

## Performance of Parallel Conjugate Gradient Solvers in Meshfree Analysis

Youngjoon Kim, Graduate Research Assistant  
Mechanical Engineering, Center for Computer-Aided Design  
The University of Iowa

Colby C. Swan<sup>♦</sup>, Associate Professor  
Civil & Environmental Engineering, Center for Computer-Aided Design  
The University of Iowa

Jiun-Shyan Chen, Associate Professor  
Civil & Environmental Engineering  
University of California, Los Angeles

**ABSTRACT:** Meshfree analysis methods, on a per degree of freedom basis, are typically more computationally expensive and yet more accurate than finite element methods. For very large models, whether meshfree or finite element, the memory and computational effort associated with direct equation solvers makes them prohibitively expensive. In this work, the performance of different linear equation solvers with meshfree analysis methods is explored. In particular, parallel conjugate gradient solvers with both Jacobi diagonal preconditioning and incomplete Cholesky factorization preconditioning are tested on a number of different meshfree analysis applications and compared against the performance of a fast direct sparse equation solver. It is found in these exploratory computations that as the support size of meshfree shape functions increases, the condition number of the associated stiffness matrices increases, and the relative efficiency of iterative solvers suffers somewhat. Nevertheless, for normalized support sizes between one and two, the performance of both conjugate gradient solvers compares very favorably with that of the sparse direct solver for intermediate ( $N \approx O(10^5)$ ) and large problems.

Keywords: meshfree methods; conjugate gradient; parallel solvers; preconditioners; high-performance computing.

---

<sup>♦</sup> Corresponding author. E-mail: [colby-swan@uiowa.edu](mailto:colby-swan@uiowa.edu); Ph: 1 319 335-5831; Fax: 1 319 335-5660; Dept. of Civil & Environmental. Engineering, 4120 SC, The University of Iowa, Iowa City, Iowa 52242, USA.

## 1. INTRODUCTION

Conventional finite element methods suffer a number of difficulties in dealing with specific classes of problems such as those involving large deformation and those involving crack propagation, which require the treatment of discontinuities that do not coincide with the original mesh lines [5]. The objective of recently developed meshfree methods is to eliminate these difficulties by constructing the function approximations entirely in terms of nodes. That is, meshfree methods do not require any explicit meshes for approximation, and the problem domain is completely described by particles, with the approximation solution constructed using a set of meshfree shape functions [9,32].

Despite these significant benefits, the high computational cost of meshfree methods has been perceived as problematic. The unique aspects of meshfree methods compared to finite element methods that result in greater computational expense are: larger bandwidths of the stiffness matrix, nontrivial imposition of essential boundary conditions, lack of simple element-based data structures, more expensive spatial numerical integration procedures, and nontrivial computation of nodal shape functions. In recent years, many of these efficiency issues with meshfree methods have been successfully addressed. For example, using singular kernel shape functions for the essential boundary nodes greatly simplifies the imposition of essential boundary conditions [8], and usage of stabilized conforming nodal integration dramatically reduces the cost of spatial integration [9]. Furthermore, for large and nonlinear problems, the initial nontrivial cost of computing shape functions becomes a relatively insignificant fraction of the overall computing effort in comparison to the nonlinear solving operations. With greatly improved efficiency in these aspects of meshfree analysis, the need to address efficiency issues in nonlinear solving operations becomes even more important.

To further reduce the computational expense associated with meshfree methods, this effort will focus on the reduction of linear solving times which lie at the heart of most nonlinear solving algorithms. Many efficient linear equation solvers that utilize sparse storage schemes in solving  $A\mathbf{x} = \mathbf{b}$  have been used successfully with finite element methods. However few of these solvers have yet been applied or explored with meshfree methods due to the *a priori* difficulty in determining the sparse storage requirements. Part of the difficulty in determining the sparse storage requirements with meshfree analysis arose from the need to apply essential boundary conditions using transformation procedures that modified the sparsity characteristics in a manner that was very difficult to predict [8]. However, with the boundary singular kernel shape functions mentioned previously, the essential boundary nodes recover Kronecker delta properties, obviating the need for transformation procedures. Accordingly, the sparsity characteristics of meshfree equation systems become very tractable making it very easy to use sparse solvers in linear solving operations.

In large-scale structural analyses, the computational expense is often dominated by the cost solving systems of coupled linear equations. In the past two decades much research has therefore been done on fast and efficient linear solving algorithms for large, potentially ill-conditioned systems. In the 1980's and early 1990's numerous very fast vector and parallel direct equation solvers were developed based on different implementations of Choleski factorization [29,30] of symmetric positive definite systems of equations into triangular factors ( $\mathbf{LL}^T$ ), followed by the forward and backward solution of the resulting triangular systems. Notable works have been accomplished in the development of Cholesky factorization algorithms that use skyline, profile and sparse matrix storage schemes [30] of which sparse solvers tend to show the greatest performance. A key aspect of fast and efficient sparse solvers is that of re-ordering the system of equations in such a way as to minimize the amount of "fill-in" that occurs

during factorization, since this can significantly reduce both the number of factorization operations and the memory storage requirements for the factorization of the stiffness matrix. In spite of the significant gains in efficiency with optimum re-ordering of sparse systems, the asymptotic performance of even the best sparse solvers indicates that their required memory and cpu-operations grow in proportion to  $(N^2 - N^{2.5})$  as  $N$  becomes large, and thus will not be competitive with scalable iterative solvers for sufficiently large problem sizes. Nevertheless, numerous domain decomposition methods (e.g. [13]) have been investigated with the objective of breaking extremely large systems into moderately sized and coupled subdomains on which the fast, direct solvers can still be effectively employed. Since development of efficient solving methods appropriate for large nonlinear meshfree analysis problems is the goal of this research, parallel iterative conjugate gradient equation solvers with different preconditioning methods are explored here as potential solutions and their performance is compared with that of a very fast direct, sparse solver.

It is believed that CPU and memory requirements that scale like  $N$  as  $N$  becomes large can be achieved by employing conjugate gradient solvers in conjunction with appropriate preconditioning methods and parallelization. The recognized advantages of conjugate gradient solvers that make them potentially attractive here are: 1) cpu-effort scales linearly with problem size when used with diagonal preconditioning; 2) explicit storage of assembled global tangent stiffness operator with fill-in is not required; 3) they are easy to parallelize; and 4) they have been used successfully with FEM. Therefore, the performance of conjugate gradient solvers with meshfree methods is investigated and issues arising from the nature of meshfree methods are noted. In addition, alternative preconditioning methods in conjunction with a conjugate gradient solver are implemented and tested. All the operations of the preconditioned conjugate gradient (PCG) solvers are parallelized for shared memory environments, and the parallel

performance characteristics of the PCG solvers are tested.

The remainder of this paper is organized as follows. Section 2 presents a brief review of meshfree methods including shape functions, variational continuum formulation, and notable characteristics of meshfree methods. In Section 3 preconditioned conjugate gradient solvers and their implementation for meshfree analyses are discussed along with the details of an incomplete Cholesky factorization preconditioning algorithm. Issues pertaining to parallel implementation of the PCG solver with IC factorization are covered in Section 4. Many numerical examples are presented in Section 5 to compare the performance of direct and different PCG solvers. The manuscript closes with both discussion and conclusions in Section 6.

## 2. OVERVIEW OF MESHFREE METHODS

### **2.1 Shape Functions**

The major difference between finite element methods and meshfree methods lies in the spaces from which the approximation functions are constructed. In FEM, approximation functions are developed with shape functions that are both node and element based. In meshfree methods, the shape functions are only node-based. The Reproducing Kernel (RK) approximation in the Reproducing Kernel Particle Method (RKPM) [24] is one of the most commonly used function approximation methods, and it is employed for this research. The discrete RK approximation field of a variable field  $u$  is

$$u^h(\mathbf{x}) = \sum_{I=1}^{NP} \overline{\Phi}_a(\mathbf{x}; \mathbf{x} - \mathbf{x}_I) d_I \quad (2.1)$$

where  $NP$  is the number of discrete points used in the approximation,  $d_I$  are the coefficients of the approximation functions, and  $\overline{\Phi}_a(\mathbf{x}; \mathbf{x} - \mathbf{x}_I)$  is the reproducing kernel, which is expressed by

$$\bar{\Phi}_a(\mathbf{x}; \mathbf{x} - \mathbf{x}_I) = \mathbf{C}(\mathbf{x}; \mathbf{x} - \mathbf{x}_I) \Phi_a(\mathbf{x} - \mathbf{x}_I) \quad (2.2)$$

where  $\Phi_a(\mathbf{x}-\mathbf{x}_I)$  is a kernel or weight function,  $a$  is a measure of the size of the support, and  $\mathbf{C}(\mathbf{x}; \mathbf{x}-\mathbf{x}_I)$  is a correction function that is used to satisfy the  $n$ -th order reproducing conditions

$$\sum_{I=1}^{NP} \bar{\Phi}_a(\mathbf{x}; \mathbf{x} - \mathbf{x}_I) x_{1I}^p x_{2I}^q x_{3I}^r = x_1^p x_2^q x_3^r \quad \text{for } p+q+r=0,1,\dots,n; p \geq 0; q \geq 0; r \geq 0 \quad (2.3)$$

where  $x_{iI}$  is the nodal value of the  $i^{\text{th}}$  Cartesian component  $x_i$  at node  $I$ . The correction function  $\mathbf{C}(\mathbf{x}; \mathbf{x}-\mathbf{x}_I)$  is constructed by a linear combination of complete  $n$ -th order monomial functions

$$\mathbf{C}(\mathbf{x}; \mathbf{x} - \mathbf{x}_I) = \sum_{p+q+r=0}^n (x_1 - x_{1I})^p (x_2 - x_{2I})^q (x_3 - x_{3I})^r b_{pqr}(\mathbf{x}) \equiv \mathbf{H}^T(\mathbf{x} - \mathbf{x}_I) \mathbf{b}(\mathbf{x}) \quad (2.4)$$

where  $b_{pqr}(\mathbf{x})$  are the coefficients of the monomial basis functions that are functions of  $\mathbf{x}$ ,  $\mathbf{b}(\mathbf{x})$  is a vector of  $b_{pqr}(\mathbf{x})$ , and  $\mathbf{H}(\mathbf{x}-\mathbf{x}_I)$  is a vector containing the monomial basis functions

$$\mathbf{H}^T(\mathbf{x} - \mathbf{x}_I) = \left[ 1, x_1 - x_{1I}, x_2 - x_{2I}, x_3 - x_{3I}, (x_1 - x_{1I})^2, \dots, (x_3 - x_{3I})^n \right] \quad (2.5)$$

Substituting Eq. (2.4) into Eq. (2.3), the coefficients  $\mathbf{b}(\mathbf{x})$  are solved by

$$\mathbf{M}(\mathbf{x}) \mathbf{b}(\mathbf{x}) = \mathbf{H}(\boldsymbol{\theta}) \quad (2.6)$$

where

$$\mathbf{M}(\mathbf{x}) = \sum_{I=1}^{NP} \mathbf{H}(\mathbf{x} - \mathbf{x}_I) \mathbf{H}^T(\mathbf{x} - \mathbf{x}_I) \Phi_a(\mathbf{x} - \mathbf{x}_I) \quad (2.7)$$

is the moment matrix of  $\Phi_a(\mathbf{x} - \mathbf{x}_I)$ . Using the solution of Eqs. (2.2), (2.4) and (2.6), the reproducing kernel function approximation is obtained by

$$\mathbf{u}^h(\mathbf{x}) = \sum_{I=1}^{NP} \mathbf{H}^T(\boldsymbol{\theta}) \mathbf{M}^{-1}(\mathbf{x}) \mathbf{H}(\mathbf{x} - \mathbf{x}_I) \Phi_a(\mathbf{x} - \mathbf{x}_I) \mathbf{d}_I \equiv \sum_{I=1}^{NP} \Psi_I(\mathbf{x}) \mathbf{d}_I \quad (2.8)$$

in which  $\Psi_I(\mathbf{x})$  ( $I = 1, 2, \dots, NP$ ) are the RK shape functions

$$\Psi_I(\mathbf{x}) = \mathbf{H}^T(\boldsymbol{\theta})\mathbf{M}^{-1}(\mathbf{x})\mathbf{H}(\mathbf{x} - \mathbf{x}_I)\Phi_a(\mathbf{x} - \mathbf{x}_I). \quad (2.9)$$

A cubic spline function is employed as the kernel function in this research such that

$$\Phi_a(\mathbf{z}) = \begin{cases} \frac{2}{3} - 4\mathbf{z}^2 + 4\mathbf{z}^3 & \text{for } \mathbf{z} \leq \frac{1}{2} \\ \frac{4}{3} - 4\mathbf{z} + 4\mathbf{z}^2 - \frac{4}{3}\mathbf{z}^3 & \text{for } \frac{1}{2} < \mathbf{z} \leq 1 \\ 0 & \text{for } \mathbf{z} > 1 \end{cases} \quad (2.10)$$

where  $\mathbf{z}$  is  $|\mathbf{x} - \mathbf{x}_I|/a$ . The quantity “ $a$ ” is commonly called shape function support size, and is a principal factor in determining the connectivity among discrete nodal points of the model.

