SOME REMARKS ON COMPLETELY VECTORIZING POINT GAUSS–SEIDEL WHILE USING THE NATURAL ORDERING

CRAIG C. DOUGLAS*

Abstract. A common statement in papers in the vectorization field is to note that point SOR methods with the natural ordering cannot be vectorized. The usual approach is to re–order the unknowns using a red–black or diagonal ordering and vectorize that. In this paper, we construct a point Gauss–Seidel iteration which completely vectorizes and still uses the natural ordering. The work here also applies to both point SOR and single program, multiple data (SPMD) parallel computer architectures. When this approach is reasonable to use is also shown.

Key words. iterative methods, vectorization, SPMD.

1. Preliminaries. In this paper a completely vectorizable point Gauss–Seidel iteration using the *natural ordering* is constructed for solving the system of linear equations

$$Ax = b, \tag{1}$$

where A is symmetric and positive definite. Gauss–Seidel,

$$(D + L)x^{i+1} = b - Ux^i, \quad i = 1, 2, \cdots \text{ and } A = D + L + U \tag{2}$$

(D, the main diagonal of A, and L and U are the strictly lower and upper triangular parts of A), is assumed to converge for (1).

While this is presented for fairly general matrices A, the real problems of interest to the author are ones derived from discretizing partial differential equations. Hence, the examples used here will typically be the five and nine point operators (and will be referred to as A_5 and A_9, respectively). The idea in this paper generalizes to higher dimensional partial differential equations problems trivially.

Assume that A has the following block structure:

$$A = \left[\begin{array}{c|c|c} D_1 & U_1 & \\ \hline L_2 & D_2 & U_2 \\ \hline & \ddots & \\ \hline & L_p & D_p \end{array}\right], \qquad L_1 = 0,\ U_p = 0. \tag{3}$$

The blocks U_i and L_i can have any structure, including be dense. However, they are quite sparse for the five and nine point operators. The $n_i \times n_i$ blocks D_i, $1 \le i \le p$, are assumed to be tridiagonal:

$$D_i = [\ell_i, d_i, u_i].$$

* Mathematical Sciences Department, IBM Research Division, T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598-0218 and Department of Computer Science, Yale University, P. O. Box 208285, New Haven, CT 06520-8285. E-mail: *na.cdouglas@na-net.ornl.gov*.

Associated with either A_5 or A_9 is an $M \times N$ point lattice G_{MN}. For $h_x > 0$ and $h_y > 0$, define

$$(4)\quad G_{MN} = \{(x_i, y_j) :\ x_i = x_0 + ih_x,\ y_j = y_0 + jh_y,\ i = 1, \cdots, M,\ j = 1, \cdots, N\}.$$

By the natural ordering, the points are ordered in (4) as (x_1, y_1), (x_2, y_1), $\cdots$, (x_M, y_1), (x_1, y_2), $\cdots$, (x_M, y_N).

2. Standard procedures. Before introducing the vectorized iteration, a review of standard methods for vectorizing (partially or fully) Gauss–Seidel is worthwhile. A good reference is Hayes [1].

2.1. Red–black ordering. The usual way of vectorizing (2) when $A = A_5$ is to use a red–black ordering of the unknowns. This causes the approximate solution and right hand side vectors to pass through cache twice, even though only half of the elements are accessed each time. For long vectors (i.e., on the order of the length MN), this also requires that M in G_{MN} be odd, which is not always an option. Finally, the red–black ordering does not produce a vectorized iteration for A_9 nor for the more general A in (3).

The cache issue can be resolved by programming red–black by strips of the lattice that just fit into cache. However, this is a highly nonstandard manner for programming this example and is not always possible anyway.

