MPI-based parallel finite element approaches for implicit nonlinear dynamic analysis employing sparse PCG solvers

A. Rama Mohan Rao*

Structural Engineering Research Centre CSIR Campus, Taramani, Chennai 600113, India

Abstract

This paper presents three formulations combining domain decomposition based finite element method with linear preconditioned conjugate gradient (LPCG) technique for solving large-scale problems in structural mechanics on parallel processing machines. In the first formulation called the Global Interface Formulation (GIF), the PCG algorithm is applied on the assembled interface stiffness coefficient matrices of all submeshes. The second formulation called Local Submesh Formulation (LSF) operates on the local unassembled submesh matrices and the preconditioner is constructed using the local submesh information. In the third formulation called Local Interface Formulation (LIF), the sparse PCG algorithm is formulated using the unassembled local schur complement matrices of submeshes. Both diagonal and incomplete Cholesky preconditioners have been employed. These domain decomposition based PCG algorithms have been implemented within a finite element code for nonlinear implicit transient dynamic analysis. Time integration is performed using Newmark-β constant average acceleration method. The parallel finite element code uses an MPI-based message passing approach to provide portable parallel execution on shared, distributed and distributed shared memory computers. Numerical experiments have been conducted on PARAM-10000, an Indian parallel supercomputer to evaluate the performance of the implicit parallel nonlinear finite element code employing the three proposed PCG formulations. Numerical studies indicate that the proposed parallel PCG formulations are highly adaptive for parallel computing and superior in performance when compared to the conventional domain decomposition algorithm with parallel direct solver. The LSF formulation, which is amenable for efficient implementation of communications by way of overlapping with computations found to be superior in performance compared to other two PCG formulations.

Keywords: Parallel finite elements; PCG formulations; Sparse matrices; Message passing; Domain decomposition

1. Introduction

The computational complexities of transient dynamic nonlinear finite element analysis of structures may become very large for many practical engineering problems. Parallel and distributed processing, which permits the engineer to undertake the finite element analysis in a considerably shorter time, is therefore of increasing significance.

For structures subjected to quasi-static and moderate loading rates, an implicit formulation of the finite element method often provides the only viable approach to resolve the complete response history. Repeated solution of the linearised equilibrium equations for each time step (within an incremental, iterative Newton strategy to solve nonlinear equations), dominates the computational cost of implicit methods.

The choice of an appropriate parallel solver thus becomes crucial and often determines the feasibility of proposed analyses. Sparse direct methods factorize the coefficient matrix to provide, within numerical error, an exact solution for the equations in a fixed number of operations [1]. Although these methods compute reliable solutions for nearly any nonsingular matrix, operation counts and memory requirements increase rapidly with the problem size. Several current projects focus on the development of sparse direct solvers for distributed (and shared) memory machines using explicit message passing. However, iterative methods generally have better scalability for parallel execution and are well suited for distributed as well as shared memory machines and requires far less
memory [2]. Unfortunately, these methods do not always converge within a reasonable amount of time frame and, for systems with large condition numbers, may not converge at all. The choice of an appropriate preconditioner often accelerates convergence. The linear preconditioned conjugate gradient (LPCG) algorithm represents one of the most commonly used iterative methods [3]. Other alternatives under intensive search include multigrid methods [4–6] and domain decomposition solvers, such as the finite element tearing and interconnecting (FETI) method [7–9]. The multigrid approach uses multiple, nested meshes for a model to smooth errors in the solution, while the FETI method decomposes a mesh into domains then applies direct solvers within domains, and a modified LPCG algorithm to enforce continuity across domain boundaries. Implementations of both approaches show good scalability in parallel execution [9]. A wide range of other parallel solution algorithms using domain decomposition techniques has been reported in the literature [10–12].

This paper presents three formulations combining domain decomposition based finite element method with LPCG algorithm for solving large-scale nonlinear dynamic analysis problems on coarse grain parallel computers. Three different parallel implementations for LPCG algorithm are presented. They are termed as Global Interface Formulation (GIF), Local Submesh Formulation (LSF) and Local Interface Formulation (LIF). Both diagonal as well as incomplete Cholesky preconditioners have been employed in the present work. The parallel algorithms discussed in this paper have been implemented using portable Message Passing Interface (MPI) software development environment [13].

2. Implicit finite element analysis

The implicit finite element method for nonlinear analyses of structures derives from the principle of virtual work expressed on the current configuration, denoted as \( n + 1 \), as

\[
\int_{V_{n+1}} \delta \varepsilon \sigma_{n+1} \, dV + \int_{V_{n+1}} \rho \dot{u}_{n+1} \frac{\partial^2 V_{n+1}}{\partial t^2} \, dV = \int_{S_{n+1}} T_{n+1} \partial u_{n+1} \, dS = 0
\]

where \( \sigma_{n+1} \) denotes the stress, \( T_{n+1} \) defines the tractions applied to the surface of the model at \( n + 1 \), \( u_{n+1} \) and \( \dot{u}_{n+1} \) define the real and virtual displacement fields, respectively, \( \delta \varepsilon \) represents the symmetric rate of the virtual deformation tensor relative to the current configuration, and \( \rho \) specifies the material mass density. \( V_{n+1} \) denotes the volume at \( n + 1 \), and \( S_{n+1} \) specifies the external surface at \( n + 1 \).

Nonlinear effects in Eq. (1) included here arise from the material constitutive response and, the strain displacement relations. Application of standard finite element procedures [14,15] yields a set of nonlinear equilibrium equations at \( n + 1 \) and are of the form

\[
r_{n+1} = f_{n+1} - p_{n+1} - Ma_{n+1}
\]

where \( a_{n+1} \) denotes the nodal accelerations, \( M \) is the mass matrix for the structure and \( f_{n+1} \) defines the effective nodal forces due to applied force. \( p_{n+1} \) denotes the forces exerted on element nodes consistent with the stresses and deformation and is given by

\[
p_{n+1} = \frac{1}{\text{NEL}} \sum_{j=1}^{\text{NEL}} \int_{V_{j}} \delta \varepsilon \sigma_{n+1} \, dV_{j}
\]

\[
= \sum_{j=1}^{\text{NEL}} [B_{n+1}]^T \sigma_{n+1} \, dV_{j}
\]

where the summation symbol implies a standard assembly process. \( r_{n+1} \) defines the unbalanced nodal forces that vanish when the nonlinear solution process resolves correct values for the nodal displacements and accelerations at \( n + 1 \).