## **2.2 Variational Continuum Formulation**

Meshfree analysis has been implemented for a variety of linear and nonlinear structural computations of static problems using reproducing kernel particle methods [7]. We can consider a body that is deformed from initial configuration  $\Omega_X$  with boundary  $\Gamma_X$  to a deformed configuration  $\Omega_x$  with deformed boundary  $\Gamma_x$ . The body is subjected to body force field  $b_i$  in  $\Omega_x$ , a surface traction field  $h_i$  on the natural boundary  $\Gamma_x^{h_i}$ , and a prescribed displacement field  $g_i$  on the essential boundary  $\Gamma_x^{g_i}$ . We denote the particle positions in the initial configuration  $\Omega_X$  by  $\mathbf{X}$  and those in the deformed configuration  $\Omega_x$  at time  $t$  by a mapping function  $\boldsymbol{\varphi}$  where  $\mathbf{x} = \boldsymbol{\varphi}(\mathbf{X}, t)$ . The strong form of equilibrium for general linear or nonlinear continuum problems is as follows: Given  $b_i(\mathbf{X}, t)$ ,  $h_i(\mathbf{X}, t)$ , and  $g_i(\mathbf{X}, t)$ , find  $\mathbf{u} : \overline{\Omega} \otimes [0, T] \mapsto \mathfrak{R}^3$  such that

$$\tau_{ij,j} + b_i = 0 \quad \text{in } \Omega_x \quad (\text{Equilibrium condition}) \quad (2.11)$$

$$\tau_{ij,j} n_j = h_i \quad \text{on } \Gamma_x^{h_i} \quad (\text{Natural boundary condition}) \quad (2.12)$$

$$u_i = g_i \quad \text{on } \Gamma_x^{g_i} \quad (\text{Essential boundary condition}) \quad (2.13)$$

Above  $\tau_{ij}$  is Cauchy stress obtained from a general constitutive law of the form  $\boldsymbol{\tau} = \hat{\boldsymbol{\tau}}(\mathbf{F})$  in which  $\mathbf{F} = \partial \mathbf{x} / \partial \mathbf{X}$  is the deformation gradient. In this paper, a Saint Venant-Kirchhoff elastic material is assumed for which the second Piola-Kirchhoff stress  $S_{IJ}$  is calculated by

$$S_{IJ} = \lambda E_{KK} \delta_{IJ} + 2\mu E_{IJ} = C_{IJKL} E_{KL} \quad (2.14)$$

in which  $E_{KL}$  is the Green-Lagrange strain tensor  $[E_{KL} = \frac{1}{2}(F_{kK} F_{kL} - \delta_{KL})]$ . The second Piola-Kirchhoff stress is related to the Cauchy stress by

$$\tau_{ij} = \frac{1}{J} F_{iM} S_{MN} F_{jN} \quad (2.15)$$

where  $J = \det(\mathbf{F})$ . Using standard methods in vector calculus, it can be shown that the weak or variational form corresponding to the strong form above is:

$$\int_{\Omega_x} \delta u_{i,j} \tau_{ij} d\Omega = \int_{\Omega_x} \delta u_i b_i d\Omega + \int_{\Gamma_x^{h_i}} \delta u_i h_i d\Gamma. \quad (2.16)$$

### **2.3 Introduction of Meshfree Shape Functions**

By introducing the RK approximation from Eq. (2.8) into Eq. (2.16), the following matrix equilibrium equations at the  $A$ -th unrestrained node are obtained

$$\mathbf{r}_A = \mathbf{f}_A^{int} - \mathbf{f}_A^{ext} = \mathbf{0} \quad (2.17)$$

in which

$$\mathbf{f}_A^{int} = \int_{\Omega_x} \mathbf{B}_A^T : \boldsymbol{\tau} d\Omega \quad (2.18)$$

$$\mathbf{f}_A^{ext} = \int_{\Omega_x} \Psi_A \mathbf{b} d\Omega + \int_{\Gamma_x^h} \Psi_A \mathbf{h} d\Gamma \quad (2.19)$$

where:  $\mathbf{B}_A$  is the nodal strain displacement matrix;  $\Psi_A$  denotes the nodal shape function for the  $A^{\text{th}}$  node;  $\mathbf{f}_A^{\text{int}}$  represents the internal forces on node  $A$ ; and  $\mathbf{f}_A^{\text{ext}}$  represents the external forces on node  $A$  due to body forces and surface tractions. The tangent stiffness matrix associated the system (2.17) is written:

$$\frac{df_{Ai}^{\text{int}}}{du_{Bl}} = K_{il}^{AB} = \int_{\Omega_x} B_{ji}^A c_{jk} B_{kl}^B d\Omega_x + \int_{\Omega_x} \Psi_{,j}^A \tau_{jk} \Psi_{,k}^B \delta_{il} d\Omega_x \quad (2.20)$$

Above,  $c_{jk}$  is the condensed form of the spatial elasticity tensor associated with (2.14) and (2.15).

## 2.4 General Newton's Methods

For linear static problems, the internal force vector is the product of the linear stiffness matrix  $\mathbf{K}$  and the displacement vector  $\mathbf{d}$ , and only a single factorization is required. For nonlinear static problems, the system of nonlinear global force-balance equations at a given time or load step  $n+1$  has the form

$$\mathbf{r}(\mathbf{d}_{n+1}) = \mathbf{0} \quad (2.21)$$

which is generally solved by Newton's method with line searching for a fixed time step ([15,31],for examples). The sequence to update the global displacement vector at the  $(n+1)$ -th time step is shown in the algorithm of Box 2.1 in which  $\alpha_v$  is the line search parameter chosen to satisfy the standard line search criterion

$$\left| \delta_v \cdot \mathbf{r}^{v+1}(\mathbf{d}_{n+1}^{v+1}) \right| < \text{STOL} \quad (2.22)$$

where STOL is a tolerance parameter controlling the accuracy of the search. For the linear

solving phase, the global stiffness matrix  $\mathbf{K}$  needs to be updated each iteration in a full Newton-Raphson method, but only occasionally if a modified Newton's method is used.

## **2.5 Definition of Nodal Neighbors and Data Structure**

For enhanced accuracy and efficiency, the global stiffness matrix and force vector in Eqns. (2.18) – (2.20) are integrated in this work using a stabilized conforming nodal integration method (SCNI) [9]. In SCNI, the nodal representative areas are defined using Voronoi cells (as shown in Figure 2.1) for local strain smoothing to meet linear exactness in the Galerkin approximation for optimum rate of integration [9]. It follows that the connective topology of the global stiffness matrix structure in this work is determined by such factors as Lagrangian nodal coordinates, Voronoi integration cell specifications, and shape function support sizes. It is necessary to explicitly quantify the connective topology of the stiffness matrix in order to store and manipulate it with the data structures used by sparse solvers.

To facilitate this discussion, a number of definitions are presented here. First, the set of all nodes in a meshfree model, with each identified by an integer value, is denoted  $\eta$ . Each node will have its own support region which is the space covered by its nodal shape function. For example, the support region of node  $I$  is denoted by  $\Omega_I = \{\mathbf{X} \in \Omega_X \mid \Psi_I(\mathbf{X}) \neq 0\}$ . In addition, each node  $I$  in the model will have its own Voronoi integration cell region denoted  $V_I$ . Furthermore, each node  $I$  in the model will have its own set of coupled nodes  $C_I$  and its own set of neighbor nodes  $N_I$  both of which are defined mathematically as follows:

$$\begin{aligned} C_I &= \{J \in \eta \mid \Omega_I \cap \Omega_J \neq \{\emptyset\}\} \\ N_I &= \{J \in \eta \mid V_I \cap \Omega_J \neq \{\emptyset\}\} \end{aligned} \quad (2.23)$$

In other words,  $C_I$  is node  $I$ 's set of coupled nodes whose support regions overlap that of node  $I$ .

In a similar fashion,  $N_I$  is the set of node  $I$ 's neighbor nodes, with members being those whose support region overlaps node  $I$ 's integration Voronoi cell  $V_I$ . Figure 2.1(a) shows a typical node  $I$  with its support region  $\Omega_I$  covered by node  $I$ 's nodal shape function and Figure 2.1(b) shows the typical Voronoi integration cell for a node  $B$ . Since a Lagrangian continuum formulation is used, these overlap computations of each node's set of coupled nodes  $C$  and neighbor nodes  $N$  can be done once and for all in the undeformed configuration. Each node's set of connected and neighbor nodes is stored in the following data structures:

- $IP(i), (i=1, \text{numnp})$ : number of nodes whose "support" covers the integration cell of node  $i$ . This is the number of "neighbors nodes" for a node  $i$ .
- $INODE(i, j), (i=1, \text{numnp}; j=1, IP(i))$  : list of neighbor nodes for node  $i$ .
- $NB, (i=1, \text{numnp})$ : number of coupled nodes for each node  $i$
- $NBL(i, j), (i=1, \text{numnp}; j=1, NB(i))$  : list of coupled nodes for each node  $i$

The algorithm for assembly of the global stiffness matrix from nodal stiffness matrices is provided in Box 2.2.

## **2.6 Special Characteristics of Meshfree Methods**

The special characteristics of meshfree methods that contribute to the high computational cost of solving linear systems  $\mathbf{K} \cdot \boldsymbol{\delta} = -\mathbf{r}$  are large bandwidths of the global stiffness matrix and severe ill conditioning of the global stiffness matrix with increasing shape function support size. Figure 2.2 shows an equally spaced discretization in order to illustrate why meshfree methods have global stiffness matrices with large bandwidths. When a normalized shape function support size of unity is used, the number of coupled nodes at the  $i$ -th integration point in Figure 2.2 is 5, whereas the number of neighbor nodes with piecewise linear finite element shape functions is 3. The number of neighbor nodes is 25 in two-dimensional problems and 125 in three-dimensional

problem with meshfree methods, whereas it is 9 in two-dimensional problems and 27 in three-dimensional problems with finite element methods using bilinear and tri-linear shape functions. When it is considered that normalized support sizes of 1.5~2.5 are commonly used with meshfree methods, it is clear that the bandwidth of  $\mathbf{K}$  is typically much larger than that for FEM. This aspect of meshfree methods that causes large bandwidths of the global stiffness matrix consequently requires more CPU time in solving linear systems  $\mathbf{K} \cdot \boldsymbol{\delta} = -\mathbf{r}$ . Understanding this characteristic of meshfree methods, the performance of direct equation solvers and iterative equation solvers is investigated and compared.

