

An Equation by Equation Method for Large FE Problems in a Distributed Computing Environment

Ganesh Thiagarajan¹ and Anoop G. Varghese²

Civil Engineering, School of Computing and Engineering

350J RHFH, 5100 Rockhill Road, University of Missouri-Kansas City, MO, 64110

Abstract

For finite element problems involving millions of unknowns, iterative methods setup in a parallel computing platform, are more efficient than direct solver techniques. Considerations that are important in the solution of such problems include the time of computation, the memory required and the type of platform being used to solve the problem. Traditionally shared memory machines were popular. However, distributed memory machines are now gaining wider acceptance due to their relatively low cost and ease of setup. The conventional approach to setup an iterative finite element solver is the Element-by-Element (EBE) method with the preconditioned conjugate gradient (PCG) solver. The EBE method is reported by Horie and Kuramae (1997, *Microcomputers in Civil Engineering*, 2, 12) to be suitable for shared memory parallel architectures, but has certain conflicts in distributed memory machines. This paper proposes a new algorithm that is developed for the parallelization of the solution algorithm of the finite element setup for a *distributed memory* environment. The new method, called the Equation-by-Equation (EQBYEQ) method, is based on generating and storing the stiffness matrix on an equation-by-equation basis in contrast to the element-by-element basis. This paper discusses the algorithm and implementation of details. The advantages of the EQBYEQ scheme when compared to the EBE scheme, in distributed environment, is discussed

Keywords: Conjugate Gradient, Finite Element, Element-by-Element, Solver algorithms, Equation-by-Equation, Distributed memory.

1 Introduction

Current day developments in both software and hardware technology have made distributed memory architectures based on multiple instruction multiple data (MIMD) machines very inexpensive. Advances in the parallel computing environment in the form Beowulf type clusters and Message Passing Interface (MPI) and compilers such as High Performance Fortran (HPF) can significantly change the way engineering analysis will be done in the future. While the main attraction of Beowulf clusters is their low cost, one must also keep in mind the problem of intercommunication amongst the processors while developing algorithms. Hence, new and suitable algorithms are required to take advantage of these architectures.

Finite element problems can be parallelized in two ways; namely by domain decomposition (Al-Nasra and Nguyen(10), Farhat (11), Hodgson and Jimack(7), Shahid et. al. (5), Wang and Bruch (2), Wang et. al

¹Assistant Professor

²Graduate Student

(9), Yugawa et. al. (8) etc.) or by parallelizing the solution algorithm. The time spent in solving the resultant equations is about ninety to ninety-five percent and about eighty percent of the total time in a large linear elastic problem and a non-linear problem respectively. Hence, finding efficient ways to parallelize the solution algorithm greatly reduces the total run time. For extremely large problems, involving anywhere from a hundred thousand to two million unknowns, direct solution techniques are not feasible due to their heavy demand on memory requirements. Hence, iterative solution methods such as the Krylov subspace techniques are particularly attractive due to their super-linear convergence rates. The setup in which extremely large problems can be solved, while keeping the memory requirements low, is the element-by-element (EBE) strategy where the global stiffness matrix is never assembled. Numerous works in the area of the parallel implementation of EBE method have been reported in literature (Barragy et. al. (1), King and Sonnad (4), Smith (3) etc.). King and Sonnad (4) for instance, have reported the effectiveness of element by element preconditioned conjugate gradient method in shared memory computers. Adeli (13) has presented a review of material published in all archived journals in parallel computing since 1994.

It has been observed that the parallel implementation of the EBE technique in a distributed memory environment has certain inherent conflicts (Horie and Kuramae (6)). The methodology proposed in this work resolves some of the conflicts and is better suited for distributed memory architecture systems. This strategy, termed as the Equation-by-Equation (EQBYEQ) (due to the nature of its solution) is setup to solve extremely large finite element problems within the realm of Krylov subspace methods, namely the Preconditioned Conjugate Gradient technique. The aim of this paper is to compare EQBYEQ method with EBE method implemented with a simple preconditioner - the diagonal preconditioner. The problems and conflicts of the EBE method are described and the solutions that the EQBYEQ method offers are presented. The entire development work was done in FORTRAN 90 and HPF (High Performance Fortran). Size and performance comparisons are made and outlined. The EQBYEQ technique has not been reported elsewhere to the best of the authors' knowledge.

1.1 Objectives of the research

The main objectives of the research study presented in this paper are as follows

1. Introduce a new algorithm called EQBYEQ method, which has better work efficiency than the EBE method in a distributed memory environment.
2. Implement the proposed EQBYEQ algorithm and compare it with the traditional EBE method; using the diagonal preconditioner.

2 Finite Element Setup

A complete Finite Element (FE) package tied in with an automatic finite element mesh generator has been developed for the purpose of this study. The parallelization of the CG iterative scheme requires certain variables, called *mapping variables*, at the time of the generation of the mesh. Hence, an automatic isoparametric mesh generator for multiply connected regions was developed. Another advantage of the integrated mesh generator is that very large meshes can be easily generated. As the generation is done only once, it is setup in a serial mode.