2.2. Diagonal orderings. Another re-ordering class is based on diagonals. For example, the D1 ordering of the points in (4) is (x_1, y_1), (x_1, y_2), (x_2, y_1), (x_1, y_3), (x_2, y_2), (x_3, y_1), $\cdots$, (x_M, y_N). Once again, this produces a vectorizable Gauss–Seidel method for A_5, but not for A_9. Sometimes a red–black diagonal ordering (D2) is used, which produces a vectorizable method for A_9.

2.3. Mostly vectorized, some scalar, but the natural ordering. When the natural ordering is used, the common computational method is to do as much of the updates in vector mode as possible followed by a scalar mode update. If the vectors x and b are partitioned into pieces of length n_i, similar to (3), then the complete (k+1)-st (vector) update in block i, $i = 1, 2, \cdots, p$, is given by

$$(5)\qquad x_i^{(k+1)} = d_i^{-1}[b_i - L_i x^{(k+1)} - U_i x^{(k)} - u_i x^{(k)}] - d_i^{-1}\ell_i x^{(k+1)}.$$

Note that only the last term needs to be done in scalar mode since all of the earlier ones vectorize. If the matrix A is complicated (i.e., has a large number of nonzeros per row), the machine in question has fast scalar arithmetic units, or no cache, then this is a very good choice for implementing Gauss-Seidel with the natural ordering.

3. Completely vectorized.

3.1. The algorithm. The technique of this paper advocates solving (5) in two phases. First, the obviously vectorizable part is computed and stored in a vector register:

$$(6)\qquad \hat{x}_i^{(k+1)} = d_i^{-1}[b_i - L_i x^{(k+1)} - U_i x^{(k)} - u_i x^{(k)}].$$

Then the bidiagonal system of equations

$$S_i x_i^{(k+1)} = \hat{x}_i^{(k+1)}, \quad where\ S_i = [\ell_i, d_i, 0], \tag{7}$$

is solved in vector mode using a highly simplified form of cyclic reduction, which will be outlined. Lambiotte and Voigt [2] is followed, but more efficient variations exist for certain problems (see Sweet [3] and its references).

There are 3 steps to solving (7): factorization, forward substitution, and backward substitution. If extra storage is available, then it is worthwhile to save the factorization.

3.2. Factorization. First, assume that the matrix A has been scaled so that $a_{jj} = 1$, all j. Then the factorization step in any block i is just computing a sequence of new bidiagonal matrices using a divide (i.e., re–ordering) and conquer scheme. For simplicity, assume that for some block i, $n_i = 2^q$, some $q > 0$. Note that the algorithm will work for general n_i's with a minor change. Note that S_i is already factored into LU form, namely,

$$L = S_i = \begin{bmatrix} 1 & & & 0 \\ c_2 & 1 & & \\ & c_3 & 1 & \\ 0 & & \ddots & \ddots \end{bmatrix} \quad and \quad U = I.$$

Let

$$\begin{aligned} G &= c_{2j}, \quad && 1 \le j \le 2^{q-1}, \\ E &= c_{2j+1}, && 0 \le j \le 2^{q-1} - 1, \end{aligned} \tag{8}$$

then

$$\hat{C} = -GE$$

results in a $2^{q-1} \times 2^{q-1}$ bidiagonal matrix similar to S_i:

$$\begin{bmatrix} 1 & & & 0 \\ \hat{c}_2 & 1 & & \\ & \hat{c}_3 & 1 & \\ 0 & & \ddots & \ddots \end{bmatrix}.$$

This can be done in 2 vector compressions (length 2^q to 2^{q-1}), 1 vector multiply (length 2^{q-1}), and 1 vector sign change (length 2^{q-1}). On machines with stride > 1 capabilities and multiply–add chaining, this is only 1 vector compression (length 2^q to 2^{q-1}) and 1 vector multiply–add (length 2^{q-1}). While $\log_2 n_i$ steps of this can be done recursively, the starting costs of the vector operations will be a limiting factor (see §3.5). In the ensuing discussion, a superscript may be added to E and G to denote recursion.