The ill conditioning of the global stiffness matrix with larger shape function support sizes is another unique feature of meshfree methods that can result in higher computational cost. The reason for this characteristic is that as the normalized support sizes increase, the resulting shape functions become more and more linearly dependent if the order of basis functions stays unchanged. While ill conditioning of the stiffness matrix does not generally present much of a problem for direct equation solvers, it can present a very serious problem with iterative equation solvers.

### 3. PRECONDITIONED CONJUGATE GRADIENT SOLVERS

#### **3.1 Jacobi Preconditioned Conjugate Gradient Solvers**

The conjugate gradient method, originally developed by Hestenes and Stiefel [12], is an iterative algorithm for solving systems of linear equations  $\mathbf{Ax} = \mathbf{b}$ , for real symmetric positive definite matrices  $\mathbf{A} \in \mathbb{R}^{n \times n}$  without assembling or factorizing the matrix  $\mathbf{A}$ . The conjugate gradient method works well on matrices that are either well conditioned or have just a few distinct eigenvalues, and so we often attempt to precondition a linear system so that the matrix of coefficients assumes one of these nice forms. Given a symmetric positive definite  $\mathbf{A} \in \mathbb{R}^{n \times n}$ ,  $\mathbf{b} \in$

$\mathbb{R}^n$ , a symmetric positive definite preconditioner  $\mathbf{B}$ , and initial guess  $\mathbf{x}_0$  ( $\mathbf{A}\mathbf{x}_0 \approx \mathbf{b}$ ), the algorithm of Box 3.1 for modified PCG - based upon the conjugate gradient algorithm - solves the linear system  $\mathbf{A}\mathbf{x} = \mathbf{b}$ .

The advantage of iterative equation solvers such as Preconditioned Conjugate Gradient (PCG) methods (Box 3.1) and memoriless quasi-Newton methods [31] is that they require less storage than sparse or banded solvers, since the global stiffness matrix  $\mathbf{K}$  need not be formed and factorized. A goal that is sometimes achievable with PCG iterative solvers is to have the asymptotic number of operations grow in proportion to the problem size  $N$  as  $N$  becomes large. PCG solvers are very attractive due to their relative ease of parallelization on most computing architectures and the fact that when a high degree of accuracy is not required in the solution, the number of iterations can be dramatically reduced thus trimming down the computational expense accordingly [26]. Despite these strong points favoring iterative equation solvers that can lead them to sometimes dramatically outperform even the fastest direct solvers on moderately sized problems, the major disadvantage of iterative methods is that when the system of equations to be solved is ill-conditioned, the convergence characteristics of the method suffer quite dramatically.

The equation for forming incremental force for each node in the PCG algorithm is

$$\mathbf{s}_{m+1} = \mathbf{s}_m - \alpha_m \mathbf{A} \cdot \mathbf{g}_m \quad (3.1)$$

where the computation of product  $\mathbf{A} \cdot \mathbf{g}_m$  is performed not at global level, but at the nodal level by storing the nodal contribution to  $\mathbf{A}$ , and is composed of following steps: 1) localize global incremental displacement vector  $\mathbf{g}$  to nodal level; 2) gather all coupled nodal stiffness matrices  $\mathbf{A}$  from storage; 3) apply nodal stiffness and nodal displacements to compute the product  $\mathbf{A} \cdot \mathbf{g}$ ; 4) assemble all nodal contributions into global force vector.

Different preconditioning methods for conjugate gradient solvers lead to quite a range of different performance characteristics making the choice of an appropriate preconditioner  $\mathbf{B}$  quite

important for the practical applications of the PCG algorithm. The rate of convergence in the energy norm  $\|\mathbf{x}_m\|_{\mathbf{A}} = (\mathbf{x}_m \cdot \mathbf{A} \cdot \mathbf{x}_m)^{\frac{1}{2}}$  is bounded as follows:

$$\|\mathbf{x}_{m+1} - \mathbf{x}\|_{\mathbf{A}} \leq \frac{\sqrt{C} - 1}{\sqrt{C} + 1} \|\mathbf{x}_m - \mathbf{x}\|_{\mathbf{A}} \quad (3.2)$$

where  $C$  is the condition number of  $\mathbf{B}^{-1}\mathbf{A}$ . Since the right hand side increases with growing condition number, lower condition numbers usually accelerate convergence. It is therefore desired to have  $\mathbf{B}^{-1}$  approximate  $\mathbf{A}^{-1}$  in the sense that  $C \approx 1$ , while retaining a computationally efficient structure. An extremely simple and yet sometimes very effective preconditioner is diagonal scaling in which  $\mathbf{B}^{-1} = (\text{diag } \mathbf{K})^{-1}$ . This is often referred to as Jacobi acceleration [14,15] or the Jacobi conjugate gradient method (JCG).

### **3.2 Alternative Pre-conditioning Strategies**

When the convergence rate of the JCG method is slow, other more robust preconditioning strategies should be used that do not compromise the inherent benefits of iterative equation solvers. Among popular preconditioning methods are Incomplete Cholesky Factorization (IC) preconditioning [3,16], element-by-element preconditioning [19], polynomial preconditioning [3,16] and multigrid preconditioning [1]. IC preconditioning methods are very effective in terms of accelerating the convergence in accordance with (3.4). However IC factorization algorithms can be difficult to parallelize due to the recursive nature of the computation. On the other hand, polynomial preconditioners are easy to parallelize since they only involve the computation of matrix-vector operations, but this method is not as powerful as the IC preconditioning method. To overcome difficulties with parallelization of IC factorization, incomplete block Cholesky factorization preconditioner using matrix blocks as basic entities were proposed [2,10,25].

The Crout element-by-element preconditioner was proposed by Hughes et al [19] to maintain the element-based data structure in finite element codes. With this approach, additional memory required for preconditioner is reduced and CPU-effort required to define a new data structure is not necessary. Recently Grote and Huckle [17] proposed a more sophisticated method for parallel preconditioning with sparse approximate inverses. With this method, a sparse approximate inverse is computed explicitly in parallel and it provides full control over the sparsity and the quality of preconditioner. These capabilities enable one to choose an optimal preconditioner for a particular problem and architecture.

### **3.3 The Incomplete Cholesky Factorization Preconditioner**

One of the most important preconditioning strategies involves computing an IC factorization of  $A$ . The idea behind this approach is to calculate a lower triangle matrix  $H$  with the property that  $H$  has some tractable sparsity structure and is somehow close to  $A$ 's exact Cholesky factor  $G$ . The preconditioner is then taken to be  $B^{-1} = H^{-1}H^{-T}$ . To appreciate this choice, consider the two following facts: First, there exists a unique symmetric positive definite matrix  $C$  such that  $B = CC^T$  and  $B^{-1} = C^{-1}C^{-T}$ . Second, there exists an orthogonal transformation  $Q$  such that  $C = QH^T$ . Accordingly,

$$\begin{aligned} B^{-1}A &= C^{-1}AC^{-1} \\ &= (HQ^T)^{-1}A(QH^T)^{-1} = Q(H^{-1}GG^TH^{-T})Q^T \approx I \end{aligned} \quad (3.3)$$

Thus, the better  $H$  approximates  $G$  the smaller the condition number of  $B^{-1}A$ , and the better the performance of the PCG algorithm.

A simple, effective  $H$  that approximates  $G$  is computed by stepping through the sparse Cholesky factorization neglecting fill-in, and it is described in the algorithm of Box 3.2. The

number of operations required for the IC factorization neglecting fill-in is as follows

$$\sum_{k=1}^N (IP_k \times NDOF)^2 \quad (3.4)$$

where  $N$  is the number of integration points or nodes in the model, and  $IP_k$  is the number of neighbor nodes coupled to the  $k^{\text{th}}$  node.

For what is called here the Incomplete Cholesky Conjugate Gradient (ICCG) method, the global stiffness matrix is formed and stored in the so-called *compressed row* sparse format, which stores only non-zero terms of stiffness matrix row by row. Box 3.3 is an example of matrix storage with the *compressed row* sparse format. Since the proposed algorithm for IC factorization neglects fill-in the data structure for the lower triangle matrix  $\mathbf{H}$  is identical to that of the stiffness matrix in sparse format. The size of diagonal preconditioner is very small in comparison with global stiffness matrix in large-scale problems, and thus the memory required for the ICCG method is almost twice as big as that for the JCG (Jacobi preconditioned conjugate gradient) method.

### **3.4 Required Storage with PCG Solvers**

In the finite element method (FEM), storage for the element level stiffness matrices is well defined in that one can predict the size according to the element type used (i.e. the size of the element stiffness) and the number of elements in the model. In addition symmetry in element level stiffness matrices enables further savings in storage. Typically, the storage of all the element level stiffness matrices requires far less memory than that of the assembled global stiffness matrix in skyline, profile, or banded form.

Since there are no elements in meshfree methods, the nodal stiffness matrix assumes a role

analogous to that of the element level stiffness matrix in FEM. The nodal stiffness matrix stores the contributions to the global stiffness matrix for any two coupled nodes. To illustrate, consider the simple one-element four-node model shown in Figure 3.1 in which each node has two degrees of freedom. In FEM, the element level stiffness matrix is symmetric, and thus only the upper triangle needs to be stored. The size of the element level stiffness matrix is  $(NEE \times (NEE + 1)) / 2 = 36$  words where  $NEE = NEN \times NDOF$  is the number of element equations (8). If the very simple model in Figure 3.1 were treated as a meshfree model, each contribution to the global stiffness matrix by two coupled nodes would be given by  $\mathbf{K}_{ab}$  where a and b indicate specific nodes. The sixteen nodal stiffness matrices for this model are:  $\mathbf{K}_{aa}, \mathbf{K}_{ab}, \mathbf{K}_{ac}, \mathbf{K}_{ad}, \mathbf{K}_{ba}, \mathbf{K}_{bb}, \mathbf{K}_{bc}, \mathbf{K}_{bd}, \mathbf{K}_{ca}, \mathbf{K}_{cb}, \mathbf{K}_{cc}, \mathbf{K}_{cd}, \mathbf{K}_{da}, \mathbf{K}_{db}, \mathbf{K}_{dc}, \mathbf{K}_{dd}$ , and except for  $\mathbf{K}_{aa}, \mathbf{K}_{bb}, \mathbf{K}_{cc}, \mathbf{K}_{dd}$  they are generally nonsymmetrical unlike element level stiffness matrices in FEM. However it is not necessary to store the entire set of nodal stiffness matrices, since  $\mathbf{K}_{ij}$  is equal to  $\mathbf{K}_{ji}^T$  where  $i$  and  $j$  are node numbers of coupled nodes. Consequently, the minimum set of nodal stiffness matrices to be constructed and stored is:  $\mathbf{K}_{ab}, \mathbf{K}_{ac}, \mathbf{K}_{ad}, \mathbf{K}_{bc}, \mathbf{K}_{bd}, \mathbf{K}_{cd}$  as well as the upper (or lower) triangular parts of  $\mathbf{K}_{aa}, \mathbf{K}_{bb}, \mathbf{K}_{cc},$  and  $\mathbf{K}_{dd}$ . The cumulative size of all the nodal stiffness matrices here would be  $6 NDOF^2 + 4 (NDOF \times (NDOF + 1)) / 2 = 36$  words which is exactly the same as that for the stiffness matrix in sparse form without fill-in. In general, the memory  $S$  required to store the nodal stiffness matrices in a meshfree model would be computed as follows

$$S = \sum_{k=1}^N \frac{1}{2} [IP_k \times NDOF \times (NDOF + 1)] = \frac{1}{2} NDOF \times (NDOF + 1) \sum_{k=1}^N IP_k \quad (3.6)$$

where  $N$  is the number of unrestrained nodal points in the model and  $IP_k$  is number of nodes coupled with the  $k^{\text{th}}$  node. In the current example, each of the three nodes has three coupled nodes yielding  $S = 36$  words.