Plane stress/plane strain, linear elastic modeling is considered in this study using an 8-noded quadrilateral isoparametric element. Boundary conditions are incorporated to eliminate zero-valued degrees of freedom (DOF) from the global stiffness matrix. Since automatic mesh generation is setup, boundary conditions and applied loads are specified at coordinates and the code automatically identifies the node number appropriate to this location to make the node number identification. A brief mathematical description of the finite element analysis and the conjugate gradient iterative scheme used for the solution algorithm follows.

2.1 Mathematical Basis

Using the variational principles the expression for the virtual work for the finite element discretization of a static problem with no body forces (Oden and Carey,(12), Cook et. al.(14)) is given as

$$-\int_v \delta \text{Tr}(\epsilon \sigma) dV + \int_{S_\sigma} \delta \mathbf{x}^T \mathbf{T} dS = 0 \quad (1)$$

where $\delta \mathbf{x}$ is the virtual displacement field compatible with the virtual strain $\delta \epsilon$ and σ is the Cauchy stress which is in equilibrium with the applied traction field \mathbf{T} . Applying strain-displacement and constitutive relations, the displacement based FE problem can be reduced to a set of linear equations using the principle of minimization of the total potential energy as

$$[K]\{u\} = \{F\} \quad (2)$$

where $\{F\}$ is the applied load vector, $\{u\}$ is the unknown nodal DOF vector and $[K]$ is the global stiffness matrix. The method used to solve the set of linear equations in equation (2) is the Preconditioned Conjugate Gradient (PCG). The finite element method is a well researched area and the reader is referred to other excellent resources (15; 16; 12) for further reading.

2.2 Preconditioned Conjugate Gradient Method

Since FE problems with unknowns of the order of a million are being investigated, an iterative solver based on Krylov subspace is setup for the solution of the resulting linear algebraic equations. The solution of the equations using the conjugate gradient technique is based on the minimization of following quadratic functional.

$$\Gamma(x) = \frac{1}{2} \{x\}^T [K] \{x\} - \{x\}^T \{F\} \quad (3)$$

The basic steps involved in a PCG method for the solution of linear equations are summarized below:

1. Start with an initial approximation $\{x\}^0$
2. Determine the direction of the search vector $\{p\}$. The Jacobi (diagonal) preconditioner is a highly parallelizable choice. Hence, starting with an initial approximation $\{u\}^0$, the initial preconditioned residual $\{r\}^0$ and the initial search direction vector $\{p\}^0$ can be expressed as

$$\{r\}^0 = \{F\} \quad (4)$$

$$\{d\}^0 = [P]\{r\}^0 \quad (5)$$

$$\{p\}^0 = \{d\}^0 \quad (6)$$

where $[P]$ is the preconditioner.

3. The magnitude along the search vector α^k is the one that minimizes $F(x^k + \alpha p^k)$ with respect to α . The equations for the iterative procedure for the k^{th} iteration can now be expressed as

$$\{u\}^k = [K]\{p\}^k \quad (\text{matrix-vector product}) \quad (7)$$

$$\alpha^k = \frac{(\{r\}^k)^T \{d\}^k}{(\{p\}^k)^T \{u\}^k} \quad (2 \text{ vector-vector products}) \quad (8)$$

4. A new solution point can be setup as

$$\{x\}^{k+1} = \{x\}^k + \alpha^k \{p\}^k \quad (\text{vector+scalar*vector}) \quad (9)$$

5. The residual and search direction for the next iteration can thereafter be determined as shown below.

$$\{r\}^{k+1} = \{r\}^k - \alpha \{u\}^k \quad (\text{vector+scalar*vector}) \quad (10)$$

$$\{d\}^{k+1} = [P]\{r\}^{k+1} \quad (11)$$

$$\beta^k = \frac{(\{r\}^{k+1})^T \{d\}^{k+1}}{(\{r\}^k)^T \{d\}^k} \quad (2 \text{ vector-vector products}) \quad (12)$$

$$\{p\}^{k+1} = \{d\}^{k+1} + \beta^k \{p\}^k \quad (\text{vector+scalar*vector}) \quad (13)$$

An L1 error norm to exit the loop is adopted as follows,

$$\text{abs}(\max(\{r\}^{k+1})) \leq TOL \quad (14)$$

3 Problem Statement

In a distributed memory environment each processor owns the data present in its local memory and follows the *owner-computes* paradigm. Any information not present in the local memory is fetched via interprocessor communications. It is found that the best performance of a distributed problem occurs when there are no interprocessor communications. Hence, the primary goal is to setup the problem with as much data present locally as possible.

