Physics of the Earth and Planetary Interiors, 59 (1990) 195–207
Elsevier Science Publishers B.V., Amsterdam – Printed in The Netherlands

ConMan: vectorizing a finite element code for incompressible two-dimensional convection in the Earth's mantle

Scott D. King, Arthur Raefsky and Bradford H. Hager *

252-21 Seismological Laboratory, California Institute of Technology, Pasadena, CA 91125 (U.S.A.)

(Received May 23, 1989; accepted August 2, 1989)

King, S.D., Raefsky, A. and Hager, B.H., 1990. ConMan: vectorizing a finite element code for incompressible two-dimensional convection in the Earth's mantle. Phys. Earth Planet. Inter., 59: 195–207.

We discuss some simple concepts for vectorizing scientific codes, then apply these concepts to ConMan, a finite element code for simulations of mantle convection. We demonstrate that large speed-ups, close to the theoretical limit of the machine, are possible for entire codes, not just specially constructed routines. Although our specific code uses the finite element method, the vectorizing concepts discussed are widely applicable.

1 Introduction

Many large computational projects in geophysics are now being run on vector supercomputers such as the Cray X-MP, yet after more than a decade since the introduction of the Cray-1, most geophysicists simply compile their original codes, making use of the fast clock, without taking full advantage of the vector hardware or achieving anywhere near supercomputer speed.

To illustrate, let us consider the Cray X-MP with a 9.5 ns clock. It takes six clock cycles to compute a floating point addition (7 clock cycles for a floating point multiplication). This leads to a theoretical peak scalar rate of 9.5 MFLOPS (million floating point operations per second). This is about 25 times faster than a Sun 3/260 workstation (Dongarra, 1987). The theoretical maximum for vector code on the Cray X-MP is 210 MFLOPS, over 20 times faster than the scalar code and 500 times faster than the Sun. In reality, the theoretical speeds are never reached; however, speeds of 50–200 MFLOPS are attainable for many codes (Dongarra and Eisenstat, 1984).

It is often mistakenly believed that for a general code, special tricks are needed to obtain vector performance. However, although vectorizing compilers are becoming more sophisticated, a code that does not have data structures suitable for vector operations will not perform well on a vector computer. We show that ConMan (Convection, Mantle), a finite element code for two-dimensional, incompressible, thermal convection, which uses the simple concepts we present and no special tricks, runs up to 65 MFLOPS for the entire code on a Cray X-MP (including i/o and subroutine overhead).

Understanding vectorization is becoming even more important because of the recent introduction of high-performance pipelined workstations. The pipeline architecture is similar to a vector register, and many of the same concepts from vector programming apply to obtaining the maximum performance from a pipelined computer.

In the next section we discuss the basic concepts of vectorization, including the concepts of chaining and unrolling. Although we illustrate vectorization with the finite element code Con-

* Present address: Department of Earth, Atmospheric and Planetary Sciences, Massachusetts Institute of Technology, Cambridge, MA 02139, U.S.A.

0031-9201/90/$03.50

Man, the concepts we present are general. Because we use a general formulation of the finite element method, it would be especially easy to generalize ConMan to solve other equations or other geometries. We then review the equations for incompressible thermal convection and briefly describe the techniques used to solve them using the finite element method. We present benchmarks and timings for several computers and we include an example subroutine from ConMan in the Appendix for illustration.

2. Vectorization

Consider the simple addition

$$c_i = a_i + b_i \quad \text{for } i = 1, 2, \ldots, N_{\text{comp}} \tag{1}$$

on a generic scalar architecture. A new result c_i is available after N_{add}, the total number of cycles needed for one addition (on the Cray X-MP this is six cycles). On a vector computer, after the summands a_1 and b_1 move to the second step in the addition unit, the next summands a_2 and b_2 can enter the addition unit. The first result c_1 is available after N_{add} cycles. Then, unlike a scalar operation, after $(N_{\text{add}} + 1)$ cycles c_2 is available, and so on. After the start-up cost, a new result is available every cycle. On the scalar machine, the second result would not be available until after $2N_{\text{add}}$ cycles. For long vectors (i.e, N_{comp} large), the vector time is asymptotically reduced by $1/N_{\text{add}}$ relative to scalar processing.

In addition to increasing speed by vectorization, a vector computer can chain operations. The classic example is the linear algebra operation SAXPY (Single precision A times X Plus Y)

$$c_i = a \cdot x_1 + y_1 \quad \text{for } i = 1, 2, \ldots, N_{\text{comp}} \tag{2}$$

Chaining allows the result from $a \cdot x_1$ to be added to y_1 while $a \cdot x_2$ is being computed, and so on. As the Cray X-MP has eight vector registers, up to seven vectors could appear on the right-hand side and be chained (one register is needed for the result vector c_i). In this way, an addition and a multiplication can be computed each clock cycle, leading to the 210 MFLOP theoretical rate.

