FFT routines in many cases do not employ an advanced algorithm. For instance, the FFT routine supplied by Intel on the iPSC/860 employs a conventional algorithm that is efficient only for transforms small enough to fit into cache (see Figure 2). Of the major RISC vendors, only IBM appears to have implemented an advanced FFT algorithm [1] similar to the one described above.

9. Iterative Methods

Iterative methods are increasingly important in solving large, sparse systems of linear equations. They are typically found at the center of economically important, large-scale computations. The most important methods in use today include the multigrid and Krylov subspace methods, of which the conjugate gradient method is the prototype. The book of Golub and Van Loan [10] is a good reference for these methods. Unfortunately, these

methods cannot be restructured to work well with memory hierarchies.

The problem is to solve the equation $Ax = b$ where A is a given sparse nonsingular matrix and b a given vector. The set of nonzero elements of A is generally much too large for cache, since it typically requires many Mbytes of storage. The conjugate gradient and related methods compute a sequence $\{x_k\}$ of vector iterates that converges to the solution x.

The nature of these algorithms is that all of A must be accessed from memory, once per iteration, before any of it is accessed again, so the computation proceeds at main memory rather than cache speeds. In the case of the Krylov subspace methods, a dot product involving the vector Ax_k has to be computed before x_{k+1} can be. Thus, the matrix-vector product at one iteration cannot be pipelined in any way with the matrix-vector product at the preceding or following iteration.

With the multigrid methods, data on fine grids is accessed only one or two times before computation passes to coarser grids. These grids thus represent a bottleneck analogous to the dot products in Krylov subspace methods.

At the present time, there is a significant amount of research activity in this field, and there is some hope for improved algorithms in the future. Improved preconditioners, for example, may reduce the number of iterations required for the conjugate gradient method to converge. Domain decomposition methods, which work by solving a sequence of smaller systems of equations for subsets of the unknowns, use much smaller sparse matrices that may fit in cache. There is also some hope for "block iterative methods," which may provide the power of the iterative schemes in a more cache-efficient design. Some other iterative methods, such as the Jacobi, Gauss-Seidel, and successive over-relaxation methods, may be organized to work well with memory hierarchies. In general, however, these latter methods are much slower and less reliable than multigrid and Krylov subspace methods. In any event, none of these alternate schemes can be automatically derived from the codes for other iterative methods.

Referring again to the four questions of section five, we conclude that codes employing iterative methods will not perform well on cache-based computer systems. Unfortunately, it appears that these difficulties cannot be helped by superficial code modifications, even those made by expert programmers, much less by automatic software tools. And while there is some hope that these difficulties can be surmounted by substituting advanced algorithms, these schemes are as yet experimental.

10. Crystal Balls

The problem of quantifying data locality in various types of scientific applications is a very important and timely one. This problem is closely related to the question of whether an application can be efficiently implemented on a distributed memory multiprocessor. Indeed, some distributed memory parallel systems are being designed to treat distant memory as simply another level of the memory hierarchy, of which the lowest levels are the registers and internal cache. Clearly this problem deserves a great deal of study by the research community, and the answers will only be evident after many scientists have

studied many different applications.

We have seen that in each of the four cases studied above, there are either known algorithms and implementation techniques that are better suited to cache memories, or else there is some hope that advanced algorithms now being investigated will provide scientists with a cache-efficient alternative in the future. It might turn out that it is always possible to find algorithms and implementation techniques that are reasonably cache-efficient, in the sense that data once fetched into the cache can be profitably accessed numerous times before being returned to main memory. In that case the RISC vendors' assumption that caches are broadly effective in sustaining high performance will be upheld.

On the other hand, it might be that there remain a significant number of important scientific applications which feature an unavoidably low degree of cache-level data locality. For these problems, RISC systems with large ratios in performance between data in cache and main memory will have much less of an edge in sustained performance per dollar, compared with other architectures, and may even be surpassed in this statistic by other designs.

While there is some uncertainty on the long-term outcome on this question, in the near term (within the next two or three years), the answer is considerably clearer: a significant segment of important scientific applications will not perform at optimal rates on current cache-based RISC systems. Thus even at this point in time, it seems essential that designers of cache systems be reasonably modest in their designs.