3.3. Forward substitution. Forward substitution is defined recursively. At the k-th step,

$$\mathcal{L}_k Z_k = \mathcal{P}_k Z_{k-1}$$

is solved where

$$Z_k = \left[\frac{u}{v}\right] \quad and \quad \mathcal{P}_k Z_{k-1} = \mathcal{P}_k \left[\frac{w}{x}\right] = \left[\frac{w}{x'}\right],$$

where v, x, and x' are of length $\alpha = 2^{q-k+1}$ and u and w are of length $n_i - \alpha$. Further,

$$x' = (x_1, x_3, \cdots, x_{\alpha-1}, x_2, x_4, \cdots, x_\alpha)^T.$$

For cyclic reduction of a bidiagonal system, the forward substitution is just

$$\begin{cases} u = w, & \\ v_j = x'_j, & j = 1, \cdots, \alpha/2, \\ v_{j+\alpha/2} = x'_{j+\alpha/2} - G^{(k)} v_j, & j = 1, \cdots, \alpha/2, \end{cases}$$

where $G^{(k)}$ is derived from (8). This requires 2 vector compressions (length α to $\alpha/2$), 1 vector multiply (length $\alpha/2$), and 1 vector subtract (length $\alpha/2$). On machines with stride > 1 capabilities and multiply–add chaining, this is only 1 vector compression (length α to $\alpha/2$) and 1 vector multiply–add (length $\alpha/2$).

3.4. Backward substitution. Backward substitution is also defined recursively. A sequence of problems of the form

$$\mathcal{U}_k \mathcal{P}_k \cdots \mathcal{U}_1 \mathcal{P}_1 Z_0 = Z_k$$

must be solved. Let

$$Z'_{k-j} = \left[\frac{u'}{v'}\right] = \mathcal{P}_{k-j+1} \left[\frac{u}{v}\right] \quad and \quad Z_{k-j+1} = \left[\frac{w}{x}\right],$$

where v, v', and x are of length $\alpha = 2^{q-k+j}$ and u, u', and w are of length $n_i - \alpha$. Similar to the forward substitution case, this is calculated by

$$\begin{cases} u' = w, & \\ v'_j = x_j, & j = \alpha/2 + 1, \cdots, \alpha, \\ v'_{j+\alpha/2} = x_j - E^{(k-j)} v_{\alpha/2+j-1}, & j = 1, \cdots, \alpha/2, \end{cases} \tag{9}$$

where $E^{(k-j)}$ is derived from (8). The permutation, $Z_{k-j} = \mathcal{P}^T_{k-j+1} Z'_{k-j}$, is computed by

$$\begin{cases} u = u', & \\ (v_1, \cdots, v_\alpha)^T = (v'_1, v'_{\alpha/2+1}, v'_2, v'_{\alpha/2+2}, \cdots, v'_\alpha)^T. & \end{cases} \tag{10}$$

The latter is a perfect shuffle merging of the upper and lower halves of v'. This requires 1 vector merge (length $\alpha/2$ to α), 1 vector copy (length $\alpha/2$), 1 vector multiply (length $\alpha/2$), and 1 vector subtract (length $\alpha/2$). On machines with stride > 1 capabilities and multiply–add chaining, (10) can be done for free by storing information computed in (9) appropriately. Hence, this can be done in only 1 vector copy (length $\alpha/2$) and 1 vector multiply–add (length $\alpha/2$).