In FORTRAN (or C), vectorization is implemented in the innermost do loop. Within the innermost do loop there must be no subroutine calls or i/o statements because these inhibit vectorization of the loop (as compilers become more sophisticated this statement will no longer be true). Because it is the innermost loops which are vectorized, these should be in general the longest loops (the loop whose index runs over the largest range) and they should not have any dependent statements, where a newly computed result is used in the right-hand side of the same assignment during a future pass of the loop. A simple example is

```
DO 10 I = 1, N
    A(I) = A(I) + A(I - 1)
10 CONTINUE
```

This causes a problem because the result A(I) is dependent on the previous value A(I − 1). In real codes this problem is often hidden in a statement which uses indirect referencing, as in the following finite element example. This loop assembles the element contribution for the IELth element into the global equations, LM(IEL, 1 − 4), for the four local nodes.

```
DO 10 IEL = 1, NEL
    A(LM(IEL,  1)) = A(LM(IEL,  1)) +
      EL_LOCAL(IEL, 1)
    A(LM(IEL,  2)) = A(LM(IEL,  2)) +
      EL_LOCAL(IEL, 2)
    A(LM(IEL,  3)) = A(LM(IEL,  3)) +
      EL_LOCAL(IEL, 3)
    A(LM(IEL,  4)) = A(LM(IEL,  4)) +
      EL_LOCAL(IEL, 4)
10 CONTINUE
```

where LM(IEL, 1) is an integer array of indices with IEL the element number and LM(IEL, 1) the global equation number of the first node of the IELth element. Here, the compiler cannot assume that LM(IEL, 1) is not equal to LM(IEL − J, 1) for some arbitrary J.

Many of the operations in a finite element code (or finite difference code) have this kind of structure and the LM arrays usually have the unfortunate property that LM(IEL, 1) is not a unique value (see Fig. 1 for an example). This is not a

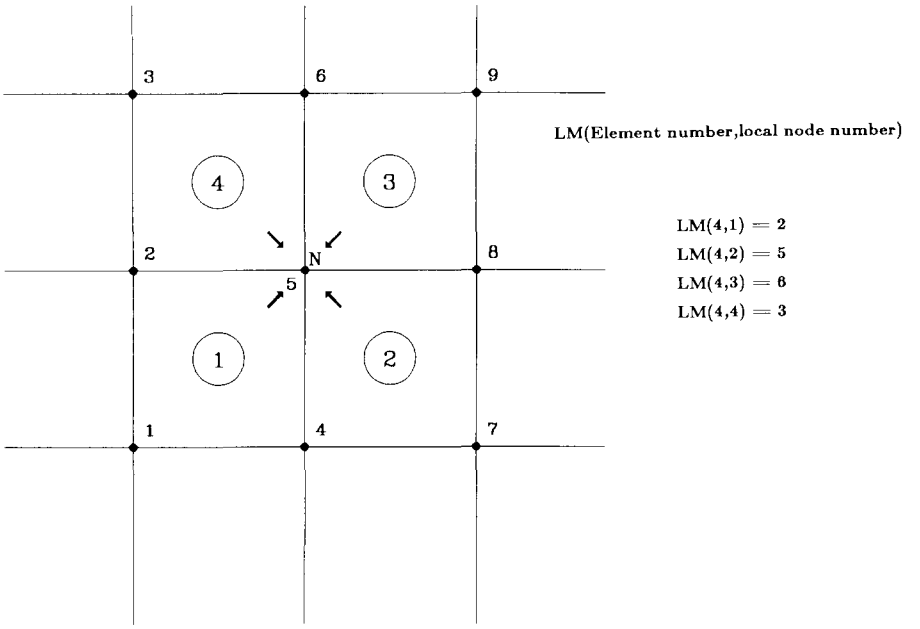

Fig. 1. The four bilinear quadrilateral elements (1–4) all contribute to the global nodal equation at node N, which they all share. The element numbers are circled and the global node numbers are not. The LM array for element 4 is listed at the right.

serious problem because we can rearrange the elements (by shuffling the LM array) such that there is a small number of groups of elements that do not share global nodes (Fig. 2). We can then loop over the total number of groups and over the elements in each group, which are independent and will safely vectorize.

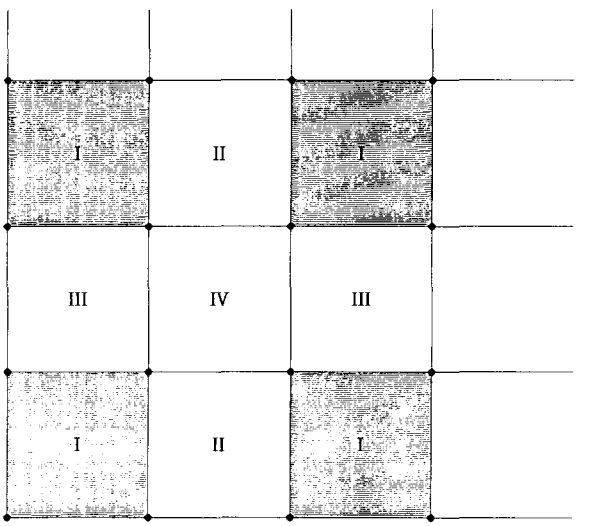

Fig. 2. The 'four-color' ordering scheme used in ConMan. It should be noted that the shaded elements (group I) do not share nodes with any other group I element. This group can be operated on safely with a vector operation.

The final point illustrated here is that as we want our innermost loop to be the loop over the elements, we do not want small loops over the local node numbers (i.e., I = 1, 4). We unroll the innermost loop by writing out the expression four times, explicitly putting in the value of the local node in each line (see Appendix for an example).

3. Equations and implementation

We now turn to the specific example program, ConMan, and discuss in brief the equations and the method used to implement their solutions on a vector computer. We give only a brief description of the method and refer to other work for convergence proofs, stability proofs and detailed analyses.

