INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS, VOL. 17, 589–603 (1993)

VECTORIZED POISSON SOLVERS FOR THE NAVIER–STOKES EQUATIONS

S. W. ARMFIELD

Department of Civil and Environmental Engineering, University of Western Australia, Perth 6009, Western Australia

SUMMARY

The advent of vector and massively parallel computers offers researchers the possibility of enormous gains in execution time for scientific and engineering programs. From the numerical point of view, such programs are frequently based on the inversion of sparse, diagonally banded matrices. Conventional scalar solvers often perform poorly on vector machines due to short effective vector lengths, and thus appropriate methods must be chosen for use with vector machines. In this paper a number of commonly used solvers are tested for the Navier–Stokes equations, in both scalar and vector form, on two vector architecture machines. A new method is presented which performs well in both vector and scalar form on a range of vector architectures.

KEY WORDS Vector Poisson Navier–Stokes Supercomputer Thomas

1. INTRODUCTION

The advent of modern high-speed vector machines, with peak performance of the order of gigaflops, means that many previously intractable problems in computational fluid dynamics may now, at least theoretically, be addressed. Unfortunately, the rate of development of vectorized algorithms has not kept pace with the hardware development and the tantalizing prospect of gigaflop speeds still seems far away for many engineering codes, which may exploit only a few per cent of the peak vector performance of modern supercomputers. As a result, much effort is currently being expended on adapting existing scalar codes for use with supercomputer architectures[1–3] and developing new numerical techniques suited to vector and parallel machines.[4]

The numerical solution of the Navier–Stokes equations (the governing equations of fluid flow and the basis of most fluid codes) reduces at the numerical level, once a local discretization such as a finite-difference or finite-element method has been applied, to the inversion of a number of sparse, regularly structured, diagonally banded matrices. These matrices are, at least in their ordering, similar to that obtained when the discretization is applied to Poisson's equation. Such a matrix system is shown in Section 2. Due to this similarity at the matrix level, the methods presented here will be described in the context of a Poisson solver, while results will also be presented for the methods embedded in a full Navier–Stokes solver.

Although direct solvers are available for the matrix inversion described above, iterative methods are frequently used due to their relatively simple programming requirements. In this paper we will consider iterative methods only. One of the most popular iterative methods is the Alternating Direction Implicit (ADI) code in which the domain is swept repeatedly in orthogonal directions. Thus, in two dimensions the domain is swept first in, for instance, the x direction and then in the y direction in a total iteration. The ADI algorithm is presented in detail in the next

0271–2091/93/190589–15$12.50

Received June 1992
Revised April 1993

section. Briefly, it consists of the Thomas tridiagonal inversion algorithm being applied successively to each of the lines in each orthogonal direction until a converged solution is attained. The ADI algorithm performs well on scalar machines, converging in relatively few iterations; unfortunately, it vectorizes poorly and therefore is not well suited to vector machines.[5]

Classical explicit codes, such as the Jacobi method described in the next secion, vectorize very well but perform poorly on scalar machines due to the large number of iterations generally required to attain a converged solution.[6] Jacobi-type codes in which a different ordering is used to speed the convergence are those such as the red/black (RB) code, also presented in the next section. Between the fully vectorized Jacobi-type codes and the fully scalar codes such as the ADI are a number of methods which partially vectorize, but attain convergence in fewer iterations than the Jacobi methods. Examples of these intermediate methods considered in the present paper are the line by line relaxation (LRLX) and Zebra methods, presented in the next section. These codes will typically have effective vector lengths of the order of the square root of the total number of points for two-dimensional problems which, together with the use of strided memory addressing, limits their performance on many of the relatively inexpensive vector machines that are now available. One such machine is the STAR 910 VP, a SPARC-based vector architecture machine for which timing results are presented in this paper. The purpose of this paper is to present a vectorized Poisson solver that gives a good vectorized performance on machines such as the STAR, while also performing well on scalar machines and vector architecture supercomputers such as the FUJITSU VP2200, for which timing results are also presented in this paper.

The new method presented in this paper is a modification of the Zebra method, ensuring that most of the floating point operations are carried out in loops with contiguous memory addressing and effective vector lengths of the order of the total number of points, but with the same convergence characteristics as the unmodified method. It will be called the Modified Zebra (MZ). The modification results in the degree of vectorization approaching that of the fully explicit RB method, and is particularly suited to machines requiring long vectors and contiguous memory addressing.

Timing results for the methods, obtained in both scalar and vector form on the STAR 910 VP and FUJITSU VP2200, will be presented. The present code has been optimized specifically for two-dimensional problems of order 100×100 on the STAR; however, as will be shown the method also performs well on the FUJITSU.

The contents of the remainder of the paper are as follows. In Section 2 the methods are described in detail and in Section 3 timing results obtained using the various methods in both scalar and vector modes are presented. Section 4 consists of a discussion of the results and Section 5 contains the conclusions.

2. METHOD

A Poisson's equation $u_{xx}+u_{yy}=f$, where f is a known function of (x, y), is to be solved for $u(x, y)$ in the domain $0 \geq x \geq 1$, $0 \geq y \geq 1$, with suitable boundary conditions. The domain is discretized by the set x^i, y^j, $1 \leq i \leq I$ and $1 \leq j \leq J$, with a constant spacing between points of h such that $x^1=0$, $x^I=1$, $y^1=0$, $y^J=1$.