For time integration of nonlinear system of equations given by Eq. (2), Newmark’s constant average acceleration technique has been adopted with Newmark parameters \( \beta \) as 1/4 and \( \gamma \) as 1/2, which yields

\[
r_{n+1} = f_{n+1} - p_{n+1} - \frac{1}{\beta \Delta t^2} M \dot{u}_{n+1}
\]

where \( \Delta t \) defines the analysis time step and \( f_{n+1} \) represents the effective nodal forces as modified by the terms from the Newmark integrator [16]. Application of a Newton Raphson procedure to eliminate the residual nodal forces at \( n + 1 \) yields the following system of linear equations at Newton iteration \( i \) to advance the solution from \( n \) to \( n + 1 \)

\[
(K_T + \frac{1}{\beta \Delta t^2} M) \Delta \tilde{u}_{n+1} = K_T \Delta \tilde{u}_{n+1}^{i-1}
\]

\[
= f_{n+1} - p_{n+1} - \frac{1}{\beta \Delta t^2} M \Delta u_{n+1}^{i-1}
\]

The right hand side represents the force imbalance (residual) from previous iteration, where \( \Delta \tilde{u}_{n+1}^{i-1} \) is the displacement increment for step \( n + 1 \) as of the previous iteration. On the left hand side, \( K_T \) defines the dynamic (consistent) tangent stiffness matrix for the current iteration, and \( \Delta \tilde{u}_{n+1}^{i} \) denotes the correction to the displacement increment over the step within the linear approximation that brings the residual to zero. Fig. 1 summarizes the computational process to advance the solution over a time step in nonlinear dynamic analysis.

Computation of equivalent nodal loads for the step precedes the Newton iterations, and includes contributions
from applied loads and inertial forces. The norm of the equivalent nodal loads provides one of several measures to assess convergence of the global solution. During each Newton iteration, the material constitutive relations provide the (consistent) tangent moduli to support computation of updated element stiffness. The tangent stiffness for the structure and the updated residual form a linear system of equations, which when solved yields a new correction to the displacement increment. Solution of the linear system proceeds using a linear preconditioned conjugate gradient solver (LPCG) with diagonal and incomplete Cholesky preconditioning. The corrected displacement increment enables computation of the updated strains, stresses and the internal force vector. The revised internal force vector and equivalent nodal loads define a new residual. When residuals meet the convergence criteria, the newly updated stresses, strains and displacements define the solution at \( n+1 \). Otherwise, the solution requires another Newton iteration.

3. Sparse storage schemes

In the recent past, the research for sparse storage schemes has become increasingly active and this trend is likely to accentuate because of the growing need to design efficient sparse matrix algorithms for modern parallel computers. There are varieties of sparse storage schemes proposed in the literature. The purpose of each of these schemes is to gain efficiency both in terms of memory utilisation and arithmetic operations. As a result, many different ways of storing sparse matrices have been devised to take advantage for the structure of the matrices or the specificity of the problem from which they arise. Some of the sparse storage schemes are Harwell–Boeing (HB) format, Compressed Row Storage (CRS) format, Compressed Sparse Column (CSC) format, Diagonal format, coordinate format, linked list storage format, etc. Description on these various sparse formats is not attempted here. However, these are available in Duff et al. [1]. The Compressed Row Storage (CRS) scheme is popular among the finite element community and same has been employed in the present work to store the global matrices.

In CRS format, a global stiffness matrix of size \( n \) is stored by defining three vectors \( A, JA, \) and \( JD \). \( A \) is a float point vector of length \( ne \), containing \( nne \) non-zero coefficients of the global stiffness matrix stored contiguously. The coefficients are stored row-wise, starting with the diagonal of each row. An integer vector, \( JD \) of length \( n+1 \) stores the absolute relative starting position of each row of the global stiffness matrix. Another integer vector \( JA \) of length \( ne \) contains the column position of each non-zero stiffness coefficient matrix stored in the vector \( A \).

4. Domain decomposition technique

In order to impart parallelism into the implicit finite element method for nonlinear dynamic analysis, domain decomposition technique has been employed in the present work. In the domain decomposition based parallel processing techniques, an efficient domain partitioning algorithm (usually built within a preprocessor) is employed to partition the finite element mesh into desired number of submeshes and assigns one submesh to each processor. Each processor performs calculations for ‘local’ elements, i.e. those within the domain it owns and conducts these calculations concurrently with the other processors. Performance of the domain decomposition based parallel algorithms strongly depends on the efficiency of the domain partitioning algorithm employed to partition the finite element mesh. A high quality domain partitioning algorithm balances the computational load among processors, while simultaneously minimising the communication costs between processors. Recent research efforts in graph partitioning have yielded several high quality domain partitioning algorithms which include METIS [17–20], JOSTLE [21–23], CHACO [24,25], etc. In the present work, a domain partitioning algorithm developed by the author [26] employing micro genetic algorithm has been employed for partitioning the finite element meshes. The GA based partitioning algorithm works on the dual graph associated with the finite element mesh and recursively partitions the graph into desired number of subgraphs by using graph slicing technique. Conventional form of parallel domain decomposition algorithm for implicit nonlinear dynamic analysis is given in Fig. 2.