## 4 PARALLEL SOLUTION SCHEMES

### **4.1 Paradigms and Overview**

Many recently developed structural analysis programs take full advantage of parallelism. Those codes in which parallelism is considered from the outset of development can take greatest advantage of parallelism, with the capacity to use a large number of heterogeneous CPUs with mesh partitioning, and message passing. On the other hand when codes that were initially developed to run serially are made parallel, the shared memory paradigm with a relatively small number of CPUs and yet significant gains in performance can be achieved with relative ease

Message passing is used in parallel finite element solution methods based on domain decomposition approaches [12]. The finite element domain is decomposed into a set of sub-domains, and each of these is assigned to processor. It is critical to balance the computational load among processors while also minimizing the number of partition interfaces. Toward this end, software packages like “Metis” [22] are commonly used to provide finite element mesh partitions of nearly equal size that achieve load balancing. Danielson et al. [11] presented a parallel computational implementation of meshfree methods for explicit dynamic analysis. Several partitioning schemes were considered and explicit MPI message passing statements were used for all communications among partitions on different processors.

Many compiler directives and libraries have been developed for parallel computing. Among them MPI and OpenMP are widely used due to portability. OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism. It is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel application. Message Passing Interface (MPI) is a library specification for message passing. It is a practical, portable,

efficient, and flexible standard for message passing in a distributed memory communication environment. Parallel programming languages such as “Charm++” [20,21] also provide portability, dynamic load balancing, and many advanced features.

## 4.2 Parallel JCG Solver

The procedure of forming incremental force in conjugate gradient algorithm is parallelized as shown in Box 4.1. When all nodal contributions are assembled into a global vector, simultaneous actualization of the global vector by many different processors can cause waiting time by these processors, leading to a reduction in the parallel speed-up rate. To avoid this concurrency, each iteration of the outer loop in Box 4.1 is responsible for generating specific entries in the vector  $\mathbf{A}\cdot\mathbf{g}$ . This procedure for a simple four-node model with three degrees of freedom per node is illustrated in Figure 4.1. One iteration of the inner loop of Box 4.1 computes the product of nodal stiffness matrix  $\mathbf{A}_{IJ}$  and localized nodal displacement  $\mathbf{g}_J$  for each neighbor node, and one iteration of the outer loop assembles three rows of  $\mathbf{A}\cdot\mathbf{g}$ . [OpenMP – the specification for shared memory parallelism - is employed to parallelize this algorithm and an HP Exemplar S-class 16 processor computer is used to test the parallel performance. For wall-clock timing of processes, Fortran 90 offers an intrinsic function “call date\_and\_time”, with a resolution of one millisecond.] Although this parallel algorithm is based on inner loop-level parallelism in order to avoid “critical sections” that can cause significant processor waiting time, the workload of the inner loop cannot be perfectly balanced among the processors and some processors will typically be idle while  $\mathbf{s} = \mathbf{A}\cdot\mathbf{g}$  is computed for the last few nodes. This idle time will be proportional to the maximum number of coupled nodes for the last few nodes processed in each conjugate gradient iteration and naturally accumulates with the number of required conjugate gradient

iterations. Consequently, this algorithm will not show ideal scalability.

The inner loop in the algorithm above is typically not used (at least by the authors) when using conjugate gradient solvers with FEM. Instead, “critical sections” are used to avoid updating the global vector simultaneously. The total number of element stiffness matrices, which are much larger than nodal stiffness matrices, is much less than the number of nodal stiffness matrix in the same size model, and therefore the number of procedures to assemble global vector in FEM is less than that of meshfree methods. As a result, the effect of “critical sections” tends not to have too deleterious an effect on the scalability of parallel conjugate gradient FEM computations.

### **4.3 Parallel IC Factorization**

The parallel Cholesky factorization method for shared memory machines was discussed in [14,16] while for distributed memory machines it was presented in [4]. Since it is easy to implement with the data structures described in Section 3, the IC factorization algorithm implemented and tested in this research is that based on the submatrix-Cholesky method, which is just one of three forms of Cholesky decomposition methods illustrated in Figure 4.2 [14]. The “Modified region” in Figure 4.2 is updated in parallel, and its procedure corresponds to the parallel inner loop in the algorithm of Box 3.2. As such, parallelism of IC factorization is easily obtained avoiding dependency that could exist if the outer loop were parallelized. In general the CPU waiting time in inner loop parallelization is longer than outer loop parallelization, therefore speed-up is less than ideal as increasing numbers of processors are used. It is noted that outer loop parallelism can be used [23] resulting in reduced processor waiting time, but this is dependent upon sophisticated reordering. Due to the high degree of nodal coupling in meshfree

methods, it is not clear that such reordering would make outer loop parallelism in the IC factorization process more efficient than the inner loop parallelism used.

## 5. NUMERICAL EXAMPLES

### **5.1 The Boussinesq Problem**

The Boussinesq problem involves application of a point load to a linear elastic half-space. The analytical solution of displacement beneath point load is singular and can be expressed by the following equation.

$$u_z = \frac{P_z}{2\pi\mu R} \left[ (1-\nu) + \frac{z^2}{2R^2} \right] \quad (5.1)$$

where R the distance from point of load application. The simple material model and uniform mesh for a cube form a well-conditioned problem that is ideal for testing the upper limit of PCG performance. Beyond these factors, the spectrum of the solution field for this problem is dominated by very high wavenumbers (or very short wavelengths), and iterative solvers such as the conjugate gradient method tend to converge very rapidly for such problems as they first obtain the high wavenumber contributions to a solution spectrum, followed subsequently by the lower wavenumber contributions [3,6]. By first studying the comparative performance of different solvers on this ideal problem for iterative solvers, convergence rates and crossover points where iterative solvers begin to outperform direct solvers can be identified. The problem described in Figure 5.1 is solved with six different levels of model refinement having discretizations of 3×3×3, 5×5×5, 9×9×9, 11×11×11, 13×13×13, and 17×17×17 equally spaced nodes.

Using the sparse direct solver VSS [33] the computational costs and solution accuracies associated with both meshfree (with normalized support size a=1) and FEM (with trilinear shape

functions) are compared in Table 5.1 and Figure 5.2. When accuracy of solutions is considered, meshfree analysis is actually very competitive. Table 5.2 compares storage required to solve the linear problem  $\mathbf{K} \cdot \mathbf{u} = \mathbf{f}$  with the sparse direct solver (VSS), the JCG solver, and the ICCG solver for normalized support sizes of  $a=1$ ;  $a=2$ ; and  $a=3$ . The VSS solver uses a very effective reordering algorithm that minimizes the fill that occurs during factorization, thereby minimizing both the required memory and number of processor operations for the solving process. Nevertheless, analysis of the results in Table 5.2 indicates that the required memory with VSS grows in proportion to  $N^{1.5}$  as the problem size  $N$  increases whereas the required memory for the JCG and ICCG solvers grows in proportion to  $N^{1.1}$  and  $N^{1.3}$ , respectively. For a given problem size, the required memory for the ICCG solver is greater than that for the JCG solver since both require the equivalent of the sparse matrix  $\mathbf{K}$  but the ICCG solver also requires the factorized  $\mathbf{K}$  without fill in. For small and moderate sized problems, the ICCG solver actually requires more memory than the VSS solver, since the storage required for two sparse matrices without fill-in can exceed that of a sparse-matrix with fill-in. However, with increasing problem size, the trend is clearly for the ICCG solver to use substantially less memory than the VSS solver.

With normalized support sizes of  $a=1$  and  $a=2$ , the JCG solver becomes faster than VSS on the Boussinesq problem for problems, respectively, greater in size than  $7 \times 7 \times 7$  nodes and  $16 \times 16 \times 16$  nodes. However, for normalized support sizes of  $a=3$  the crossover point at which the JCG solver overtakes the direct VSS solver was not observed due to the fact that with a support size of  $a=3$  the stiffness matrix of the problem becomes extremely ill-conditioned greatly slowing down the rate of convergence within the jacobi-preconditioned conjugate gradient algorithm. While such a crossover point should exist for sufficiently large problem sizes, it is not worth pursuing, due to the slow convergence rate of the JCG algorithm. The ICCG solver provides a much faster rate of convergence than the JCG solver especially when normalized

support sizes of  $a=2$  or  $a=3$  are used. As shown in Table 5.2 and Figure 5.3, the ICCG solver begins to outperform the VSS solver in terms of speed on the Boussinesq problem with support size  $a=2$  for a problem of approximately  $NEQ=10^4$  equations, and extrapolating the results of Figure 5.3c, the ICCG solver would begin to outperform the VSS solver with support size  $a=3$  for problem sizes well before  $NEQ=10^5$ .

Both the JCG and ICCG solving algorithms were parallelized for shared memory environment and tested with small number of CPUs ( $\leq 12$ ) for this example. The computations were performed on an HP Exemplar S-class 16 processor computer, and a normalized support size of  $a=2$  was used. The speed-up factors associated with parallelism are shown in Tables 5.3 and 5.4 and while they are reasonably good, they are a little less than ideal as increasing numbers of CPUs are used. It is especially noteworthy that the parallel speed-up factors with meshfree methods diminish somewhat with increasing problem sizes. This is believed by the authors to be due in part to the increase of idle time during stiffness and force assembly caused by increasing numbers of neighbor nodes with increasing problem size as was mentioned in Section 4.2. The Boussinesq problem was also solved by FEM with a parallel JCG solver for comparison of speed-up factors between two methods. As shown in Figure 5.4(a) almost perfect scalability is obtained for sufficiently large FEM problems because the proportion of idle cpu-time decreases with increasing problem size, while unlike the proposed implementation of the meshfree method, the load associated with each parallel loop iteration stays the same. The parallel ICCG solver with parallelization of the inner loop of the IC factorization was also tested with small number of CPUs in shared memory environment. In general, the occurrence of idle CPUs in inner loop parallelization is more frequent than that with outer loop parallelization. Consequently, speed-up factors shown in Figure 5.4c are lower than ideal.

### **5.2 The 3D Cylinder Problem**

Cylindrical structures subjected to end loadings have been considered by Nour-Omid [26] to assess the performance of various linear solving algorithms. While this problem is essentially two-dimensional, it is solved as a 3D problem as shown in Figure 5.5 to produce larger bandwidth of stiffness matrix. Two different refinements with  $11 \times 21 \times 11$  and  $13 \times 25 \times 13$  equally spaced nodes have been considered. The material is linearly elastic, and a static loading is applied. The results in Table 5.5 and Figure 5.6 share the same tendencies as the preceding results obtained for the Boussinesq problem. When a normalized support size of  $a=1$  is used, both the JCG and ICCG solvers outperform the VSS solver in terms of memory and CPU time. When normalized support size of  $a=2$  is used, the JCG again converges very slowly due to the somewhat ill-conditioned stiffness matrix. Alternatively, the ICCG solver appears superior to both of the other solvers in terms of both memory and cpu-time for this problem.

### **5.3 The 3D Beam Problem**

A cantilever beam test problem is described in Figure 5.7. All the degrees of freedom at the clamped end are fixed, and uniform loading is applied at the tip of beam in the transverse direction. Three cases are considered: a linear elastic beam, an elastic beam with geometric non-linearity, and an elasto-plastic beam with geometric and material non-linearity. The material properties are Young's modulus  $E = 21,000$  MPa, the Poisson's ratio  $\nu = 0.3$ , and elastoplastic isotropic hardening  $\sigma_y(\bar{\epsilon}^p) = 21 + 1000 \bar{\epsilon}^p$  Mpa. The beam is discretized into two levels of refinements with  $6 \times 11 \times 21$  and  $9 \times 17 \times 31$  equally spaced nodes, and analyzed using ten equal loading steps for the nonlinear cases. This example is employed to see how the different conjugate gradient solvers work for nonlinear problems. First, however, the performance of the

JCG and ICCG solvers for this cantilever beam geometry are compared to the performance on the Boussinesq problem (see Table 5.6). While the problem sizes for the 3D linear elastic beam problem are comparable to those of the largest Boussinesq problems solved (Table 5.2), it is seen that a considerably larger number of conjugate gradient iterations are required to solve these problems. Nevertheless, even for though the ICCG solver requires a larger number of iterations in this problem, it still shows a clear trend toward outperforming both the JCG and VSS solvers.