The Poisson equation is discretized, using the standard five-point centred-difference approximation, to produce the linear system

$$\frac{(u^{i+1,j}+u^{i,j+1}-4u^{i,j}+u^{i-1,j}+u^{i,j-1})}{h^2}=f^{i,j}.$$

This discrete formulation is rewritten in terms of one-dimensional vectors using the following transformation. The data points $u^{i,j}$ are expressed as u^k, where $k=(j-1)I+i$. u^k is then a one-dimensional vector of length $K=I\times J$ and the above system can be written as

$$\frac{(u^{k+1}+u^{k+I}-4u^k+u^{k-1}+u^{k-I})}{h^2}=f^k.$$

When written in matrix form, and with the inclusion of boundary values, this system has a sparse banded matrix on the left-hand side, operating on the unknown vector u, of the form shown in Figure 1, where the non-zero diagonal bands of the matrix are labelled as α, β, γ, δ and ε for convenience. Typical values for the diagonals are then

$$\alpha^k=1/h^2,$$
$$\beta^k=1/h^2,$$
$$\gamma^k=-4/h^2,$$
$$\delta^k=1/h^2,$$
$$\varepsilon^k=1/h^2,$$

All the techniques described in this paper consist of repeatedly sweeping through this system, correcting an estimated u with each sweep, until a preset convergence criterion is attained, based on the residual of the discrete equation. The stored u at the final iteration is then said to be the solution of the algebraic system and an approximation to second degree accuracy of the solution of the continuous system.

Using the notation given above, the Jacobi method with relaxation factor *Relx* may be written as follows:

Do 1 while(*reside*$<$*conv*)

Do 2 $k=1, K$

$$v^k=((f^k-\delta^k u^{k+1}-\beta^k u^{k-1}-\varepsilon^k u^{k+I}-\alpha^k u^{k-I})\hat{\gamma}^k-u^k)\,Relx+u^k$$

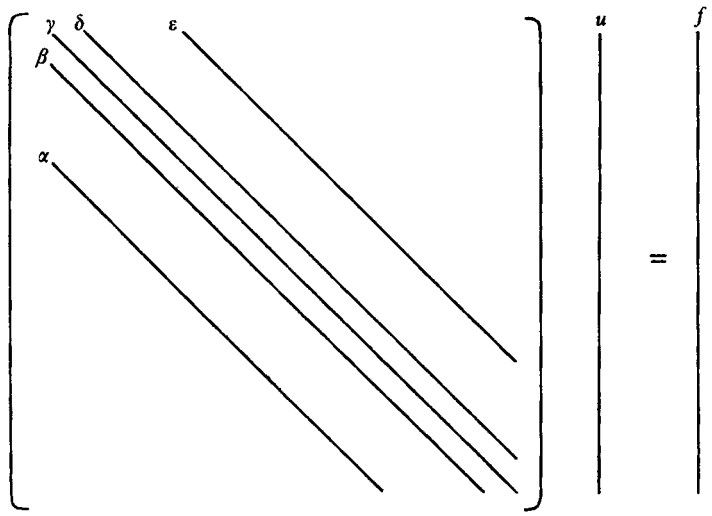

Figure 1

end 2

Do 3 $k=1, K$

$u^k=((f^k-\delta^k v^{k+1}-\beta^k v^{k-1}-\varepsilon^k v^{k+I}-\alpha^k v^{k-I})\hat{\gamma}^k-v^k)\,Relx+v^k$

end 3

Do 4 $k=1, K$

$resid=resid+abs(f^k-\delta^k u^{k+1}-\beta^k u^{k-1}-\varepsilon^k u^{k+I}-\alpha^k u^{k-I}-\delta^k u^k)$

end 4

end 1,

where $\hat{\gamma}=1/\gamma$. Typically, divisions take about three times as long to perform as multiplications, so whenever possible it is advisable to replace divisions with multiplications. However, a commensurate change in performance will only be seen if the speed is not otherwise limited. Generally, for poorly vectorized codes or for scalar codes little improvement is achieved by replacing divides. As can be seen, the Jacobi method is very simple to code and fully explicit, with no dependencies, and thus the effective vector length is K. A total iteration consists of 35 floating point operations per node. Such a scheme will be said to be fully vectorized and will run at a substantial percentage of the peak speed on a vector machine. However, the lack of dependency that allows good vectorization means that the coupling is poor and, as a result, the Jacobi method is inefficient in terms of the number of iterations required to attain a converged solution. On the STAR 910 VP a 50% increase in performance is obtained by explicitly caching the arrays v and u using the compiler directive $C*\$*CACHE::u, v$; timing results presented in the next section were obtained with these variables cached. The residual calculation loop shown above is identical for all the methods presented and so will not be included in the subsequent descriptions.

An improvement in the efficiency of the basic Jacobi method may be obtained when the system is stepped through with a stride of 2, alternately starting on the first and second element, as follows:

Do 1 $k=1, K, 2$

$u^k=((f^k-\delta^k u^{k+1}-\beta^k u^{k-1}-\varepsilon^k u^{k+I}-\alpha^k u^{k-I})\hat{\gamma}^k-u^k)\,Relx+u^k$

end 1

Do 2 $k=2, K, 2$

$u^k=((f^k-\delta^k u^{k+1}-\beta^k u^{k-1}-\varepsilon^k u^{k+I}-\alpha^k u^{k-I})\hat{\gamma}^k-u^k)\,Relx+u^k$

end 2.

