Contemporary Mathematics
Volume 157, 1994

A Parallel Element-by-Element Method for Large-Scale Computations with $h-p-$ Finite Elements

S. FORESTI, S. HASSANZADEH, AND V. SONNAD

ABSTRACT. Element-by-Element (EBE) methods represent the lowest degree of domain decomposition in finite element analysis. The use of EBE strategies with iterative solvers allows to minimize storage requirements. However, the process of recomputing elemental matrices can be exorbitantly expensive in computation time, particularly when using high order finite elements.

An operator, based on tensor products factorizations, can be used within high order elements, to minimize the number of floating point operations. We show in this paper a strategy for optimal computational efficiency, when $h-p-$ adaptation is used. The method is amenable to parallel computations and we describe the strategy for implementation on both shared and distributed memory parallel computers.

1. Introduction

High order finite elements and $h-p-$ adaptive strategies have been studied and succesfully used to more accurately and efficiently solve finite element applications [**3, 2, 12**].

Despite the advantages and convenience of these methods, the solution of real problems and industrial applications still leads to large-scale discretized systems, that require very high storage and computational capabilities. Solution methods and computational techniques that are scalable and take advantage of the architecture of parallel and distributed computers, and superscalar microprocessors, are the necessary complement for solving very large applications in practical time.

Solution methods that rely on factoring the global matrix usually have high storage and correspondingly high computational requirements because of the fill-

1991 *Mathematics Subject Classification.* Primary 65Y05, 65N30; Secondary 65F10, 65Y10.
This paper is in final form and no version of it will be submitted for publication elsewhere

0271-4132/94 $1.00 + $.25 per page

in between the bands. The storage requirements increase almost quadratically with the number of degrees of freedom: this severely limits the size of three-dimensional problems that can be solved with reasonable memory requirements.

The storage required by iterative methods, if the global matrix is formed, is lower than direct methods, because no fill-in occurs, but it still grows more than linearly with the number of degrees of freedom; therefore, iterative methods by themselves do not offer the best scalability potentials, and implementation on massively parallel computers is still limited by increasingly large data sets.

Element-by-Element (EBE) methods, have been succesfully used in conjunction with iterative solution techniques to minimize the storage requirements of traditional ($h-$) finite element applications, and therefore to solve larger problems on a given computer [**11, 4, 13**]. However, the use of $p-$adaptive finite elements dramatically increases the computational complexity, and EBE methods in the traditional approach lead to a serious bottleneck. An operator, based on tensor products factorizations, can be used within higher order elements, to minimize the number of floating point operations [**8, 9, 6**]. We show in this paper a strategy for optimal computational efficiency when $h-p-$ adaptation is used.

EBE methods represent the lowest degree of domain decomposition in finite element analysis, and are naturally implemented in parallel: we describe the strategy for implementation on both shared and distributed memory parallel computers.

2. Rapid operator for matrix-vector products

At the heart of any iterative solver is a matrix-vector product. In the finite element approach, the matrix is obtained by assembling matrices corresponding to individual elements. The result of a matrix-vector multiplication $C_i = A_{ij}U_j$, carried out on an element-by-element basis, consists of assembling the result of elemental matrix-vector products C_{i_e}. This is equivalent to assembling the result of element based integrals:

$$C_i = \biguplus_e C_{i_e} = \biguplus_e A_{i_e j_e} U_{j_e} = \sum_e \int_{\Omega_e} \Psi_{i_e} \Psi_{j_e} U_{j_e} \, d\Omega_e \tag{2.1}$$

We consider hexahedral elements (quadrilateral in 2-D), and we use the mass operator to simplify the description. More details can be found in [**8**].

The computation can be carried out either way: the former is to form elemental matrices and perform matrix-vector products; the latter is to compute elemental integrals. If tensor-product basis functions are used, the calculations of the integrals can be performed by using tensor-product factorization, (Tensor-product basis functions are usually used in high order elements, or p-version [**1**]).