The relative performance of the different solvers on the nonlinear 3D beam problems are summarized in Table 5.7 and Figure 5.8. For both the geometrically nonlinear problem, and the fully nonlinear problem, the magnitude of loading applied to the beam was selected so that the maximal deflection at the tip is approximately one third of the beam length. For the case involving material non-linearity in the form of hardening elasto-plasticity, approximately 79 percent of the nodes in the model undergo plastic deformation. The results in Table 5.7 and Figure 5.8 indicate that both types of nonlinearity, geometric and material, significantly degrade the performance of the JCG solver with the convergence rates for the elasto-plastic problem being worse than those for the nonlinear elastic case. For example, Figure 5.8 (c) shows the performance of the JCG solver as comparable to that of VSS for the linear elastic case, but it becomes 1.4 times slower than VSS with introduction of geometric non-linearity, and 3.8 times slower with further introduction of material non-linearity. The relative performance of the ICCG solver also deteriorates somewhat in solving nonlinear problems, although not as severely as that of the JCG solver. Nevertheless, the ICCG solver remains competitive with the VSS direct solver for large-scale nonlinear problems in terms of CPU time and required memory.

## 6. DISCUSSION and CONCLUSIONS

In this paper two different parallel, pre-conditioned conjugate gradient (PCG) solvers have been implemented and tested in a meshfree analysis framework on both linear and nonlinear continuum mechanics problems. The first iterative solver was based on using the inverted diagonal of the stiffness matrix as a preconditioner (JCG), while the second used the Cholesky factorization of the sparse stiffness matrix neglecting fill-in (ICCG) as the preconditioner. In addition, the high-performance direct sparse solver VSS was also used in this study as a baseline against which to measure the performance of the PCG solvers.

The required memory and cpu-time for in-core meshfree solution of the Boussinesq problem with normalized shape function support size  $a=2$  tends to grow roughly in proportion to  $N^{1.5}$  and  $N^{2.25}$ , respectively, with the VSS solver, in proportion to  $N^{1.1}$  and  $N^{1.72}$ , respectively, for the JCG solver, and in proportion to  $N^{1.3}$  and  $N^{1.52}$ , respectively, for the ICCG solver. Similar relative trends for the three solvers are observed on different test problems, and with different normalized shape function support sizes. It can therefore be stated that above a certain size of analysis problem, PCG solvers will always be more efficient than direct sparse solvers based on complete Cholesky factorization, both in terms required CPU operations and required memory. Moreover by exploiting relatively simple shared-memory parallelism within the PCG solvers, significant additional computational efficiency is easily gained with relatively small numbers of processors. Here the parallel performance of the solvers has been explored with up to 12 processors.

Many of the well-established performance characteristics of iterative PCG solvers within FEM frameworks have been found to be apparent within a meshfree analysis framework as well. In particular, the performance of iterative solvers is quite problem dependent. On problems where the solution field has primarily a high wavenumber content, in a spectral sense, iterative PCG solvers can be extremely efficient and outperform direct solvers even on fairly small sized

applications. The Boussinesq problem solved in Section 5.1 above is a classic example of such a problem. However in problems where the solution field has a significant low wavenumber spectrum, the iterative PCG solvers can require substantially more cpu-effort and iterations. This was demonstrated in the cylinder and beam problems in Sections 5.2 and 5.3 above.

A new aspect of using PCG solvers within a meshfree analysis framework was also identified in this work. The usage of relatively large shape-function support sizes in meshfree methods can result in ill conditioning of the stiffness matrix due to loss of linear independence between shape functions. Iterative PCG solvers are much more sensitive to this ill-conditioning than direct equation solvers. In particular, the rate of convergence of the JCG solver was found to be greatly diminished with larger normalized shape function support sizes such as  $a=3$ . Alternatively, the ICCG solver was found to be significantly more robust and able to deal with this problem. Nevertheless, even with the ICCG solver, the analyst should avoid choosing the normalized support size  $a>2$  without good reason, since doing so will entail more computational effort.

## 7. ACKNOWLEDGEMENTS:

The authors gratefully acknowledge a grant NSF DMS-9874015 that facilitated this research.

## 8. REFERENCES

1. Adams, M. and Taylor, R.L. "Parallel multigrid solvers for 3D-unstructured large deformation elasticity and plasticity finite element problems." *Finite Elements in Analysis and Design* 36 (2000) 197-214
2. Axelsson, O. (1986), "A general incomplete block-matrix factorization method," *Linear Algebra Appl.*, 74,179-190.
3. Axelsson, O. (1994), "Iterative solution methods," Cambridge University Press.

4. Basermann, A., Reichel, B., and Schelthoff, C. (1997), "Preconditioned CG methods for sparse matrices on massively parallel machines," *Parallel Computing*, 23, 381-398.
5. Belytschko, T., Krongauz, Y., Organ, D., and Fleming, M. (1996), "Meshless methods: An overview and recent developments," *Comput. Methods Appl. Mech. Engrg.*, 139, 3-47.
6. Briggs, W.L., Henson, V.E. and McCormick, S.F. *A Multigrid Tutorial*. 2<sup>nd</sup> ed. SIAM, 2000
7. Chen, J. S., Pan, C., Wu, C. T., and Liu, W. K. (1996), "Reproducing kernel particle methods for large deformation analysis of non-linear structures," *Comput. Methods Appl. Mech. Engrg.*, 139, 195-227.
8. Chen, J. S. and Wang, H. P. (2000), "New boundary condition treatments in meshfree computation of contact problems," *Comput. Methods Appl. Mech. Engrg.*, 187, 441-468.
9. Chen, J. S., Wu, C. T., Yoon, S., and You, Y. (2001), "A stabilized conforming nodal integration for Galerkin mesh-free methods," *Int J. Numer. Meth. Engng*, 50, 435-466
10. Concus, P., Golub, G. H., and Meurant, G. (1985), "Block preconditioning for the CG method," *SIAM J. Sci. Stat. Comput.*, 6, 220-252.
11. Danielson, K. T., Hao, S., Liu, W. K., Uras, A., and Li, S. (2000), "Parallel computation of meshless methods for explicit dynamic analysis," *Int J. Numer. Meth. Engng*, 47, 7, 1323-1341.
12. Danielson, K. T. and Namburu, R. R. (1998), "Nonlinear dynamic finite element analysis on parallel computers using FORTRAN 90 and MPI," *Advances in Engineering Software*, 29, 179-186.
13. Farhat, C. and Roux, F. X. (1991), "A method of finite element tearing and interconnecting and its parallel solution algorithm," *Int. J. numer. Meth. Engng*, 32, 1205-1227.
14. George, A., Heath, M. T., and Liu, J. (1986), "Parallel Cholesky factorization on a shared-memory multiprocessor," *Lin. Alg. and Its Applic.*, 77, 165-187.
15. Gerardin, M. and Hogge, M. (1987), "Solving systems of nonlinear equations," H. Kardestuncer (ed.), *The Finite Element Handbook*, McGraw-Hill, New York.
16. Golub, G. H. and Van Loan, C. F. (1996), "Matrix Computations," The Johns Hopkins University Press, Baltimore, MD.
17. Grote, M. J. and Huckle, T. (1997), "Parallel preconditioning with sparse approximate inverses," *SIAM J. Sci. Comput.*, 18, 838-853.
18. Hestenes, M. R. and Stiefel, E. (1952), "Methods of conjugate gradients for solving linear systems," *J. Comput. Appl. Math.*, 25, 153-168.
19. Hughes, T. J. R., Ferencz, R. M., and Hallquist, J. O. (1987), "Large-scale vectorized implicit calculations in solid mechanics on a Cray X-MP/48 utilizing EBE preconditioned conjugate gradients," *Comput. Meth. appl. Mech. Engng*, 61, 215-248.
20. Kale, L. V., Ramkumar, B., Sinha, A. B., and Gursoy, A. (1994a), "The Charm Parallel Programming Language and System: Part I --- Description of Language Features," IEEE

Transactions on Parallel and Distributed Systems.

21. Kale, L. V., Ramkumar, B., Sinha, A. B., and Gursoy, A. (1994b), "The Charm Parallel Programming Language and System: Part II - The Runtime System," IEEE Transactions on Parallel and Distributed Systems.
22. Karypis, G. and Kumar, V. (1995), "A fast and high quality multilevel scheme for partitioning irregular graphs," Technical Report TR 95-035, Department of Computer Science, University of Minnesota.
23. Lin, W. Y. and Chen, C. L. (1999), "Minimum communication cost reordering for parallel sparse Cholesky factorization," *Parallel computing*, 25, 943-967.
24. Liu, W. K., Jun, S., and Zhang, Y. F. (1995), "Reproducing Kernel Particle Method," *Int. J. Numer. Methods Fluids* 20, 1081-1106.
25. Meurant, G. (1984), "The block preconditioned conjugate gradient method on vector computer," *BIT*, 24, 623-633.
26. Nour-Omid, B. (1984), "A preconditioned conjugate gradient method for solution of finite element equations," in: W. K. Liu, T. Belytschko and K. C. Park, eds., *Innovative Methods for Nonlinear Problems*, Pineridge Press, Swansea, U. K., 17-40.
27. Ortega, J. M. (1988), "Introduction to parallel and vector solution of linear systems," Plenum Press New York, London.
28. Pini, G. and Gambolati, G. (1990), "Is a simple diagonal scaling the best preconditioner for conjugate gradients on supercomputers?," *Adv. Water Resour.*, 13, 147-153.
29. Poole, E. L., Knight, N. F., and Davis, D. D. (1992), "High-performance equation solvers and their impact on finite element analysis," *Int J. Numer. Meth. Engng*, 33, 855-868.
30. Storaasli, O. O., Nguyen, D. T., and Agarwal, T. K. (1990), "A parallel-vector algorithm for rapid structural analysis on high-performance computers," NASA Technical Memo, 102614, Hampton, VA.
31. Swan, C. C. and Kosaka, I. (1997), "Homogenization-based analysis and design of composites," *Computers and Structures*, 64, 603-621.
32. Yoon, S (2001), "Stabilized conforming nodal integration for Galerkin meshfree methods," Ph.D. Thesis, University of Iowa.
33. Solversoft, [www.solversoft.com](http://www.solversoft.com) (2002).

## Box 2.1 Global Newton solving algorithm

*Predictor phase*

$v = 0$  : counter initialization  
 $\mathbf{d}_{n+1}^v = \tilde{\mathbf{d}}_{n+1}$  : displacement predictor  
 form  $\mathbf{r}_{n+1}^v(\mathbf{d}_{n+1}^v)$  : initial residual

*Corrector phase*

**while** ( $\|\mathbf{r}_{n+1}^v\| > \text{RTOL}$ ),  
      $\mathbf{K}\boldsymbol{\delta}_v = -\mathbf{r}_{n+1}^v$  : linear solving phase for  $\boldsymbol{\delta}_v$   
      $\mathbf{p}_v = \alpha_v \boldsymbol{\delta}_v$  : line search for step size  $\alpha_v$   
      $\mathbf{d}_{n+1}^{v+1} = \mathbf{d}_{n+1}^v + \mathbf{p}_v$  : displacement update  
     form  $\mathbf{r}_{n+1}^{v+1}(\mathbf{d}_{n+1}^{v+1})$  : residual update  
      $v = v + 1$  : counter update  
**end-while**

Box 2.2 Algorithm for assembling global  $\mathbf{K}$  on a node-by-node basis.