The main issue in the parallelization of the above algorithm in a distributed memory environment is that there are two conflicting loops involved in each iteration. The outer loop index represents the equation number. Equations (9), (10), (12) and (13) are involved with the outer loop index only and hence inherently parallel. On the other hand the computation of equation (7) involves an additional inner loop, which represents the elements. Hence, in order to minimize communications in the matrix-vector operation the stiffness matrices of all elements must be present in each processor, thereby imposing a huge memory requirement. A new methodology, called the EQBYEQ method is proposed and described in the following sections to reduce this huge memory requirement, while attempting to minimize communications.

4 Theoretical Basis of the EQBYEQ Method

The primary issue in the iterative scheme is the matrix-vector product in equation (7). Three techniques, two existing well known ones, and a third - the proposed method, are illustrated here with the help of an example. They are

- Global Stiffness Strategy
- Element-by-Element Strategy (EBE)
- Proposed Equation-by-Equation Strategy (EQBYEQ)

The mathematical validity of the proposed method is now described and compared to the other two methods. Consider a two element (4 node, 8 DOF elements) plate as shown in Figure 1 (a). Figure 1 (b) shows the associated local DOF for each element. The example presented here is demonstrated for the computation of one component of the global variable, namely u_1 .

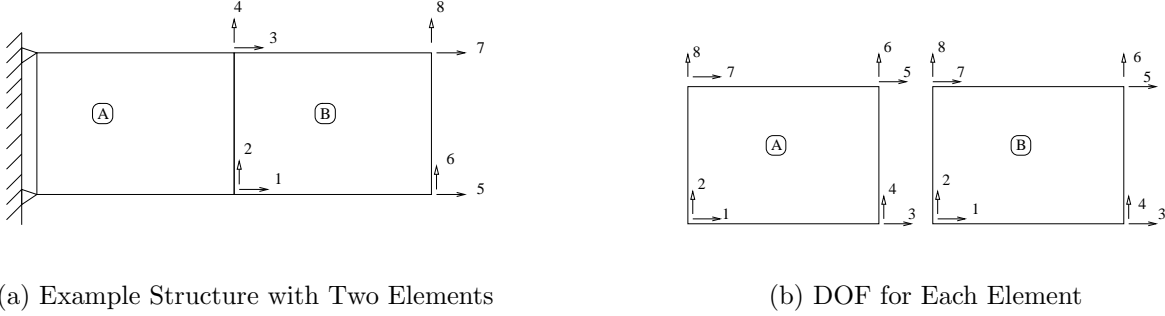


Figure 1: An example problem

4.1 Global Stiffness Strategy

In the computation of the u vector using the global stiffness method the matrix-vector is set up as follows.

$$\sum_{j=1}^{N^{\text{eqn}}} K_{ij} p_j = u_i \quad \text{where } j = 1, \dots, N^{\text{eqn}} \quad (15)$$

where K_{ij} is the global stiffness matrix set up from the element stiffness matrices. In the example considered with two elements A and B the total number of equations is eight. Considering the first row in full detail, the global stiffness matrix components can be expressed as

$$\begin{aligned} K_{11} &= k_{33}^A + k_{11}^B \\ K_{12} &= k_{34}^A + k_{12}^B \\ K_{13} &= k_{35}^A + k_{17}^B \\ K_{14} &= k_{36}^A + k_{18}^B \\ K_{15} &= k_{13}^B \\ K_{16} &= k_{14}^B \\ K_{17} &= k_{15}^B \\ K_{18} &= k_{16}^B \end{aligned} \quad (16)$$

where k_{ij}^e is the element stiffness matrix of the element e . Hence the global $\{u\}$ vector is computed using the matrix-vector product and the first component can be expanded as shown below,

$$u_1 = K_{1j}p_j \quad \text{where } j = 1, 8 \quad (17)$$

$$= [k_{33}^A p_1 + k_{34}^A p_2 + k_{35}^A p_3 + k_{36}^A p_4] + [k_{11}^B p_1 + k_{12}^B p_2 + k_{13}^B p_5 + k_{14}^B p_6 + k_{15}^B p_7 + k_{16}^B p_8 + k_{17}^B p_3 + k_{18}^B p_4] \quad (18)$$

The primary issue with this method is the size of the global stiffness matrix, even in a sparse storage scheme as the memory requirements grows geometrically with the problem size.