The "rapid" evaluation with the use of tensor-product basis functions is here

illustrated with the mass operator: We define p basis functions per dimension:

$$\Psi_\alpha(r,s,t) = \psi_{\alpha_r}(r)\ \psi_{\alpha_s}(s)\ \psi_{\alpha_t}(t) \qquad \forall \begin{cases} \alpha & = 1 \dots p^3 \\ \alpha_r, \alpha_s, \alpha_t & = 1 \dots p. \end{cases} \tag{2.2}$$

The evaluation of the vector C_{i_e} can be represented in this form:

$$C_{i_e} = C_\alpha = C_{\alpha_r \alpha_s \alpha_t} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \psi_{\alpha_r}\psi_{\alpha_s}\psi_{\alpha_t}\ \psi_{\beta_r}\psi_{\beta_s}\psi_{\beta_t} U_{\beta_r\beta_s\beta_t}\ dr\ ds\ dt. \tag{2.3}$$

($\forall\ \alpha_r, \alpha_s, \alpha_t = 1 \dots p$ and contracting the indices $\beta_r, \beta_s, \beta_t$).

When numerical integration is performed, integrals are replaced by weighted sums of the functions evaluated at the integration points. Let us indicate with $r^{g^r}, s^{g^s}, t^{g^t}$ $(\forall\ g^r, g^s, g^t = 1 \dots g)$ the points of a numerical quadrature and $w^{g^r}, w^{g^s}, w^{g^t}$ the corresponding weights. The one dimensional basis functions, evaluated at the integration points, are $p \times g$ matrices, denoted by $\psi^{g^r}_{\alpha_r} = \psi_{\alpha_r}(r^{g^r})$, $\psi^{g^s}_{\alpha_s} = \psi_{\alpha_s}(s^{g^s})$, $\psi^{g^t}_{\alpha_t} = \psi_{\alpha_t}(s^{g^t})$. Therefore,

$$C_{\alpha_r\alpha_s\alpha_t} = \psi^{g^r}_{\alpha_r}\ \psi^{g^s}_{\alpha_s}\ \psi^{g^t}_{\alpha_t}\ \psi^{g^r}_{\beta_r}\ \psi^{g^s}_{\beta_s}\ \psi^{g^t}_{\beta_t}\ U_{\beta_r\beta_s\beta_t}\ w^{g^r} w^{g^s} w^{g^t}. \tag{2.4}$$

The computation is performed by contracting the indices one step at a time (in order of $\beta_r, \beta_s, \beta_t, g^r, g^s, g^t$), forming a sequence of temporary tensors with three indices.

The computational requirements vary according to the order of polynomials: we are trying to address the problem of choosing the most efficient way to carry out these results. Table 1 shows the number of operation and storage required to evaluate the vector C_{i_e}, for every element e. The exact operation count depends on the operator used: in [**6**] the 3-dimensional equations of elasticity are considered. However, the numerical counts showed that the threshold of the inequalities for most differential operators lies in the same polynomial number p: greater than 3 in 3-D, and greater than 5 in 2-D. The optimal strategy consists then in using rapid integral evaluation in those in elements that are more than cubic (5-th order in 2-D), and choosing integral evaluation with no tensor product factorization in cubic (or less) elements (5-th order or less in 2-D). In this latter case, forming an elemental matrix and performing a matrix-vector product is not convenient, unless the elemental matrix was managed to be formed once and for all the iterations, and kept in main storage.

3. Parallel implementation

The evaluation of the global matrix-vector product C_i, carried out on an element-by-element basis, is naturally implemented in parallel by the simple strategy of defining elemental computations as processes, and assigning them to different processors. The computation of each elemental vector C_{i_e} is completely

	2-Dimensions		3-Dimensions	
	storage	flops	storage	flops
form matrix	p^6	$o(p^6)$	p^4	$o(p^4)$
matrix-vector	p^6	$2p^6$	p^4	$2p^4$
integral	$o(p^3)$	$o(p^6)$	$o(p^2)$	$o(p^4)$
rapid evaluation	$o(p^3)$	$o(p^4)$	$o(p^2)$	$o(p^3)$

TABLE 1. Storage and operation count for computing elemental results.

independent of other elements, and can be performed either with rapid integral evaluation, or matrix-vector product as discussed in the previous paragraph.

$$C_i = \biguplus_e C_{i_e} = \biguplus_e A_{i_e j_e} U_{j_e} = \sum_e \int_{\Omega_e} \Psi_{i_e} \Psi_{j_e} U_{j_e} \, d\Omega_e \tag{3.1}$$

The algorithm has a parallel content that is close to 100%, and relatively coarse granularity, particularly when using high order basis functions.