3.5. Costs. Associated with a vector computer are costs (machine cycles or some unit of time) involving both a startup time and time associated with getting results out of a pipeline. Typically, for an operation on a vector of length n, the cost is

$$\mathcal{S}^{(V)} + \mathcal{T}^{(V)}(n).$$

Even for a single scalar operation, the cost may well be modeled by

$$\mathcal{S}^{(S)} + \mathcal{T}^{(S)}(1).$$

The basic operations that are used in this paper are the following:

Operation	Symbols	Description
Compression	$\mathcal{S}_{CM}, \mathcal{T}_{CM}$	Copy every other element of a vector into a new vector.
Merge	$\mathcal{S}_{MG}, \mathcal{T}_{MG}$	Interleave the components of 2 stride 1 vectors to produce a new vector
Copy	$\mathcal{S}_{CP}, \mathcal{T}_{CP}$	Copy a vector to a new location.
Add	$\mathcal{S}_{A}, \mathcal{T}_{A}$	Do an element–wise vector addition. This is assumed to be equivalent to subtraction.
Sign change	$\mathcal{S}_{SC}, \mathcal{T}_{SC}$	Do an element–wise vector sign change.
Multiply	$\mathcal{S}_{M}, \mathcal{T}_{M}$	Do an element–wise vector multiplication.
Multiply–add	$\mathcal{S}_{MA}, \mathcal{T}_{MA}$	Do an element–wise vector multiplication and addition to another vector, chained in hardware.

As noted earlier, a superscript will be added to denote scalar or vector costs. These costs are assumed to be for a single processor or a parallel processor with a shared memory. However, in a single program, multiple data environment, e.g., a distributed memory parallel processor, some of these costs could be of a slightly different form, taking into account data movement times between processors.

Assume that k levels of recursion occur in a block of the cyclic reduction process. Then the cost of factorization is $\mathcal{T}_{Fac}^{(V)}$:

$$\sum_{j=1}^{k} \left[2\mathcal{S}_{CM}^{(V)} + \mathcal{S}_{M}^{(V)} + \mathcal{S}_{SC}^{(V)} + 2\mathcal{T}_{CM}^{(V)}(2^{q-j}) + \mathcal{T}_{M}^{(V)}(2^{q-j}) + \mathcal{T}_{SC}^{(V)}(2^{q-j})\right]$$

or

$$\sum_{j=1}^{k} \left[\mathcal{S}_{CM}^{(V)} + \mathcal{S}_{MA}^{(V)} + \mathcal{T}_{CM}^{(V)}(2^{q-j}) + \mathcal{T}_{MA}^{(V)}(2^{q-j})\right]$$

depending on the hardware (no assumptions versus stride > 1 and multiply–add chaining capability). The combined cost of the forward and backward substitutions is $\mathcal{T}_{Subst}^{(V)}$:

$$\sum_{j=1}^{k} \left[2\mathcal{S}_{CM}^{(V)} + \mathcal{S}_{MG}^{(V)} + \mathcal{S}_{CP}^{(V)} + 2\mathcal{S}_{M}^{(V)} + 2\mathcal{S}_{A}^{(V)} + 2\mathcal{T}_{CM}^{(V)}(2^{q-j}) + \mathcal{T}_{MG}^{(V)}(2^{q-j}) \right.$$

$$\left. + \mathcal{T}_{CP}^{(V)}(2^{q-j}) + 2\mathcal{T}_{M}^{(V)}(2^{q-j}) + 2\mathcal{T}_{A}^{(V)}(2^{q-j})\right] \tag{11}$$

or

$$(12) \qquad \sum_{j=1}^{k} \left[\mathcal{S}_{CM}^{(V)} + \mathcal{S}_{CP}^{(V)} + 2\mathcal{S}_{MA}^{(V)} + \mathcal{T}_{CM}^{(V)}(2^{q-j}) + \mathcal{T}_{CP}^{(V)}(2^{q-j}) + 2\mathcal{T}_{MA}^{(V)}(2^{q-j}) \right]$$

depending on the hardware. Assume that the smallest vector length in the cyclic reduction is greater than 1, say of length N'. Then there will be a term of the form

$$N' \left[\mathcal{S}_{M}^{(S)} + \mathcal{S}_{A}^{(S)} + \mathcal{T}_{M}^{(S)}(1) + \mathcal{T}_{A}^{(S)}(1) \right]$$

added to (11) and (12).

