# NAS Applications and Advanced Architectures

David H. Bailey, Rupak Biswas and Rob Van Der Wijngaart

NAS Technical Report NAS-97-031
1997-11-21

**Abstract**

This paper examines the applications most commonly run on the supercomputers at the Numerical Aerospace Simulation (NAS) facility. It analyzes the extent to which such applications are fundamentally oriented to vector computers, and whether or not they can be efficiently implemented on hierarchical memory machines, such as systems with cache memories and highly parallel, distributed memory systems.

Bailey: NASA Ames Research Center, Mail Stop T27A-1, Moffett Field, CA 94035-1000; `dbailey@nas.nasa.gov`. Bailey is an employee of NASA.
Biswas: NASA Ames Research Center, Mail Stop T27A-1, Moffett Field, CA 94035-1000; `biswas@nas.nasa.gov`. Biswas is an employee of MRJ Technology Solutions. This work was performed under contract NAS2-14303.
Van Der Wijngaart: NASA Ames Research Center, Mail Stop T27A-1, Moffett Field, CA 94035-1000; `wijngaar@nas.nasa.gov`. Van Der Wijngaart is an employee of MRJ. This work was performed under contract NAS2-14303.

## 1. Introduction

The usefulness of traditional vector computers for the solution of the discretized equations governing viscous, compressible (and incompressible) fluid flow has been demonstrated rather convincingly by the success of codes like OVERFLOW and INS3D on vector supercomputers, such as the Cray C90 and T90. These programs combine the numerical efficiency of implicit solution algorithms (which permit large time steps) with the computational efficiency of regular grids and easily exploited vector parallelism. But should they always be run on vector machines? Are there some inherent features of these applications that make them poorly suited for other architectures, notably hierarchical or non-uniform memory access (NUMA) systems? In the following, we will consider distributed memory parallel systems as another form of NUMA system.

Before examining this question, it is good to keep in mind that there are a number of distinct, numerically efficient algorithms available for the solution of equations of interest to NASA and the community it serves. Not all of them are implicit schemes. Explicit schemes, which are usually quite well suited for NUMA systems, may be acceptable for some applications. Other types of algorithms, such as multigrid, may be used as well. Thus the question at hand is whether or not, for certain important classes of applications, all known numerically efficient algorithms have a strong preference for one type of architecture or another.

Some important factors governing the efficiency of a particular scientific application on a particular system include

- Spatial locality, such as the presence of nonunit strides.
- Temporal locality, such as the number of operations performed per memory access.
- Regularity of memory accesses, such as whether or not strides are constant.
- Data dependencies, such as recurrences.
- Loop vector lengths.
- Complexity of inner loops.
- Operation mix ratios.

These factors are related to certain characteristics typical of many modern computer architectures. These characteristics in turn stem from a fundamental challenge in all modern computer architectures: dealing with the latency between processors and main memory.

Vector processors attempt to ameliorate this latency by working with long streams of regularly spaced data. In this way, after the penalty of latency is paid for the first element, the remaining data elements can be loaded, and possibly processed and stored as well, at a rate of up to one per clock period. Vector systems typically have difficulties with data that is spaced with power-of-two strides (since such strides result in repeated accesses of the same memory banks), highly memory intensive applications (since in such cases memory bank collisions with other processors are frequent), as well as with shorter loops and, of course, loops with recursions or other features that inhibit vector processing.

Computers with hierarchical memory systems attempt to deal with latency by employing a hierarchy of storage systems (with main memory as one of the higher levels) which attempts to exploit data locality in the computation by keeping frequently used data close to the processor. Hierarchical and NUMA systems have difficulties with non-unit stride data (since only one or a few elements are used from each block of main memory fetched to cache). These difficulties are amplified if strides are certain particular values, usually powers of two (since these may result in repeated access of the same cache lines, as well as problems in the translation look-aside (TLB) buffer). Even with loops featuring stride one data access, performance on a cache-based may be sub-optimal if each data element fetched from main memory is used only once.

The question is this: do these architectural differences also distinguish fundamentally algorithms that can be implemented efficiently on these systems? Are some algorithms fundamentally better suited for one type of architecture than another? Here we will investigate these issues for the kind of problems that NASA faces in large-scale 3D CFD simulations.

