NUMERICAL ANALYSIS PROJECT
MANUSCRIPT NA-92-w 20

DECEMBER 1992

Use of Linear Algebra Kernels to Build An Efficient Finite Element Solver

by

H.C. Elman
D. K-Y Lee

NUMERICAL ANALYSIS PROJECT
COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305

USE OF LINEAR ALGEBRA KERNELS TO BUILD AN EFFICIENT FINITE ELEMENT SOLVER

HOWARD C. ELMAN • AND DENNIS K.-Y. LEE †

Abstract. For scientific codes to achieve good performance on computers with hierarchical memories, it is necessary that the ratio of memory references to arithmetic operations be low. In this paper, we show that Level 3 BLAS linear algebra kernels can be used to satisfy this requirement to produce an efficient implementation of a parallel finite element solver on a shared memory parallel computer with a fast cache memory.

Key words. Finite element, hpversion, cache memory, Level 3 BLAS, parallel.

1. Introduction.

The finite element method divides domains into elemental subdomains, and it is therefore a natural candidate for parallel numerical solution of elliptic partial differential equations. In this paper, we describe the results of an experimental study of a parallel finite element solver, using the hpversion of the finite element method for the discretization. This technique can achieve accuracy by either refining a mesh or increasing the order of the polynomial basis functions [4]. Our computational algorithm is to distribute elements among available processors; construct local stiffness matrices; eliminate by Gaussian elimination (or static condensation) certain unknowns associated with basis functions whose supports are entirely contained within individual elements; and solve for unknowns associated with element interfaces by a preconditioned conjugate gradient method. The tests were done on an Alliant FX/8 computer, which contains eight vector processors and a shared memory, including a fast cache memory.

These results build on [3], where it was observed that the steps of local matrix construction and static condensation dominate the overall cost of the computation. The implementation of [3] used software from the Level 2 Basic Linear Algebra Software (BLAS) library [6], in which the basic computation is the matrix-vector product. These kernels have the property that the ratio of memory references to arithmetic operations is of order one. Although in principle the local matrix construction and static condensation are fully parallelizable, if memory references take more time than arithmetic, as on the Alliant when data is not in the cache, then data movement dominates the cost. Indeed, it was observed in [3] that the local computations displayed good parallel efficiency provided all matrices that are treated in parallel fit into the cache, but performance degraded otherwise.

We show here how to restructure both the construction of local stiffness matrices and the static condensation to eliminate any degradation of performance due to memory hierarchies. This is done using matrix-matrix oriented computations of the type available in the Level 3 BLAS library [5]. With these kernels, for matrices of

• Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, e-mail: elman@cs.umd.edu. The work of this author was supported in part by the U. S. Army Research Office under grant DAAL-0392-G-0016 and the National Science Foundation under grant ASC-8958544, aud by the National Science Foundation under grant CCR-8821078 at Stanford University. Computer time was provided by the Advanced Computing Research Facility at Argonne National Laboratory.

† National Computer Board and Defence Science Organization, Science Park Drive, Singapore 0511, Republic of Singapore, email: dlee@iti.gov.sg. The work of this author was supported in part by the U. S. Army Research Office under grant DAAL-0389-K-0016 while the author was at the Department of Computer Science, University of Maryland, College Park, MD 20742.

order n, the ratio of memory references to arithmetic operations is of order $1/n$. Consequently, the costs of data movement become insignificant, and the resulting finite element computations display essentially full parallelism regardless of the size of the local matrices. In numerical experiments, the static condensation step reaches the performance (in terms of megaflop rates) achievable for this class of computation. The matrix construction step fails short of this performance, mainly because of limits on vector lengths.

An outline of the paper is as follows. In $2, we give a brief overview of the problem addressed and the mathematical methods used to solve it. In $3, we show how the static condensation is performed using Level 3 BLAS tools and describe its performance, and in §4, we present the algorithm and results for local matrix construction. Finally, in $5, we briefly discuss the results and compare the performance of these two local computations.