The outer loop of this method, associated with the calculation of the residual and the repeated sweeping, is identical to that of the Jacobi method and so is not included in this or any of the following methods. When the discretization is chosen so that I is odd, this is the red/black method, so named because the first of the loops effectively covers the red squares of an imaginary draft board, while the second covers the black. The degree of vectorization will be of the order of half the total vector length, $K/2$; however, the need to calculate a stride of 2 will reduce the effective vector speed on some computers. If I is even, the above scheme will no longer be red/black, but will be a one-directional banded algorithm, in which the first loop covers the odd j lines and the second covers the even j lines. This is a limitation of this method of implementing a red/black algorithm, which will be discussed in Section 4. To ensure vectorization of the loops in the RB method it is necessary to include a compiler directive identifying these loops as having no vector dependencies; otherwise the compiler would assume that terms of the form u^{k-I} produce

a dependency and would not attempt vectorization. The directive used on the STAR is *CDIR$IVDEP*, and on the FUJITSU, **VOCL LOOP, NOV REC*. The maximum speed for this method on the STAR was obtained by explicitly caching u.

The line by line relaxation method consists of sweeping the domain in orthogonal directions, with the solution on each line being coupled to the most recently calculated solution on the previous line, as follows:

Do 1 $j=1, J$

  Do 2 $i=1, I$

    $k=(j-1)I+i$

    $v^k=((f^k-\delta^k v^{k+I}-\beta^k u^{k-1}-\varepsilon^k u^{k+I}-\alpha^k u^{k-I})\hat{\gamma}^k-u^k)\,Relx+u^k$

  end 2

end 1

Do 3 $i=1, I$

  Do 4 $j=1, J$

    $k=(j-1)I+i$

    $u^k=((f^k-\delta^k v^{k+I}-\beta^k u^{k-1}-\varepsilon^k v^{k+I}-\alpha^k v^{k-I})\hat{\gamma}^k-v^k)\,Relx+v^k$

  end 4

end 3.

The LRLX method has a line by line dependency and thus the effective vector length will be I for loop **2** and J for loop **4**. Additionally, loop **4** has a stride of I, further reducing the speed of this method on machines such as the STAR. Compiler directives were included, identifying the inner loops as having no vector dependencies, to ensure vectorization.

For the sake of brevity, only the first of the orthogonal sweeps comprising a total iteration will be presented for the ADI, Zebra and MZ methods. For the ADI method the sweep in the j direction is as follows:

Do 1 $j=1, J$

  Do 2 $i=1, i$

    $k=(j-1)I+i$

    $r^k=f^k-\alpha^k u^{k-I}-\varepsilon^k u^{k+I}$

  end 2

  Do 3 $i=2, I$

    $k=(j-1)I+i$

    $v=\beta^k/\gamma^{k-1}$

    $\gamma^k=\gamma^k-v\,\delta^{k-1}$

    $r^k=r^k-v\,r^{k-1}$

  end 3

  Do 4 $i=I, 1, -1$

    $k=(j-1)\,I+i$

    $u^k=((r^k-\delta^k u^{k+I})/\gamma^k-u^k)\,Relx+u^k$

  end 4.

end 1

The ADI algorithm solves a tridiagonal system along each j line using the Thomas algorithm, the results of which are immediately coupled into the solution on the next j line, followed by an equivalent sweep in the orthogonal direction for the i lines, which is not shown. The use of the Thomas algorithm on each line means that each node is directly coupled to its neighbours on that line while the line by line nature of the algorithm means that the solution on any given line is coupled to that on the preceding line, in a single sweep. This strong coupling is the reason for the efficiency of the ADI method; however, it prevents vectorization. Thus, loop **2** will have an effective vector length of I while the remainder of the loops will have an effective vector length of 1. A single iteration of the ADI method requires 49 floating point operations per node, with divides counted as three operations and including the residual calculation. Replacing divides with multiplies resulted in no increase in performance for the ADI method, or the Zebra method that follows.

The Zebra method consists of two sweeps in each orthogonal direction. On the first sweep the new solution on the odd numbered lines is calculated and on the second sweep the most recently calculated solution on the odd numbered lines is used to update the solution on the even numbered lines, with a partially vectorized Thomas algorithm being used on each of the lines. The sweeps in the j direction are as follows:

Do 1 $n=1, 2$
  Do 2 $i=2, I$
  Do 3 $j=n, J, 2$
    $k=(j-1)I+i$
    $v=\beta^k/\gamma^{k-1}$
    $\gamma^k=\gamma^k-v\,\delta^{k-1}$
    $f^k=f^k-\alpha^k\,u^{k-I}-\varepsilon^k\,u^{k+I}-v\,f^{k-1}$
  end 3
  end 2
  Do 4 $i=I, 1, -1$
  Do 5 $j=n, J, 2$
    $k=(j-1)\,I+i$
    $u^k=((f^k-\delta^k\,u^{k+1})/\gamma^k-u^k)\,Relx+u^k$
  end 5
  end 4
end 1.

As can be seen, the ordering of the indices for loops **3** and **5** has been interchanged when compared to the ADI method, which allows a partial vectorization of the Thomas algorithm, with effective vector lengths of I. The zebra banding (whereby first odd and then even lines are calculated, producing a line by line coupling), although of a form different from that of the ADI method, introduces stride and further reduces the effective vector length to $I/2$. Compiler directives must be used to ensure that the inner loops are vectorized, while caching u and f provides a further $\sim 10\%$ improvement in performance on the STAR. In a total iteration, including the residual calculation, the Zebra method has 49 floating point operations, of which all but the 11 residual operations have effective vector lengths of $I/2$ and $J/2$ with stride.