## 2. Alternating Direction Implicit (ADI) Schemes

Let us first examine the Alternating-Direction Implicit (ADI) technique used in OVERFLOW. It comes in two flavors, namely a block tridiagonal (BT) and a scalar pentadiagonal (SP) form (more about this distinction later). For both SP and BT there are three independent families of equations to be solved, each corresponding to a particular coordinate direction. These families are called factors. Each factor requires the solution of a large set of (again independent) linear equations, one for each grid line. Within a grid line a certain data dependence is incurred. If we solve in the $y$-direction with `j` as a running index, we have to solve for point `j-1` before point `j` can be visited. This seems to be a problem; it is natural to view each grid line as an independent task that needs to be finished before moving on to the next line.

But if arrays are stored as `u(i,j,k)`, for example, than for the solves in the y-direction the second index will change fastest, resulting in a memory stride the size of the first ($x$) grid dimension. This will be unfavorable for a cache-based system, since it is likely that only one data element per cache line is used. However, this is an artifact of the seeming indivisibility of the line solves. Since all $y$-line equation systems are independent, we can solve for all the first points (`j=k=1`, with `i` as a running index) of each $y$-line before moving to the next point. In other words, we can always code a loop that implements an ADI-like data dependence in a cache-favorable way, with `i` as the inner loop index (provided multi-dimensional arrays are stored in column major order, as in Fortran).

Here is a simple but illustrative example. Assume that the line solve in the $y$-direction consists of adding each previous value at point `j-1` to that of point `j`, in other words`u(i,j,k) = u(i,j,k) + u(i,j-1,k)`. The "natural" code for this operation is something like

```
do k = 1, kmax
  do i = 1, imax
    do j = 2, jmax     ! the line solve starts here
      u(i,j,k) = u(i,j,k) + u(i,j-1,k)
    enddo              ! the line solve ends here
  enddo
enddo
```

A cache-friendly version of this code is

```
do k = 1, kmax
  do j = 2, jmax       ! the line solves in the i,j-plane start here
    do i = 1, imax
      u(i,j,k) = u(i,j,k) + u(i,j-1,k)
    enddo
  enddo                ! the line solves in the i,j-plane end here
enddo
```

Evidently, the recurrence imposed by the ADI scheme along grid lines did not lead to an unfavorable data access pattern, because there was enough parallelism in the other coordinate directions. Notice that the second code fragment also vectorizes well, since all inner iterations are independent. Ironically, on a vector machine the canonical implementation of the $x$-factor does not perform well, because it exhibits a recurrence in the inner loop:

```
do k = 1, kmax
  do j = 1, jmax
    do i = 2, imax      ! the line solve starts here
      u(i,j,k) = u(i,j,k) + u(i-1,j,k)
    enddo               ! the line solve ends here
  enddo
enddo
```

Again, there is no inherent problem. Advanced vector compilers can easily recognize that a simple loop index interchange (`i<->j`) will make the inner loop vectorizable again. This has been observed in the SP benchmark from the new NAS Parallel Benchmarks suite (NPB-2), which implements the essentials of the diagonalized Beam-Warming version of OVERFLOW. When compiled on the NASA Ames Cray C90 using the currently available Cray Fortran-90 compiler, this code vectorizes completely, despite the presence of these inner loop recurrences. Of course, if a compiler does not automatically recognize the opportunity for loop interchange, such a change can easily be done by hand or by using a preprocessor tool.

Thus we conclude that on single-processor systems there is no natural preference of the ADI algorithm for vector over cache-based systems. Further, there are now parallel ADI algorithms of sufficient sophistication to limit the data transfer between processors to an acceptable minimum (e.g. the multi-partition method [5, 6]) on most scalable machines. The major part of the performance degradation of the NPB2 SP code on systems like the CRAY T3D or IBM SP2, as the number of processors grows, is attributable to the degradation of performance on individual nodes, not to the communication overhead, except when the number of nodes is very large compared to the grid size.

Most NUMA machines exhibit better performance on the BT benchmark from the NPB-2 than they do on the SP benchmark. This is because the BT benchmark (and most BT-derived application codes) solves for each grid point a $5 \times 5$ dense system of equations as part of the overall block-tridiagonal line solve. Dense matrix computations have excellent data reuse, meaning not only that all data elements in each cache line are used (this has to do with a good data access pattern), but also that several computations per element are performed on a datum once it is placed in a CPU register. This is much less the case in SP, which does not solve dense systems. The situation is reversed for vector machines: here SP does best (440 Mflop/s on one processor of a Cray C90), because it vectorizes well and the line solve loops are fairly "thin" (not too many instructions, constants and arrays in the loops, so that there are always enough vector registers). BT, by contrast, achieves "only" 260 Mflop/s on a C90 processor. The code still vectorizes well, but since the inner $5 \times 5$ loops are unrolled (partly to pre-empt the possible vectorization of these very short loops, and partly to remove inner loop recurrences), the loops are very "fat", resulting in register allocation problems.