This paper presents three domain decomposition formulations combining with LPCG solvers for nonlinear dynamic analysis with implicit time integration technique. In the first formulation, the interface stiffness coefficients are assembled and the preconditioner is constructed using the global interface stiffness coefficient matrix and the parallel PCG solver is applied for the interface solution. This is termed as Global Interface Formulation (GIF). In the second formulation, the schur complement matrices of each submesh are neither computed nor the assembly of local
1. Input and setup basic analysis data
2. In each processor:
   i. Compute internal force vector
   ii. Compute effective global stiffness matrix and load vector
   iii. Condense internal degrees of freedom
3. Communication:
   Send relevant parts of condensed stiffness and load vectors to adjacent processors
   Receive relevant parts of data of condensed stiffness and load vectors from adjacent processors
   Assemble the relevant columns of the global interface stiffness coefficient matrix and force vector
4. Use parallel solver to solve boundary degrees of freedom
5. In each processor:
   i. Solve internal degrees of freedom
   ii. Compute velocities and accelerations
   iii. Compute stresses and strains
6. Check equilibrium for nonlinear analysis
   i. Assemble internal force and load vector
   ii. Compute unbalanced force vector
   iii. Assemble unbalanced force norm
   iv. If converged go to (7)
   v. Solve for displacements (same as 4, 5 (i))
   vi. Compute velocities and accelerations (same as 5(ii))
   vii. Recover stresses and strains (same as 5(iii)) and go to (6(i))
7. Go to step 2 for the next time step

Fig. 2. Parallel algorithm for nonlinear dynamic analysis (conventional form).

The matrix is partitioned into as many row-wise segments as the number of processors. These partitions are carried out in such a way that the number of non-zero elements in each segment of the matrix is approximately equal. This distribution of contiguous rows allocated to each processor makes SAXPY operations particularly efficient and produces excellent load balancing. Parallel implementation of various components of the algorithm is described below.

Since MPI implementation details are also discussed in this paper, some of the MPI function calls are specified, wherever it is essential to improve clarity on the implementation details of the proposed formulations. The detailed explanation on each of the referred MPI function

\begin{verbatim}
  i. u_i = u_0, r_i = f and w_i = z_0
     For i = 0, 1, 2, ..., MITER
       Do
       ii. z_i = P^{-1} r_i
           i^* Preconditioning step
       iii. a_i = (r_j) T z_i
           i^* vector operations
       iv. if (i = 0) w_i = z_i
           else
             w_i = z_i + (a_i z_i) w_i
             i^* SAXPY operation
       v. z_i = K w_i
           i^* Matrix vector product
       vi. a_i = (w_i) T z_i
           i^* vector operations
       vii. u_i = u_i + (a_i b_i) w_i
           i^* SAXPY operation
       viii. r_i = r_i - (a_i) b_i z_i
           i^* SAXPY operation
       ix. if (a_i / c_i) < e, then exit
       End for
\end{verbatim}

Fig. 3. PCG algorithm for Global Interface formulation.
calls is also given in the paper whereever they are mentioned.

### 4.1.1. Global sparse matrix assembly

As already mentioned, a sparse storage scheme called Compressed Row Storage (CRS) format has been employed to store the global matrices. First, the sequential method proposed for assembling the global sparse matrices in CRS format using the element matrices and connectivity details is discussed. Later the parallel implementation of the sparse assembly procedure is presented.

The interface stiffness coefficient matrix of any typical submesh obtained by condensing out the internal degrees of freedom can be treated as a super-element matrix with the number of nodes equivalent to the border nodes of the submesh. Hence, there is hardly any difference in assembling the interface stiffness coefficient matrix (super element matrix) or the element stiffness matrices.

The basic steps involved in the sparse matrix assembly can be described as follows:

i. Construct a square integer matrix ND of size NXN, where \( N \) is the total active degrees of freedom of the whole mesh. The ND matrix contains ‘1’ at the location of non-zero coefficients of the global stiffness matrix and ‘0’ at the remaining locations. These zero’s and one’s in the ND array reflects the zero and nonzero coefficients of the global assembled stiffness matrix. This matrix can be constructed by using the degrees of freedom of the individual elements. This can be accomplished by filling the ND matrix with ‘1’ for the location of non-zero coefficient of the global stiffness matrix. The location of the non-zero coefficient can be obtained by using the global degrees of freedom of the individual elements. Fig. 4 shows the portion of the ND matrix that has been updated for a typical element \( e \), whose global degrees of freedom are also indicated in the figure. It may be noted that only upper triangular

\[
\begin{bmatrix}
1 & \cdots & p & \cdots & q & \cdots & r & \cdots & s \\
p & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
q & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
r & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
s & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{bmatrix}
\]

Fig. 4. Formation of ND array in sparse matrix assembly for element ‘e’ with global locations as \{p,q,r,s,…\}.

ii. Using this ND matrix, the JA array containing the column indices and the pointer array, which contains the addresses in the global stiffness matrix, \( K \) and \( JA \) at the beginning of each row can be formed. This can be accomplished by scanning through the elements of the ND array.

iii. The assembly of the element stiffness matrices into global sparse matrix can be carried out by making use of the active global degrees of freedom of the elements and their connectivity. The appropriate location of any element stiffness coefficient in the global sparse matrix can be identified with the help of JA array and the pointer array formed in the previous step.

Formation of ND matrix needs considerable storage especially for large size problems. In view of this, in the present work, the ND matrix is not formed. Instead, the element nodal connectivity data and their corresponding global degrees of freedom are scanned to form A, JA and JD arrays. It can easily be verified that the sparse assembly of global stiffness matrix can be performed efficiently by assembling node wise (rather active degrees of freedom wise) rather than element wise. In this approach, each row of the sparse global matrix corresponding to a nodal active degrees of freedom (d.o.f) is formed assembling all the corresponding contributions from elements connected to that particular d.o.f. The JA array is updated with the entries related to non-zero column indices of the current row of the global matrix. The row is then condensed removing all non-zero entries. The JD array is then updated before proceeding to the assembly of next row in the sparse matrix corresponding to the next active degrees of freedom.

Even though various basic steps given earlier in this section related to formation of ND matrix, JD and JA arrays, are not strictly followed in our approach for sequential sparse assembly, this explanation is in fact necessary for our subsequent discussion on parallel sparse matrix assembly.

### 4.1.2. Parallel sparse matrix assembly

In order to perform parallel operation in the PCG algorithm, the sparse global matrix is partitioned as row wise segments and distributed to the processors. In view of this, the sparse matrix assembly of only the allocated row segments need to be carried out in each processor.