An analogy with the history of vector computing is instructive in this regard. One of the early vector systems, the CDC 205, featured superior performance on codes with long, unit stride loops. Other vector systems, notably the Cray-1 and Cray X-MP, featured a more flexible design, with more modest assumptions about typical vector lengths. These latter systems, while they could not compete with the CDC 205 on specially selected, highly-tuned codes, exhibited respectable performance across a much wider spectrum of real-world applications. This was a principal reason that they ultimately prevailed.

11. Conclusions and Recommendations

Many in the scientific computing community have observed that both the workstation market and the high-end supercomputer market have the same design goal: a high level of sustained performance on real scientific and engineering applications. These same persons have further argued that the massive investment being made in RISC microprocessors will force supercomputer vendors to employ these "commodity" processors in their systems.

However, this may not necessarily happen — it may turn out that the workstation market has sufficiently distinct requirements from high-end supercomputing that each will pursue its own path in processor development. Indeed, it is interesting to note that of the principal contenders in the latest generation of highly parallel supercomputers, only Intel and Cray are relying on commodity RISC processors for high-performance floating-point computation. Others, including Thinking Machines, nCUBE, Kendall Square Research, Fujitsu and Meiko, are relying on custom processors for this purpose.

If this divergence does occur, then much of this discussion may prove moot. On the

other hand, if it becomes clear that the requirements of these two markets are converging, then it is essential that RISC processor designers carefully consider these issues. It is also essential that scientists who traditionally have used vector systems for their computations understand in greater detail the implications of RISC processors on their programs. Otherwise both the high-end supercomputing market and even the workstation market itself may suffer from less-than-optimal sustained performance on many important applications.

While one could voice concern about several features of current RISC processors, two stand out and have been discussed in length above: (1) the large cost of divide operations, relative to the cost of adds and multiplies, and (2) the limited bandwidth between processors and main memory, relative to the bandwidth between processors and cache, or in other words overly optimistic ratios between cache and main memory performance. Both are problems of degree rather than fundamental concept.

What does the author recommend? First of all, I suggest that the cost of an integer divide operation be no more than about ten times the cost of an integer add or multiply, and that the cost of a floating point divide operation be no more than about ten times the cost of a floating point add or multiply. This ratio is higher than the ratio on Cray systems (four), but it seems acceptable even for applications that make heavy use of divisions.

With respect to main memory bandwidth, I certainly urge designers of future systems to make this as large as possible, recognizing that there will always be design costs and trade-offs that will limit this. What about the balance between bandwidth and latency to main memory, as compared with bandwidth and latency to cache? A factor of two appears tolerable, and a factor of four is not fatal. But I believe that ratios beyond this level will not add significant value for many large scientific codes, and whatever resources were devoted to such caches could more profitably be employed in seeking ways to obtain higher bandwidth and lower latency to main memory.

One additional useful design point can be deduced as follows. Just as programmers of vector systems can be expected to tune their codes so that inner loops are reasonably long and so that strides are not powers of two, it is reasonable to ask some tuning effort by programmers of cache-based RISC systems. For example, it is reasonable to assume that with some effort many scientific application can be revised to feature mainly unit stride accesses (for instance by employing computational steps interleaved with array transpositions). In return, it seems reasonable to insist that future RISC-based systems be able to access vectors of contiguous main memory data into the processor without significant slowdown. For instance, it seems reasonable to insist that a unit stride DAXPY loop run equally fast for data in cache as for long vectors that cannot be held in cache.

It may be that fundamental changes can be made to the design of the cache systems in RISC processors to make them more broadly effective for large-scale scientific computation. Some have suggested special instructions that load data from main memory, at a stride, directly into the registers (the Intel i860 has an instruction along these lines). Another possibility is to make changes to the basic cache organization, such as the number associativity classes. The author is not sufficiently knowledgeable about processor architectures to comment on these suggestions, but hopefully discussions such as this will highlight the

issues involved and lead to improved designs in the future.