The equations for incompressible convection (in dimensionless form) are the equations of momentum

$$\nabla^2 u = -\nabla p + \mathrm{Ra}\ \theta \hat{k} \tag{3}$$

continuity

$$\nabla \cdot u = 0 \tag{4}$$

and energy

$$\frac{\partial \theta}{\partial t} = u \cdot \nabla \theta + \nabla^2 \theta \tag{5}$$

where u is the dimensionless velocity, θ is the dimensionless temperature, p is the dimensionless pressure, $\hat{k}$ is the unit vector in the vertical direction and t is the dimensionless time. In this form all the material properties are combined into one dimensionless parameter, the Rayleigh number, given by

$$\mathrm{Ra} = \frac{g\alpha \Delta T d^3}{\kappa \mu} \tag{6}$$

where g is the acceleration due to gravity, α is the coefficient of thermal expansion, ΔT is the temperature drop across the box, d is the depth of the box, κ is the thermal diffusivity, and μ is the dynamic viscosity.

The momentum and energy equations form a simple coupled system of differential equations. We treat the incompressibility equation as a constraint on the momentum equation and enforce incompressibility in the solution of the momentum equation using a penalty formulation. As the temperatures provide the buoyancy (body force) to drive the momentum equation and as there is no time dependence in the momentum equation, the algorithm to solve the system is a simple one: given an initial temperature field, calculate the resulting velocity field. Use the velocities to advect the temperatures for the next time step and solve for a new temperature field. If the time stepping for the temperature equation is stable, then this method is stable and converges as $\Delta t \to 0$.

3.1. Momentum equation

The momentum equation is solved using the penalty method to enforce incompressibility. The formal statement of the problem is as follows:
Given:

f: $\Omega \to R^n$ body force vector

g: $\Gamma_g \to R^n$ imposed velocity vector

h: $\Gamma_h \to R^n$ imposed traction vector

$\overline{\Gamma_g \cup \Gamma_h} = \Gamma$

$\Gamma_g \cap \Gamma_h = \Phi$

where Γ is the boundary of the domain Ω. Γ_g and Γ_h are the parts of the boundary where velocities and tractions are specified. Φ is the empty set.
Find u: $\Omega \to R^n$ and p: $\Omega \to R$

$$t_{ij,j} + f_i = 0 \quad \text{on } \Omega \tag{7}$$

$$u_{i,i} = 0 \quad \text{on } \Omega \tag{8}$$

$$u_i = g_i \quad \text{on } \Gamma_g \tag{9}$$

$$t_{ij} n_j = h_i \quad \text{on } \Gamma_h \tag{10}$$

with the constitutive equation for a Newtonian fluid

$$t_{ij} = -p\delta_{ij} + 2\mu u_{(i,j)} \tag{11}$$

where t_{ij} denotes the Cauchy stress tensor, p is the pressure, δ_{ij} is the Kronecker delta and $u_{(i,j)} = (u_{i,j} + u_{j,i})/2$.
In the penalty formulation, (11) is replaced by

$$t_{ij}^{(\lambda)} = -p^{(\lambda)}\delta_{ij} + 2\mu u_{(i,j)}^{(\lambda)} \tag{12}$$

where

$$p^{(\lambda)} = -\lambda u_{i,i}^{(\lambda)} \tag{13}$$

and λ is the penalty parameter (repeated subscripts means summation over all indices).

This formulation automatically enforces incompressibility (8) as the solution to (7), (12) and (13) converges to the incompressible Stokes equation as λ approaches infinity (Temam, 1977). Also, the unknown pressure field is eliminated. This is useful not only because the amount of computational work is decreased because no pressure equation is solved, but also because it eliminates the need to create artificial boundary conditions for the pressure equation. There are no pressure boundary conditions in the formal specification of the problem. By examining the equation we see that the role of pressure is to balance the system, so physically the penalty formulation makes sense.

The equation is cast in the weak form and the Galerkin formulation (i.e, the weighting functions are the same as the basis functions) is used to solve the weak form of the equation.
Let

$$V = \left\{ w \in H^1 \mid w = 0 \text{ on } \Gamma_g \right\} \tag{14}$$

where V is the set of all weighting functions w which vanish on the boundary. Similarly V^h is a subset of V parameterized by h, the mesh parameter. Let g^h denote an approximation of g which

$$
\begin{array}{cccccccc}
\mathrm{i} = & 1 & 2 & 3 & 4 & & \mathrm{j} = \\
\left[\mathrm{K}\right]^{\mathrm{e}} = & \begin{array}{|cc|cc|cc|cc|}
1 & 2 & 4 & 7 & 11 & 16 & 22 & 29 \\
 & 3 & 5 & 8 & 12 & 17 & 23 & 30 \\
\hline
 & & 6 & 9 & 13 & 18 & 24 & 31 \\
 & & & 10 & 14 & 19 & 25 & 32 \\
\hline
 & & & & 15 & 20 & 26 & 33 \\
 & & & & & 21 & 27 & 34 \\
\hline
 & & & & & & 28 & 35 \\
 & & & & & & & 36
\end{array} & & & & & \begin{array}{c} 1 \\ \\ 2 \\ \\ 3 \\ \\ 4 \end{array}
\end{array}
$$

$$
\left[\mathrm{K}\right]^{\mathrm{e}} = \left[\mathrm{K}\right]_{\nu}^{\mathrm{e}} + \left[\mathrm{K}\right]_{\lambda}^{\mathrm{e}}
$$

$$
\left[\mathrm{K}\right]_{\nu} = \nu \begin{bmatrix} 2\mathrm{N_x(i)N_x(j)} + \mathrm{N_y(i)N_y(j)} & \mathrm{N_x(j)N_y(i)} \\ \mathrm{N_x(i)N_y(j)} & \mathrm{N_x(i)N_x(j)} + 2\mathrm{N_y(i)N_y(j)} \end{bmatrix}
$$

$$
\left[\mathrm{K}\right]_{\lambda} = \lambda \begin{bmatrix} \mathrm{N_x(i)N_x(j)} & \mathrm{N_x(i)N_y(j)} \\ \mathrm{N_x(j)N_y(i)} & \mathrm{N_y(i)N_y(j)} \end{bmatrix}
$$