The method proposed in the present paper modifies the Zebra method described above to place most of the floating point operations in loops with effective vector lengths of K as follows:

count = 0
Do 1 *while* (*resid* < *conv*)
  count = *count* + 1
  Do 2 $k = 1, K$
    $q^k = \alpha^k\ u^{k-I} + \varepsilon^k\ u^{k+I}$
    $r^k = f^k + q^k$
  end 2
  if (*count* = 1) *then*
    Do 3 $i = 2, I$
      Do 4 $j = 1, J$
        $k = (j-1)\ I + i$
        $v^k = \beta^k / \gamma^{k-1}$
        $\gamma^k = \gamma^k - v^k\ \delta^{k-1}$
        $r^k = r^k - v^k\ r^{k-1}$
        $\hat{\gamma}^k = 1/\gamma^k$
      end 4
    end 3
    Do 5 $i = I, 1, -1$
      Do 6 $j = 1, J$
        $k = (j-1)\ I + i$
        $\mu^k = \delta^k\ \hat{\gamma}^{k+I}$
        $r^k = r^k - \mu^k\ r^{k+I}$
      end 6
    end 5
  else
    Do 7 $i = 2, I$
      Do 8 $j = 1, J$
        $k = (j-1)\ I + i$
        $r^k = r^k - v^k\ r^{k-1}$
      end 8
    end 7
    Do 9 $i = I, 1, -1$
      Do 10 $j = 1, J$
        $k = (j-1)\ I + i$
        $r^k r^k - \mu^k\ r^{k+I}$
      end 10
    end 9
  endif
  Do 11 $k = 1, K$
    $u^k = ((r^k - q^k + \varepsilon^k\ r^{k+I}\ \hat{\gamma}^{k+I} + \alpha^k\ r^{k-I}\ \hat{\gamma}^{k-I})\ \hat{\gamma}^k - u^k)\ Relx + u^k$
  end 11
end 1.

This method uses two partially vectorized sweeps to invert the tridiagonal system in the same manner as the Zebra, loops **7** and **9**, but rather than solving for u in the second of the sweeps, the system is reduced to diagonal form. A final sweep of the form $u = r/\gamma$ is then required to obtain the

new estimate for u which is accomplished in loop **11**, with the $1/\gamma$ replaced with $\hat{\gamma}$. A line by line coupling is included in this loop by subtracting the direct effect of the ε and α components from the previous iteration, which have been stored in the vector q, and then adding back their influence obtained from the present iteration. This is an explicit loop over the entire domain that will fully vectorize.

In the first iteration of the MZ method, $count = 1$, the vectors v, μ, $\hat{\gamma}$ are used to store quantities that will be used unchanged on subsequent iterations. This is a particularly important part of the algorithm and effectively doubles the speed of the code. All but the first iteration consist of 49 floating point operations per node, including the residual calculation, of which 41 are contained in loops with effective vector lengths of K and 45 have contiguous memory addressing. The first iteration consists of 71 floating point operations per node. Provided the number of iterations is of order 10 or greater, the larger number of operations in the first iteration does not have a significant effect on the overall speed. Once again, compiler directives are used to ensure that the inner loops **4**, **6**, **8** and **10** will vectorize, while on the STAR $\sim 50\%$ improvement in performance was obtained by caching the arrays u, r, $\hat{\gamma}^k$ and q.

3. RESULTS

The STAR 910 VP is a SPARC-based vector architecture machine with a vector clock speed of 66 MHz. It has a single multiply/add/divide pipe and in one clock can perform either a single vector multiply/add, or a chained vector/scalar multiply/add ($x = uv + c$), where x, u and v are vectors and c is a scalar. Divides take three clocks to perform. A single load/store operation to memory can be performed in each clock, but the machine has a 1 Mbyte cache to which three load/store operations can be performed in each clock. The machine has a theoretical peak of 132 mflops which can only be attained for operations involving a scalar. For vector operations of the type used in the methods presented in this paper, the STAR has an effective peak of 66 mflops. Strided operations with constant stride require an additional clock per load/store, effectively halving the speed. Figure 2 shows a plot of vector length against Mflops for the Jacobi method. The performance of the STAR asymptotes to a peak of 60 mflops at a vector length of 6000, with 90% of the peak being attained at a vector length of 2000. In particular, it is noted that vectors of length 100 attain only 30% of the peak and vectors of length 50 attain only 16% of the peak.

The FUJITSU VP2200 is a vector architecture machine with two load/store pipes, a 32 kbyte vector register, two multiply/add pipes and a divide pipe, with a vector clock speed of 312 MHz. Each multiply/add pipe can produce a chained vector multiply and add each clock, while the divide pipe produces a divide every three clocks. The FUJITSU has a theoretical peak of 1·248 Gflops. However, as only two load/stores can be performed in each clock, for operations of the type considered here the effective peak will be between 312 and 624 mflops. For instance, a vector operation of the form $u = ax + b$ will require one clock to load a and x and one clock to load b and store u, and will thus have a theoretical peak of 312 mflops. An operation of the type $u = ax + by + c$ will have a theoretical peak of 416 mflops and an operation of the form $u = ax + by + cz + w$ will have a theoretical peak of 468 mflops. Operations of this type may be considered as n chained multiply/adds, with $n = 1$ for $u = ax + b$, $n = 2$ for $u = ax + by + c$, and so on. The number of clocks will then be $n + 1$ and the number of operations will be $2n$; thus, the speed for the FUJITSU will be $312 \times (2n/(n+1))$, which will clearly asymptote to 624 mflops for large n. The FUJITSU does not require an extra clock to compute stride; however, constant stride will still lead to a reduction in performance if it results in memory bank conflicts with the 64-way interleaved memory, as well as effectively reducing the vector length as noted above. Figure 3 shows a plot of the vector length against mflops for the Jacobi method attained with the VP2200.