For this purpose, the interface stiffness coefficient matrices and the element connectivity details are duplicated in each processor. These are stored in the temporary buffers. Making use of the connectivity information, the ND array is formed in each processor as already explained earlier. Using the ND array information, the JA array and the pointer array can be formed.

Once the ND array is formed, the number of row segments allocated to each processor can be determined
with the help of ND array. The number of rows in each segment is fixed in such a way that the nonzero elements in each segment of the global sparse matrix are approximately equal. Since ‘1’s in ND array reflects the nonzero coefficients, each processor can identify the appropriate rows allocated to it.

The assembly is carried out in each processor only in the row segments assigned for the particular processor. This is accomplished by placing the super element stiffness matrix elements in the appropriate locations of the rows in the sparse matrix with the help of the JA array and the pointer array.

The parallel assembly requires the interface stiffness coefficient matrices and the connectivity matrices to be duplicated in each processor. This requires considerable interprocessor communication and additional temporary storage. Alternatively, parallel assembly can be carried out by only considering the local super element matrix for storage. Alternatively, parallel assembly can be carried out in a temporary space and once the global sparse matrix will be of manageable size. This requires considerable conditioning require solving the system.

4.1.5. Matrix-vector product

The assemble of the global stiffness matrix is the unassembled submesh stiffness matrix in global submesh-by-submesh form. The submesh stiffness matrices \( K^m \) are neither condensed out to obtain interface stiffness coefficient matrix nor assembled to form global stiffness matrix of the entire mesh. In this formulation, a local preconditioner based on the submesh stiffness matrix is constructed and the preconditioned conjugate gradient method is applied on the distributed stiffness matrices and local vectors. The main feature of the algorithm is it’s inherent parallelism as the formulation of preconditioner and the dominant matrix vector products are computed locally using the assigned submesh matrices within the processor.

4.2. Local submesh formulation (LSF)

In this formulation, PCG algorithm is implemented on the global stiffness matrix in submesh-by-submesh form. The submesh stiffness matrices \( K^m \) are neither condensed out to obtain interface stiffness coefficient matrix nor assembled to form global stiffness matrix of the entire mesh. In this formulation, a local preconditioner based on the submesh stiffness matrix is constructed and the preconditioned conjugate gradient method is applied on the distributed stiffness matrices and local vectors. The main feature of the algorithm is it’s inherent parallelism as the formulation of preconditioner and the dominant matrix vector products are computed locally using the assigned submesh matrices within the processor.

4.1.6. Preconditioning

Here, both diagonal as well as incomplete Cholesky preconditioners have been employed. The diagonal preconditioning require solving the system \( z_i = p^{-1}r_i \). Solving the preceding system is equivalent to dividing each element of \( r \) by a diagonal entry of the corresponding row of \( A \). Since the rows of the stiffness matrix and the corresponding elements of the vectors reside in the same processors, this operation does not require any inter-processor communications. The incomplete Cholesky preconditioner is constructed using optimized row-wise implementation proposed by Bitoulas and Papadrakakis [27].
submesh quantities from all incident elements to the global system. Similarly, given an assembled vector \( x \), the operation \( Cx \) denotes distributing the quantity from the global system to local submeshes. Hence, the operation \( CC'x' \) represents the procedure of assembling the element quantities to the global system and broadcasting the assembled vector back to the submesh level. When the local coordinates of the submeshes are oriented in the same directions as the global coordinates, \( C \) is a Boolean matrix populated with zero and unit values. This allows the operation \( CC'x' \) to be accomplished by adding to \( x' \) those contributed from the nearest-neighbour elements.

In the Local Submesh Formulation, both the matrices and vectors are stored on the processor nodes in a ‘distributed form’ or in an ‘accumulated form’. An accumulated matrix or vector contains full entries for both internal and border nodes. In a distributed matrix, the submesh matrix or vector contains entries assembled from contribution of elements belonging to that particular submesh. Entries in the distributed matrix contain full values for internal nodes and only partial values for boundary nodes.

It can easily be shown that, in general, matrix-vector product can not be performed with accumulated arrays. In view of this, the global stiffness matrix and preconditioning matrix is stored in distributed form. The distributed submesh stiffness matrix is obtained automatically by assembling element stiffness matrices of elements belonging to that particular submesh. Similarly, the preconditioner is constructed using the distributed (local) submesh stiffness matrix. However, for vectors, both distributed as well as accumulated are required during the vector inner products and matrix vector products, which is evident from the following example.

Consider the computation of an inner product of two vectors \( x \) and \( y \):

\[
\alpha = x^T y = \sum_i x_i y_i
\]  

(6)

Where subscript \( i \) denote the \( i \)th entry of the vector. If vectors \( x \) and \( y \) are distributed among the processors as \( x_j \) and \( y_j \) (\( j \) is the processor number) then the inner product can be computed as:

\[
\alpha = \sum_j (x^T y)_j = \sum_j \sum_i (x_i y_j)_j
\]  

(7)

If both vectors are stored as accumulated then quantities \( x_i y_j \) for interface nodes repeat several times during the inner product computation and the result is incorrect. One way of handling this is to divide \( x_i y_j \) for interface nodes by a multiplicity factor of this node. This multiplicity factor of an interface node shows how many times the same node is repeated in all the submeshes. The other simple approach is to perform correctly the inner product with one vector as accumulated and other as distributed. Similarly, in the matrix vector product, the correct result can be obtained by multiplying a distributed matrix by an accumulated vector. The resulting vector will be a distributed vector.

As already shown earlier, the PCG iteration procedure, require matrix vector products and vector inner products. Hence, it is necessary to transform vector from a distributed form to an accumulated form. This can be accomplished by using inter processor communication of data. For example, to transform a vector from distributed form \( x \), to accumulated form \( \bar{x} \), each processor performs the following steps:

- Collect the boundary entries of \( x \) into array segments \( x^b \) corresponding to external submesh boundaries.
- Send array \( x^b \) to neighbouring submeshes and receive array segments corresponding to the resident submesh interface entries \( \bar{x}^b \) from neighbouring submeshes.
- Assemble external interface entries \( \bar{x}^b \) to the distributed vector \( \bar{x} = x + \bar{x}^b \), where \( \bar{x} \) is the accumulated vector.