Fig. 3. (a) The location and numbering of the 8×8 element stiffness matrix for the velocity equation used in ConMan. It should be noted that the matrix is made up of 16 2×2 matrices with i, j indices as shown. The i and j refer to the local node numbering of the element and can be thought of as the effect of node i as felt at node j. (b) The 8×8 element stiffness matrix is made up of two terms: a viscosity contribution $[K]_{\nu}^{e}$, and the penalty contribution $[K]_{\lambda}^{e}$. (c) The 2×2 submatrix for the viscous contribution to the element stiffness matrix. $N_x(i)$ is the x derivative of the shape function evaluated at node i (i, j here refer to the location of the 2×2 matrix in the 8×8 element stiffness matrix). (d) The 2×2 submatrix for the penalty contribution to the element stiffness matrix (i, j here refer to the location of the 2×2 matrix in the 8×8 element stiffness matrix).

converges to g as $h \to 0$. Find $u^h = w^h + g^h$, $w^h \in V^h$, such that for all $\bar{w}^h \in V^h$

$$
\begin{aligned}
&\int_{\Omega} \left(\lambda w^h_{j,j} \bar{w}^h_{i,i} + 2\mu w^h_{i,j} \bar{w}^h_{i,j}\right) \mathrm{d}\Omega \\
&\quad = \int_{\Omega} f_i \bar{w}^h_i \, \mathrm{d}\Omega + \int_{\Gamma_h} h_i \bar{w}^h_i \, \mathrm{d}\Omega \\
&\qquad - \int_{\Omega} \left(\lambda g^h_{j,j} \bar{w}^h_{i,i} + 2\mu g^h_{i,j} \bar{w}^h_{i,j}\right) \mathrm{d}\Omega
\end{aligned} \tag{15}
$$

$$
w^{h_1} = \Sigma N_A u_{iA} \tag{16}
$$

$$
u_{iA} = u_i(x_A) \tag{17}
$$

where N_A is the shape function for node A for the element.

The element stiffness matrix (Fig. 3) is made up of the two terms from the left-hand side of the integral equation. The integration is done using 2 × 2 Gauss quadrature, which is exact when the elements are rectangular and bilinear shape functions are used. The λ term is under-integrated (one point rule) to keep the large penalty value from effectively locking the element (Malkus and Hughes, 1978). The right-hand side is made up of three known parts, the body force term (f_i), the applied tractions (h_i) and the applied velocities (g_i). The momentum equation is equivalent to an incompressible elastic problem, and the resulting stiffness matrix will always be positive definite (Hughes, 1987, pp. 84–89). This allows us to assemble only the upper triangular part of the stiffness matrix and save both storage and operations using Cholesky factorization. More details of the method and a formal error analysis were given by Hughes et al. (1979).

3.2. *Energy equation*

The energy equation is an advection–diffusion equation. The formal statement is
Find T: $\Omega \to R$ such that

$$
\dot{T} + u_i T_{,i} = \kappa T_{,ii} + H \quad \text{on } \Omega \tag{18}
$$

$$
T = b \quad \text{on } \Gamma_b \tag{19}
$$

$$
T_{,j} n_j = q \quad \text{on } \Gamma_q \tag{20}
$$

where T is the temperature, u_i is the velocity, κ is the thermal diffusivity and H is the internal heat source. The weak form of (18) is given by

$$
\begin{aligned}
\int_{\Omega} (w + p)\dot{T} \, \mathrm{d}\Omega = &- \int_{\Omega} (w + p)(u_i T_{,i}) \, \mathrm{d}\Omega \\
&- \kappa \int_{\Omega} w_{,i} T_{,i} \, \mathrm{d}\Omega + \int_{\Gamma_q} w T_{,j} n_j \, \mathrm{d}\Gamma_q
\end{aligned} \tag{21}
$$

where $\dot{T}$ is the time derivative of temperature, $T_{,i}$ is the gradient of temperature, w is the standard weighting function and $(w + p)$ is the

Petrov–Galerkin weighting function with, p, the discontinuous streamline upwind part of the Petrov–Galerkin weighting function given by

$$p = \tau u \cdot \nabla T = \tilde{k} u_i \frac{w_{,i}}{\| u \|^2} \tag{22}$$

The energy equation is solved using Petrov–Galerkin weighting functions on the internal heat source and advective terms to correct for the under-diffusion and remove the oscillations which would result from the standard Galerkin method for an advection-dominated problem (Hughes and Brooks, 1979). The Petrov–Galerkin function can be thought of as a standard Galerkin method in which we counterbalance the numerical underdiffusion by adding an artificial diffusivity of the form

$$(\xi u_\xi h_\xi + \eta u_\eta h_\eta)/2 \tag{23}$$

with

$$\xi = 1 - 2\frac{\kappa}{u_\xi h_\xi} \tag{24}$$

$$\eta = 1 - 2\frac{\kappa}{u_\eta h_\eta} \tag{25}$$

where h_ξ and h_η are the element lengths and u_ξ and u_η are the velocities in the local element coordinate system ($\xi\eta$ system) evaluated at the element center. This form of discretization has no crosswind diffusion because (23) acts only in the direction of the flow (i.e., it follows the streamline), hence the name Streamline Upwind Petrov–Galerkin (SUPG). This makes it a better approximation than straight upwinding, and it has been demonstrated to be more accurate than Galerkin or straight upwinding in advection-dominated problems (Hughes and Brooks, 1979; Brooks, 1981). It has recently been shown that the SUPG method is one of a broader class of methods for advection–diffusion equations referred to as Galerkin/least-squares methods (Hughes et al., 1988).

