Tuning the Rank-n Update in a Wavefront Solver for Peak Performance

Sen-ming Chang
Mark A. Johnson

Applications Technology Center
IBM Corporation
Neighborhood Road
Kingston, NY 12401

Abstract

The wavefront solver is a type of linear equation solver that is suitable for solving the system of linear equations that arises in many finite-element applications. A new version of the wavefront solver was recently introduced into the ANSYS® program that uses a rank-n update. The rank-n update has properties that allow it to make very efficient use of a hierarchical memory structure. In addition, the rank-n update can exploit both vector and parallel processing to increase its performance. We discuss several general techniques for tuning the rank-n update, using the IBM ES/3090 VF as an example of a computer that incorporates hierarchical memory and vector and parallel processing capabilities. We then report the performance of the tuned rank-n update, both in isolation and in the context of ANSYS jobs.

Keywords: Wavefront solver, Rank-n update, Hierarchical memory, Vector processing, Parallel processing, IBM ES/3090 VF.

Introduction

The computation that dominants many finite-element applications is assembling and solving a set of linear equations. One type of linear equation solver is the wavefront solver, which works on only a small subset of the whole problem, the wavefront, at any given time. Rather than assembling all of the equations at once and then solving them, the wavefront solver assembles and solves the equations together in small steps. While assembling new equations adds variables to each step, partially solving the resulting equations eliminates variables from subsequent steps. When Irons[1] first introduced the wavefront method, he proposed eliminating one equation at a time. A large part of the process of eliminating an equation is updating the remaining equations, which is accomplished with a rank-one update of the matrix representing the equations. Recently Swanson Analysis Systems, Inc. introduced a new version of the wavefront solver into the ANSYS® program, a general-purpose, finite-element, structural analysis program.[2] The new solver can eliminate many equations at once, which involves a rank-n update of the remaining rows of the matrix. We will focus our attention on the matrix update because it often accounts for a large portion of the CPU time spent in the ANSYS program.

Carefully tuning the matrix update to exploit the hardware features of a computer can substantially increase the performance of the update. We will describe techniques for taking advantage of such features as hierarchical memory, vector processing, and parallel processing. In particular, we will examine the properties of the rank-n update that allow it to utilize a hierarchical memory structure efficiently. Although the details of tuning the matrix update and the resulting gains in its performance vary on different computers, the techniques are still fairly general. We use the IBM ES/3090 VF as a specific instance of a computer with a hierarchical memory and with vector and parallel processing capabilities.

Following a more detailed description of the problem being solved and the architecture of the IBM 3090 family, we describe the tuning techniques as applied to the rank-n matrix update. We then give performance results that show the effect of tuning the update procedure. In addition to reporting the performance of matrix update alone, we show the resulting performance improvement in the ANSYS program and discuss the implications of Amdahl's Law.

Wavefront Solver

The linear equation solver in a finite-element application assembles and solves a system of linear equations having the form,

$$AX = V$$

where A is the positive-definite global stiffness matrix, X is the solution vector, and V is the load vector. Although the global stiffness matrix may or may not be symmetric, we will

consider only the symmetric case in the following discussion. Instead of assembling all of the element matrices to form the full global stiffness matrix and then solving the complete system of equations, the wavefront method interleaves the steps that assemble and solve the equations. As the solver processes each of the elements, it assembles new equations into the global stiffness matrix and solves some of the equations in the partially constructed matrix. The elements involved in the partially constructed matrix constitute the current wavefront. Although the size of A increases and decreases during the solution process, it is always much smaller than the full global stiffness matrix. Thus, the wavefront solver requires much less memory to store the stiffness matrix than solvers that assemble the full matrix.

Because detailed descriptions of the wavefront method already exist, [3-4] we present only a summary of the steps. In the following description, the nodes currently in the stiffness matrix are the active nodes.

1. A new element is assembled into the global stiffness matrix by obtaining its stiffness coefficients from a scratch file. If the nodes of the element are being activated for the first time, their stiffness coefficients are added to the global matrix. If the nodes are already active, their stiffness coefficients are summed into the existing equations.

2. If some nodes are appearing in the global stiffness matrix for the last time, the corresponding equations are solved in terms of the other active nodes. After rearranging the global stiffness matrix, A, so that the nv equations being solved reside in its first nv rows, each of the nv rows is scaled by its diagonal element and stored in the matrix B.

$$B_{ij} = \frac{A_{ij}}{A_{jj}}$$

The remaining rows in A are then updated using the scaled rows in B and the nv rows being eliminated, according to the following equation.

$$A_{ij} = A_{ij} - \sum_{k=1}^{nv} B_{ki} A_{kj}$$

After A has been updated, the nv equations are eliminated from A, and the corresponding nodes are deactivated, decreasing the size of the global stiffness matrix and making space available for subsequent steps. The matrix B is stored in a scratch file for use in the subsequent back substitution.