The send and receive operations are shown in Fig. 5 for upper left submesh. The global connectivity matrix (matrix \( C \)) as explained earlier can be used to compute the accumulated vectors in the submesh. This has been implemented as given below:

- Compute \( x^b = C_m x_m^b \) in each submesh, \( m \), where \( x_m^b \) is the distributed value corresponding to the boundary nodes. \( x^b \) is submesh boundary quantities in global coordinates. Send these values to the neighbouring processes which are sharing these nodes
- Receive \( x^b \) from neighbouring processors and transform to local coordinates. \( \bar{x}^b = C_m x_m^b \). Add external interface entries to the distributed vector \( x : \bar{x} = x + \bar{x}^b \).

With this discussion, the PCG algorithm for Local Submesh Formulation (LSF) on any typical submesh can be described as given in Fig. 6.
4.2.1. Local submesh formulation with overlapped communications

It can be observed from the LSF algorithm presented here, that the processors need to communicate twice in each iteration, to send the resident boundary nodal vectors and also to receive the boundary nodal vectors from the neighbouring processors in order to form the accumulated vectors. As already discussed earlier, in the LSF formulation, the PCG algorithm operates on the full submesh matrices and not on the condensed submesh matrices. In any of these partitioned submeshes, it is reasonable to assume that the internal nodes are large when compared to the interface nodes. This paves way to optimise the communications by overlapping with part of computations. This can be accomplished using non-blocking communication calls. Since internal and boundary nodes are separated, communication of data related to boundary nodes can be overlapped with the computations for internal nodes. First the interface entries are collected and ‘send and receive’ operations are initiated. Then matrix and vector computations corresponding to the internal nodes are performed. After waiting for the receipt of the submesh interface vectors from neighbouring processors, the vectors are assembled to form the accumulated vector. Another possibility to optimise the communications is by overlapping the communications with the solution update

\[ u_{i+1} = u_i + \alpha_i \beta_i \tilde{w}_i \]

as \( u \) is not used in any computation.

These operations are implemented by non-blocking send and receive calls like `MPI_Isend()` and `MPI_Irecv()` function calls respectively of MPI library, where ‘I’ stands for ‘incomplete’. These are non-blocking functions and they initiate the communication and return immediately, so that the program may continue its work on the core of the submesh. Temporary send and receive buffers are used in order to avoid memory conflicts between the main program, (which performs the computations and provides interface entries are collected and ‘send and receive’ operations are initiated. Then matrix and vector computations corresponding to the internal nodes are performed. After waiting for the receipt of the submesh interface vectors from neighbouring processors, the vectors are assembled to form the accumulated vector. Another possibility to optimise the communications is by overlapping the communications with the solution update

\[ u_{i+1} = u_i + \alpha_i \beta_i \tilde{w}_i \]

as \( u \) is not used in any computation.

These operations are implemented by non-blocking send and receive calls like `MPI_Isend()` and `MPI_Irecv()` function calls respectively of MPI library, where ‘I’ stands for ‘incomplete’. These are non-blocking functions and they initiate the communication and return immediately, so that the program may continue its work on the core of the submesh. Temporary send and receive buffers are used in order to avoid memory conflicts between the main program, (which performs the computations and provides interface
nodal values to be transmitted to neighbouring processors) and the communication threads for sending and receiving data across processors. The send routine makes a copy of the interface nodal values of the submesh in a temporary send buffer space before calling MPI_Isend(). The background thread MPI_Isend() operates only on this temporary send buffer memory area. Similarly, the back ground thread launched by MPI_Irecv() collects the received values into another temporary receive buffer memory area which gets refreshed before every receive call. Even though the main program is finished with all the communications and computations in that particular iteration loop, the execution of the next iteration cannot be commenced unless the communications are completed. In order to ensure this, processors need to be synchronised and it is accomplished by using the MPI_Waitall() function call for the sends and a series of MPI_Waitany() function calls for receives. The synchronisation function calls for send and receive are different because, the threads launched by send routine need to simply check for the completion of send operation and no additional work need to be done. In contrast, the threads launched by receive routine, needs further work on receiving the data into the buffer. The data need to be added to the respective submesh interface nodal values. The MPI_Waitany() function call returns when a particular receive operation is completed and the data collected in the temporary receive buffer is added to the respective interface values of the local submesh and the buffer is refreshed.

The optimised LSF algorithm with overlapped communications is as shown in Fig. 7.

---

i. Construct the preconditioning matrix $P$
ii. Initialise the vectors $u_0 = d^T$, $r_0 = f$
iii. Send $r_i^b$, receive $r_i^{eb}$ from neighbouring processors
iv. From $\tilde{r}_i = r_i + r_i^{eb}$
v. Loop over iterations for $i = 0, 1, 2, \ldots$ MITER
vi. Compute $z_i^b = P^{-1}(\tilde{r}_i)^b$

vii. Initiate communications: Send $z_i^b$, Receive $z_i^{eb}$ from neighbouring processors

viii. Compute $z_i^{in} = P^{-1}(\tilde{r}_i)^{o_i}$
ix. Compute $\alpha_i = (\tilde{r}_i)^T z_i$

x. Reduce $\alpha_i$, $\alpha_i = \sum \alpha_i$
xii. Wait for receive $z_i^{eb}$

xiii. Compute $\tilde{z}_i = z_i + z_i^{eb}$
xiv. If $(i = 0)$ $\tilde{w}_i = \tilde{z}_i$
xv. Else $\tilde{w}_i = z_i + \frac{\alpha_i}{\alpha_i - \alpha_{i-1}} - \tilde{z}_{i-1}$
xvi. Initiate communications: Send $z_i^b$, Receive $z_i^{eb}$ from neighbouring processors

xvii. $z'_i = K(\tilde{w}_i)^b$
xviii. $\beta_i = (\tilde{w}_i)^T z_i$

xix. Reduce $\beta_i$, $\beta_i = \sum \beta_i$
xx. $u_{i+1} = u_i + (\alpha_i / \beta_i)\tilde{w}_i$
xxi. Wait for receive $z_i^{eb}$

xxii. Compute $\tilde{z}_i = z_i + z_i^{eb}$

xxiii. $\tilde{r}_{i+1} = \tilde{r}_i - (\alpha_i / \beta_i)\tilde{z}_i$
xxiv. If $(\alpha_i / \alpha_0) < \varepsilon$ then exit