The resulting matrix equation is not symmetric, but as the energy equation only has one degree of freedom per node, whereas the momentum equation has two or three, the storage for the energy equation is small compared with that for the momentum equation. As we use an explicit time-stepping method, the energy equation is not implemented in matrix form. The added cost of calculating the Petrov–Galerkin weighting functions is much less than the cost of using a refined grid with the Galerkin method. The Galerkin method requires a finer grid than the Petrov–Galerkin method to achieve stable solutions (Travis et al., in preparation).

3.2.1. Time stepping

Time stepping in the energy equation is done using an explicit predictor–corrector algorithm. The form of the predictor–corrector algorithm is

Predict:

$$T_{n+1}^{(0)} = T_n + \Delta t(1-\alpha)\dot{T}_n \tag{26}$$

$$\dot{T}_{n+1}^{(0)} = 0 \tag{27}$$

Solve:

$$M^* \Delta \dot{T}_{n+1}^{(i)} = R_{n+1}^{(i)} \tag{28}$$

$$R_{n+1}^{(i)} = -\left[\dot{T}_{n+1}^{(i)} + u \cdot \left(T_{n+1}^{(i)}\right)_{,x}\right](w+p) - \tilde{k} w_{,x} \left(T_{n+1}^{(i)}\right)_{,x} + (\text{boundary condition terms}) \tag{29}$$

Correct:

$$T_{n+1}^{(i+1)} = T_{n+1}^{(i)} + \Delta t \alpha \dot{T}_{n+1}^{(i)} \tag{30}$$

$$\dot{T}_{n+1}^{(i+1)} = \dot{T}_{n+1}^{(i)} + \Delta \dot{T}_{n+1}^{(i)} \tag{31}$$

where i is the iteration number (for the corrector), n is the time-step number, T is the temperature, $\dot{T}$ is the derivative of temperature with time, $\Delta\dot{T}$ is the correction to the temperature derivative for the iteration, M^* is the lumped mass matrix, $R_{n+1}^{(i)}$ is the residual term, Δt is the time step and α is a convergence parameter. It should be noted that in the explicit formulation M^* is diagonal.

The time step is dynamically chosen, and corresponds to the Courant time step (the largest step that can be taken explicitly and maintain stability). With the appropriate choice of variables, $\alpha = 0.5$ and two iterations, the method is second-order accurate (Hughes, 1987, pp. 562–566).

4. Numerical benchmarks

Two examples are given, based on benchmarks for two-dimensional Cartesian convection codes

Table 1
The parameters used in the two benchmarks

Parameters	Benchmark	
	CVBM	TDBM
Rayleigh number	77927	100000
Time steps	5000	1000
Tempdep viscosity	no	yes
Velocity b.c.	fs	fs
Temp b.c.—top	$T=0$	$T=0$
Temp b.c.—bottom	$T=1$	$T=1$

Tempdep viscosity means the global matrix had to be factored at every step. fs means free slip boundary condition applied at the boundaries (flow allowed along the side walls as well as top and bottom). TDBM was not run to steady state, but for a fixed number of time steps for timing purposes

Table 2
Execution speeds normalized to the Cray X-MP scalar execution speed for the two benchmark problems (CVBM and TDBM) on various computers

Computer	Benchmark	
	CVBM	TDBM
Cray Y-MP V	31.18	26.07
Cray X-MP V	18.73	15.29
Cray 2 V	15.92	10.62
Convex C210 V	4.27	3.10
Convex C120 V	1.40	1.42
Cray X-MP S	1.00	1.00
Sun 4-330	0.42	0.79
Convex C120 S	0.31	0.42

Where available, both scalar (S) and vector (V) speeds are shown. The speed-up (the ratio of scalar to vector times) is greater on the Crays than on the Convex because the slower memory access on the Convex limits vector performance

given by Travis et al. (in preparation). The parameters for these benchmarks are given in Table 1. There are two purposes for these benchmarks: (1) to verify the code against standard existing codes; and (2) to demonstrate the speed of the vector code on representative problems. Benchmark CVBM has a constant viscosity, and the velocity stiffness matrix is factored only once, whereas TDBM (not from Travis et al.) has temperature-dependent viscosity and the velocity stiffness matrix is factored at every time step. This allows us to observe the difference between the speed-up in the finite element forms and assemblies and the matrix factorization and back-substitution. Vectorizing sparse matrix solvers is an area of active research (e.g., Ashcraft et al., 1987; Lucas, 1988), and our implementation has used a stan-

Table 3
Execution times (in seconds) for individual routines for benchmarks CVBM and TDBM in scalar and vector mode on various computers