3. Steps 1 and 2 are repeated until all of the elements in the problem have been processed. After assembling the last element, the number of equations equals the number of unknowns, allowing the remaining equations to be solved explicitly.

4. The resulting triangularized matrix in the scratch file is used in a back substitution, completing the solution process.

The original version of the wavefront solver eliminates only one equation at a time, so it can use only one row to update the remaining rows of the global stiffness matrix. Such an update procedure is called a rank-one update of the matrix. However, the new version of the wavefront solver eliminates multiple equations in a single step, allowing the use of a rank-n update. The rank-n update uses all the rows being eliminated to update the remaining rows of the global stiffness matrix. In order to use a rank-n update, the solver continues to assemble new elements into the global stiffness matrix until a predetermined number of rows can be eliminated together. The solver then uses the rank-n update in the process of eliminating the rows from the matrix.

In the remainder of this paper, we will focus on the matrix update because it accounts for most of the CPU time spent in the wavefront solver. In turn, the wavefront solver accounts for a large portion of the CPU time consumed by many ANSYS jobs, especially those performing linear static analysis. The advantage of using the rank-n update instead of the rank-one update is that the rank-n update performs many updates at once and thus allows more opportunity for tuning. While the rank-one update is a matrix-vector operation, the rank-n update is a matrix-matrix operation, which can often be tuned to use a hierarchical memory structure very efficiently.

Architectural Description

The various techniques for tuning the rank-n update apply to many computers that have hierarchical memory structures and support vector and parallel processing. However, we apply our techniques to the IBM ES/3090 VF to make the following discussion more clear and concrete. Before investigating the techniques for improving the performance of the rank-n update, we briefly summarize the relevant features of the 3090 family.

The hierarchical memory in the 3090 consists of cache, main memory, expanded storage, and disk storage. Although the wavefront solver uses expanded storage to contain its large scratch files, the rank-n update deals with only the cache and main memory. The cache is a four-way, set-associative memory organized into 128 byte cache lines, having a total size of 64 kbytes on the 3090 and 3090E models and 128 kbytes on the 3090S models. Depending on the addressing pattern of a program, the effective size of the cache can vary from its full size down to only four cache lines. When data is addressed with a stride between adjacent elements that is divisible by a high power of two, the cache lines compete for space in the four-way, set-associative scheme, reducing the

effective size of the cache. We will see the effect of ignoring stride in the section on performance.

The vector architecture of the 3090 family consists of 16 short-precision or 8 long-precision vector registers and a variety of vector instructions. Each of the vector registers can contain at most 128 elements on the 3090 and 3090E models and 256 elements on the 3090S models. The length of the vector registers is the vector section size (VSS) and is the largest number of elements that a single vector instruction can process. The vector instructions have a variety of addressing modes, including register-register operations and memory-register operations. The memory-register operations allow only one memory access per element. If the data that the memory operand addresses resides in the cache, memory-register operations are as fast as register-register operations. However, if the data does not reside in the cache, the resulting cache miss slows the execution of the memory-register operation. Although most of the vector instructions perform a single operation on each element, several instructions perform both a multiplication and an addition or a subtraction on each element. The compound operations execute in the same length of time as the single operation, so they should be used wherever possible.

The IBM ES/3090 family of computers has models with from one to six processors, and software support is available to allow parallel processing. Each of the processors in a multiprocessor system share the same main memory, expanded storage, and disk storage, but each processor has its own cache, whose integrity is guaranteed by the hardware. The software support that we used is the VS FORTRAN Multitasking Facility (MTF), which has been part of the VS FORTRAN library since Version 1 Release 4. The MTF allows the main task to dispatch a subroutine as a subtask on another processor and to synchronize the main task with the completion of the dispatched subroutine. Such a simple interface was completely adequate for enabling the rank-n update to exploit parallel processing.

Tuning The Rank-n Update

FORTRAN Implementation

As we mentioned previously, the input to the rank-n update procedure consists of the global stiffness matrix, A, and a matrix, B, containing the scaled rows being eliminated. Figure 1[3] illustrates the storage order of A, which is the upper triangular portion of the square matrix representing the currently active nodes. Figure 1 also illustrates the storage order of B. Example 1 gives a simple FORTRAN implementation of the rank-n update algorithm, where n is the total number of rows in A, and nv is the number of rows being eliminated. The rank-one update can be recovered from the more general rank-n update by eliminating the argument nv and the loop over nv on line 10 of Example 1.