**For** each nodal (integration) point I  
     **For** each member  $J \in N_I$   
         **For** each member  $L \in C_J$   
             Assemble  $\mathbf{K}_{JL} \rightarrow$  global  $\mathbf{K}$   
         **End-for**  
     **End-for**  
**End-for**

## Box 3.1 Algorithm of preconditioned conjugate gradient solver (PCG).

Step 1: Initialize

$$m = 0; \mathbf{x}_0 = \mathbf{0}; \mathbf{s}_0 = \mathbf{b};$$

$$\mathbf{g}_0 = \mathbf{z}_0 = \mathbf{B}^{-1} \cdot \mathbf{s}_0;$$

Step 2: Line search & updates

$$\alpha_m = \frac{\mathbf{s}_m \cdot \mathbf{z}_m}{\mathbf{g}_m \cdot \mathbf{A} \cdot \mathbf{g}_m}; \quad \mathbf{x}_{m+1} = \mathbf{x}_m + \alpha_m \mathbf{g}_m; \quad \mathbf{s}_{m+1} = \mathbf{s}_m - \alpha_m \mathbf{A} \cdot \mathbf{g}_m$$

Step 3: Convergence check

if  $\|\mathbf{s}_{m+1}\| \leq \delta_L \|\mathbf{s}_0\|$ , then return

Step 4: Update conjugate search direction

$$\mathbf{z}_{m+1} = \mathbf{B}^{-1} \cdot \mathbf{s}_{m+1}; \quad \beta_{m+1} = \frac{\mathbf{s}_{m+1} \cdot \mathbf{z}_{m+1}}{\mathbf{s}_m \cdot \mathbf{z}_m}; \quad \mathbf{g}_{m+1} = \mathbf{z}_{m+1} + \beta_m \mathbf{g}_m$$

$$m = m + 1$$

Go to step 2

Box 3.2: Algorithm for incomplete Cholesky factorization with parallel region.

```
for k = 1 : n
    A(k,k) =  $\sqrt{A(k,k)}$ 
    for i = k + 1 : n
        if A(i,k)  $\neq$  0
            A(i,k) = A(i,k)/A(k,k)
        end
    end
    DO_PARALLEL
    for j = k + 1 : n
        for i = j : n
            if A(i,j)  $\neq$  0
                A(i,j) = A(i,j) - A(i,k)A(j,k)
            end
        end
    end
    END_PARALLEL
end
```

} Parallel region



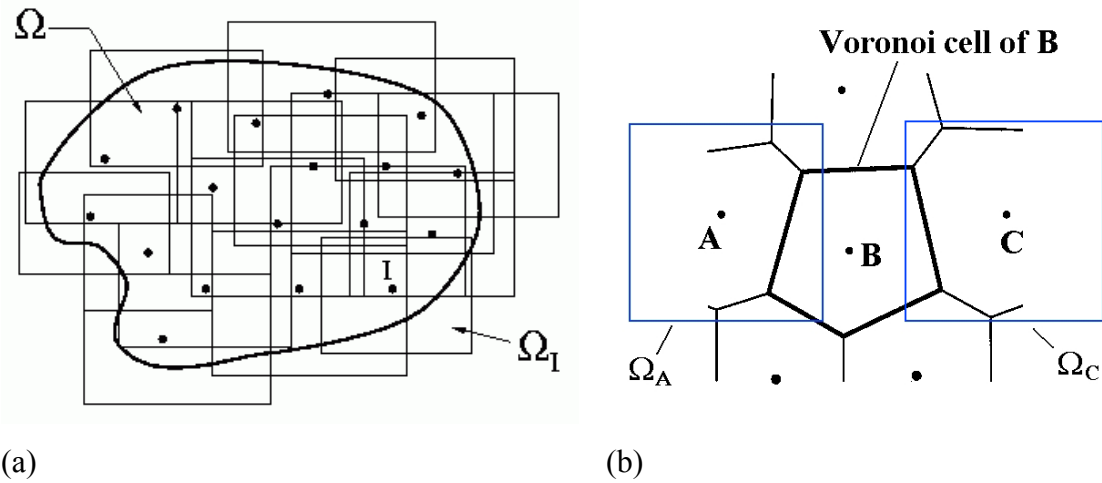


Figure 2.1. Relationship between shape function support size and nodal connectivity (a) node  $I$  and its support  $\Omega_I$ , (b) Voronoi integration cell for a node  $B$ .

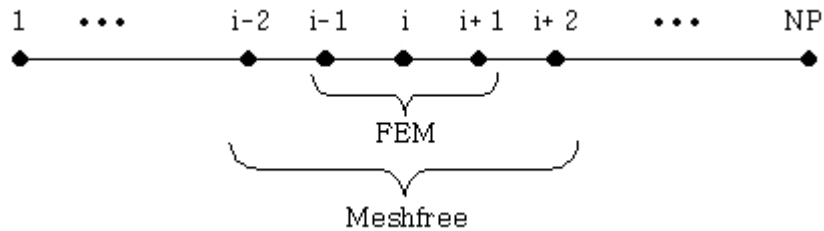


Figure 2.2. Nodal connectivity in one-dimensional equally spaced discretization. With piecewise linear FEM shape functions, each three consecutive nodes are “coupled.” With meshfree shape functions with normalized support size of unity, each five consecutive nodes are “coupled.”

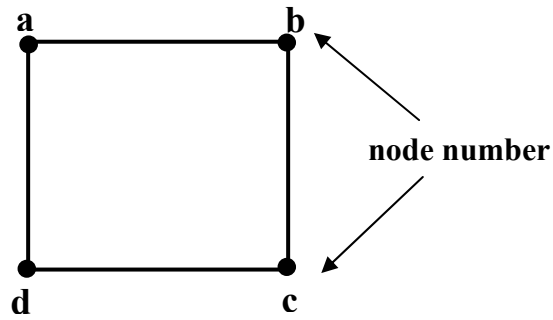


Figure 3.1. Simple four-node model.

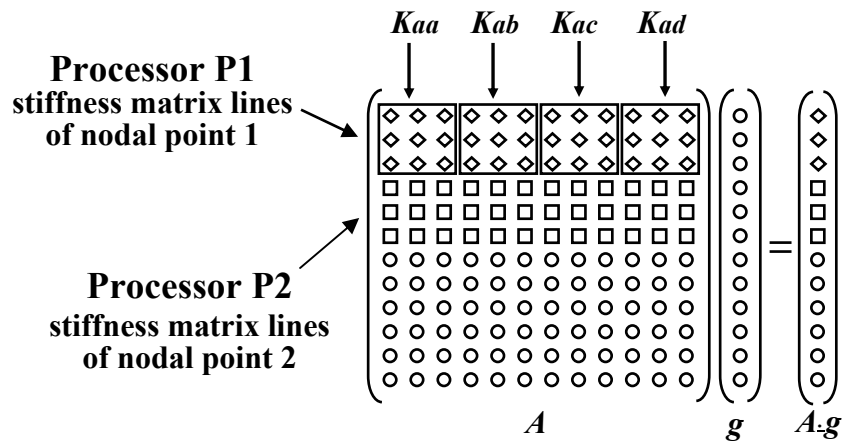


Figure 4.1. The procedure to assemble all nodal contributions into global vector

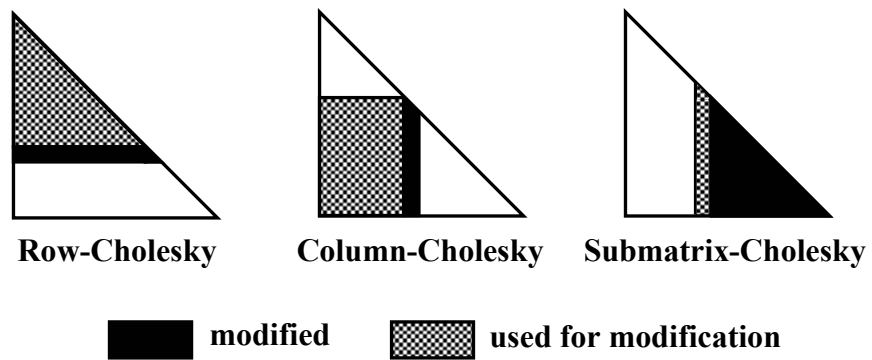


Figure 4.2 Three forms of Cholesky factorization.

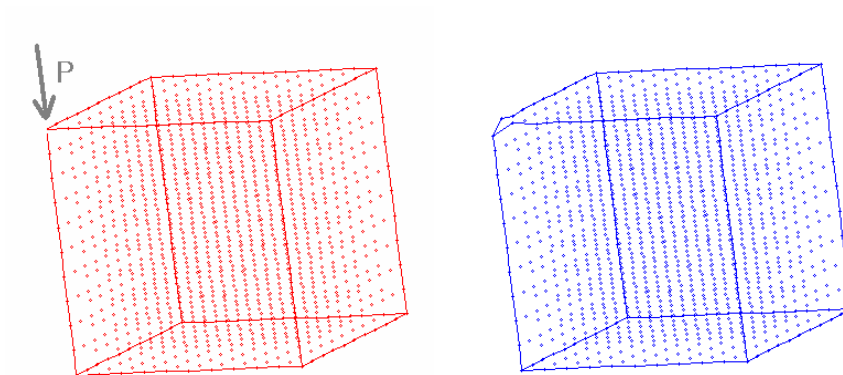


Figure 5.1 Problem description of a Boussinesq problem; Length = 16,  $P = 500$ ; Young's modulus:  $E = 30,000$ ; Poisson's ratio:  $\nu = 0.3$ .

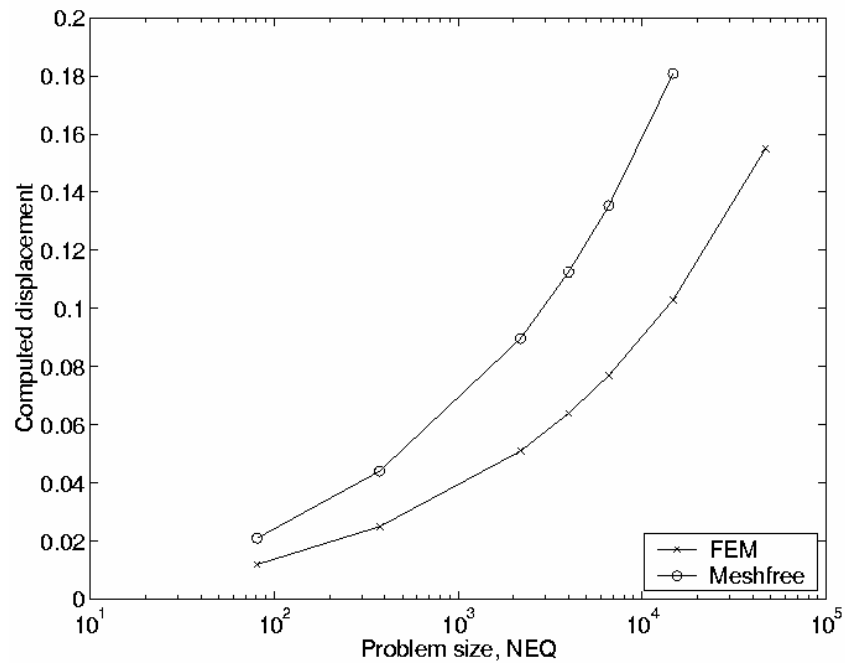


Figure 5.2 Comparison of solution accuracy and problem size ( $NEQ$ ) for meshfree methods and FEM with tri-linear basis function, showing that for a given problem size the meshfree method is much more accurate than FEM