2. Overview of computations. We are interested in the numerical solution of the two-dimensional elliptic equation

$$(1) \qquad -[(a\,u_x)_x + (b\,u_y)_y] = f \quad \text{on } \Omega \subset \mathbf{R}^2,$$

where Ω is a-rectangular domain and a, b, f and $u|_{\partial\Omega}$ are such that the solution to (1) is uniquely defined. The discrete problem is to find the weak solution to (1) in a finite dimensional **subspace** of H'(R) [8]. We use the Q(p) variant of the hpversion of the finite element method; see [1, 4] for theoretical properties of the discretization and a comprehensive set of references, and [2, 3] for additional details concerning the computations. For our purposes here, the following points are relevant;

1. The domain Ω is partitioned into a collection of rectangular elements $\{\Omega_i\}$ such that $\Omega = \cup\bar{\Omega}_i$. We will assume that the number of elements divides the number of processors on our parallel computer.

2. On each element Ω_i, the basis functions of the finite element discretization have the form $\Phi_{jk}(x, y) = \phi_j(x)\phi_k(y)$ **where** ϕ_j is **a** polynomial of degree j, I $\leq$ j $\leq$ p. **If** $j = k =$ 1, then Φ_{jk} is the bilinear function used in the usual piecewise bilinear discretization [8]; on any element there are four such functions, which we refer to as ***nodal*** basis functions. If j $=$ 1 and 2 $\leq k \leq p$, or if k = 1 and 2 $\leq$ j $\leq p$, then Φ_{jk} is **nonzero** on one side of Ω_i and zero on the other three sides; there are $4(p-1)$ such **side** basis functions. If 2 $\leq j$, k $\leq$ p, then the support of Φ_{jk} is entirely contained in $\bar{\Omega}_i$; we refer to these as internal basis functions, of which there are $(p-1)^2$. The local stiffness matrix S_i associated with Ω_i is of order $(p+1)^2$ with entries

$$(2) \quad B(\Phi_{jk},\Phi_{lm}) = \int\int a\,\phi_j'(x)\phi_k(y)\phi_l'(x)\phi_m(y) + b\,\phi_j(x)\phi_k'(y)\phi_l(x)\phi_m'(y)\,dxdy,$$

where $\Phi_{jk}(x,y) = \phi_j(x)\phi_k(y)$ and $\Phi_{lm}(x, y) = \phi_l(x)\phi_m(y)$ range over all the basis functions defined on Ω_i. Except in special cases, such **as** where a and b are constant functions, (2) must be evaluated using a quadrature rule.

3. The discrete solution to (1) is obtained by solving a system of linear equations $S\alpha$ = s, where the global matrix S has the form S $= \sum S_i$ **and** S_i is the local stiffness matrix associated with Ω_i.[1] S_i can be identified with the Gramm matrix determined

[1] **The right hand side s is defined in a similar manner, and the computations in which it is involved are similar to those outlined below for matrices; see [2, 3] for details.**

by (2), where Φ_{jk} and Φ_{lm} range over all basis functions. Under this identification, S_i can be ordered to have the form

$$S_i = \begin{pmatrix} A_i & B_i^T \\ B_i & C_i \end{pmatrix}, \tag{3}$$

where A_i corresponds to interactions among internal basis functions, C_i corresponds to interactions among side and nodal basis functions, and B_i corresponds to interactions between internal basis functions and side and nodal basis functions. The internal functions have local support, so that the unknowns associated with them can be decoupled from the system. This entails computing the Schur complement

$$\tilde{C}_i = C_i - B_i A_i^{-1} B_i^T. \tag{4}$$

4. Once the "internal unknowns" are decoupled from the system as above, the result is a system of linear equations $\hat{S}\hat{\alpha} = \hat{s}$ for the unknowns associated with the boundaries of $\{\Omega_i\}$. These unknowns can be solved efficiently using a preconditioned conjugate gradient method where the preconditioner comes from the portion of S associated with nodal unknowns [2, 3]. The cost of this step is lower than the cost of the local matrix computations [3], [9], and we omit a discussion of it here.