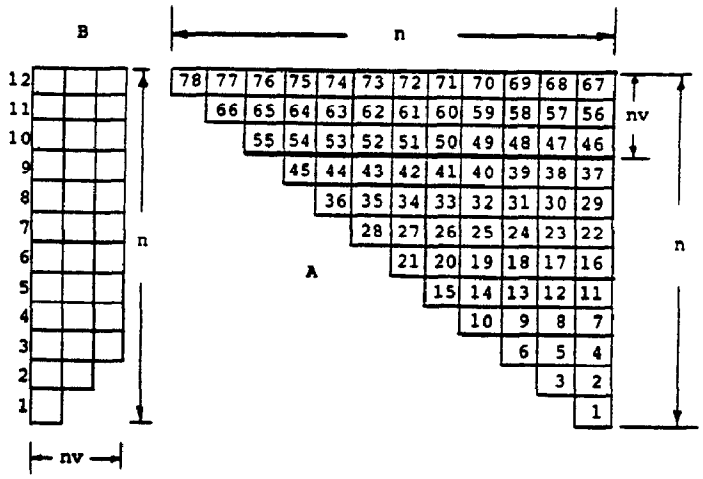

Figure 1. The storage order of matrices A and B that the rank-n update uses. The size of A is n, and the number of rows being eliminated is nv.

```
 1      subroutine rankn (A, n, B, nv, ka, kb, ja, jb)
 2      integer in, ir, jn, jr, n, nv, ka, kb, ja, jb, i, j, k
 3      double precision A(*), B(n, nv)
 4
 5      in = n - ja + 1
 6      ir = (in * (in - 1)) / 2
 7      do 10 j = ja, min(n - ka + 1, jb)
 8        jn = n
 9        jr = (n * (n - 1)) / 2
10        do 20 i = 1, nv
11          do 30 k = ka, min(in, kb)
12            A(ir + k) = A(ir + k) - B(in, i) * A(jr + k)
13 30       continue
14          jn = jn - 1
15          jr = jr - jn
16 20     continue
17        in = in - 1
18        ir = ir - in
19 10   continue
20      return
21      end
```

Example 1.

Rather than updating the whole matrix in the example, we allow the programmer to specify a range of rows (ja to jb) and columns (ka to kb) to update. As we will describe later, the additional flexibility is necessary in a version of the wavefront solver that exploits parallel processing.

Cache Usage

In both the rank-one and the rank-n updates, the row or rows being eliminated are used many times once they are loaded into the cache. However, the rank-one update uses each row that is being updated in only one operation (line 12 of Example 1) during the update. In contrast, the the rank-n update uses each row being updated in nv operations, resulting in much more effective use of the hierarchical memory. Increasing the number of rows being eliminated improves the usage of cache and thereby improves performance, as long as all the rows being eliminated can remain in the cache.

Since the cache is relatively small and we want to make the number of rows being eliminated as large as possible without overflowing cache, we modified the basic algorithm to reduce the cache requirements of each row. The modification consists of updating strips of the matrix, as Figure 2 illustrates. Each strip consists of a range of columns of the matrix being updated. The modified algorithm requires that only a portion of each row being eliminated reside in cache, so more such rows can be used without overflowing the cache. In the next section, we will discuss vector processing considerations and examine the balance between the width of each strip and the number of rows being eliminated.

In the basic algorithm, the rows being eliminated were contiguous in their storage order, so they efficiently filled the cache. In contrast, the modified algorithm uses only segment of each such row, and the segments are not contiguous. The noncontiguous segments may or may not fill the cache efficiently, depending on the stride between adjacent segments. As we mentioned in the description of the IBM 3090's cache, certain strides, such as those divisible by a high power of two, cause competition for the certain cache lines in the four-way, set-associative scheme. A technique for ensuring that the cache is filled efficiently, regardless of the stride between adjacent segments, is to copy the segments of the rows being eliminated into a contiguous block.[5] The storage space and the amount of time required for the copying operation are small under all circumstances because the size of the cache limits the amount of data that must be copied for each strip.