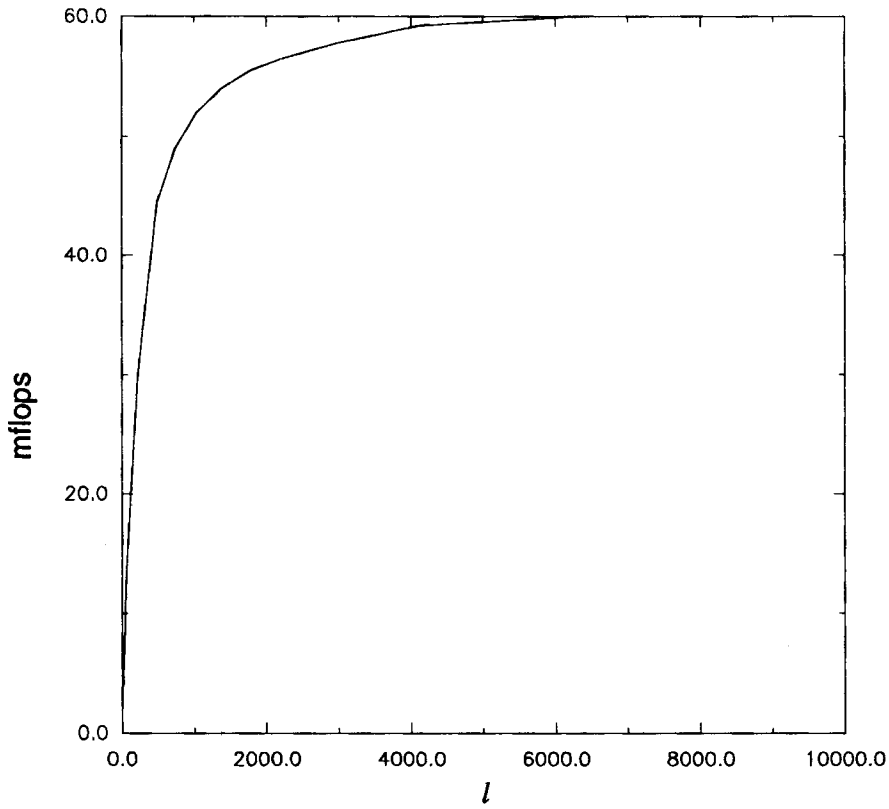

Figure 2. Floating point speed (mflops) against vector length for the STAR

The performance asymptotes to a peak of 406 mflops, with 90% of the peak being attained at $l = 200$. Vectors of length 100 attain 80% of peak performance, while vectors of length 50 attain 70% of peak performance.

Comparing the STAR to the FUJITSU, it is apparent that the FUJITSU is a lot faster, simply as a function of a higher clock speed and the ability to perform more operations per clock. It is also clear that the FUJITSU has proportionally considerably better performance for short vectors and strided access. These points will be considered in the next section.

Timing results have been obtained for two problems. The first is a solution of Poisson's equation with mixed Neumann and Dirichlet boundary conditions. The second is a solution of the Navier–Stokes equations for natural convection in a cavity with an applied horizontal temperature gradient. The results presented were obtained on the University of Western Australia's Centre for Water Research STAR 910 VP and on the Australian National University Super Computing Centre FUJITSU VP2200. Both these machines have vectorizing compilers which were used for compilation; no direct vector calls were used. Tables I and II present timing results for the Poisson's problem on the STAR and the FUJITSU, respectively, with the convergence set to $1{\cdot}0 \times 10^{-4}$ and no relaxation. It should be noted that the performance obtained included the residual calculation, which was always fully vectorized. This will influence the performance of the poorly vectorized and non-vectorized codes.

In scalar form on the STAR the most efficient algorithm is the ADI, followed by the MZ, the RB and the Zebra methods. In vector form the MZ is the most efficient, with the Jacobi and RB also obtaining a significant vector speed-up. The Zebra and ADI perform poorly in vector form,