Details of the implementation are as follows. The tests were performed on an Alliant FX/8 with a cache memory of size 512 kbytes. All computations were done in double precision. Although the local stiffness matrices are symmetric, to enable use of the Level 3 BLAS library, they were stored in square arrays, in full nonsymmetric form. The Level 3 BLAS kernels used were those provided by the manufacturer. Parallelism was achieved using compiler constructs, so that an outer loop over the the number of elements distributed independent computations over separate processors. To determine parallel efficiency, runs on one processor were performed using the runtime command "execute -c 1".

3. Static condensation of local stiffness matrices. We first consider the implementation of the static condensation (4), which entails a straightforward generalization of block Cholesky factorization described e.g. in [5]. Let M denote a symmetric positive-definite matrix represented in block form as

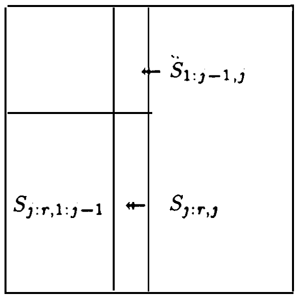
$$M = \begin{pmatrix} M_{11} & M_{12} & \cdots & M_{1r} \\ M_{21} & M_{22} & \cdots & M_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ M_{r1} & M_{r2} & \cdots & M_{rr} \end{pmatrix}, \tag{5}$$

where M_{ij} is itself a matrix and M_{ii} is square for each i. A block version of the Cholesky factorization of M is as follows.

(6)
```
for j = 1 to $r$ do
    $M_{j:r,j} \leftarrow M_{j:r,j} - M_{j:r,1:j-1} M_{1:j-1,j}$
    Compute Cholesky factorization $M_{jj} = L_{jj} L_{jj}^T$
    $M_{j+1:r,j} \leftarrow M_{j+1:r,j} L_{jj}^{-T}$
enddo
```

Here, we use the "Matlab–like" notation $M_{j:k,l:m}$ to refer to the entries of A4 in block rows j through k and block columns 1 through m. If only one block row or column,

FIG. 1. Submatrices of S participating in *the* dominant computation *of block condensation.*



say j, is referenced, then the subscript j : j is replaced simply by j. During the course of the computation (6), the contents of the block lower triangle of the array M are overwritten with the Cholesky factor of M. In an implementation, the entries of $M_{1:j-1,j}$ (which lie above the block diagonal) would be obtained as the transpose of $M_{j,1:j-1}$.

Now, consider a modification of (6) for computing the Schur complement (4). To simplify notation, we omit the subscript i from the matrices of (3) and (4). Assume that S is blocked into r blocks as in (5), where A occupies the first r_0 x r_0 subblock. The following computation overwrites the part of S containing the block lower triangle of A with its Cholesky factor L, places BL^{-T} into the part containing B, and overwrites the block lower triangle of C with the lower part of the Schur complement.

```
     for j = 1 to r do
         $j_{max} = \min(j - 1, r_0)$
         $S_{j:r,j} \leftarrow S_{j:r,j} - S_{j:r,1:j_{max}} S_{1:j_{max},j}$
         if $j \leq r_0$ then
(7)          Compute Cholesky factorization $S_{jj} = L_{jj} L_{jj}^T$
             $S_{j+1:r,j} \leftarrow S_{j+1:r,j} L_{jj}^{-T}$
         endif
     enddo
```

If the blocks are of order 1 (i.e., they are just the entries of S), then the quantity $S_{1:j_{max},j}$ appearing in the third line of (7) is a vector, and the dominant part of the computation is a matrix-vector product; this was implemented using Level 2 BLAS software in [3]. For other block sizes, this operation is a matrix-matrix product, and we implemented here it using the Level 3 BLAS routine DGEMM. The submatrices in the case j $\leq r_0$ are shown in Fig. 1. In addition, the block triangular solve $M_{j+1:r,j} L_{jj}^{-T}$ was implemented using the Level 3 BLAS routine DTRSM. The Cholesky factorization of S_{jj} was computed using the algorithm (6) with block size 1, i.e., in “BLAS2” style.