Vector Processing

The loop over k in Example 1 is the loop that is most suitable for vectorization because it produces sufficiently long vectors with a unit stride. The IBM VS FORTRAN Version 2 compiler will vectorize the loop over k, using a compound operation that performs both the multiplication and the subtraction on line 12 of Example 1. However, the vector A(ir + k) is loaded from memory and stored back to memory during each iteration of the loop over i. Such redundant loads and stores can often be eliminated by moving

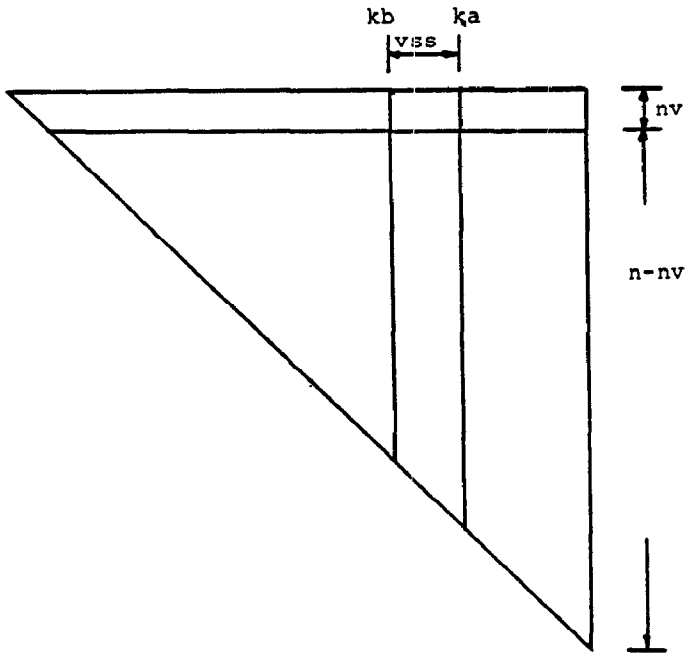

Figure 2. An example of the use of ka and kb to divide the matrix being updated into strips. A good choice for the width of each strip is the vector section size (VSS) of the machine.

the vector loop outside other loops, e.g., by changing the order of the loops over i and k in Example 1.[5] Unfortunately, such a technique will not work in Example 1 because of the complicated addressing that the storage order of matrix A requires. We eliminated the redundant loads and stores in the rank-n update by programming it in assembler, rather than in FORTRAN. We note that if a rank-one update were used instead, the loop over i would not be present, and a load and store would be necessary for each multiply-subtract operation.

The assembler implementation of the rank-n update consists of the following steps.

1. Load a segment of the row being updated into a vector register.
2. Load the scale factor, B(in, i) in line 12 of Example 1, into a scalar floating-point register.
3. Perform a vector multiply-subtract operation on the segment of the row being updated, using a memory operand to obtain the segment of A(jr + k), a row being eliminated.
4. Repeat steps 2 and 3 for all nv rows being eliminated.
5. Store the updated segment of the row loaded in step 1 back into matrix A.

Increasing the number of operations performed on each segment of a row between loading it and storing it improves the performance of the update procedure. Eliminating many rows at once both reduces the overhead of loads and stores and improves the usage of the cache, as long as the cache does not overflow. If the cache overflows, cache misses degrade the performance of the rank-n update, negating the benefits of further reducing the overhead of loads and stores. As we mentioned in the section describing the architecture of the IBM 3090 family, memory-register operations are as fast as register-register operations if the memory operand resides in cache. So in step 3 of the assembler algorithm, we are able to use a memory operand, which allows a loop over the nv rows being eliminated without sacrificing performance. In contrast, Hessel et al.[3] used register-register operations, limiting the number of operations between loading and storing a row to one less that the number of registers.

In order to increase the number of rows being eliminated without overflowing the cache, we updated the matrix in strips. The width of each strip should be small so that segments of many rows can reside in the cache simultaneously. However, making the strips too narrow reduces the vector performance because short vectors have a relatively greater overhead associated with them. We achieved a good balance between the two effects by choosing the width of each strip to be the vector section size of the machine. Such a choice ensures that most vector operations use maximum length vectors, giving optimal performance, while allowing segments of an adequate number of rows to reside in the cache simultaneously.

Parallel Processing

The reason that we are able to reduce cache requirements by operating on strips of the matrix being updated is that each of the strips is independent of the others, i.e., they do not share data. Since the strips are independent, they can be updated in parallel if more than one processor is available. The IBM ES/3090 family can have up to six processors, so parallel processing can reduce the turnaround time of a high-priority job. Of course, enabling an application program, such as the ANSYS program, to exploit parallel processing does not guarantee that a reduction in elapsed time will occur when the job is run. Whether a user has the dispatching priority to acquire multiple processors while running a parallel job is a political decision that is often made by such people as system administrators.