4.2 Element-by-Element Strategy (EBE)

In the EBE strategy the global stiffness matrix is never assembled. The global $\{u\}$ vector is obtained by performing the matrix-vector product at the element level as seen in equation (19)

$$(u_i)^{\text{elem}} = \sum_{j=1}^8 k_{ij}^{\text{elem}} p_j \quad \text{where } i = 1, \dots, 8 \quad (19)$$

and the correspondence between the local and global DOF ($u_{\text{globalDOF}} \Rightarrow u_{\text{localDOF}}^{\text{elem}}$) is based on the identified global DOF corresponding to the local DOF. If any local DOF is constrained globally, then its contribution is dropped. For example in element A the assignment is set as

$$\begin{aligned} u_1 &\Rightarrow u_3^A \\ u_2 &\Rightarrow u_4^A \\ u_3 &\Rightarrow u_5^A \\ u_4 &\Rightarrow u_6^A \end{aligned} \quad (20)$$

An algorithm is set up to identify the non-zero global DOF that correspond to the local DOF for each element. To complete the illustration for the entire problem, the two element $\{p\}$ vectors are as follows

$$\{p\}^A = \{0, 0, p_1, p_2, p_3, p_4, 0, 0\} \quad (21)$$

$$\{p\}^B = \{p_1, p_2, p_5, p_6, p_7, p_8, p_3, p_4\} \quad (22)$$

Performing the matrix vector product for the global u_1 and observing that the first global DOF corresponds to the third DOF of element A and the first DOF of element B, it can be shown that

$$\begin{aligned} u_1 &= u_3^A + u_1^B \\ &= \sum_{j=1}^8 k_{3j}^A p_j^A + \sum_{j=1}^8 k_{1j}^B p_j^B \\ &= [k_{33}^A p_1 + k_{34}^A p_2 + k_{35}^A p_3 + k_{36}^A p_4] + [k_{11}^B p_1 + k_{12}^B p_2 + k_{13}^B p_5 + k_{14}^B p_6 + k_{15}^B p_7 + k_{16}^B p_8 + k_{17}^B p_3 + k_{18}^B p_4] \end{aligned} \quad (23)$$

Comparing equations (18) and (23) it can be seen that they yield identical results.

4.3 Proposed Equation-by-Equation Strategy (EQBYEQ)

The new EQBYEQ strategy performs the same matrix-vector computations in a manner suitable to a distributed memory environment. The idea driving this method is to generate all the *relevant rows* of the stiffness matrix of the *all the elements* that contribute to the stiffness of a *particular equation* - locally in each processor. This would ensure that all the computations are done independently in each processor, and also reduces the memory required in each processor by manifold. Realizing that in a two-dimensional environment only *four elements surround a node* at most, only these four elements contribute to the global stiffness matrix of that DOF. A three index variable $K_{i,jj,dof}^{EQBYEQ}$ is used where the index i represents the equation number, jj represents the four elements that can surround a node and dof represents the local DOF in an element. The second index essentially represents the *rows of the element stiffness matrix* that contribute to the global stiffness matrix.

The critical aspect of the EQBYEQ approach is to set up a technique to *identify the elements* contributing to a global DOF and the *rows of the element stiffness matrix* (for example, 3, 4, 5 and 6 for element A) that are required for the element matrix-vector product. Once this identification is made the assignment can be made using dot products (4 here) instead of the complete matrix-vector products (8 for a 8 DOF element). The strategy is now outlined in the following steps.

Step 1: Identification of elements contributing to a particular equation: The identification of the elements that contribute to a global equation number is set up as follows.

$EQ_ELEM = 0$

for each element $elem$

for each local DOF j of element $elem$

$eqn \leftarrow$ global equation number corresponding to the local DOF j

$EQ_ELEM_{eqn,j} \leftarrow elem$

end for

end for

In the setup described above, the 2-dimensional mapping variable EQ_ELEM stores all the elements that contribute to a particular equation. The first index has a size equal to the total equation numbers in the problem. The second index has a size equal to the number of DOF per element. In a two-dimensional physical FE problem only four values of the second index is expected to have non-zero element numbers, due to the fact that only four local DOF will correspond to a global DOF. The second index value of the non-zero EQ_ELEM will be utilized in the next step. In the algorithm described above the mapping variable is set up in a reverse manner by identifying all the equations that an element contributes.

Step 2: Identification of the rows: The next step is to identify the row of each element matrix-vector product that contributes to the global $\{u\}$ vector. The pseudo code to do this is shown below.

for each equation i

$jj = 1$

for each local DOF dof in an element

$elem \leftarrow EQ_ELEM_{i,dof}$

if *elem* exists **then**

$K^{\text{elem}} \leftarrow$ the local stiffness matrix of the element *elem*

$K_{i,jj,1\dots 8}^{\text{EQBYEQ}} \leftarrow K_{j,1\dots 8}^{\text{elem}}$

$jj \leftarrow jj + 1$

end if

end for

end for

The index *jj* is the most important index here. This can vary from one to four depending on the number of elements surrounding the node. For inner nodes it is four and for outer nodes it is one or two. Although the number of elements surrounding a node varies from one to four, more significantly, it stores *the 8 values of the row of the element stiffness matrix* that contributes to the dot product.