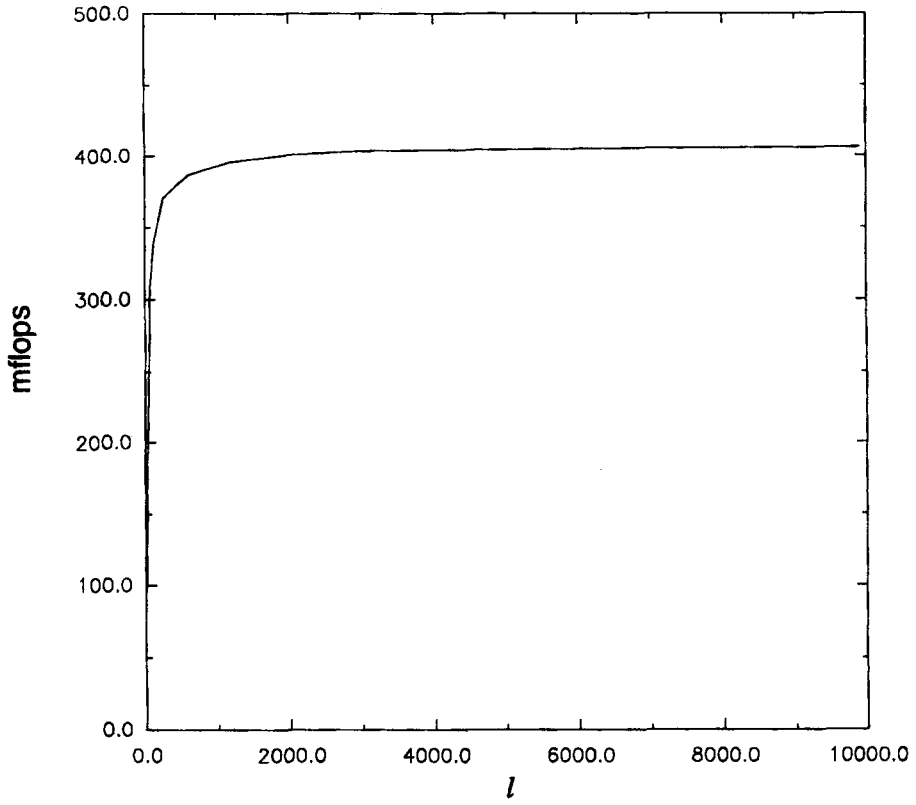

Figure 3. Floating point speed (mflops) against vector length for the FUJITSU

Table I. Timing results for Poisson's equation on STAR

Solver	Jacobi	Red/black	LRLX	Zebra	ADI	MZ
Iterations	6715	6724	5027	2090	1668	2179
mflops (scalar)	4·6	4·2	3·7	2·4	2·7	2·9
mflops (vector)	60	42	28	8·6	3·3	35
Time (sec—scalar)	508	365	484	418	297	346
Time (sec—vector)	38	36	63	115	247	29
Vector/scalar	13	10	7·7	3·6	1·2	12

Table II. Timing results for Poisson's equation on FUJITSU

Solver	Jacobi	Red/black	LRLX	Zebra	ADI	MZ
mflops (scalar)	28·0	27·0	16·3	19·3	13·4	22·6
mflops (vector)	400	400	314	230	37	307
Time (sec—scalar)	79	55	105	52	57	43
Time (sec—vector)	5·6	3·9	5·5	4·4	20	3·3
Vector/scalar	14	14	19	12	2·8	14

with the ADI, as expected, showing virtually no improvement in performance at all. These results were obtained on a uniform 99×99 mesh; changing to either a 100×99 or a 98×99 mesh, i.e. a mesh with an even number of points in the i direction, degrades the performance of the RB method by a factor of 50%, while the other five methods perform equivalently on odd and even meshes.

On the VP2200 there is a considerable variation in floating point speed in scalar as well as in vector form, with the MZ having a significantly greater scalar floating point speed than the ADI. The MZ is the most efficient scalar code, followed by the Zebra, RB and then the ADI. In vector form all methods except the ADI achieve significant improvements in floating point speed. The MZ is again the most efficient vector code; however, the Zebra achieves a good vector speed and performs proportionally considerably better than on the STAR, being the second most efficient vector code on the FUJITSU.

Tables III and IV present the timing results obtained for the algorithms embedded in a full unsteady two-dimensional Navier–Stokes solver. Details of the solver and the problem considered are available in Armfield[7] and Patterson and Armfield[8] and will not be presented here. In the Navier–Stokes solver the methods presented in this paper are used to sequentially invert the matrices resulting from a finite-volume discretization of the temperature equation, the two momentum equations and the pressure correction equation. For the inversion of the temperature and momentum equations an under-relaxation factor of 0·4 is used, while for the pressure correction equation no relaxation is used. The Navier–Stokes solver cycles iteratively through these four equations at each time step until a converged solution is attained, with the process then repeated at the next time step. In Navier–Stokes solvers of this type it is found most efficient to place a limit on the number of iterations of the pressure correction equation in each cycle. Best results were obtained when this parameter was tuned for each of the methods considered. For the Jacobi, RB, LRLX, Zebra, ADI and MZ methods, respectively the optimal pressure correction

Table III. Timing results for Navier–Stokes equations on STAR

Solver	Jacobi	Red/black	LRLX	Zebra	ADI	MZ
Time (sec—scalar)	109	79	85	54	36	47
Time (sec—vector)	13·3	11	13	16·3	28·2	5·6
Vector/scalar	8·2	7·2	6·5	3·7	1·3	8·4
Pressure iterations	1350	1260	810	190	160	231
Pressure time—scalar	100	68	65	37	23	36
Pressure time—vector	11·6	8·8	9·0	12·2	20	3·6

Table IV. Timing results for Navier–Stokes equations on FUJITSU

Solver	Jacobi	Red/black	LRLX	Zebra	ADI	MZ
Time (sec—scalar)	24·4	12·6	16	6·7	7·32	6·11
Time (sec—vector)	1·74	0·90	0·88	0·56	2·02	0·47
Vector/scalar	14	14	18	12	3·6	13
Pressure iterations	1350	1260	810	190	160	231
Pressure time—scalar	16·5	11·0	13·5	5·4	5·23	4·94
Pressure time—vector	1·18	0·79	0·71	0·45	1·87	0·38

iteration limits were found to be 150, 80, 45, 10, 10 and 20. The times presented are the execution times for a single time step obtained when the flow was approximately half way to full development for a divergence residual of 1×10^{-4} obtained on a non-uniform 99×99 mesh. Timings obtained at other stages during the flow development yield similar results.