We have restricted our attention to the case where all blocks of S are square, except possibly those of the last block row and column. Figure 2 shows how the CPU times for the static condensation varies with the blocksizes, for several values of the basis function degree, p. The data corresponds to an 8 x 8 element grid for $p = 6$ and $p = 8$, a 4 x 4 grid for $p = 8$, and a 2 x 4 grid for $p = 16$. These grid sizes have no effect on the overall trend of the data, and they were chosen so that the four curves fit into the graph. The results indicate that a block size of 7 (highlighted by a vertical line) is optimal. Table 1 shows the megaflop (Mflop) rates and parallel efficiency of

FIG. 2. *CPU times vs. blocksize for static condensation.*

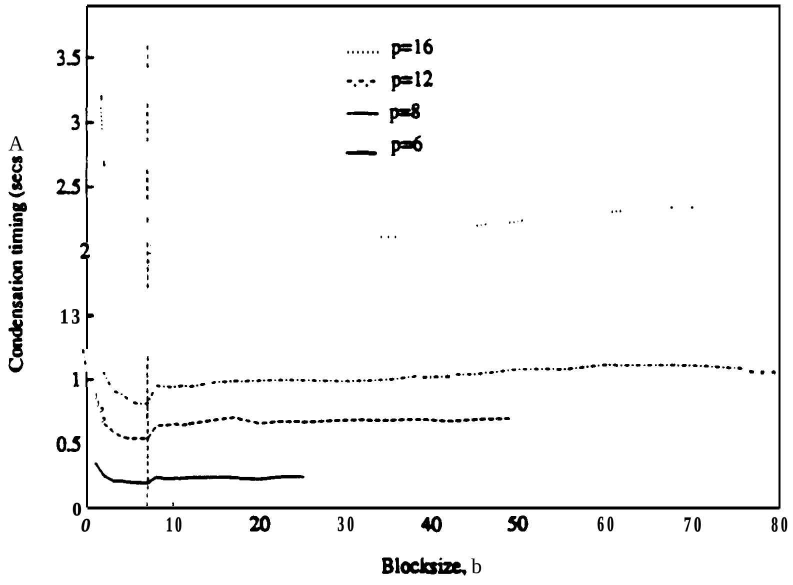


TABLE 1
Timings *and performance* of static *condensation, for a 2 × 4* grid, *on 8 processors.*

p	Time (secs.)	Operations per matrix	Rate (Mflops)	Efficiency (%)
4	.0088	4,700	4.3	86
6	.0244	37,600	12.3	91
8	.0670	173,700	20.7	95
10	.1816	585,300	25.8	93
12	.4117	1,602,800	31.1	92
14	.8992	3,791,900	33.7	93
16	1.826	8,045,800	35.2	94
18	3.483	15,692,700	36.0	93
20	6.263	28.614.400	36.6	93

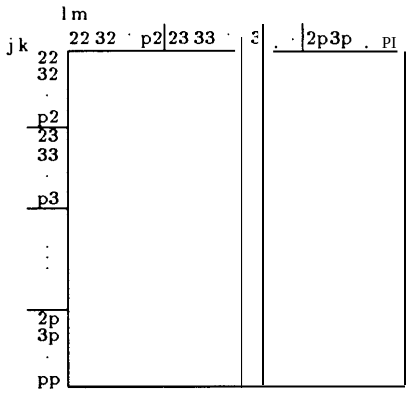
the static condensation on eight processors, for values of p between 4 and 20, and block size 7. Here, efficiency is defined to be

$$\frac{1}{8}\,\frac{\text{CPU time on 1 processor}}{\text{CPU time on 8 processors}} \times 100.$$

We see that there is no degradation in efficiency as p increases, although the 8 local matrices treated in parallel fit into cache only for $p \leq 8$. The performance of 37 Mflops for p = 20 (where the matrix is of order 441) is virtually identical to empirical limits observed for LU decomposition of comparably sized problems on the Alliant, **see** [7, p. 97].