This critical step identifies the rows of the corresponding element stiffness matrix, without having an extra variable (hence taking up extra memory). For the example discussed earlier the K^{EQBYEQ} is as follows

$$\begin{aligned}
K_{1,1,1\dots 8}^{\text{EQBYEQ}} &= K_{3,1\dots 8}^{\text{A}} \quad ; \quad K_{1,2,1\dots 8}^{\text{EQBYEQ}} = K_{1,1\dots 8}^{\text{B}} \\
K_{2,1,1\dots 8}^{\text{EQBYEQ}} &= K_{4,1\dots 8}^{\text{A}} \quad ; \quad K_{2,2,1\dots 8}^{\text{EQBYEQ}} = K_{2,1\dots 8}^{\text{B}} \\
K_{3,1,1\dots 8}^{\text{EQBYEQ}} &= K_{5,1\dots 8}^{\text{A}} \quad ; \quad K_{3,2,1\dots 8}^{\text{EQBYEQ}} = K_{7,1\dots 8}^{\text{B}} \\
K_{4,1,1\dots 8}^{\text{EQBYEQ}} &= K_{6,1\dots 8}^{\text{A}} \quad ; \quad K_{4,2,1\dots 8}^{\text{EQBYEQ}} = K_{8,1\dots 8}^{\text{B}} \\
K_{5,1,1\dots 8}^{\text{EQBYEQ}} &= K_{3,1\dots 8}^{\text{B}} \\
K_{6,1,1\dots 8}^{\text{EQBYEQ}} &= K_{4,1\dots 8}^{\text{B}} \\
K_{7,1,1\dots 8}^{\text{EQBYEQ}} &= K_{5,1\dots 8}^{\text{B}} \\
K_{8,1,1\dots 8}^{\text{EQBYEQ}} &= K_{6,1\dots 8}^{\text{B}}
\end{aligned} \tag{24}$$

From equation (24) it can be seen that the global DOF 1 has both elements contributing to it while the global DOF 8 has only element B contributing to it. The setup to complete the matrix vector product is as follows.

Step3: Proposed Matrix-Vector product setup:

for each equation *i*

$u_i \leftarrow 0$

$jj \leftarrow 1$

$p_{i,1\dots 8}^{\text{mul}} \leftarrow 0$

for each local DOF *j* in an element

$elem \leftarrow EQ_ELEM_{i,j}$

if *elem* exists **then**

for each local DOF *k* in an element

$eqn \leftarrow ELEM_EQ_{elem,k}$

$p_{i,k}^{\text{mul}} \leftarrow p_{eqn}$

end for

$u^{\text{tmp}} \leftarrow \sum_{k=1}^8 K_{i,jj,k}^{\text{EQBYEQ}} p_{i,k}^{\text{mul}}$ (where u^{tmp} is a scalar)

```

     $u_i \leftarrow u_i + u^{\text{tmp}}$ 
     $jj \leftarrow jj + 1$ 
  end if
end for
end for

```

To illustrate this method using the example, global equation 1 has both elements contributing to it. As seen above $K_{1,1,1\dots 8}^{\text{EQBYEQ}}$ stores the third row $K_{3,1\dots 8}^{\text{elem}}$ of element A and $K_{1,2,1\dots 8}^{\text{EQBYEQ}}$ stores the first row $K_{1,1\dots 8}^{\text{elem}}$ of element B. The $p_{1,1\dots 8}^{\text{mul}}$ stores the p vector corresponding to the two elements $(0, 0, 0, p_1, p_2, p_3, p_4, 0, 0)$ and $(p_1, p_2, p_5, p_6, p_7, p_8, p_3, p_4)$. The two dot products would now result in

$$\begin{aligned}
 u_1 &= u_3^{\text{A}} + u_1^{\text{B}} \\
 &= [k_{33}^{\text{A}}p_1 + k_{34}^{\text{A}}p_2 + k_{35}^{\text{A}}p_3 + k_{36}^{\text{A}}p_4] + \\
 &\quad [k_{11}^{\text{B}}p_1 + k_{12}^{\text{B}}p_2 + k_{13}^{\text{B}}p_5 + k_{14}^{\text{B}}p_6 + k_{15}^{\text{B}}p_7 + k_{16}^{\text{B}}p_8 + k_{17}^{\text{B}}p_3 + k_{18}^{\text{B}}p_4]
 \end{aligned} \tag{25}$$

which is identical to the results obtained in earlier two methods.

4.4 Comparison of EBE and EQBYEQ Methods

The parallel implementation of the two strategies (EBE and EQBYEQ) are compared and contrasted in the following section. The primary bottleneck in the parallel implementation of PCG algorithms is the matrix-vector product

$$u_i = \sum k_{ij}p_j \tag{26}$$

EBE:In the EBE strategy there are two *loop index variables* in any iteration as shown below. The PCG algorithm to solve the FE set of equations can be expressed by the following pseudocode

```

...
for each equation  $i$ 
  ...
  for each element  $k$ 
    Matrix-Vector product on each element
  end for
  ...
end for

```

The outer loop is parallelized but the inner loop cannot be parallelized simultaneously as it conflicts with outer loop variable. This results in the inner loop being performed independently for each equation. Another issue is the distribution of the element stiffness matrices. To avoid communication during iterations the best strategy is to adopt a degenerate distribution of K^{elem} , implying that each processor has a copy of all element stiffness matrices, distributed once before all iterations begin. This implies that a problem of size 100,000 elements would require a storage of 25,600,000 words or about 100 MB of local memory in each processor.