End for

---

Fig. 7. Optimised parallel PCG algorithm for Local Submesh Formulation (LSF).
4.3. Local interface formulation (LIF)

The global interface formulation (GIF) discussed in the earlier sections is effective for structures with less number of interface nodes. For example, three dimensional tower or framed structures are likely to have narrow interfaces among submeshes with few interface nodes. Global Interface Formulation is ideal for this sort of problems where it is feasible to form global interface stiffness matrix with out much memory, computational/communication overheads, especially on massively parallel systems. Keeping this in view, a third formulation where the PCG algorithm operates only on local submesh interface matrices and force vectors. The preconditioners are constructed locally using the local schur complement matrix of the assigned submesh.

The interface stiffness coefficient matrix, $K_B$ and interface force vector, $f_B$ of any typical submesh $m$ in a partitioned finite element mesh can be written as:

$$K_B^m = K_{bi}^m - K_{bi}^m K_{ii}^{-1} K_{ib}^m$$  \hspace{1cm} (8)

$$f_B^m = f_{bi}^m - K_{bi}^m K_{ii}^{-1} f_i^m$$  \hspace{1cm} (9)

The global interface stiffness matrix can be obtained by assembling all the submesh interface stiffness matrices.

$$\left( \sum_{m=1}^{NS} B_b^m K_{bi}^m B_b^m - \sum_{m=1}^{NS} \tilde{K}_{bi}^m K_{ii}^{-1} \tilde{K}_{ib}^m \right) \{u_b\} = \sum_{m=1}^{NS} B_b^m f_B^m - \sum_{m=1}^{NS} \tilde{K}_{bi}^m K_{ii}^{-1} \{f_i^m\}$$  \hspace{1cm} (10)

where

$$\tilde{K}_{bi}^m = B_b^m K_{bi}^m \text{ and } \tilde{K}_{ib}^m = K_{ib}^m B_b^m$$

$K_{bi}^m$, $K_{ii}^m$, $K_{ib}^m$ are the internal and border (interface) stiffness coefficient matrices respectively of a submesh $m$. The $K_{bi}^m$, $K_{ib}^m$ correspond to the interaction of the internal and border degrees of freedom of any submesh $m$. $B_b^m$ is the interface connectivity matrix of the submesh $m$. The Eq. (10) can also be written as

$$S u_b = \hat{f}_b$$  \hspace{1cm} (11)

where

$$S = \sum_{m=1}^{NS} B_b^m s^m B_b^m$$  \hspace{1cm} (12)

$$s^m = K_{bb}^m - K_{bi}^m K_{ii}^{-1} K_{ib}^m$$  \hspace{1cm} (13)

$$\hat{f}_b = f_b - \sum_{m=1}^{NS} B_b^m K_{bi}^m K_{ii}^{-1} f_i^m$$  \hspace{1cm} (14)

Matrix $S$ is the schur complement of $K_{bb}$ in $K$ and each submatrix $S^m$ corresponds to a local static condensation operator or a local schur complement.

The diagonal preconditioner and incomplete factorisation preconditioners are constructed using the local schur complement matrix $S^m$. The parallel PCG implementation in the local interface formulation is presented in Fig. 8.

5. SPANDAN: Finite element code for parallel nonlinear dynamic analysis

The parallel algorithms discussed in this paper have been implemented in the finite element code SPANDAN (Software for PARallel Nonlinear Dynamic ANalysis). SPANDAN [28] consist of a suit of parallel algorithms for nonlinear dynamic analysis employing explicit, implicit time integration techniques with parallel sparse preconditioned conjugate gradient solvers, Profile direct solvers as well as hybrid (Combination of iterative and direct) solvers. SPANDAN is developed using MPIICH implementation of MPI software development environment. SPANDAN is capable of solving problems with both material and geometric nonlinearities. Material nonlinearity due to an elasto-plastic material response is considered and anisotropy effects are included in the yielding behaviour by employing Huber Mises material response and associated flow rule. The geometric nonlinearity effects are included by employing total Lagrangian formulations and the formulation accounts for only large deformations and moderate rotations. In SPANDAN, the strain-displacement matrix is calculated once during the nonlinear process and its nonlinear part is updated using the current displacements by a simple matrix product.

At present, SPANDAN’s element library consists of only Mindlin plate and shell elements. Even though, SPANDAN permits modelling structures with serendipity, lagrangian and heterosis elements, all the numerical studies conducted in this paper uses heterosis element. SPANDAN permits use of both consistent and lumped mass matrices. However, In the present work, consistent mass matrices are employed.

6. Parallel hardware-overview

The parallel code SPANDAN has been implemented on PARAM-10000, an Indian parallel super computer at NPSF (National PARAM Supercomputing Facility), Pune, India.

PARAM Open Frame scalable parallel computer is a distributed memory machine based on the heterogeneous Open Frame architecture. The Open Frame architecture unifies cluster computing with Massive Parallel Processing (MPP). PARAM-10000 is built with a cluster of 40 workstations. Each workstation in the cluster has 4 Ultra
SPARC processors running at 300 MHz and 512 MB of Random Access Memory (RAM). It supports networks such as MYRINET, Fast ETHERNET, and PARAMNET. The architecture of PARAM-10000 permits the parallel processing system to be viewed as an assembly of independent workstations, a cluster of workstations or as MPP system connected through a scalable high-bandwidth network or any combination of these. The operating system of PARAM-10000 is Solaris 2.6, which supports multithreading and multiprocessing. The message passing libraries of PARAM-10000 consists of MPICH implementation of MPI and PVM3.