The implementation on a shared-memory computer, is accomplished by the simple strategy of assigning elemental processes to processors as they become available, with a loop over the elements. A key advantage of this approach, is that there is no need to partition the elements among the processors prior to the computations. This takes full advantage of a shared memory architecture, where all data is available to all processors at all times. The problem of load balancing is avoided entirely, because it is automatically adjusted at computation time. The most efficient data structure is to store assembled (global) data in the shared memory. A copy of elemental data is copied in temporary storage as needed, by yanking the pertinent components from the assembled data (a pointer vector indicates the location in the global vector of every term of the elemental vector). Elemental data are then processed, and results are reassembled into the global results, using the same pointer. This is the only stage with data dependencies: it may happen that a single location needs to be updated by two or more processors simultaneously. These conflicts are resolved by a locking mechanism that serializes the writing into the location, and a synchronization barrier between all processors ensures that the subsequent task in the program is executed when the assembly has been completed.

The implementation on a distributed-memory computer, requires that elements be partitioned among processors prior to the computations. This partition has to be studied according to a criterion of load balancing and minimal communications: each processors is assigned a cluster of neighboring elements. The appropriate data structure is to store a copy of elemental data for each element in the cluster in the local memory of the corrispondent processor. Components on the boundaries between elements in a cluster will be repeated; however, this allows elements to be reassigned to a different processor at a later stage of the computation in order to adjust the load balance, if changes in the polynomial order occur due to adaptation. Each processor computes sequentially elemental

processes of the assigned cluster, independently of the clusters on other processors. Coefficients that are on boundaries between elements are sequentially updated in the local memory, if the elements are in the same cluster, and exchanged via message passing, if the elements are in different clusters. The implementation of elemental processes is dominated by tensor products. In the former case, a matrix-vector product $C_{i_e} = A_{i_e j_e} U_{j_e}$ is performed, where the matrix $A_{i_e j_e} = \int_{\Omega_e} \Psi_{i_e} \Psi_{j_e} \, d\Omega_e$ has first to be formed, or can be formed and stored once and for all, if the storage is enough.

In the latter case, the rapid element operator consists of a sequence of tensor products of this type: $W^{(n)}_{j\ (ka)} = \psi^a_i \ W^{(n-1)}_{i\ (jk)}$, where i is the index being contracted at the step n. Tensors W are stored as matrices by grouping the indices that are not contracted. As a result of the appropriate order chosen for the indices, the resulting computation consists of matrix-matrix products with stride 1.

In both cases, the computation can be performed by using Basic Linear Algebra Routines (BLAS), that are optimized for a particular architecture. and the array sizes and ordering is such that the computation is carried out at close to peak performance, on vector or RISC processors. Moreover, the level of polynomials and number of integration points, used respectively in the both cases, are generally of such an order that all the tensors mentioned above fit in cache memories.

4. Numerical results and conclusions

In order to illustrate the computational efficiency of this algorithm, we have solved the three-dimensional equations of linear elasticity. The physical domain is a cube, which has been discretized with 343 elements (7 per axis direction); we have deliberately distorted the elements in order to be representative of a complex geometry. The goal of this work was to solve problems of scaling size, reaching the order of the million of degrees of freedom. We defined a set of problems by increasing the level of polynomials and solved them with the Conjugate Gradient method. We notice that, in this context, we have not used any form of preconditioner. Our previous work [**6**] showed that it is possible to improve the convergence rate, by using "lower $p-$level" preconditioners. However, the use of these preconditioners requires storage that increases more than linearly with the number of degrees of freedom: the solution of problems with one million degrees of freedom can not be solved "in core" memory. Moreover, lower level preconditioners are not inherently parallelizable. Therefore, the use of these preconditioners becomes impractical in this context. Here we show how this method allows the solution of very large and scalable problems; we will address the issue of opportune parallel and low storage preconditioners in future work.

Here we analyze the computational requirements on an IBM 3090/600 VF. a shared memory mainframe with six vector processors. For a given level of

polynomials, table 2 shows degrees of freedom, total storage, elapsed time for one iteration (v corresponds to the best implementation on 1 vector processor, p on 6 vector processors, and s is the speedup), number of iterations to achieve a tolerance of 10^{-3}, and elapsed time for the solution on 6 vector processors. The times were measured during a benchmark with dedicated system.