EQBYEQ:The EQBYEQ strategy results in only *one loop index variable* for the entire operation, which parallelizes the loop completely as shown by the following pseudocode

```

...
for each equation  $i$ 
  ...
  for each equation  $k$ 
    Matrix-Vector product on each element (equation-by-equation)
  end for
  ...
end for

```

It is relevant to mention that when the stiffness matrix entries required by an equation (rather than an element) is generated it requires more calls to the subroutine that computes the stiffness matrix. For example, the subroutine to generate the stiffness matrix in the EBE method is called once for each element, whereas the same subroutine for the EQBYEQ strategy is called up to four times for each equation. This results in increased computations but is not detrimental as no communications are involved and results in a better utilization of the idle clock cycles of the processor. The stiffness matrix for the EQBYEQ method has the size $N^{\text{eqn}} \times 4 \times 16$. Since the stiffness matrix is now *block distributed*, implying that each processor computes and stores only the entries for the equation it is responsible for, it results in each processor owning only a part of it. This results in a big reduction of the local memory used and thus much larger problems can be solved.

Operation Count Comparison : Table 1 shows details of the total operation count for the two methods. Based on the information in table 1, the floating point operations (FLOP) count *per iteration* for the two methods can be expressed as follows

$$FLOP_{(EBE)} = (12N^{\text{eqn}} + 2 \cdot N^e N^{\text{dof}}) \quad (27)$$

$$FLOP_{(EQBYEQ)} = N^{\text{eqn}}(11 + 2 \cdot 4 \cdot N^{\text{dof}}) \quad (28)$$

where N^{dof} is the number of DOF per element, N^{eqn} , N^e are the number of equations and number of elements respectively. Table 2 shows a numerical comparison of the FLOP count. It can be seen that the operation count for the proposed method is much higher. This could have the tendency to make the algorithm more coarse grained and the computational time can be effectively split by increasing the number of processors.

5 Results and Discussions

5.1 Machine used

- **SHAKTI:** SHAKTI is a 32 node Beowulf cluster each with 256 MB of local memory for a total addressable space of 8 GB. The cluster is interconnected through a 100 Mbps switched fast Ethernet network. All the work in SHAKTI was developed in FORTRAN 90 and Portland Group's High Performance Fortran (HPF). Results generated using SHAKTI (Figures 3 and Tables 4, 5, 6) are shown to demonstrate the initial viability of the method.

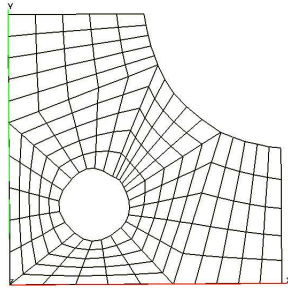


Figure 2: Sample Mesh used in the Study

Numerical experiments have been performed on a two-dimensional mesh to demonstrate the performance of the EQBYEQ method with respect to the size of the problems that can be solved and the time taken to do it. Figure 2 shows the sample mesh used in this study. The left and bottom edges are constrained from moving in the plane and a vertical load is applied at the top edge. The automatic, tied in mesh generator is capable of generating extremely large meshes for the given geometric mesh. A comparison of the memory requirements and performance of the two algorithms is presented below.

5.2 Size Comparison:

The stiffness matrix sizes for the EBE and the EQBYEQ method are shown in Table 3. The last column shows the actual memory requirement per processor. It can be seen that the EQBYEQ method requires a little less than twice the total memory than the EBE method, but this memory is distributed amongst the processors. Hence, the actual memory required per processor is only about a quarter for an eight processor system and this goes down even more for a bigger cluster.

5.3 Performance Comparison:

A comparison of the times taken by the two methods in a uniprocessor (SHAKTI) setup is shown in Table 4. It can be seen that the time taken to setup the stiffness matrix is much less for the EBE method due to the decreased number of computations. However, the time for solution for the EQBYEQ method is lesser. This is probably due to the two conflicting loop indices in the EBE method. Table 5 shows a comparison of the parallel times taken by the two methods. It is seen that the performance of the parallelized EBE method deteriorates when the number of processors is increased. This can be attributed to excessive communications due to the presence of the two conflicting loop indices. For the EQBYEQ method both the CPU and real time decreases by increasing the number of processors up to a certain stage after which it shows an increase, due to communication time increase overcoming the CPU time decrease.