4. **Construction of local stiffuess matrices.** We now consider the use of matrix-oriented routines to construct the local stiffness matrices (3), where again we omit the subscript i. It is convenient to represent the local stiffness matrix S as a quadruply-indexed array $S[(j, k), (l, m)]$, where j, k, I, and m are essentially as in

FIG. 3. Indexing scheme *for A based on quadruples,* with column corresponding *to* index $(l,m) = (p, 3)$ **highlighted.**



(2).[2] To simplify the exposition, we will concentrate our discussion on the portion of S corresponding to internal unknowns, i.e., A of (3), denoted here as $A[(j, k), (l, m)]$ where $2 \leq j, k, l, m \leq p$. *See* Fig. 3.

Using the tensor product form of the basis functions, we rewrite (2) as

$$(8) \int \phi_k(y)\phi_m(y) \left(\int a\,\phi_j'(x)\phi_l'(x)dx\right) dy + \int \phi_k'(y)\phi_m'(y) \left(\int b\,\phi_j(x)\phi_l(x)dx\right) dy.$$

This suggests the following quadrature strategy. If a and b are constant, then Gauss quadrature with $p + 1$ quadrature points in each direction will produce the exact integral for all values of $j, k, l, m \leq p$. This is more points than is needed when any of j through m are less than p, but on the other hand, for more general a and b, exact integration may not be possible. As a compromise between these two possibilities, we fix our quadrature rule to use $p + 1$ Gauss points. The approximation to (8) is then given by

$$(9) \quad \begin{aligned} A[(j,k),(l,m)] &= \textstyle\sum_{r=0}^{p} w_r\phi_k(\eta_r)\phi_m(\eta_r) \left(\sum_{q=0}^{p} a(\xi_q,\eta_r)w_q\phi_j'(\xi_q)\phi_l'(\xi_q)\right) \\ &+ \textstyle\sum_{r=0}^{p} w_r\phi_k'(\eta_r)\phi_m'(\eta_r) \left(\sum_{q=0}^{p} b(\xi_q,\eta_r)w_q\phi_j(\xi_q)\phi_l(\xi_q)\right), \end{aligned}$$

where $\{\xi_q\}_{q=0}^{p}$ and $\{\eta_r\}_{r=0}^{p}$ are the Gauss points in the horizontal and vertical directions, respectively, and $\{w_q\}_{q=0}^{p}$ are the weights. Suppose the inner sums for all j, l and r are precomputed, at a cost of $O(p^4)$ operations. The computation (9) then requires $O(p)$ operations. Since there are $O(p^4)$ entries in A, the total cost is $O(p^5)$ operations, a savings of $O(p)$ over the naive implementation. See [10] for related ideas. Note that this is asymptotically less costly than static condensation, which requires $O(p^6)$ operations.

To structure these computations in terms of matrix-matrix products, let us first

[2] We say "essentially" because, for example, there are two side functions that are linear in x, one that is identically zero on the left side of Ω and one that is zero on the right side of Ω. In the notation of (2), these would both have the form $\Phi_{1k}(x,y) = \phi_1(x)\phi_k(y)$. To define four-tuples for the local matrix, we could assign one of these the index (0, k) and the other the index (1, k), with other side functions and nodal functions handled in an analogous manner.

define the quantities

$$I_0(\eta_r,k,m) = w_r\phi_k(\eta_r)\phi_m(\eta_r), \qquad I_1(\eta_r,k,m) = w_r\phi'_k(\eta_r)\phi'_m(\eta_r),$$
$$\sigma(\eta_r,j,l) = \textstyle\sum_{q=0}^{p} a(\xi_q,\eta_r)I_1(\xi_q,j,l), \quad \tau(\eta_r,j,l) = \sum_{q=0}^{p} b(\xi_q,\eta_r)I_0(\xi_q,j,l).$$