Subroutine	Convex C120		Cray X-MP		Cray Y-MP		Cray 2	
	Scalar	Vector	Scalar	Vector	Scalar	Vector	Scalar	Vector
Times (s) for subroutines from CVBM								
Factor	10.4	3.6	4.8	0.3	–	0.2	–	0.5
Backsolve	3132.2	830.2	1341.5	80.0	–	52.8	–	100.1
f_vStf	1.6	0.4	0.6	0.1	–	0.0	–	0.1
f_vRes	531.0	92.9	154.4	6.6	–	3.8	–	9.5
f_tRes	5272.3	948.7	1212.4	50.9	–	28.2	–	53.8
Total	9383.2	2021.2	2869.3	153.2	–	92.0	–	180.2
Times (s) for subroutines from TDBM								
Factor	10883.4	3564.9	4870.0	297.1	–	199.4	–	507.7
Backsolve	646.9	167.3	270.2	16.4	–	10.6	–	20.0
f_vStf	1690.0	417.1	649.5	71.3	–	15.9	–	30.0
f_vRes	112.9	19.5	30.6	1.4	–	0.8	–	1.8
f_tRes	1088.5	188.0	242.4	9.8	–	5.5	–	10.6
Total	14513.8	4383.2	6094.7	398.6	–	233.7	–	573.4

The assembling routines (f_vStf, f_vRes, f_tRes) show better vector performance than do the factor and back-substitution routines. The performance of the factorization and back-substitution improves as the bandwidth of the matrix approaches the vector register length

dard and poorly vectorized solver. Therefore we present the times in individual units of the program as well as the total times. Results are given for a Convex-C120 running Unix compiled with fc4.1, a Convex-C210 running Unix compiled with fc5.0, a Cray X-MP 4/8 running CTSS, compiled with CFT77, a Cray 2 (one processor) running Unicos and a Cray Y-MP (one processor), also running Unicos. (We list compilers and operating systems as we found a factor of up to 1.5 difference in speed between code compiled with different compilers on the same machine!) We present execution speed relative to the Cray X-MP scalar speed (Table 2), as that is comparable with the vector speed of the original code upon which ConMan was based. It should be noted that a well-vectorized code runs better on mini-supercomputers such as the Convex C120 and C210 than a scalar code on a Cray. Hardware monitoring for CVBM gives an overall rate of 65 MFLOPS on the X-MP (95 MFLOPS on the faster Y-MP). TDBM, with an overall rate of 45 MFLOPS on the Y-MP, is slower because more time is spent factoring, a routine which is not well vectorized. A breakdown of the execution time spent in each of the major routines is given in Table 3.

We note that an old version of the SUPG (Streamline Upwind Petrov–Galerkin) code on which ConMan is based ran slower in vector mode than ConMan does in scalar. This indicates that even though the compiler indicated it was vectorizing many loops in the old code, either the time-consuming calculations were not being vectorized or the loops were too short to give any appreciable speed-up. We also note that ConMan runs faster on a Sun 4-330, which is a scalar machine, than the old SUPG code (a common finding of people who vectorize and optimize codes).

From the benchmark timings it is clear that the element assembly routines (f_vStf, f_vRes, f_tRes) have been improved significantly more than the factorization and back-substitution routines. We note that faster fully vectorized matrix solvers are being developed especially for finite element programs (e.g., Lucas, 1988). We designed ConMan so that it can be easily adapted to a new matrix solver when one becomes available.

The routine times do not add up to the total times because there are unlisted routines (for grid generation and i/o) that take a small amount of time. The MFLOP rates and relative speeds, however, do include time spent in these routines. We note that the speed-up, the ratio of scalar to vector speed, on the Convex is not as good as on the Crays. This is because in double precision the Convex memory access time, especially in the indirect referencing, is slowing the vector calculations. The Convex is simply not able to deliver numbers fast enough to the vector registers. As the Cray X-MP and Y-MP have three paths to memory, as well as a faster memory access time, they are not limited by memory access. Also, because of the Cray's larger word size we can use single

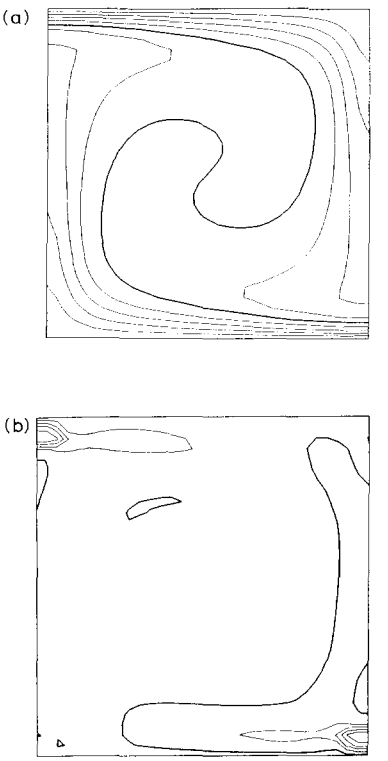

Fig. 4. (a) Final temperature field for CVBM. The contour interval is 0.1 dimensionless temperature units. The heavy contour is the 0.5 contour line. (b) Difference of final temperature field above and the high-resolution 129×129 grid from Travis et al. (in preparation). The contour interval is 0.002 dimensionless temperature units. The heavy contour is the 0.0 contour line.

precision on the Cray with the same accuracy as double precision on the Convex.

Plots of the final temperature field and the difference from a finite-difference calculation on a 129×129 grid from Travis et al. (in preparation) are shown in Fig. 4. The solution for benchmark CVBM agrees reasonably well with the Travis et al. benchmark. The error from this method is second order $O((\Delta x)^2)$, where Δx is the grid spacing. The error associated with the high resolution grid is more than an order of magnitude less than the 33×33 grid used for CVBM. Taking the 129×129 result as the exact solution, the difference plot (Fig. 4b) shows the error in the SUPG method. The SUPG method has localized the error in the corner regions where the flow is most advective, and compares exactly with the sequential SUPG method code used in the benchmark paper (see Travis et al. (in preparation) for more details of the error associated with the method).