Timing results for the STAR are presented in Table III. In Navier–Stokes solvers of this type the majority of the time is spent solving the pressure correction equation. For this reason, the time in the pressure correction equation and the total number of pressure iterations are also presented, as this then allows a direct comparison of the relative speeds of the six methods without having to consider the effect of the degree of vectorization of the remainder of the code. In scalar form, the ordering of the ADI, MZ, Zebra and RB is the same as for Poisson's equation; however, it is noted that the variation in performance is greater, particularly between the ADI and RB. In vector form, the MZ is again the most efficient method (but now by a factor of more than two from the RB, the next best method) when comparing the time spent in the pressure solver. It should be noted that the remainder of the Navier–Stokes code is written in two-dimensional array form and thus its degree of vectorization on the STAR is less than optimal. Rewriting the remainder of the code into one-dimensional form would provide a further improvement in the vectorized codes, in which, due to the efficiency of the Poisson solver, the overall performance is being degraded by the performance of the remainder of the code.

Timing results for the Fujitsu VP2200 are shown in Table IV. The proportionally better scalar floating point speed of the MZ means that it is the most efficient code in both scalar and vector form on the VP2200. However, the better vector performance of the Zebra, as compared to that on the STAR, means that in both vector and scalar forms the Zebra is only a factor of 20% less efficient than the MZ. The LRLX, the next best method on the FUJITSU in vector form, requires approximately twice the execution time of the MZ.

4. DISCUSSION

It is clear that the relative performance of the methods is machine-dependent. On the STAR those methods with predominantly effective vector lengths of order K (i.e. the Jacobi, RB and MZ codes) obtained significantly better vector speed-ups. On the FUJITSU the codes (i.e. the LRLX and Zebra) of order $\sqrt{(K)}$ and $\sqrt{(K)}/2$ also obtained significant speed-ups, of approximately the same degree as the codes of order K. On both machines the ADI method obtained very poor performance improvements under vectorization.

In addition to the machine dependence noted above, the relative performance of the methods is also problem-dependent. For the Poisson's problem the RB method performs almost as well as the MZ in vector form on both machines. However, for the Navier–Stokes problem the vector performance of the RB is a factor of two worse than the MZ. In general, it is seen that the non-Jacobi methods, the ADI, Zebra and MZ, all have a proportionally better performance with the Navier–Stokes problem than with the Poisson's problem when compared to the Jacobi-type methods. This is due to the poor smoothing properties of the Jacobi methods. As is well known, explicit methods are very efficient at smoothing the high wave number error component but less efficient at smoothing the low wave number error component, which has led to the development of the multigrid scheme.[9] Schemes such as the ADI, the Zebra and the MZ, because they obtain an exact solution via the Thomas algorithm on each of the lines, apply a much stronger smoothing to the low wave number component of the error. Of the two problems considered, it is likely that the Navier–Stokes code, due to the repeated cycling through the momentum and pressure equations, has a greater input of energy into the low wave number error component, and

the superior low wave number smoothing of the non-Jacobi methods is the reason for their proportionally better performance with the Navier–Stokes code.

It should be noted that the good performance of the RB algorithm presented here is restricted to grids with an odd number of points in the i direction. To implement a red/black scheme on an even mesh it is necessary to recalculate the stride at the end of each row or column, reducing the vectorization to the order of the length of each row or column and thus considerably reducing the vector performance. Generally, most problems may readily be defined on an odd mesh, and so this is not a major limitation of the present RB method. However, multigrid schemes are sometimes written in a manner that requires an even mesh, effectively precluding the use of the RB scheme as defined here.

Despite the considerable problem and machine-dependent variation noted above, for both problems and both machines the MZ is the most efficient algorithm in vector form. This is most significant on the STAR, where the MZ obtains a more than 3 times better vector performance than the Zebra, while on the FUJITSU although an improvement in performance is obtained it is by a smaller factor. The good vector performance of the MZ on the STAR, when compared to the standard Zebra, may be attributed to four factors.

(1) Shifting most floating point operations to fully explicit loops of order K. The Zebra has all its floating point operations, apart from the residual calculation which is the same for all methods, in loops of order $\sqrt{(K)}/2$, leading to poor performance on a machine such as the STAR, that requires long vectors for optimal performance.
(2) Reducing strided memory addressing. In the MZ only four out of 49 floating point operations have strided addressing, whereas in the Zebra all non-residual floating point operations have strided addressing.
(3) Caching arrays to ensure that in any vector/vector to vector operations ($x(i)=y(i)*z(i)$) only one read/store is made to memory while the other two read/stores are made to cache. As three read/stores can be made to cache in a single clock, while only one can be made to memory through the cache, this considerably improves the performance for the codes considered in this paper that would otherwise be severely restricted by the available memory bandwidth.
(4) Replacing divides with multiplies. As divides take three times as long as multiplies it is always good practice to replace them with multiplies. However, an improvement in performance will only be obtained if the code is not otherwise limited. Thus, replacing divides with multiplies for the Zebra leads to no improvement in performance as the vector length and strided memory addressing otherwise limit the performance.