Pol	DGF	Storage	Iteration time			Tot Iter	Tot Time
			v	p	s		
2	6,591	0.9 Mbytes	1.59"	0.32"	5.45	96	5'
6	206,763	12.5 Mbytes	6.66"	1.25"	5.43	607	17'
10	985,527	54.0 Mbytes	22.02"	4.18"	5.39	1958	141'

TABLE 2. Degrees of freedom, storage, elapsed time for one iteration (v = serial, p = parallel, and s = speedup), iterations to 10^{-3} tolerance, and elapsed time for parallel solution.

The storage as well as the elapsed time increase linearly with the number of degrees of freedom: therefore, the performance of this algorithm scales to very large problems.

In conclusion, we have presented an Element-by-Element method whereby matrix-vector products, arising from the iterative solutions of h–p–version finite element applications, can be computed with optimal computational efficiency. The method is amenable to parallel computations and we describe the strategy for implementation on both shared and distributed memory parallel computers. This method allows the solution of increasingly large finite element problems taking advantage of superscalar microprocessors, and of the scalable architecture of parallel and distributed computers, Applications of this method to large-scale elastodynamic problems can be found in [**7, 10**].

REFERENCES

1. I. Babŭska, M. Griebel, and J. Pitkaranta. The problem of selecting the shape functions for a p- type finite element. *Int. J. Num. Meth. Engg.*, 28:1891–1908, 1989.
2. I. Babŭska and M. Suri. The p- and h-p versions of the finite element method: an overview. Technical Report BN-1101, Institute for Physical Science and Technology, University of Maryland, 1989.
3. I. Babŭska, B.A. Szabo, and I.N. Katz. The p-version of the finite element method. *SIAM J. Num. Anal.*, 18:515–545, 1981.
4. G.F. Carey and B.N. Jiang. Element-by-element preconditioned conjugate gradients algorithm for compressible flow. In W.K. Liu, T. Belytschko, and K.C. Park, editors, *Proceedings of the International Conference on Innovative Methods for Nonlinear Problems*, pages 41–49. Pineridge Press, Swansea, 1984.
5. S. Foresti, G. Brussino, S. Hassanzadeh, and V. Sonnad. Multilevel solution method for the p-version of finite elements. *Computer Physics Communications*, 53:349–355, 1989.
6. S. Foresti, S. Hassanzadeh, H. Murakami, and V. Sonnad. A comparison of preconditioned iterative solution techniques with rapid operator evaluation against direct solution methods. *Int. J. Num. Meth. Eng.*, 32(5):1137–1144, 1991.

7. S. Foresti, S. Hassanzadeh, H. Murakami, and V. Sonnad. Finite element analysis of very large-scale structural problems using minimal memory. Technical Report USI-7, Utah Supercomputing Institute, University of Utah, 1991.
8. S. Foresti, S. Hassanzadeh, H. Murakami, and V. Sonnad. Parallel rapid operator for iterative finite element solvers on a shared memory machine. *Parallel Computing*, 18(12), 1992.
9. S. Foresti, S. Hassanzadeh, H. Murakami, and V. Sonnad. Parallel rapid operator for iterative finite element solvers on a shared memory machine. Technical Report USI-31, Utah Supercomputing Institute, University of Utah, 1992.
10. S. Hassanzadeh, S. Foresti, H. Murakami, and V. Sonnad. Minimal storage finite element solution of large-scale three-dimensional elastodynamic problems. In *Proceedings of Eighth National Conference on Computing in Civil Engineering*, ASCE, Dallas, Texas, 1992.
11. T. Hughes, I. Levit, and J. Winget. Element-by-element implicit algorithms for heat conduction. *J. Engg. Mech*, 109:576–585, 1983.
12. B.A. Szabo and I. Babŭska. *Finite Element Analysis.* Wiley, 1991.
13. J. Winget and T. Hughes. Solution algorithms for nonlinear transient heat conduction analysis employing element-by-element iterative strategies. *Comp. Meth. Appl. Mech. Eng.*, 52:711–815, 1985.

UTAH SUPERCOMPUTING INSTITUTE, UNIVERSITY OF UTAH, SALT LAKE CITY, UTAH 84112
E-mail address: stefano@osiris.usi.utah.edu

FUJITSU AMERICA, INC., SAN JOSE, CA 95134
E-mail address: siamak@fai.com

ADVANCED WORKSTATION DIVISION, IBM CORPORATION, AUSTIN, TEXAS 78758
E-mail address: sonnad@ausvm6.vnet.ibm.com