5. Summary

In addition to the traditional modifications for vectorization (removing subroutine calls and i/o statements from the inner loops, etc.), we believe that data structure is an important consideration for codes used on vector computers. We have presented some simple ideas for improving code performance on vector computers, specifically in the improvement of data structures. These ideas work not only on small linear algebra routines but can also be extended to full-scale scientific programs and still retain order of magnitude speedups. We have implemented these ideas in a new finite-element code, ConMan, designed to study problems in mantle convection, which takes advantage of the speed available on current vector supercomputers. An unsupported version of this code is available from the authors.

Acknowledgements

Solutions for CVBM were provided by Brian Travis and were calculated as part of the IGPP Mantle Convection Workshop at Los Alamos National Laboratory. Helpful videotape lectures on vectorizing code and Cray architecture were obtained at the Los Alamos National Laboratory. This research was supported by NSF grant EAR-86-18744, and is Caltech contribution no. 4751. Cray X-MP calculations were performed at the San Diego Supercomputer Center. Cray 2 and Cray Y-MP calculations were performed at NASA Ames with the help of Eugene Miya. Convex C210 calculations were performed by Ron Gray at Convex Computer Corporation. Sun 330 calculations were performed by Keith Bierman at Sun Microsystems, Inc.

Appendix

```
      subroutine f_vRes(shl    , det     , tl     , lcblk   ,
     &                  t      , vrhs    , mat    , ra      ,
     &                  tq     , lmv     , lmt    )
c
      include 'common.h'
c
      dimension  shl(4,5)          , det(numel,5)   , tl(lvec,4)     ,
     &           lcblk(2, nEG)     , t(numnp)       , vrhs(nEGdf),
     &           mat(numel)        , ra(numat)      , tq(lvec,5)     ,
     &           lmv(numel,8)      , lmt(numel,4)
c
      common/templ/el_rhs(lvec,8),blkra(lvec)
c
```

```
c
c
c     This routine calculates the Right Hand Side velocity Residual
c
c     Input:
c                    lvec:          max length of element groups (64 for Cray)
c                    numnp:         total number of nodes
c                    numel:         total number of elements
c                    nEGdf:         number of degrees of freedom for velocity
c                                        equation (2*numnp)
c
c                    shl(4,5):      shape functions (for mapping element)
c                    det(numel,5):  determinant of element (for transformation)
c                    lcblk(2,nEG):  element group info
c                    t(numnp):      array of nodal temperatures
c                    mat(numel):    material number of element
c                    ra(numat):     rayleigh number of material groups
c                    lmv(numel,8)   array linking element and local node number
c                                        to global equation number for velocity
c                    lmt(numel,8)   array linking element and local node number
c                                        to global equation number for temperature
c
c     Temporary:
c                    tl(lvec,4):    temporary space for temperature array
c                    tq(lvec,4):    temporary array for temperatures at integration
c                                        points
c                    el_rhs(lvec,8) temporary space for element contribution
c
c     Output:
c                    vrhs(nEGdf):   right hand side of velocity equation
c
------------------------------------------------------------------------------
c
      include 'common.h'
c
c... loop over the element blocks
c
      do 1000 iblk = 1, nelblk
c
c... set up the parameters
c
c     iel: starting element number of the group
c     nenl: number of element nodes (4 for bilinear elements)
c     nvec: number of elements in iblk th group
c
      iel    = lcblk(1,iblk)
      nenl   = lcblk(2,iblk)
      nvec   = lcblk(1,iblk + 1) - iel
```

```
c
c...  localize (gather) the temperature for the whole element group
c
      do 150 iv = 1, nvec
       ivel = iv + iel - 1
       tl(iv,1) = t( lmt(ivel,1) )
       tl(iv,2) = t( lmt(ivel,2) )
       tl(iv,3) = t( lmt(ivel,3) )
       tl(iv,4) = t( lmt(ivel,4) )
150   continue
c
c... form the temperature at integration points
c     the temperature at each integration point (tq(iv,1 - 4)) is formed by
c     adding contributions from all four nodes (tl(iv,1 - 4))
c
      do 200 iv = 1, nvec
        tq(iv,1) = shl(1,1)*tl(iv,1) + shl(2,1)*tl(iv,2)
     &           + shl(3,1)*tl(iv,3) + shl(4,1)*tl(iv,4)
        tq(iv,2) = shl(1,2)*tl(iv,1) + shl(2,2)*tl(iv,2)
     &           + shl(3,2)*tl(iv,3) + shl(4,2)*tl(iv,4)
        tq(iv,3) = shl(1,3)*tl(iv,1) + shl(2,3)*tl(iv,2)
     &           + shl(3,3)*tl(iv,3) + shl(4,3)*tl(iv,4)
        tq(iv,4) = shl(1,4)*tl(iv,1) + shl(2,4)*tl(iv,2)
     &           + shl(3,4)*tl(iv,3) + shl(4,4)*tl(iv,4)
200   continue
c
c... load the value of the rayleigh number into temp array blkra
c     for each element
c
c$dir no_recurrence
      do 300 iv = 1 , nvec
        ivel = iv + iel - 1
        blkra(iv)  =  ra(mat(ivel))
300   continue
c
c... calculate the contribution to each local element right hand side
c     due to the buoyancy (i.e. t*Ra )
c
      do 400 iv = 1 , nvec
      ivel = iv + iel - 1
        el_rhs(iv,1) = zero
        el_rhs(iv,3) = zero
        el_rhs(iv,5) = zero
        el_rhs(iv,7) = zero
        el_rhs(iv,2) = blkra(iv)*(tq(iv,1)*det(ivel,1)*shl(1,1)
     &                          + tq(iv,2)*det(ivel,2)*shl(1,2)
     &                          + tq(iv,3)*det(ivel,3)*shl(1,3)
     &                          + tq(iv,4)*det(ivel,4)*shl(1,4))
```

```
c
      el_rhs(iv,4) = blkra(iv)*(tq(iv,1)*det(ivel,1)*shl(2,1)
     &                         + tq(iv,2)*det(ivel,2)*shl(2,2)
     &                         + tq(iv,3)*det(ivel,3)*shl(2,3)
     &                         + tq(iv,4)*det(ivel,4)*shl(2,4))
c
      el_rhs(iv,6) = blkra(iv)*(tq(iv,1)*det(ivel,1)*shl(3,1)
     &                         + tq(iv,2)*det(ivel,2)*shl(3,2)
     &                         + tq(iv,3)*det(ivel,3)*shl(3,3)
     &                         + tq(iv,4)*det(ivel,4)*shl(3,4))
c
      el_rhs(iv,8) = blkra(iv)*(tq(iv,1)*det(ivel,1)*shl(4,1)
     &                         + tq(iv,2)*det(ivel,2)*shl(4,2)
     &                         + tq(iv,3)*det(ivel,3)*shl(4,3)
     &                         + tq(iv,4)*det(ivel,4)*shl(4,4))
400   continue
c
c...assemble (scatter) the local element right hand sides into the
c    global right hand side
c
c$dir no_recurrence
      do 500 iv = 1 , nvec
        ivel = iv + iel - 1
        vrhs(lmv(ivel,1)) = vrhs(lmv(ivel,1)) + el_rhs(iv,1)
        vrhs(lmv(ivel,2)) = vrhs(lmv(ivel,2)) + el_rhs(iv,2)
        vrhs(lmv(ivel,3)) = vrhs(lmv(ivel,3)) + el_rhs(iv,3)
        vrhs(lmv(ivel,4)) = vrhs(lmv(ivel,4)) + el_rhs(iv,4)
        vrhs(lmv(ivel,5)) = vrhs(lmv(ivel,5)) + el_rhs(iv,5)
        vrhs(lmv(ivel,6)) = vrhs(lmv(ivel,6)) + el_rhs(iv,6)
        vrhs(lmv(ivel,7)) = vrhs(lmv(ivel,7)) + el_rhs(iv,7)
        vrhs(lmv(ivel,8)) = vrhs(lmv(ivel,8)) + el_rhs(iv,8)
500   continue
c
c...  end loop over element blocks
c
1000  continue
c
c...     return
c
      return
      end
```