In summary, there do not appear to be any compelling preferences for either vector or NUMA computers when implementing ADI-type algorithms.

### 3. Gauss-Seidel Schemes

Another important CFD production code at NASA Ames is INS3D. It can be run with several different kernel solution algorithms. There are two that employ approximate factorizations based on a spectral decomposition of the flux Jacobians, resulting in three-factor lower-diagonal-upper schemes, which are applied in a symmetric double-sweep fashion (LU-SGS). The resulting data dependencies are equivalent to point- and line-Gauss-Seidel, respectively; in the forward sweep, no point `(i,j,k)` may be updated before all points with smaller or equal indices have been updated. Interestingly, even though in the point-relaxation mode the updates take place on a point-by-point basis, the data dependency makes vectorization hard. There is no way that a triple "Cartesian" loop (loop bounds independent of other loop indices) can be written that does not have a recursion in the inner loop. This gave rise to the skewed-hyperplane approach, in which points are updated simultaneously and independently on planes of constant `i+j+k`. Notice that intersections of such skewed planes with a grid are not of constant size, and hence vector lengths vary widely.

By contrast, a cache-based implementation, for which recursions are acceptable, can be coded in a completely straightforward fashion, using simple Cartesian loops. Again, the loop nests can be arranged in a way that preserves data locality. This can even be accomplished to a reasonable degree on a distributed memory machine, through the use of a generalized two-dimensional pipeline, as is evidenced by the NPB-2 LU code. Not surprisingly, this code does not fare well on vector machines. And a vector-based code does not perform well on a cache-based machine, because the strides along skewed hyperplanes are too large. So even though the INS3D scheme in the relaxation mode is not fundamentally unsuited for either vector or NUMA computer, it does seem not easy to write a single code that does well on both types of architectures.

## 4. Krylov Schemes

A relatively new feature of INS3D is the implementation of a Krylov subspace method (GMRES, in this case) for the solution of the linear systems. This solver can be implemented efficiently on both vector and NUMA computers, since it does not feature the complicated recurrences of the relaxation mode solvers. The most important operations are the formation of regular, ultra-sparse matrix-vector products (to compute residuals) and of dot products. On distributed memory systems, this requires mostly nearest-neighbor communications, as opposed to, for example, the conjugate gradient problem in the NPB-2 suite, which features general, randomly filled sparse matrices.

## 5. Unstructured Grid Methods

Unstructured-grid flow solvers are usually one of two types: cell-center schemes and cell-vertex schemes. In both two and three dimensions, the major computational effort is associated with the calculation of the numerical fluxes. In two dimensions, both schemes require nearly equal computational effort. Time evolution of explicit methods is performed on a control volume basis. The finite elements are the control volumes for the cell-center schemes. For the cell-vertex schemes, the control volumes are non-overlaping polyhedral regions (in three dimensions) surrounding the vertices of the mesh and can be obtained from the dual of the computational mesh.

Since the number of triangles is approximately twice the number of vertices, cell-center schemes require slightly more computation. In three dimensions, the number of flux calculations is proportional to the number of faces in the mesh for the cell-center schemes and to the number of edges for the cell-vertex scheme. Since the number of faces is usually about twice the number of edges, the flux computation is significantly more expensive for cell-center schemes. The computational effort associated with explicit time stepping also favors cell-vertex schemes since the number of tetrahedral elements is typically about five times the number of vertices.

Both cell-center and cell-vertex schemes run into similar problems when implemented on vector and NUMA computers. For example, for cell-vertex schemes, the computation of the fluxes across the faces of the control volumes is carried out by summing the contribution from each edge in the mesh. For vectorization, the edges need to be properly "colored" so that the inner do loop ranges over edges of the same "color" (in order to avoid race conditions with array data). However, for each edge, its two end-points need to be accessed. This requires indirect addressing. Similar problems are encountered for cell-center schemes.

The above discussion assumes an explicit method; implicit methods for these problems are still in an early stage of development, and will not be discussed here.