If these have been precomputed, then the local stiffness matrices can be constructed using the following algorithm:

```
for $m = 2$ to $p$ do
   for $l = 2$ to $p$ do
      for $k = 2$ to $p$ do
         for $j = 2$ to $p$ do
(10)        $A[(j,k),(l,m)] = \sum_{r=0}^{p} \sigma(\eta_r,j,l)I_0(\eta_r,k,m) + \tau(\eta_r,j,l)I_1(\eta_r,k,m)$
         enddo
      enddo
   enddo
enddo
```

The loops over indices j and k construct the entries $A[*,(l,m)]$, the column of A corresponding to the (I, m) index pair; see Fig. 3. This data can also be viewed as an unravelled version of an ordinary square matrix of order p, with row and column indices $2 \le j, k \le p-1$. Since the inner sum is a pair of inner products, this part of the computation is (the unravelled result of) a pair of matrix-matrix products

$$A[*,(l,m)] = F_\sigma^{(l)}G_{0,\eta}^{(m)} + F_\tau^{(l)}G_{1,\eta}^{(m)}, \tag{11}$$

where

$$F_\sigma^{(l)} = \begin{pmatrix} \sigma(\eta_0,2,l) & \sigma(\eta_1,2,l) & \cdots & \sigma(\eta_p,2,l) \\ \sigma(\eta_0,3,l) & \sigma(\eta_1,3,l) & \cdots & \sigma(\eta_p,3,l) \\ \vdots & \vdots & \ddots & \vdots \\ \sigma(\eta_0,p,l) & \sigma(\eta_1,p,l) & \cdots & \sigma(\eta_p,p,l) \end{pmatrix}, \quad F_\tau^{(l)} = \begin{pmatrix} \tau(\eta_0,2,l) & \tau(\eta_1,2,l) & \cdots & \tau(\eta_p,2,l) \\ \tau(\eta_0,3,l) & \tau(\eta_1,3,l) & \cdots & \tau(\eta_p,3,l) \\ \vdots & \vdots & \ddots & \vdots \\ \tau(\eta_0,p,l) & \tau(\eta_1,p,l) & \cdots & \tau(\eta_p,p,l) \end{pmatrix},$$

are of dimension $(p-1) \times (p+1)$, and

$$G_{0,\eta}^{(m)} = \begin{pmatrix} I_0(\eta_0,2,m) & I_0(\eta_0,3,m) & \cdots & I_0(\eta_0,p,m) \\ I_0(\eta_1,2,m) & I_0(\eta_1,3,m) & \cdots & I_0(\eta_1,p,m) \\ \vdots & \vdots & \ddots & \vdots \\ I_0(\eta_p,2,m) & I_0(\eta_p,3,m) & \cdots & I_0(\eta_p,p,m) \end{pmatrix},$$

$$G_{1,\eta}^{(m)} = \begin{pmatrix} I_1(\eta_0,2,m) & I_1(\eta_0,3,m) & \cdots & I_1(\eta_0,p,m) \\ I_1(\eta_1,2,m) & I_1(\eta_1,3,m) & \cdots & I_1(\eta_1,p,m) \\ \vdots & & \ddots & \vdots \\ I_1(\eta_p,2,m) & I_1(\eta_p,3,m) & \cdots & I_1(\eta_p,p,m) \end{pmatrix},$$

are of dimension $(p+1) \times (p-1)$. Moreover, defining

$$A_0 = [a(\xi_s,\eta_t)],\ B_0 = [b(\xi_s,\eta_t)],\ 0 \le s,t \le p, \tag{12}$$

we find that part of the preprocessing can be performed as

$$F_\sigma^{(l)} = [G_{1,\xi}^{(l)}]^T A_0, \quad F_\tau^{(l)} = [G_{0,\xi}^{(l)}]^T B_0.$$