We follow three guidelines when enabling an application program for parallel processing.[6] The first guideline is that the total CPU time of a parallel run should not be significantly larger than the CPU time of a serial run of the same problem. Although the purpose of parallel processing is to reduce the elapsed time of a job, the total CPU time, the sum of the CPU times on each of the processors, does not decrease. In fact, it always increases by some amount because of the overhead in handling the parallel tasks. The first guideline reminds the programmer to minimize the overhead of parallel processing. The second guideline is that parallel performance should not come at the expense of the vector performance. Thus, making the strips of the matrix narrower than the VSS in order to increase the number of parallel pieces is not an acceptable solution because doing so would degrade vector performance. Since degrading vector performance increases the CPU time of a job, the second guideline is really a specific instance of the first one. The third guideline is that a parallel job should effectively use all of the processors that are available to it. To avoid having some processors waiting while others are computing, the computational load of each parallel task should be as nearly balanced as possible.

For the overhead of a parallel application to be relatively small, the amount of CPU time required to compute a piece of the problem must be large compared to the overhead of handling the parallel task. Problems with larger independent pieces have smaller parallel overheads when the same utilities control the parallel processors. Since the size of the independent pieces of the update increases with the rank of the update when the matrix size is constant, updates involving more rows being eliminated incur smaller parallel overheads. The rank-n update with its potential for a large grain size is a good match for the VS FORTRAN Multitasking Facility, which provides facilities for controlling parallel tasks.

Satisfying the second and third guidelines requires that the update be divided into pieces that are nearly equal in size and that have a vector length of the VSS. In order to increase our flexibility in dividing the update into independent pieces while maintaining an adequate vector length, we enabled the rank-n algorithm to update only the range of rows specified by ja and jb in Example 1. In addition, we wrote two additional subroutines, MULTI and DRIVER, that enable the rank-n update to run in parallel. If the user specifies that more than one processor should be used to perform the rank-n update, MULTI partitions the update into nearly equal pieces, based on the number of floating-point operations in each piece. For instance, Figure 3 shows how a matrix would be divided among four processors. The numbers in the matrix indicate the processor on which each piece would run. After partitioning the matrix, MULTI calls the MTF routine DSPTCH to invoke DRIVER in each of the parallel tasks. Each instance of DRIVER then calls the rank-n update for each of the portions of the matrix assigned to it. For instance, in Figure 3 we see that the piece of the matrix assigned to processor 3 actually consists of portions of two strips.

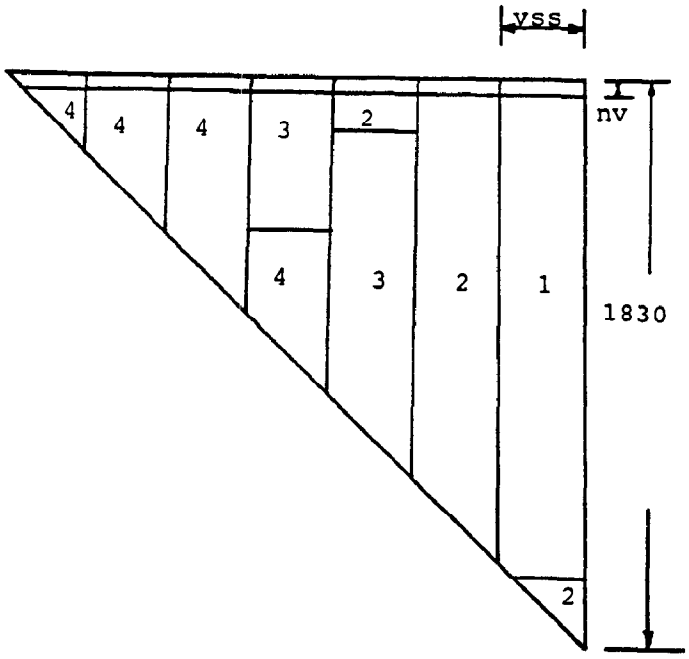

Figure 3. An example of distributing among four processors a matrix to be updated. Carefully choosing the sizes of the pieces that are assigned to each processor balances the computational load of each processor.

Performance

Having described techniques that enable the rank-n update to exploit the cache, vector processing, and parallel processing, we now discuss its performance. We begin by reporting the performance of the rank-n update itself. We then examine its performance in the context of the ANSYS program, an application which heavily uses the rank-n update. The performance measurements for the ANSYS jobs illustrate Amdahl's Law[7] and show that the scalar performance of the computer limits the speedup of the application.