As outlined earlier, both the generation of the stiffness matrix and the iterative solution parts form an integral part of the EQBYEQ algorithm. Table 6 compares the scalability of the two parts by comparing the two speedups. It can be seen that the generation of the stiffness matrix part is highly scalable as this part has no conflicting alignments and is computation intensive. However, the total solution time scales well up to 8

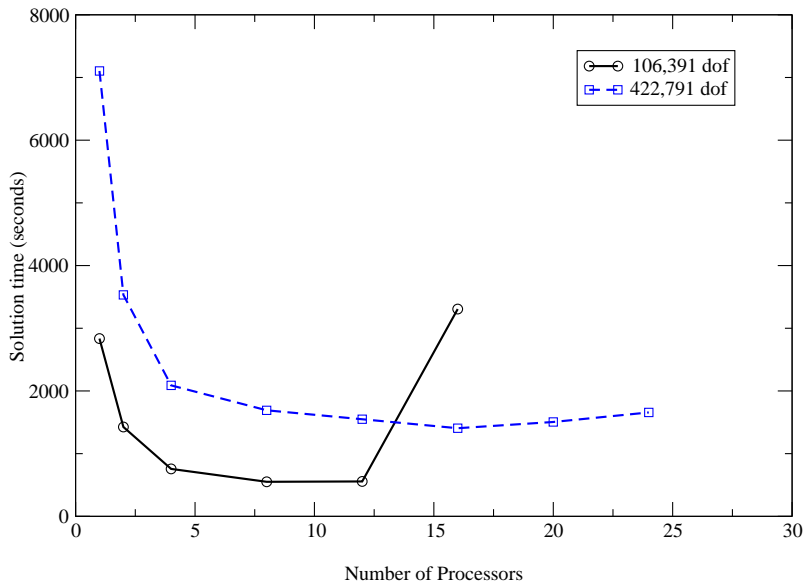


Figure 3: Variation of Solution time with processors

processors after which it shows a deterioration in performance.

Figure 3 shows a study on the scalability of the EQBYEQ algorithm using SHAKTI. It can be observed that the total solution time decreases as the number of processors increases and after a limit the time actually increases. This is because the increase in communication time over weighs the decrease in computational time after a certain point. Figure 3 shows the performance comparison for very large problems using SHAKTI. Another observation that can be made from Figure 3 is that the scalability of the problem improves with its size. This is deduced from the observation that the least time for the 100k problem is obtained by using 12 processors while for a 420k problem it is about 20 processors.

6 Conclusions

A new equation by equation strategy is proposed for the effective parallelization of the conjugate gradient, specifically the matrix-vector product, iterations for a *distributed memory* environment. This algorithm ties in the mesh generation stage to the FE code, by generating a mapping variable which is later used by the FE analysis code. Preliminary studies on memory requirement and performance comparison have been presented in this paper.

The major conclusions of this work can be stated as follows:

- The proposed method allows the solution of problems *much larger* than what a standard EBE method would allow. This is due to the equation-by-equation distribution of the stiffness matrix elements, proposed in this paper, among the processors. While this method requires more memory and computations in a uniprocessor setup, it is better suited to a distributed memory environment as the memory requirement is now distributed and the computation time divided with minimal communications. This algorithm requires only twenty percent of the memory required for a EBE algorithm for a eight processor cluster and this goes down even more as the cluster size increases.

- While the CG iterative scheme is effectively parallelized it cannot be completely communication free. The effect of this can be seen in the scalability studies reported here. The problem scales well with increasing number of processors but reaches a plateau for a certain number of processors after which the solution time actually increases.
- The scalability of the problem depends on its size. As the size increases the number of processors required to its fastest solution increases.

Acknowledgements

The authors would like to acknowledge Dr. Vibhas Aravamathan of Louisiana State University for the usage of the Beowulf cluster used in this study.

References

- [1] E. Barragy, G. Carey and R. van de Geijn, Performance and Scalability of Finite Element Analysis for Distributed Parallel Computation, *Journal of Parallel and Distributed Computing* **21** (1994) 202-212.
- [2] K. Wang and J. Bruch Jr, A Highly Efficient Iterative Parallel Computational Method for FE Systems, *Engineering Computations* **10** (1993) 195-304.
- [3] I. Smith, A General Purpose System for FE Analyses in Parallel, *Engineering Computations* **17** (2000) 75-91.
- [4] R. King and V. Sonnad, Implementation of an Element-by-Element Solution Algorithm for the Finite Element Methods on a Coarse-grained Parallel Computer, *Computational Methods in Applied Mechanics Engineering* **65** (1987) 47-59.
- [5] J. Shahid, S. Hutchinson and H. Moffat, Parallel Performance of a Preconditioned CG Solver for Unstructured Finite Element Applications *Proceedings of the Colorado Conference on Iterative Methods*, (1994).
- [6] T. Horie and H. Kuramae, Possibilities of Workstation Clusters for Parallel Finite Element Analysis, *Microcomputers in Civil Engineering* **12** (1997) 129-139.
- [7] D. Hodgson and P. Jimack A Domain Decomposition Preconditioner for a Parallel Finite Element Solver on Distributed Unstructured Grids, *Parallel Computing* **23** (1997) 1157.
- [8] G. Yugawa, N. Soneda and S. Yoshimura, A Large Scale Finite Element Analysis Using Domain Decomposition Method on a Parallel Computer, *Computers and Structures* **38** (1991) 615-625.
- [9] W. Xicheng, P. Baggio and B. Schrefler, A Multi-level Frontal Algorithm for Finite Element Analysis and its Implementation on Parallel Computation, *Engineering Computations* **16** (1999) 406-437.
- [10] M. Al-Nasra and D. Nguyen, An Algorithm for Domain Decomposition in Finite Element Analysis, *Computers and Structures* **29** (1991) 277-289.