TABLE 2

Timings *and performance of columnwise local matrix construction, for a* 2 x 4 *grid, on* 8 *processors.*

p	Time (secs.)	Operations per matrix	Rate (Mflops)	Efficiency (%)
4	.0700	12,400	1.42	91
6	.1643	55,200	2.69	93
8	.3194	174,000	4.33	93
10	.5690	442,500	6.22	92
12	.9339	971,700	8.18	92
14	1.505	1,917,500	10.19	91
16	2.235	3,488,500	12.49	91
18	3.298	5,953,300	14.44	91
20	4.662	9,648,700	16.56	92

TABLE 3

Timings *and performance of blockwise local matrix construction, for a 2 × 4 grid, on 8 processors.*

p	Time (secs.)	Operations per matrix	Rate (Mflops)	Efficiency (%)
4	.0309	14,300	3.70	90
6	.0771	68,700	7.12	93
8	.1873	231,300	9.88	92
10	.4223	620,200	11.75	88
12	.8633	1,420,000	13.11	87
14	1.629	2,897,900	14.23	88
16	2.985	5,418,200	14.52	89
18	4.886	9,458,400	15.49	91
20	7.667	15,624,200	18.10	90

That is, except for the matrices A_0, B_0, $\{G_{0,\eta}^{(l)}\}$ and $\{G_{1,\eta}^{(l)}\}$, the construction of A can be completely structured using matrix-matrix products. A_0 and B_0 require $O(p^2)$ function evaluations. There are $O(p^3)$ quantities required for $\{G_{0,\eta}^{(l)}\}$ and $\{G_{1,\eta}^{(l)}\}$, but each requires O(1) computations.[3] Thus, these preprocessing steps represent low order costs. In our experiments, preprocessing made up at most 5% of the total cost of local matrix construction.

We consider two variants of local matrix construction based on these observations:

1. ***Columnwise construction.*** A is symmetric, so that only the part of A on or below the diagonal needs to be constructed. It is fairly easy to modify (11) to avoid some unnecessary computation. Let us use

$$F^{(l)}G^{(m)} = F^{(l)}[g_2^{(m)}, \dots, g_p^{(m)}] \tag{13}$$

as a shorthand notation for each of the two matrix-matrix products of (11). The

[3] This requires the efficient evaluation of $\{\phi_k\}$ and $\{\phi'_k\}$; for this, we make use of a special choice of polynomial basis functions built from integrated Legendre polynomials, which can be evaluated recursively [3].

product is unravelled for distribution into A as

$$[(F^{(l)}g_2^{(m)})^T,\ldots,(F^{(l)}g_p^{(m)})^T]^T.$$

The part of this vector corresponding to the block diagonal of A is $F^{(l)}g_m^{(m)}$, and the part below the diagonal is $F^{(l)}g_s^{(m)}$, $m < s \leq p$. Thus, at step m of the construction (10), we can avoid computing the part of A above the block diagonal by using only the columns of $G^{(l)}$ with indices greater than or equal to m. This entails some redundant computation, of entries of the block diagonal above the diagonal, but it retains the form of a matrix-matrix product.

2. Blockwise construction. The (l, m) column of A is determined from $F^{(l)}G^{(m)}$ (in the shorthand notation of (13)), so that all the entries of A can be obtained by rearranging the contents of the block outer product

$$\begin{pmatrix} F^{(2)} \\ F^{(3)} \\ \vdots \\ F^{(p)} \end{pmatrix} \left(G^{(2)}, G^{(3)}, \ldots, G^{(p)}\right). \tag{14}$$

This variant works with larger matrices than columnwise construction, and we expect an implementation based on it to be less dependent on the ability of system software to place needed data in cache memory.

Tables 2 and 3 show the behavior of these two variants on eight processors, for values of p between 4 and 20. We used the constant values $a = b = 1$ for the coefficients of (1) but implemented Gauss quadrature as discussed above; this makes the cost of function evaluations for (12) minimal, but it has no effect on the rest of the computations. A strategy based on blockwise construction was used to compute the entries of the other parts of the matrix, B_i and C_i of (3).