As a result of the requirement for indirect addressing, stencil operations cannot be done with fixed (or even predictable) stride, and memory access is rather erratic and usually not very favorable for cache-based systems. On the other hand, because most of the addressing of grid elements takes place through indirect referencing (arrays of indices), it is difficult on vector machines to assemble vectors efficiently. So here it appears that the implementations on vector and NUMA computers are relatively inefficient on both types of systems. In short, there is again no compelling argument that this type of algorithm fundamentally prefers one architecture over the other.

**5. Fast Fourier Transforms**

Fast Fourier transforms (FFTs) are utilized by several types of computations, notably large eddy simulations and direct simulations of turbulence. FFTs are also involved in the FT benchmark of the NPB (and NPB-2) suite, wherein a large 3-D FFT is used to solve a Poisson PDE problem.

It is well known that 3-D FFTs can be implemented easily on a vector system, simply by performing vectors of 1-D FFTs in the first dimension, followed by vectors of 1-D FFTs in the second dimension, and then the third dimension. For power-of-two FFTs (by far the most commonly utilized size), a naive implementation will result in severely reduced performance due to power-of-two memory strides (due to bank conflicts). However, such difficulties are relatively easily ameliorated by simply padding the dimensions of the 3-D arrays to be, say, one larger than a power of two. In this way, vector fetches in all three dimensions will be with favorable strides.

On a cache-based or NUMA system, the large strides involved in the second and third dimension data accesses will sharply reduce performance. Power-of-two FFTs are even more troublesome, due to cache line and TLB conflicts, which are the cache system equivalent of vector bank conflicts. However, these power-of-two difficulties can be avoided by padding the 3-D arrays as above, although the size of the optimal pad is typically different on cache systems than for vector systems (a good coding practice is to use a settable parameter for this purpose). Nonetheless, the large strides remain a problem.

The typical solution of the large strides in 3-D FFTs is to employ at least one array transposition. In the case of a distributed memory system, substantial interprocessor bandwidth is required for this operation, but existing commerical highly parallel systems appear to feature sufficient bandwidth to permit such transpositions ("complete exchange" operations) to be performed without dominating the total computational cost. With array transposition, all of the actual computational steps involve only memory resident (usually cache-resident) data. One can usually employ highly optimized vendor-supplied library routines to perform the resulting 1-D FFTs. The library routine for the IBM 590 workstation, for example, runs at approximately 220 Mflop/s, which is 80% of the theoretical peak. Some RISC vendors support efficient 2-D and 3-D FFT library routines as well, although FFT support among distributed memory vendors is still weak due to the lack of standards.

The case of large 1-D FFTs is interesting in this regard, even though such FFTs are generally not used in CFD calculations (they are however a staple of the digital signal processing world). Again, efficient schemes for large 1-D FFTs on vector systems have been known for some time [2], but these usually involved large strides and disparate array accesses, which are not favorable on cache-based and distributed memory systems. A few years ago, researchers found a scheme to perform large 1-D FFTs that in effect converts the problem to a 2-D FFT, which can then be solved as described above (usually with array transpositions) [2]. In fact, it was later realized that this scheme had actually been presented in one of the earliest FFT papers [4]. Thus this scheme is not "new" — it has been in the literature longer than efficient FFT schemes for vector systems.

In summary, FFT codes written for vector systems in most cases cannot be superficially converted to efficient cache-based or NUMA FFT codes. However, efficient FFT schemes are known for these other architectures, and it is possible to construct FFT-based codes that run efficiently on a wide variety of architectures, requiring custom tuning only for array transposition operations. But there appears to be no fundamental architectural preference one way or the other for performing large 1-D, 2-D or 3-D FFTs.

## 6. Array Transpositions

As mentioned in the previous sections, array transpositions are used in FFT applications, particularly on cache-based and distributed memory systems. Along this line, it is worthwhile to mention a straightforward scheme for array transpositions that appears to be relatively efficient on a wide variety of systems. We will describe it here for the case of a 2-D matrix:

Consider first the case where $n_1 = n_2$, so that the matrix is square. In that case a block interchange technique can be used to transpose the array in a single pass, in place. This can be done by simply considering the external $n_1 \times n_2$ matrix to be decomposed into square blocks of size $b$ on a side, where $b$ is the block size of an efficient read/write memory operation. The square blocks down the diagonal can be transposed simply by fetching the blocks one at a time into main memory, transposing them using any efficient main memory scheme, and storing the resulting matrices back in the same locations. The off-diagonal square blocks can be fetched in opposing pairs, transposed in main memory, and then stored back in opposite locations. One difficulty in applying this scheme is when the main memory block size $b$ is a power of two (which it almost always is). Transposing matrices whose dimensions are powers of two in main memory, using the straightforward scheme, results in memory bank conflicts on vector computers and cache line conflicts on cache-based systems. But such conflicts can be avoided by padding the first dimension of the scratch array as mentioned above.