- [11] C. Farhat, A Simple and Efficient Automatic FEM Domain Decomposer, *Computers and Structures* **23** (1988) 579-602.
- [12] J. Oden and G. Carey, *Element Mathematical Aspects, Vol. IV*, (Prentice Hall, Englewood Cliffs, NJ, 1977).
- [13] H. Adeli, High-Performance Computing for Large-Scale Analysis, Optimization and Control, *Journal of Aerospace Engineering* **13** (2000).
- [14] R. Cook, D. Malkus and M. Plesha, *Concepts and Applications of Finite Element Analysis* (John Wiley and Sons, third edition, 1989).
- [15] T. J. R. Hughes, *The Finite Element Method: Linear Statics and Dynamic Finite Element Analysis* Dover Publications, 2000
- [16] J. N. Reddy, *An Introduction to the Finite Element Method* Tata McGraw Hill, second edition, 2003

Table 1: Operation Count Comparison for the Two Methods

Equation	EQBYEQ	EBE
$d = \text{diag_precon} * r$	N^{eqn}	N^{eqn}
$\text{rho1} = \text{dot_product}(r, d)$	$N^{\text{eqn}} + N^{\text{eqn}}$	$N^{\text{eqn}} + N^{\text{eqn}}$
$p = d + \beta * p$ (SAXPY)	$N^{\text{eqn}} + N^{\text{eqn}}$	$N^{\text{eqn}} + N^{\text{eqn}}$
$u = [K]p$ (matrix-vector product)	$N^{\text{eqn}} * (4 * (N^{\text{dof}} + N^{\text{dof}}) + 4)$	$N^{\text{e}} * (3 * N^{\text{dof}}) + N^{\text{eqn}}$
$\text{down} = \text{dot_product}(p, u)$	$N^{\text{eqn}} + N^{\text{eqn}}$	$N^{\text{eqn}} + N^{\text{eqn}}$
$x = x + p * \alpha$ (SAXPY)	$N^{\text{eqn}} + N^{\text{eqn}}$	$N^{\text{eqn}} + N^{\text{eqn}}$
$r = r - u * \alpha$ (SAXPY)	$N^{\text{eqn}} + N^{\text{eqn}}$	$N^{\text{eqn}} + N^{\text{eqn}}$

Table 2: Numerical Comparison of Operation Count (per iteration)

No. of Equations	No. of Elements	EQBYEQ (Oper. Count) $\times 10^6$	EBE (Oper. Count) $\times 10^6$	Ratio
1,181	175	0.17	0.022	7.5
106,391	17,500	15.2	2.12	7.17
422,791	70,000	60.46	5.07	11.92
1,685,591	280,000	241.04	33.66	7.16

Table 3: Memory Requirements for the Two Methods

No. of Equations	No. of Elements	Memory Required by		
		EBE MB/processor	EQBYEQ (MB)	EQBYEQ MB/processor (8 nodes)
106,391	17,500	17 MB	26 MB	3.25
422,791	70,000	68 MB	103 MB	13 MB
1,685,591	280,000	273 MB	411 MB	51 MB

Table 4: Solution times for the two methods (Uniprocessor Comparison)

No. of Equations	EBE		EQBYEQ	
	Stiffness	Solution	Stiffness	Solution
	seconds	seconds	seconds	seconds
1,181	0.16	16.63	3.4	6.8
26,941	16.35	33145	83.68	667
106,391			356	2835
422,791			1314	7014

Table 5: Total Solution times for the two methods (Multiprocessor comparison)(1181 equations)

No. of Processors	EBE seconds	EQBYEQ seconds
1	16.57	6.8
2	224	4.14
4	878	2.91
8		2.84
12		3.51
16		4.33

Table 6: Stiffness and Solution :: times and speedups for EQBYEQ (26000 equations)

No. of Processors	Time (seconds)		Speedup	
	Stiffness	Total	Stiffness	Total
1	83.38	666.0		
2	41.85	344.2	2.0	1.93
4	20.57	200.9	4.0	3.35
8	10.25	142.4	8.7	4.7
12	6.82	150.47	12.3	4.67
16	5.11	160.53	16.3	4.14