6.1. Numerical studies

Two different sets of numerical experiments have been conducted and reported in this paper. The first set of numerical studies is intended to test the accuracy of the parallel algorithms built into SPANDAN. For this purpose, numerical examples of smaller size are considered. The second set of numerical studies is conducted to test the performance of the developed parallel nonlinear dynamic analysis code on parallel processing machines using the three parallel formulations discussed earlier in this paper. For this purpose, larger finite element models are employed, which can exploit the strengths and point out the weaknesses of the parallel algorithms.

6.2. Validation

The parallel code is expected to give the identical results of its sequential counterpart. In order to verify this, two standard numerical examples available in the literature are considered

A simply supported plate shown in Fig. 9 is considered as a first numerical example to verify the nonlinear dynamic response using SPANDAN with the proposed parallel

---

Fig. 8. Parallel PCG algorithm for Local Interface Formulation (LIF).
Newmark implicit time integration algorithm. The time step $\Delta t$ is taken $0.223E-04$ sec., which is $1/48$ of the fundamental period of the plate. Four processors have been employed for the solution. The dynamic elastic-plastic response of simply supported plate for both isotropic and anisotropic (yield in different directions) cases is shown in Fig. 10, which has been compared with the solutions of Huang and Hinton [29] and Owen and Li [30] reported in the literature. The results obtained using the proposed parallel implicit algorithm are found to be in good agreement with the results reported in the literature.

A deep moderately thick spherical shell cap with central angle of $120^\circ$ is considered as a second numerical example. The cap dimensions and material properties are given in the Fig. 11. A symmetric quadrant of the shell is considered and is described by a mesh as shown in Fig. 11. Both isotropic and anisotropic (Yield in different directions) cases are considered. The time step length, $\Delta t$ is taken as $0.10 \times 10^{-4}$ s. Only two processors are employed for the solution. Fig. 12 shows the comparison of elastic, elastic-plastic responses with isotropic and anisotropic material properties. The nonlinear dynamic response obtained using the parallel PCG based implicit algorithms of SPANDAN are compared with the results reported by Nagarajan and Popov [31] and found to be in good agreement.

6.3. Performance evaluation of parallel algorithms

To evaluate the performance of the three PCG based parallel algorithms, larger size problems are employed. The problem sizes are increased either by changing the dimension of the plate or by refining the finite element mesh. For all the numerical examples considered for evaluation, damping has been neglected and emphasis is
Fig. 10. Elasto-plastic transient dynamic response of simply supported plate employing SPANDAN with parallel time integration algorithms.

Fig. 11. Problem description for deep moderately thick spherical shell cap.

(a) Geometric detail of cap
(b) Finite element mesh (nine noded elements)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radius R</td>
<td>508.00 mm</td>
</tr>
<tr>
<td>Angle α</td>
<td>60°</td>
</tr>
<tr>
<td>Uniform Load P</td>
<td>4.137 Mpa</td>
</tr>
<tr>
<td>Thickness t</td>
<td>76.20 mm</td>
</tr>
<tr>
<td>Mass density ρ</td>
<td>$7.834 \times 10^{-10}$ N·sec²/mm⁴</td>
</tr>
<tr>
<td>Poisson’s Ratio ν</td>
<td>0.30</td>
</tr>
</tbody>
</table>

**ISOTROPIC PROPERTIES**

- Young’s modulus
  - $E_x = E_y = 206.84 \times 10^3$ MPa
- Uniaxial Yield Stress
  - $\sigma_{ot} = \sigma_{ot} = \sigma_{et} = 206.84$ MPa
- Plastic modulus
  - $E_p = 1061.79 \times 10^3$ MPa

**ANISOTROPIC PROPERTIES**

- Young’s Modulus
  - $E_x = E_y = 206.84 \times 10^3$ MPa
- Uniaxial Yield Stress
  - $\sigma_{ot} = \sigma_{ot} = \sigma_{et} = 206.84$ MPa
- Plastic modulus
  - $E_p = 1061.79 \times 10^3$ MPa

Note: $\sigma_{11}$ and $\sigma_{22}$ indicate Uniaxial yield stress in 1, 2 principal axes and $\sigma_{45}$ indicate yield stress at $45^\circ$ to the 1-axes.
given to maintain higher degree of accuracy with stringent convergence tolerance levels.

Fig. 13 shows the speedup of all three PCG based parallel algorithms while solving a plate problem of size 5445 degrees of freedom on four and eight processor configurations on PARAM-10000. The results have been compared with the theoretical speedup and also the conventional domain decomposition algorithm presented in Fig. 2 with a direct solver. Similarly, Fig. 14 shows the speedup performance of parallel algorithms for problem size of 12005 degrees of freedom. It can be observed that even for small size problems the parallel sparse PCG algorithms are superior in performance when compared to parallel direct solver.

In order to test and evaluate these algorithms with larger problems and larger processor configurations, three test meshes with 21125, 32805, 83205 degrees of freedom are solved employing all the three PCG based parallel algorithms slated for evaluation in this paper. The wall clock timings for all these test problems on four, eight, sixteen and thirty-two processors are presented in Figs. 15–17 respectively. Due to paucity of computer time, the simulations have been carried out for 200 time steps on PARAM 10000. From the results...
presented in the Figs. 13 and 14, it has been observed that the overall computational performance of all the parallel algorithms improves with increase in the problem size. Similarly, it can be observed from Figs. 15–17 that the behavior of the parallel algorithms on varied number of processors is consistent.