It should also be noted that the array dimensions have been chosen to avoid memory bank conflicts for the STAR, which has 16-way interleaved memory. If such a conflict occurs it will degrade the performance of both the Zebra and MZ methods by a significant amount.

The difference in performance for the MZ and Zebra on the FUJITSU is relatively small when compared to that on the STAR. This is due to the architecture of the FUJITSU, which is considerably different from the STAR. The FUJITSU has a significantly better short vector performance as noted in the previous section. It can also handle strided memory access without reduction in performance. Thus, the main problems encountered with the Zebra on the STAR are not encountered to the same extent on the FUJITSU. The FUJITSU can simultaneously perform two adds, and two multiplies every clock and a divide every three clocks. However, for most of the time it is performing between one and two floating point operations per clock due to the limitation of only having two load/store pipes to memory. As the FUJITSU has a relatively small vector register, arrays cannot be held in cache to reduce the memory bandwidth limitation, as on

the STAR. Finally, the FUJITSU does not suffer a degradation in performance on divides provided there are enough adds and multiplies in the loop to allow continuous operation of the add/multiply pipes, as is the case in loops 3 and 5 in the Zebra method. Thus, although the MZ is seen to perform well on the FUJITSU, the dramatic improvement in performance observed on the STAR is not obtained.

To determine the effect the compiler may be having on the performance, estimates have been obtained for the theoretical peak performances of the vectorized codes based on the architectural features of each of the machines. All of the vectorized codes achieve a performance of within 20% of their theoretical peak, indicating that it is the codes in combination with the architecture of the machines that has been tested rather than the vectorizing capability of the two compilers.

5. CONCLUSIONS

A number of iterative vectorized Poisson's solvers for use with the Navier–Stokes equations have been tested. Timing results have been obtained for the solution of a Poisson's equation and, with the solvers embedded in a Navier–Stokes code, for the solution for natural convection in a cavity subjected impulsively to a horizontal temperature gradient. Results have been obtained in both scalar and vector forms on a STAR 910 VP, a relatively inexpensive SPARC-based vector architecture machine with a peak floating point speed of 132 mflops, and a FUJITSU VP2200, a high-end vector architecture supercomputer with a peak floating point speed of 1·25 Gflops.

On both the machines and for both the problems, the vectorized modified Zebra method described in this paper is the most efficient of the algorithms considered. Its superior efficiency is particularly evident for the natural convection problem on the STAR, where the Jacobi-type methods performed badly due to their poor low wave number smoothing. The modified Zebra method has been specifically optimized for the STAR by maximizing the effective vector lengths and reducing to a minimum the amount of non-contiguous memory accessing. As a result, on the STAR the modified Zebra runs in vectorized form at 3·4 times the floating point speed of the standard Zebra and it is anticipated that a good vector performance would be obtained on any machine with similar characteristics. The modifications will ensure a high floating point speed on any vector machine; however, on a machine such as the FUJITSU, with proportionally better short vector and non-contiguous memory access performance, the relative improvement in performance of the modified method is less significant.

The good performance of the modified Zebra code in both scalar and vector form on both the architectures considered is particularly important in the light of current day program development practice for scientific and engineering problems, whereby a code is frequently developed on scalar workstations and low-end vector computers and must run efficiently on scalar and low- and high-end vector machines.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the provision of time on the FUJITSU VP2200 of the Australian National University and the STAR 910 VP of the Centre for Water Research at the University of Western Australia.

REFERENCES

1. J. B. Nixon, 'From mini-computer to super-computer: optimization of a three-dimensional tidal model', *Computational Techniques and Applications CTAC-91*, Comp. Maths. Group, Div. of Applied Maths, Aust. Math. Society, Adelaide South Australia, 1992, pp. 355–364.

2. C. S. Ierotheou, C. W. Richards and M. Cross, 'Vectorization of the SIMPLE solution procedure for CFD problems–Part 1: a basic assessment', *Appl. Math. Modelling*, **13**, 524–529 (1989).
3. M. Cross, C. W. Richards, B. Knight and N. C. Markatos, 'Engineering CFD software in the next decade', in G. de Vahl Davis and C. Fletcher (eds), *Computational Fluid Dynamics*, North-Holland, Amsterdam, 1988, pp. 31–42.
4. C. Fletcher, 'Approximate factorization explicit methods', in L. Hogarth and J. Noye (eds), *Computational Techniques and Applications CTAC-89*, Hemisphere, Washington, DC, 1990, pp. 607–613.
5. J. Lambiotte and L. Howser, 'Vectorization on the STAR computer of several numerical methods for a fluid flow problem', *NASA TN D-7545*, NASA Langley Research Centre, Hampton, VA, 1974.
6. J. M. Ortego and R. G. Voigt, 'Solution of partial differential equations on vector and parallel computers', *SIAM Rev.*, **27**, 149–240 (1985).
7. S. W. Armfield, 'Finite difference solutions of the Navier–Stokes equations on staggered and non-staggered grids', *Comput. Fluids*, **20**, 1–17 (1991).
8. J. C. Patterson and S. W. Armfield, 'Transient features of natural convection in a cavity', *J. Fluid Mech.*, **219**, 469–497 (1990).
9. A. Brandt, 'Multigrid solutions to steady state compressible Navier–Stokes equations', in V. Glowinski, J. Lions and C. Shin (eds), *Computing Methods in Applied Science and Engineering*, North-Holland, Amsterdam, 1982.