We see that for smaller values of p, the blockwise version is more efficient, since it achieves a larger Mflop rate without a great deal of extra arithmetic. As the matrix sizes grow, the computation rates of the two schemes become comparable, but the number of computations required by blockwise construction is much larger, and columnwise construction requires less CPU time. As above, it is possible to avoid much of the redundant computation of the blockwise strategy by computing

$$\begin{pmatrix} F^{(2)} \\ F^{(3)} \\ \vdots \\ F^{(p)} \end{pmatrix} \left(\tilde{G}^{(2)}, \tilde{G}^{(3)}, \ldots, \tilde{G}^{(p)}\right),$$

where $\tilde{G}^{(m)} = [g_m^{(m)}, \ldots, g_p^{(m)}]$. We expect this strategy to display essentially the same computation rate as the blockwise construction tested, which would make the blockwise construction superior for all p.

5. Discussion and comparison of costs. Our primary goal was to show that this class of (fully parallelizable) finite element computations can be implemented using matrix-matrix products so that hierarchical computer memories do not have deleterious effects on performance. The efficiencies in Tables 1, 2 and 3 indicate that this is true; efficiencies on the order of 90% are achieved for all computations, irrespective of whether all data being processed in parallel fits into cache memory.

A comparison of the costs and computation rates of static condensation (Table 1) and matrix construction (Tables 2 and 3) shows that the static condensation achieves much higher computation rates than the matrix construction, and primarily because of this, construction of the local matrices requires more CPU time than static condensation for $p \leq 16$. This is not related to data movement. Note that the static condensation entails matrix computations with matrices of order p^2, and as the computation (7) proceeds, the vector lengths of the rows of $S_{1:j_{max},j}$ and columns of $S_{j:r,1:j_{max}}$, which produce the dominant cost, become large. In contrast, the vector lengths in (13) and (14) are of order p. Since the Alliant is a vector computer with vector length equal to 32, the relatively small sizes of the matrices used in the matrix construction limit the computation rates that can be achieved there. Thus, even though their asymptotic costs are lower than for static condensation and (except for $p = 4$ and $p = 6$) they require fewer computations, matrix construction dominates the local cost for practical values of p.

REFERENCES

[1] I. Babuška, A. Craig, J. Mandel, and J. Pitkäranta. Efficient preconditionings for the p-version finite element method in two dimensions. *SIAM J. Numer. Anal.*, 28:624–661, 1991.

[2] I. Babuška and H. C. Elman. Some aspects of parallel implementation of the finite element method on message passing architectures. *J. Comp. Appl. Math.*, 27:157–187, 1989.

[3] I. Babuška, H. C. Elman, and K. Markley. Parallel Implementation of the hpversion of the finite element method on a shared-memory architecture. *SIAM J. Sci. Stat. Comput.*, 13:1433–1459, 1992.

[4] I. Babuška and M. Suri. The p- and *h-p* versions of the finite element method, an overview. *Computer Methods* in *Applied hlechanics and* Engineering, 80:5–26, 1990.

[5] J. J. Dongarra, J. du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[6] J. J. Dongarra, J. du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *A CM Trans. Math. Soft.*, 14:1–17, 1988.

[7] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM* Review, 32:54–135, 1990.

[8] C. Johnson. Numerical Solution *of Partial Differential* Equations *by the* Finite *Element Method.* Cambridge University Press, New York, 1987.

[9] Dennis K.-Y. Lee. Use of linear algebra kernels to build an efficient finite element solver. Master's Thesis, Department of Computer Science, University of Maryland at College Park, 1992.

[10] A. Weiser, S. C. Eisenstat, and M. H. Schultz. On solving elliptic equations to moderate accuracy. *SIAM J. Numer. Anal.*, 17:908–929, 1980.