A close look at the results presented in Figs. 13–17, indicates that the proposed parallel PCG algorithms performs better than direct solvers, when repetitive solution of equations are required as in the case of nonlinear dynamic analysis. Among the PCG solvers, the LSF is found to be slightly faster for the problems solved in this paper. It can also be observed that the algorithms based on diagonal preconditioner are found to be superior to the PCG implementations with incomplete Cholesky factorisation as preconditioner. The diagonal preconditioner is computationally less intensive and also converges faster for well-conditioned matrices. The dynamic stiffness matrices in implicit time marching scheme for the problems solved in this paper are generally well conditioned. It is well known that diagonal preconditioner is expected to perform more efficiently than the preconditioner based on incomplete factorisation for solution of well-conditioned matrices like the ones obtained from implicit dynamic analysis. However, in situations like solving interaction problems and also thin shell problems modelled with 3-D finite elements, the PCG algorithm with incomplete factorisation is expected to perform better. Moreover, handling of the large preconditioning matrices in parallel environment in case of
incomplete factorisation imposes an overhead on the computing time in each iteration and also increases substantially the storage requirements.

The accuracy of the solution in PCG based algorithms depends on the convergence tolerance chosen for the conjugate gradient iterations. In order to maintain higher degree of accuracy in the solution, highly conservative tolerances have been considered during the numerical studies. The convergence tolerance of conjugate gradient iterations is taken as $1 \times 10^{-8}$ and convergence of Newton Raphson iterations is taken as $1 \times 10^{-6}$. The rejection parameter in incomplete Cholesky preconditioner is taken as $0.10 \times 10^{-6}$. However, the performance of these PCG based parallel algorithms can be further improved by optimising these tolerance values without sacrificing the quality. This requires further work in order to identify the optimum tolerances for conjugate gradient iterations. Another aspect related to the PCG algorithms, which require attention and also further studies, is to evaluate the effects of infrequent updating of the incomplete Cholesky preconditioner. In the present work, the preconditioner is updated once in every two steps in order to maintain good accuracy. However, identifying the optimum interval is beyond the scope of the present work.

Finally the communication overheads of various PCG formulations discussed in this paper are evaluated for a problem size of 32805 degrees of freedom and the results are shown in Fig. 18. A close look at the results indicate that, the communication overheads increase with the increase in number of processors. PCG formulations with diagonal preconditioner have less interprocessor communication overheads and it is very well reflected in the overall performance as already discussed earlier. It can also be observed that the communication overheads are comparatively less for LSF formulation as the algorithm is amenable for overlapping communications to computations.

7. Conclusions

In this paper, parallel processing techniques for computing nonlinear dynamic response employing finite element method is presented. The most time consuming segment in implicit nonlinear dynamic analysis is linear equation solver. This paper proposes three parallel formulations combining domain decomposition based finite element method with linear preconditioned conjugate gradient solvers for computing nonlinear dynamic response. They are Global Interface Formulation, Local Submesh Formulation, and Local Interface Formulation. The Compact Row Storage (CRS) scheme has been employed for sparse storage of the global matrices. All the parallel PCG solver implementations have been developed for this sparse data

![Fig. 17. Performance of parallel PCG based algorithms for a problem size of 83,205 degrees of freedom on PARAM10000.](image)

![Fig. 18. Comparison of communication timings of three parallel time integration algorithms for a problem size of 32,805 degrees of freedom on PARAM10000.](image)
structure to enhance the computational efficiency. Parallel as well as sequential approaches for efficient assembly of global matrices for CRS data format have been proposed. In the PCG solver, both diagonal scaling as well as incomplete Cholesky based preconditioners have been employed.

All the three parallel sparse PCG based formulations presented in this paper have been implemented in a nonlinear dynamic analysis parallel finite element code called SPANDAN using MPI software development environment. The performance of the parallel code SPANDAN (with the proposed parallel LPCG formulations) is evaluated by solving number of test meshes of varying size and employing variable number of processors. Numerical experiments indicate that the parallel PCG algorithms are faster when compared to parallel direct solvers even for small size problems.

Based on the numerical experiments the following remarks can be made.

i. All the three PCG formulations discussed in this paper are scalable. The diagonal preconditioner is relatively superior in performance in all three PCG formulations both in terms of communication overheads and also the overall performance. One main reason could be the nature of the test problems solved in this paper. The Cholesky preconditioner is likely to become more competitive and even superior when the matrices to be solved are not well conditioned.

ii. The GIF formulation will be more attractive for problems with narrow interfaces like three dimensional towers or frame like structures as the overheads associated with the assembly of the schur complement matrices will be relatively smaller.

iii. On coarse grain computing environment, for problems with larger interfaces, the LIF formulation will be handy as it permits to operate on local schur complement matrices and thus avoids completely, the computational/communication overheads associated with assembly of large schur complement matrices.

iv. For problems like large three-dimensional meshes, where computing schur complement matrices is compute intensive task, the Local Submesh Formulation is more attractive. As already shown, this formulation is amenable for efficient implementation of interprocessor communications by way of overlapping with computations. Among all the three formulations, the Local Submesh Formulation (LSF) is found to be superior in performance when compared to the other two formulations. Moreover, it did not show any degradation in terms of convergence with the increase in number of submeshes. It is however appropriate to mention here that the problems solved in the present paper yields very well conditioned matrices. As local preconditioner is being used in this formulation, the performance of the algorithm depends rather heavily on the quality of the preconditioner to reduce the number of iterations, especially while solving ill-conditioned matrices. Hence this algorithm is likely to work better for large size problems with well-conditioned matrices.

v. The numerical studies are conducted on PARAM-10000, an Indian parallel super computer. The results presented here are on the hardware where the processor nodes are networked through 100 Mbps fast Ethernet. Faster and better networks such as MYRINET or PARAMNET can further enhance the performance and scalability of the proposed algorithms.

vi. As already mentioned earlier, stringent convergence criterion has been applied for PCG algorithms in order to maintain accuracy. A rather relaxed convergence criterion also likely to enhance the performance of the proposed algorithms.

vii. Further numerical experiments are needed to understand the effect of infrequent updating of Cholesky preconditioner both on the performance and convergence of the PCG formulations.

Acknowledgements

The authors wish to acknowledge the support of National PARAM Supercomputing Facility (NPSF), CDAC, Pune, India, for extending their computing facilities for this work. This paper is being published with the permission of the Director, Structural Engineering Research Centre, Madras

References