If designers of RISC processors really want to get serious about supporting the highly parallel supercomputer market, they ought to consider adding features to facilitate very high speed, low latency interprocessor communication (including fast synchronization). At the present time, manufacturers of parallel systems that employ RISC processors utilize separate custom-designed devices to provide this functionality, and the cost of this custom circuitry is emerging as a dominant factor in the total cost of the system. Even in a workstation system, such features could prove valuable, for example as a basis for implementing multi-workstation, networked parallel computing.

In any event, it is a pity that there is not a greater degree of mutual communication between the RISC processor community and the scientific computing community, especially the parallel scientific computing community. Clearly both communities have much to gain from such interaction. This article is written with the hope of fostering this dialogue.

Acknowledgments

The author wishes to acknowledge a significant contribution by Robert S. Schreiber of the RIACS organization at NASA Ames Research Center. Others who have made valuable suggestions to this manuscript include E. Barszcz of NASA Ames; R. Fatoohi, H. Simon, V. Venkatakrishnan and S. Weeratunga of Computer Sciences Corp.; P. Bjorstad, currently visiting RIACS; A. Gupta of Stanford University; A. Karp of Hewlett-Packard; S. Oberlin of Cray Research, Inc.; B. Parady of Sun Microsystems, Inc.; M. Humphrey of Silicon Graphics, Inc., D. Scott of Intel Scientific Computers; and D. Smitley of the Supercomputing Research Center.

References

[1] R. C. Agarwal, and J. W. Cooley, "Fourier Transform and Convolution Subroutines for the IBM 3090 Vector Facility", *IBM Journal of Research and Development*, vol. 30 (1986), p. 145 - 162.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorenson, *The LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

[3] D. H. Bailey, "FFTs in External or Hierarchical Memory", *Journal of Supercomputing*, vol. 4 (1990), p. 23 - 35.

[4] D. H. Bailey, "A High-Performance FFT Algorithm for Vector Supercomputers", *International Journal of Supercomputer Applications* vol. 2 (1988), p. 82 - 87.

[5] D. H. Bailey, "Unfavorable Strides in Cache Memory Systems", RNR Technical Report RNR-92-015, NASA Ames Research Center, 1992.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks", *Intl. Journal of Supercomputer Applications*, v. 5, no. 3 (Fall 1991), pp. 63 – 73.

[7] D. H. Bailey, E. Barszcz, L. Dagum and H. D. Simon, "NAS Parallel Benchmark Results", *Proceedings of Supercomputing '92*, IEEE, p. 386 – 393.

[8] D. H. Bailey, R. Krasny and R. Pelz, "Multiple Precision, Multiple Processor Vortex Sheet Roll-Up Computation," Technical Report RNR-90-028, NAS Applied Research Branch, NASA Ames Research Center, October 1992.

[9] S. Carr and K. Kennedy, "Compiler Blockability of Numerical Algorithms", *Proceedings of Supercomputing '92*, IEEE, to appear.

[10] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins, Baltimore, 1989.

[11] K. Lee, "On the Floating Point Performance of the i860 Microprocessor", Technical Report RNR-90-019, NAS Applied Research Branch, NASA Ames Research Center, October 1990. Also in the *Intl. Journal of High Speed Computing*, to appear.

[12] D. Pase, "MPP Programming Model", Cray Research, Inc. 1992.

[13] R. S. Schreiber, "Block algorithms for Parallel Machines," in Martin Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, IMA Volumes in Mathematics and Its Applications, vol. 13, Springer-Verlag, New York, 1988, p. 197 – 207.

[14] R. S. Schreiber and G. Shroff, "On the Convergence of the Cyclic Jacobi Method for Parallel Block Orderings," *SIAM Journal on Matrix Analysis and Applications*, vol. 10 (1989), p. 326 – 346.

[15] R. S. Schreiber and C. Van Loan, "A Storage Efficient WY Representation for Products of Householder Transformations," *SIAM Journal on Scientific and Statistical Computing*, vol. 10 (1989), p. 53 – 57.

[16] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm", *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, ACM, June 1991.