Rank-n Update Performance

We measured the performance of the rank-n update with a program that timed the update for different values of matrix size and rank of the update. Varying both parameters allowed us to determine how effectively the update used the cache, vector processing, and parallel processing under varying conditions. In particular, the measurements show the ranges of matrix size and rank of the update that attain the best performance. As we mentioned previously, increasing the rank of the update improves cache reuse as long as the cache does not overflow. The performance measurements guide us in choosing the rank of the update to use in the wavefront solver and indicate how sensitive the performance of the update is to the particular choice. In all of the measurements, we divided the total number of floating-point operations in the update by the time for the update to obtain the values in terms of millions of floating-point operations per second (Mflops).

Figure 4 shows the performance of two versions of the rank-n update on the IBM 3090 Models 180 and 180S. We performed the measurements on dedicated machines running the MVS/XA 2.2.0 operating system. The rank of the update is 32, and the matrix size, n, varies from about 100 to 2100. We show the performance of both our original rank-n update and the new version, in which we improved the usage of cache. In our original version, we neglected to copy the segments of the rows being eliminated into a contiguous block. Clearly, many values of n cause inefficient use of the cache, degrading performance. The new version performs the copying operation, so it reduces the effects associated with having a stride that is divisible by a high power of two. The remaining narrow valleys in the plot of the performance result from cache misses involving the stride of the B matrix. Unfortunately, the wavefront solver generates the B matrix in the stride-one direction, and the rank-n update uses it only once, so we cannot improve the overall performance by simply creating B in the transposed order. The difference between the 3090 and 3090S models is mostly due to cycle time improvements that result in a nearly constant speedup of about 1.4. However, the 3090S model also reduces the effect of stride on performance because of its larger cache.

Figure 5 shows the performance of the same two versions of the rank-n update on the same models as Figure 4. In Figure 5 the size of the matrix is fixed at 1502 and the rank of the update, nv, varies from 1 to 50. We see that both

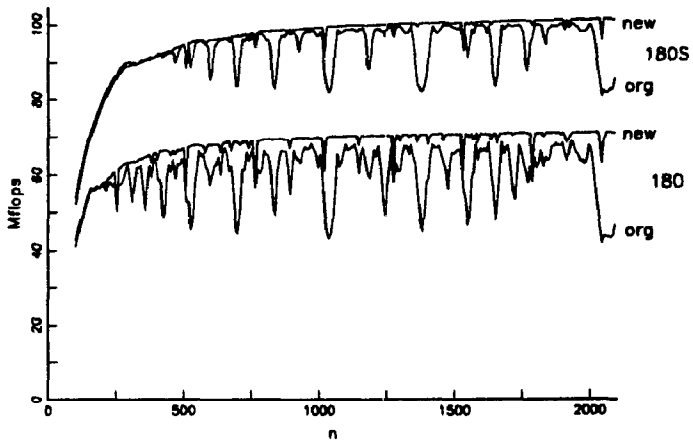

Figure 4. The performance of the rank-32 update as a function of matrix size. Performance measures of two versions of the rank-n update on the IBM 3090 Models 180 and 180S are plotted.

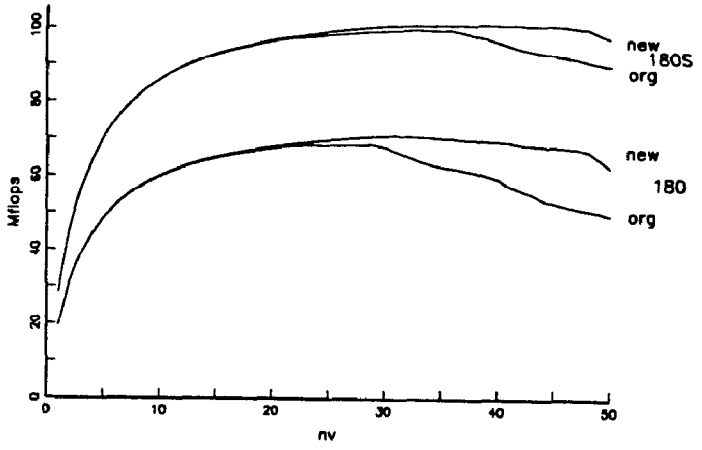

Figure 5. The performance of the rank-n update for a matrix of size 1502 as a function of the rank of the update. Performance measures of two versions of the rank-n update on the IBM 3090 Models 180 and 180S are plotted.