Let $\mathcal{T}_{Fac}^{(V)}$ and $\mathcal{T}_{Subst}^{(V)}$ be the total cost of computing the factorization and forward and backward substitutions. For r iterations of Gauss-Seidel, the algorithm described in this paper is only useful when (the obvious, but trivial)

$$(13) \qquad \begin{aligned} \mathcal{T}_{Subst}^{(V)} + r^{-1}\mathcal{T}_{Fac}^{(V)} &< \mathcal{T}_{MA}^{(S)}(N) - \mathcal{T}_{MA}^{(S)}(N') \\ &= (N - N') \left(\mathcal{S}_{M}^{(S)} + \mathcal{S}_{A}^{(S)} + \mathcal{T}_{M}^{(S)} + \mathcal{T}_{A}^{(S)} \right) \end{aligned}$$

Hence, for this algorithm to be practical, the vector unit must be much faster than the scalar one. In this case, N' will be quite small in comparison to N. A quick look at vector supercomputers of today shows that this requirement is currently with us (at least for a while).

3.6. Example. This algorithm was tested on an IBM 3090J processor with a vector unit. The problem was based on Poisson's equation on a square with a uniform lattice for G_{MN}. This machine has vector registers of length 256. A routine was coded in vector assembly language in order to avoid a problem with Fortran discussed in §3.7.

The technique proposed here became cost effective when $M > 40$. With long enough vectors, the (almost) completely vectorized method was 50% faster than the scalar version for both A_5 and A_9 in the part of the code that was scalar only before. Since this was 25% and 12.5% of the total work per iteration, the savings were actually only 12.5% and 6.25%, respectively.

A 3090 has quite fast scalar floating point units in comparison to many vector machines. On a machine where the vector units are significantly faster than the scalar unit, a higher savings will be achieved.

3.7. Drawbacks. A feature worth noting is that current compilers (and vector preprocessors) for Fortran and C do not globally optimize well enough to catch that in calculating (7), the right hand side $\hat{x}_i$ was just computed in (6). Hence, it can be left in a vector register and not moved to main memory. Further, in Fortran, the language standard seems to require this completely wasteful data movement. Unfortunately, moving $\hat{x}_i$ to and then immediately from main memory destroys most, if not all, of the usefulness of this algorithm. Hence, assembly language seems to be required for now, which is utterly repugnant.

There is one exception to this worth noting, namely, certain SIMD (single instruction, multiple data) parallel processors that can be considered large vector processors. The startup time is essentially zero for vector operations, so that much shorter vectors can be handled than on traditional vector supercomputers.

A final drawback is that the scalar update can be done in $O(M)$ cost while the left hand side of (13) can conceivably be $O(M \log_2 M)$. Hence, the vector lengths may be limited to a closed interval where the vectorized method is better than the scalar method.

4. SOR and concluding remarks. The techniques of this paper are immediately applicable to point SOR–like methods with the natural ordering. Whether or not this is worth using is dependent on the hardware, the size of the problem in (1), how much time the scalar part of the iteration actually takes, and on some machines whether or not anyone is willing to program in assembly language (unless the compiler writing community improves their technology to accommodate the type of coding style required here).

Finally, this method works conveniently (at least in theory) for nine point and more general operators, which is important when solving partial differential equation problems using finite element procedures on traditional vector supercomputers.

REFERENCES

[1] L. Hayes, *Comparative analysis of iterative techniques for solving LaPlace's equation on the unit square on a parallel processor*, master's thesis, University of Texas, Austin, TX, 1974.

[2] J. J. Lambiotte and R. G. Voigt, *The solution of tridiagonal linear systems on the CDC STAR–100 computer*, ACM Trans. Math. Soft., 1 (1975), pp. 308–329.

[3] R. A. Sweet, *A parallel and vector variant of the cyclic reduction algorithm*, SIAM J. Sci. Stat. Comp., 9 (1988), pp. 761–765.