For the common case of power-of-two FFTs, it can be assumed that either $n_1 = n_2$ or else $n_1 = 2n_2$. In the second case, it does not appear possible to transpose the array in one pass, in place, using only full block I/O transfers. However, such arrays can be transposed in just two passes, in place, using only full block transfers, as follows. First, consider the $n_1 \times n_2$ external array as two blocks of size $n_2 \times n_2$, and transpose each of these two square blocks in place, as described in the previous paragraph. This completes the first pass. Now consider the resulting data array in external memory to be a $n_2 \times n_1$ matrix. Inspection of an example shows that the columns of the resulting array need to be de-interleaved: column $2j$, $0 \leq j < n_2$ needs to be moved to column $j$, and column $2j+1$ needs to be moved to column $j+n_2$ (here the columns are numbered beginning with zero). When this de-interleaving is performed, the transposition is complete.

It should be emphasized that array transpositions are in fact a potentially very powerful tool for large 2-D and 3-D computations from a variety of disciplines. This is because array transpositions, in effect, convert multi-dimensional array accesses, which are fatal to the performance of many cache-based and distributed memory architectures, into contiguous data accesses, which are ideal for vector, cache-based and distributed memory systems. In other words, a typical 3-D application whose implementation for a vector system involves array accesses in all three dimensions may be performed in six steps, as follows:

1. Perform computations in the first (contiguous) dimension.
2. Transpose the array so that the second dimension is now the contiguous dimension.
3. Perform computations in the second (now contiguous) dimension.
4. Transpose the array so that the third dimension is now the contiguous dimension.
5. Perform computations in the third (now contiguous) dimension.
6. Transpose the array back to the original dimension ordering.

In many or even most cases, other schemes may be preferable for a particular application. But the point here is that this transposition scheme is generally applicable — it is an "existence proof" of sorts that multi-dimensional array calculations can always be localized. Further, array transpositions are "latency tolerant". This feature may be very important on future very highly distributed systems, such as the petaflops systems now on the drawing boards. Closer to the present, such schemes may enable certain large 3-D physical simulations to be performed with reasonable efficiency on networks of workstations, where latency is often unavoidably large.

It is also worth nothing that while efficient transpositions require fairly high data communications bandwidth, especially on distributed memory systems, this is something that can in principle be provided in a good design. By contrast, providing low latency appears to be increasingly difficult, since semiconductor memory devices are not improving greatly in operation speed, while processors continue to advance fairly aggressively in performance. Thus the disparity in performance between processors and memory is destined to grow even more acute in the future. Indeed, the challenge of managing latency is likely to dominate future design, both in hardware and software [3].

## 7. Conclusions

In this paper, we have reviewed only a few applications, namely those of most interest to scientists using the Numerical Aerospace Simulation (NAS) supercomputers. These applications include the key CFD solution schemes (mostly implicit), as well as some other methods widely used in aeronautical computation. It appears from our experience, including the experience of implementing the NAS Parallel Benchmarks on various architectures, that these computations can be efficiently implemented on both vector and cache-based or NUMA systems (including distributed memory, highly parallel systems). We see no fundamental architectural preference one way or the other.

# References


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

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

[3] David H. Bailey, "Onward to Petaflops Computing", *Communications of the ACM*, to appear, 1997.

[4] W. Morven Gentleman and George Sande, "Fast Fourier Transforms — For Fun and Profit", *AFIPS Proceedings*, vol. 29 (1966), p. 563–578.

[5] Rob F. Van der Wijngaart, "Efficient Implementation of a 3-Dimensional ADI method on the iPSC/860", *Proceedings of Supercomputing '93*, IEEE, 1993, p. 102–111.

[6] Rob F. Van der Wijngaart, Maurice Yarrow and Merritt H. Smith, "An Architecture-Independent Parallel Implicit Flow Solver with Efficient I/O", *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, to appear, 1997.