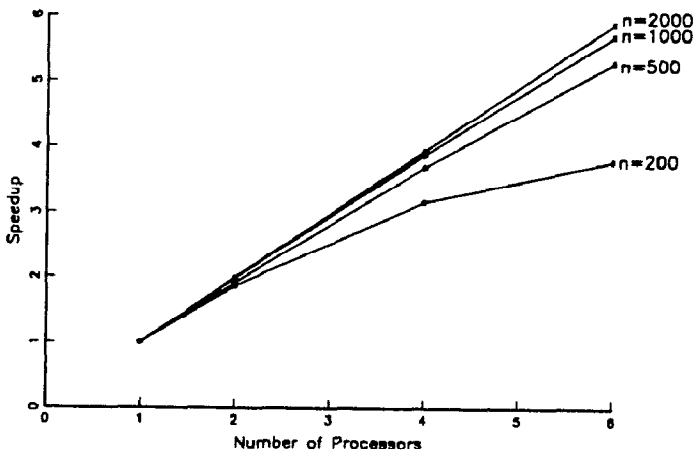

Figure 6. The speedup in elapsed time of the parallel rank-32 update as a function of the number of processors on a dedicated IBM 3090 Model 600S. The size of the matrix being updated is indicated for each plot.

versions of the update first increase in performance as nv increases. As we mentioned previously, the improvements in performance result from increasing the reuse of cache and from reducing the overhead of load and store operations. However, the rank at which cache overflows, causing cache misses and degrading performance, occurs at a lower value of nv in the original version of the update. Copying the segments of the rows being eliminated into a contiguous block clearly improves the efficiency of filling the cache, increasing the effective size of the cache. In addition to increasing the maximum performance of the update, the range of good performance is substantially larger, as both Figures 4 and 5 illustrate.

We now examine the effect of parallel processing on the performance of the rank-n update. First, we check to see how well we satisfied our first and second guidelines for parallel processing, which state that the increase in total CPU time should be small. We measured the total CPU times of the parallel updates and computed the CPU overhead, the increase in the total CPU time over that of the serial update, for each measurement. A parallel rank-32 update of a matrix of size 200 running on six processors has a CPU overhead of less than 4%, which decreases to much less than 1% as the size of the matrix increases to 1000.

Figure 6 illustrates the speedup in elapsed time that parallel processing achieves for a rank-32 update of several sizes of matrices on a dedicated IBM 3090 Model 600S. Because the overhead is small and the computational loads of the processors are balanced, the speedup for the large matrices is nearly equal to the number of processors. The parallel speedups clearly indicate that we satisfied all three guidelines for parallel processing. We measured a maximum value of 595 Mflops during a rank-32 update of a matrix of size 2000 on a dedicated IBM 3090 Model 600S running the MVS/XA 2.2.0 operating system.

Application Performance

Obtaining large Mflop numbers is rewarding, but to understand the true relevance of such numbers, we must examine them in the context of an application that incorporates the rank-n update. We provide such a context with the ANSYS program, which introduced a new version of the wavefront solver that invokes the rank-n update. Swanson Analysis Systems, Inc. provides several ANSYS benchmarks for measuring the performance of the ANSYS program. The benchmarks are static analysis problems that study a cantilevered plate with a force loading applied to the free end of the plate. Eight-node, three-dimensional solid isoparametric elements (ANSYS STIF45) represent the plate. The plate has one element through its thickness, while the number of elements along its length and width vary in each of the benchmarks. We report results for the S3 benchmark, which uses 10 elements along the length and 300 elements along the width of the plate. The S3 benchmark contains 6622 nodes and 18060 degrees of freedom, and its maximum and RMS wavefronts are 1818 and 1619, respectively. We have reported results for other ANSYS benchmarks previously.[8]

Figure 7 shows the speedup in elapsed time of the ANSYS S3 benchmark as a function of the Mflops in the rank-n update. The Mflop numbers for each of the data points are typical values for each version of the rank-n update on several IBM 3090S models. Starting from the left of Figure 7, the data points represent the scalar rank-one update, the

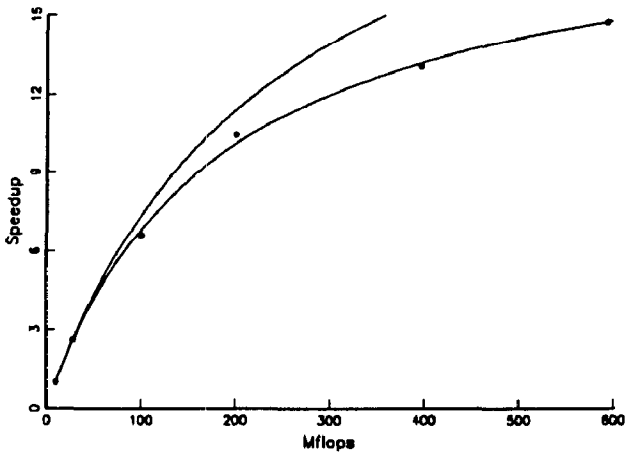

Figure 7. An illustration of Amdahl's Law. As the performance of the rank-n update increases, the speedup of the ANSYS S3 benchmark increases more and more slowly. If the remaining scalar component of the job were reduced, the speedup would follow a higher curve, as illustrated.