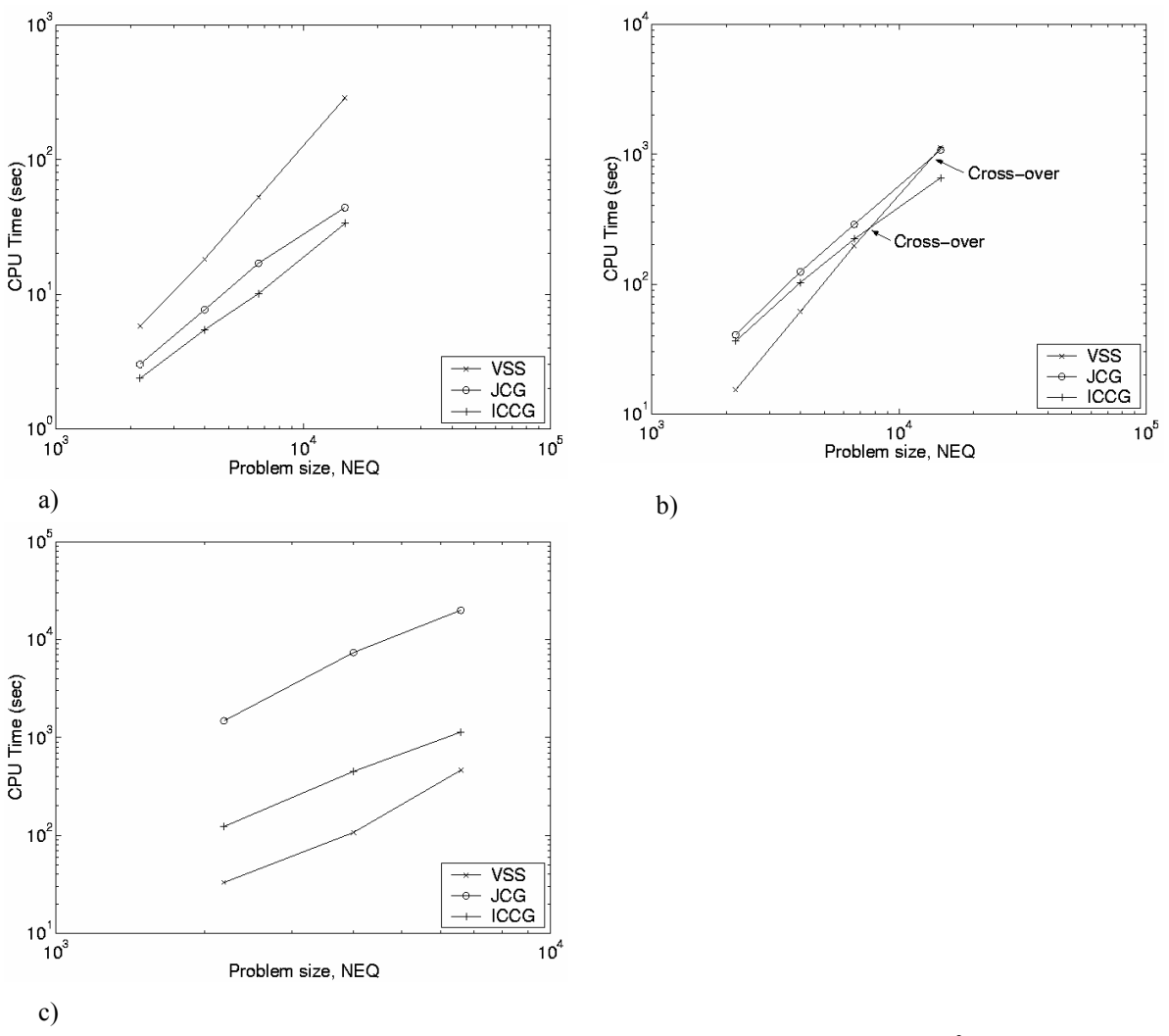


Figure 5.3 CPU time comparison for the Boussinesq problem with  $NEQ > 10^3$  (VSS vs. JCG vs. ICCG) (a) support size  $a=1$ ; (b) support size  $a=2$ ; (c) support size  $a=3$ . Computations were performed on HP S-class processors.

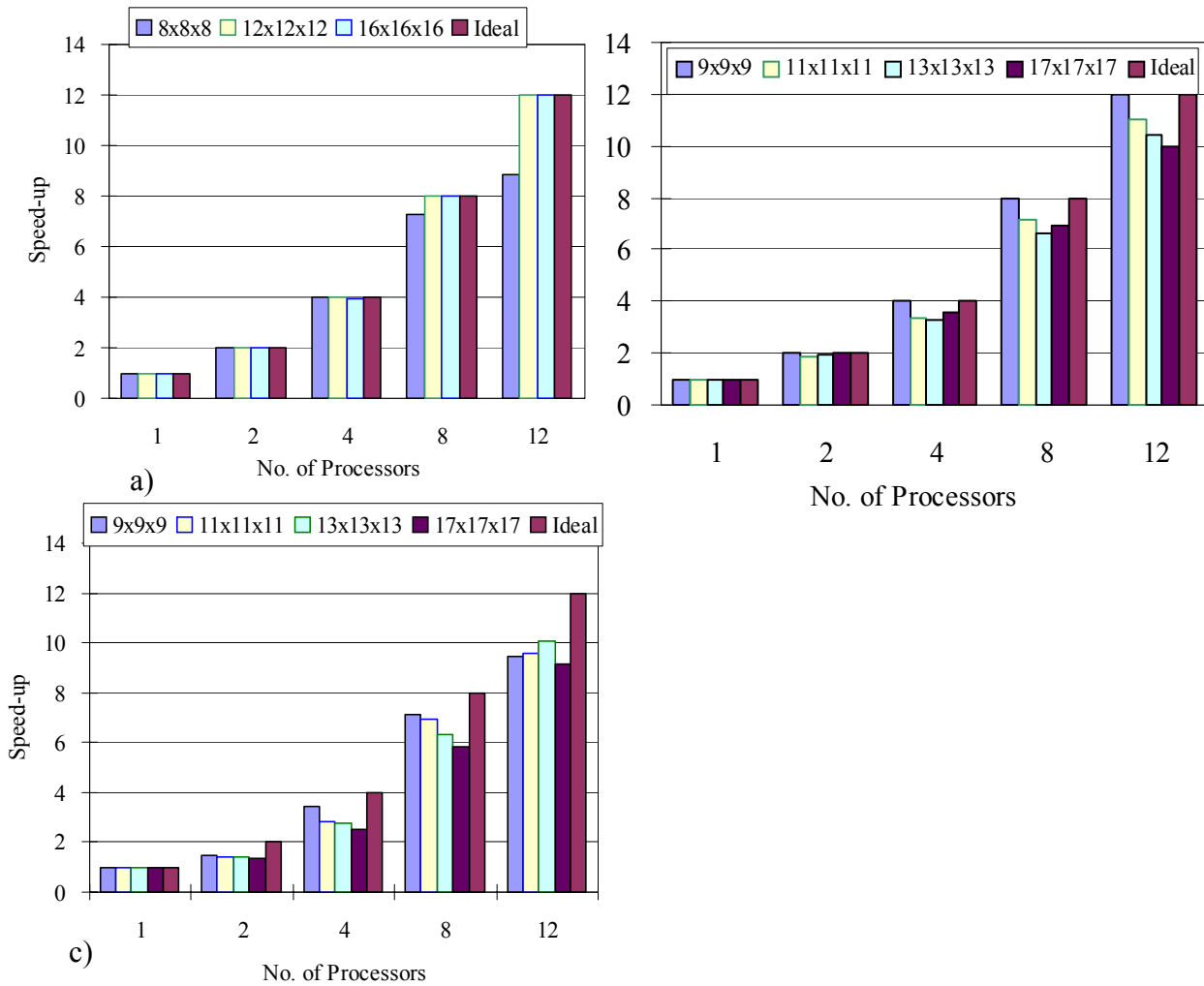


Figure 5.4. Speed-up factors for different problem sizes and solving algorithms; a) FEM with JCG solver; b) Meshfree with JCG solver; c) Meshfree with ICCG solver.

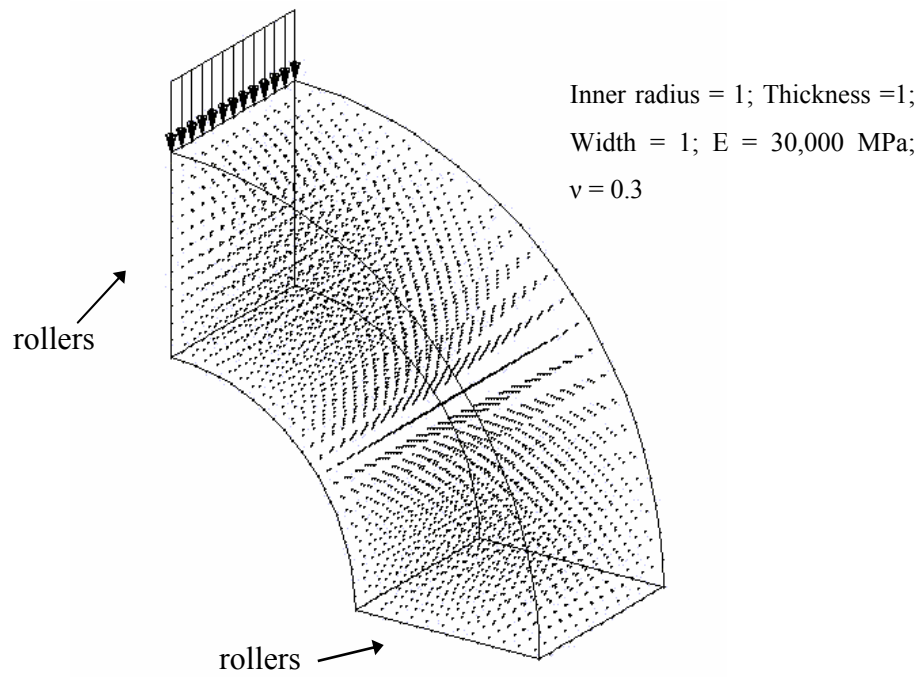


Figure 5.5 Description of 3D cylinder problem.

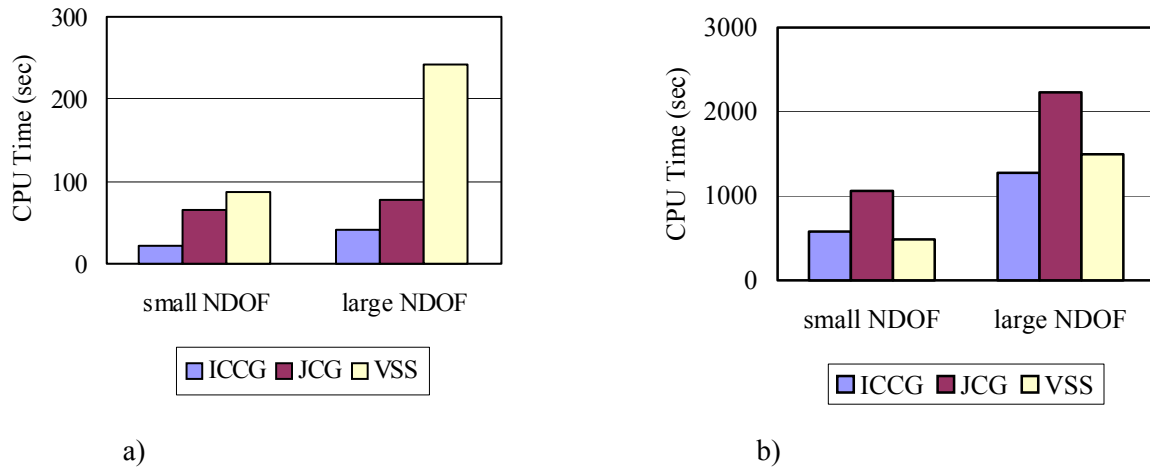


Figure 5.6 CPU time comparison on HP J-class processors for the 3D cylinder problem (VSS vs. JCG vs. ICCG) (a) support size  $a=1$  (b) support size  $a=2$ .