References

Ashcraft, C.C., Grimes, R.G., Peyton, B.W. and Simon, H.D., 1987. Progress in sparse matrix methods for large linear systems on vector supercomputers. Int. J. Supercomput. Appl., 1: 10–30.

Brooks, A., 1981. A Petrov–Galerkin Finite-element Formulation for Convection Dominated Flows. Ph.D. Thesis, California Institute of Technology, Pasadena, CA.

Dongarra, J.J., 1987. Performance of various computers using standard linear equation software in a FORTRAN environment. Argonne National Laboratory Tech. Memo, 23.

Dongarra, J.J. and Eisenstat, S.C., 1984. Squeezing the most out of an algorithm in Cray FORTRAN. ACM Trans. Math. Software, 10: 219–230.

Hughes, T.J.R., 1987. The Finite Element Method. Prentice-Hall, Englewood Cliffs, NJ, 631 pp.

Hughes, T.J.R., and Brooks, A., 1979. A multi-dimensional upwind scheme with no crosswind diffusion. In: Finite Element Methods for Convection Dominated Flows. ASME, New York, Vol. 34 pp. 19–35.

Hughes, T.J.R., Liu, W.K. and Brooks, A., 1979. Finite element analysis of incompressible viscous flows by the penalty function formulation. J. Comput. Phys., 30: 19–35.

Hughes, T.J.R., Franca, L.P., Hulbert, G.M., Johan, Z. and Shakib, F., 1988. The Galerkin / least-squares method for advective-diffusion equations. In: T.E. Tezduyar (Editor), Recent Developments in Computational Fluid Dynamics. ASME, New York, Vol. 95, pp. 75–99.

Lucas, R.F., 1988. Solving Planar Systems of Equations on Distributed Memory Multi-processors. Ph.D. Thesis, Stanford University, Palo Alto, CA.

Malkus, D.S. and Hughes, T.J.R., 1978. Mixed finite element methods–reduced and selective integration techniques: a unification of concepts. Comput. Meth. Appl. Mech. Eng., 15: 63–81.

Temam, R., 1977. Navier–Stokes Equations: Theory and Numerical Analysis. North-Holland, Amsterdam, pp. 148–156.

Travis, B.J., Olson, P., Hager, B.H., Raefsky, A., O'Connell, R.J., Gable, C., Anderson, C., Schubert, G. and Baumgardner, J., in preparation. Comparison of codes for infinite Prandtl number convection: 2-D Cartesian cases. Los Alamos National Laboratory Tech. Rep.