vector rank-one update, and the vector rank-n updates on one, two, four, and six processors. Clearly, the speedup of S3 increases more and more slowly as the performance of the rank-n update increases, illustrating Amdahl's Law. We can closely fit the data points in Figure 7 with the function:

$$S = \frac{1}{(1-p) + (\frac{p}{S_{rankn}})}$$

where S is speedup in elapsed time of the ANSYS program over its scalar version, p is the portion of the scalar job spent in the rank-n update, and S_{rankn} is the speedup of the rank-n update over its scalar version. In Figure 7, the curve that follows the data points uses $p = .948$.

Running on the six processors of the IBM 3090 Model 600S, the ANSYS program that uses the rank-n update achieves a speedup of 14.7 over the scalar version. Even if the number of Mflops obtained by the rank-n subroutine could be increased without bound, the speedup of the S3 benchmark would be improved by only 31% over the speedup achieved on the IBM 3090 Model 600S. On the other hand, if the portion of the ANSYS job outside the rank-n update were decreased from 5.2% to 4% in the scalar version, the speedup of the S3 benchmark would follow the higher curve in Figure 7, where $p = .96$. In that case, the speedup on the IBM 3090 Model 600S would be 21% greater than it is for the current version. Thus, we see that the scalar performance of the computer affects the speedup of the ANSYS job much more strongly than an additional improvement of rank-n update. The portion of the ANSYS job outside the rank-n update is very important because the rank-n update consumes a smaller fraction of the elapsed time of the job as its performance improves. Further improvements in the rank-n update would yield little improvement in the ANSYS program on the IBM 3090 Model 600S, unless the other portions of the program were improved substantially. Of course, further improvements in the rank-n update would have a significantly larger effect on ANSYS jobs running on only one processor.

Conclusions

The new wavefront solver in the ANSYS program uses a rank-n update, which can be implemented more efficiently than the rank-one update of the older solver, especially on a machine with hierarchical memory. Carefully tuning the rank-n update to exploit the cache and the vector processing capability of the IBM ES/3090 VF achieved large reductions in both CPU time and elapsed time for the ANSYS jobs that we timed. In addition, the rank-n update is suitable for parallel processing, which can further reduce the elapsed time of an ANSYS job. Because the performance of the matrix update increased so much from the scalar, rank-one version to the vector-parallel, rank-n version, the performance of the remainder of the ANSYS job limits the overall speedup. Besides achieving good performance, the parallel rank-n update incurs a CPU overhead of only a few percent, indicating that we satisfied our guidelines for parallel processing.

Acknowledgements

We thank S. A. Murgie, P. Narducci, J. A. Swanson, L. Wagner, and C. R. Rogers of Swanson Analysis Systems, Inc. for their assistance with the ANSYS program.

References

1. B. M. Irons, *A Frontal Solution Program for Finite Element Analysis*, International Journal for Numerical Methods in Engineering 2 (1970) 5.

2. G. J. DeSalvo and R. W. Gorman, *ANSYS Engineering Analysis User's Manual*, (Swanson Analysis Systems, Inc., Houston, Pennsylvania, 1987).

3. R. Hessel, M. Myszewski, G. Brussino, J. Swanson, and L. Wagner, *Timing the ANSYS Kernel LSOLVE for a Parallel Computer*, Supercomputing '88, November 14-18, 1988, Orlando, Florida.

4. E. Hinton and D. R. J. Owen, *Finite Element Programming*, (Academic Press, 1977) pp. 170-206.

5. B. Liu and N. Strother, *Programming in VS Fortran on the IBM 3090 for Maximum Vector Performance*, Computer **21** (June, 1988) 65.

6. R. C. Agarwal and F. Gustavson, *A Parallel Implementation of Matrix Multiplication and LU Factorization on IBM 3090*, IBM Symposium on Parallel Processing, October 19-21, 1988, Poughkeepsie, New York.

7. G. M. Amdahl, *AFIPS Conference Proceedings* **30** (Thompson, Washington D.C., 1967) 483.

8. S. M. Chang and M. A. Johnson, *An Improved Version of the ANSYS Program for the IBM 3090 VF*, ANSYS 1989 Conference Proceedings, 4-75.