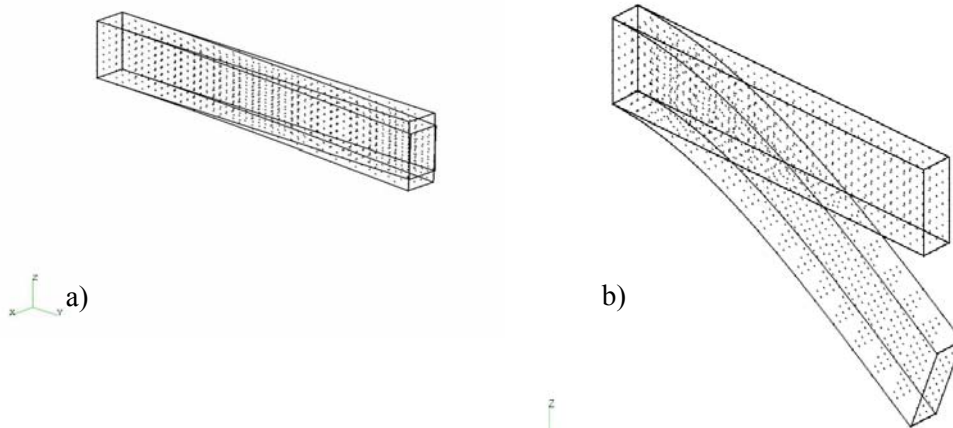


Figure 5.7 Problem description of 3D beam problem (a) small deformation (linear elastic) (b) large deformation (geometric and material nonlinear). Length = 12, Thickness = 2, Width = 1, Young's modulus:  $E = 21,000$  MPa; Poisson's ratio:  $\nu = 0.3$ ; Hardening modulus:  $H = 1000$  MPa; Mixed hardening parameter:  $\beta = 1$ ; Initial yield stress:  $^0Y = 21$  MPa.

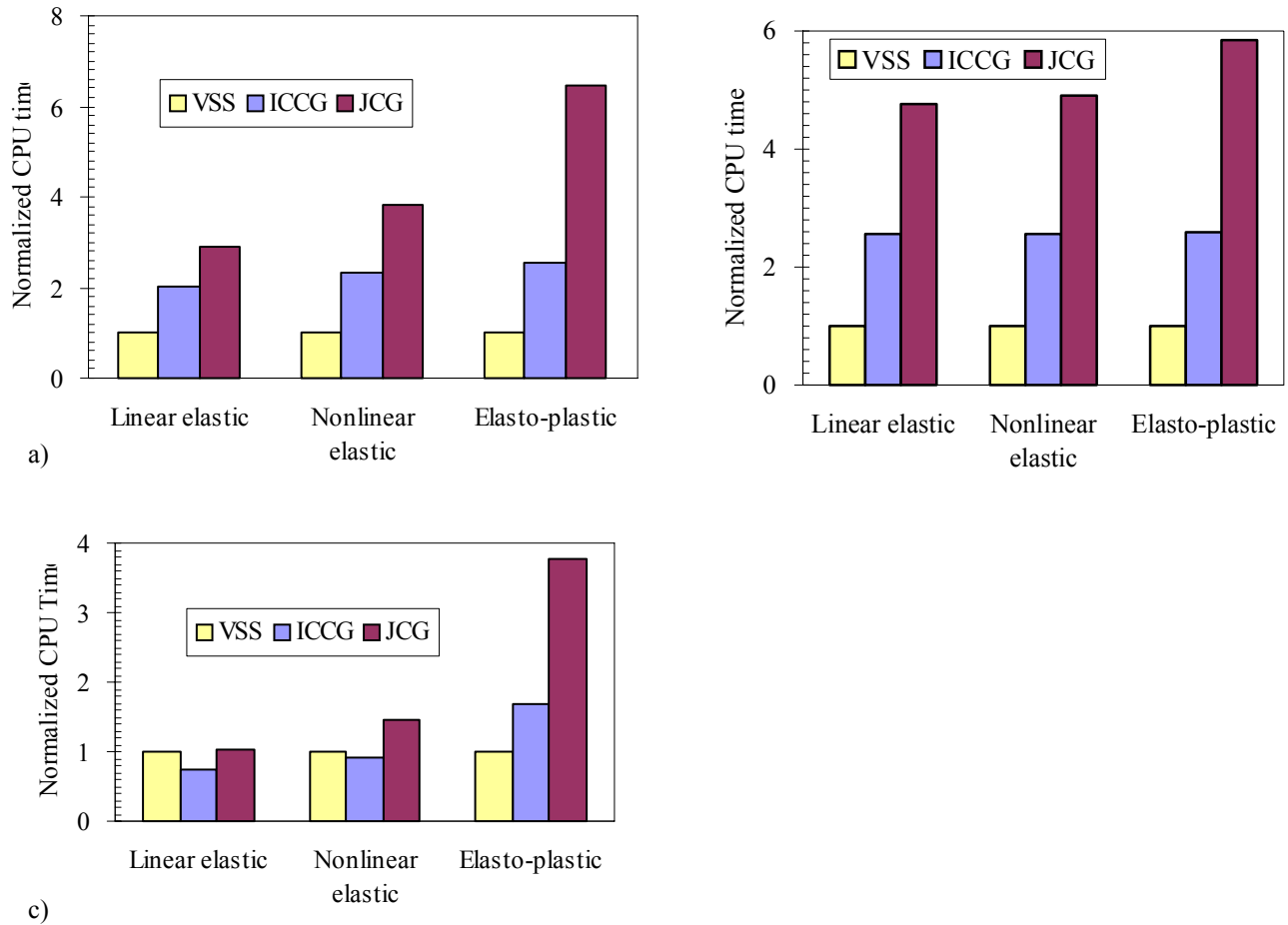


Figure 5.8. CPU time comparison for three cases of the 3D beam problem (VSS vs. JCG vs. ICCG): linear elastic, nonlinear elastic, and elasto-plastic (a) small NDOF, support size  $a=1$ ; (b) small NDOF, support size  $a=2$ ; (c) large NDOF, support size  $a=1$ .

Table 5.1 Comparison of solution accuracy and cost between meshfree methods and FEM with a direct solver for the Boussinesq problem. (Computations performed on HP S-class processors.)

Refinement	Meshfree		FEM	
	CPU (sec)	$\delta u$	CPU (sec)	$\delta u$
3x3x3	.010	-.0208	.024	-.012
5x5x5	.09	-.0441	0.18	-.025
9x9x9	5.8	-.0898	.94	-.051
11x11x11	18.2	-.113	2.81	-.064
13x13x13	52.6	-.135	8.47	-.077
17x17x17	286	-.181	45.4	-.103
25x25x25	-	-	384	-.155

Table 5.2 CPU time and number of iterations in VSS, JCG, and ICCG for Boussinesq problem. (Computations performed on HP S-class processors.)

Size	NEQ	Support size	VSS		JCCG			ICCG		
			Mem. (Mw)	cpu (sec)	Mem. (Mw)	cpu (sec)	Iter.	Mem. (Mw)	cpu (sec)	Iter.
3x3x3	81	1	0.003	0.01	0.003	0.02	20	0.006	0.02	2
		2	0.003	0.01	0.003	0.02	21	0.006	0.03	2
		3	0.003	0.02	0.003	0.03	31	0.006	0.03	2
5x5x5	375	1	0.043	0.09	0.030	0.14	31	0.060	0.15	8
		2	0.054	0.14	0.048	0.39	64	0.096	0.45	7
		3	0.063	0.18	0.058	2.52	351	0.12	0.47	7
9x9x9	2,187	1	0.92	5.8	0.26	3.02	51	0.52	2.37	12
		2	1.45	15.4	0.79	40.66	201	1.58	36.41	9
		3	1.64	33	1.35	1481	>4000	2.7	123.4	11
11x11x11	3,993	1	2.2	18.2	0.51	7.67	60	1.02	5.45	15
		2	4.1	61.4	1.8	124.5	265	3.6	102.9	10
		3	5.2	107	3.5	7374	>8000	7.0	450.3	11
13x13x13	6,591	1	4.8	52.6	0.89	17.0	69	1.78	10.1	16
		2	9.5	197	3.39	288	322	6.78	222	11
		3	13.6	464	7.24	>10 <sup>4</sup>	>10 <sup>4</sup>	14.5	1137	17
17x17x17	14,739	1	16.6	285.9	2.19	44.1	60	4.38	33.5	17
		2	36.0	1115.	9.06	1074	416	18.1	658	12

Table 5.3 Speedup of total solving time in JCG solver for the Boussinesq problem with support size = 2.

Size	No. of Processors				
	1	2	4	8	12
9x9x9	1.00 (0.92)	2.03 (0.84)	3.18 (0.71)	4.82 (0.52)	4.98 (0.43)
11x11x11	1.00 (0.92)	1.73 (0.87)	2.90 (0.79)	4.59 (0.59)	6.21 (0.52)
13x13x13	1.00 (0.93)	1.85 (0.87)	2.86 (0.80)	4.76 (0.67)	6.50 (0.58)
17x17x17	1.00 (0.94)	1.86 (0.89)	3.11 (0.82)	5.02 (0.68)	6.38 (0.60)

Note: Time in parentheses is % of solving time in parallel region

Table 5.4 Speedup in parallel region of JCG solver for the Boussinesq problem with support size a =2.

Size	No. of Processors				
	1	2	4	8	12
9x9x9	1.00	2.00	4.00	8.00	11.91
11x11x11	1.00	1.83	3.38	7.17	11.00
13x13x13	1.00	1.97	3.31	6.60	10.46
17x17x17	1.00	1.97	3.56	6.93	10.01

Table 5.5 CPU time, memory, and no. iterations with VSS, JCG, and ICCG solvers for the 3D cylinder problem. (Computations performed on HP J-class processors.)

Size	Support size	VSS		JCG			ICCG		
		Memory (Mw)	CPU (sec)	Memory (Mw)	CPU (sec)	Iter.	Memory (Mw)	CPU (sec)	Iter.
11x11x21 NEQ=7,623	1	6.74	87.6	1.18	66.4	212	2.36	22.9	38
	2	15.66	486.1	6.01	1060.0	674	12.02	579.7	84
13x13x25 NEQ=12,675	1	14.32	241.2	2.07	78.6	144	4.14	41.1	41
	2	36.3	1494.8	11.06	2232.1	731	22.12	1275.1	127

Table 5.6 CPU time, memory, and no. iterations with VSS, JCG, and ICCG solvers for linear elastic 3D beam problem. (Computations performed on HP J-class processors.)

Size	Support size	VSS		JCG			ICCG		
		Memory (Mw)	CPU (sec)	Memory (Mw)	CPU (sec)	Iter.	Memory (Mw)	CPU (sec)	Iter.
6x11x21 NEQ=4158	1	1.74	10.2	0.50	29.7	259	1.00	20.5	118
	2	3.71	46.6	1.94	221.9	465	3.88	119.4	85
9x17x31 NEQ=14,229	1	12.98	177.1	1.96	184.3	377	3.92	131.6	172
	2	29.48	875.0	9.01	1837.5	816	18.02	699.9	88

Table 5.7 CPU times<sup>a</sup>, memory, and no. iterations with VSS, JCG, and ICCG solvers for nonlinear 3D beam problems.

Size	Material	Support Size a	VSS		JCG			ICCG		
			Mem. (Mw)	CPU (s·10 <sup>3</sup> )	Mem. (Mw)	CPU (s·10 <sup>3</sup> )	Iter.	Mem. (Mw)	CPU (s·10 <sup>3</sup> )	Iter.
6x11x21	NE*	1	1.74	.354	0.5	1.344	333	1.00	8.201	132
		2	3.71	1.699	1.94	8.336	481	3.88	4.349	82
	EP**	1	1.74	4.929	0.5	31.918	567	1.00	12.590	148
		2	3.71	20.358	1.94	119.035	602	3.88	52.730	98
9x17x31	NE	1	12.98	6.922	1.96	10.007	490	3.92	6.274	219
	EP	1	12.98	24.325	1.96	92.015	905	3.92	40.732	227

Note: Number of iteration is averaged value per one solving.

<sup>a</sup> Measured on HP J-class workstations.

\*Nonlinear Elastic; \*\*Elasto-plastic