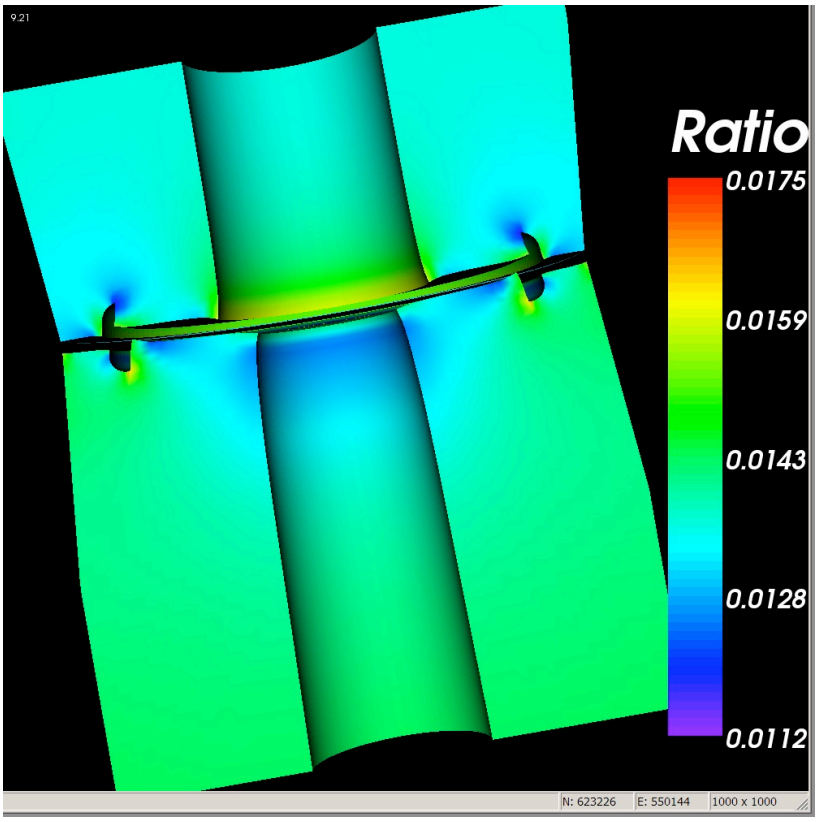
ROCSOLID Developer's Guide

Version 3.4
March 17, 2005
Ali Namazifard



Center for the Simulation of Advanced Rockets
University of Illinois
Urbana, Illinois

TABLE OF CONTENTS

LIST OF FIGURES

C H A P T E R 1

INTRODUCTION

ROCSOLID is an implicit finite element code to solve very large scale, 3D structural mechanics problems subjected to static and dynamic loads. ROCSOLID is under continuing development as a research code and has the following capabilities:

- A small-strain plasticity model based on rate independent von Mises material model with the associated flow rule utilizing a bilinear uniaxial material response in addition to linear elasticity exists in this version of ROCSOLID.
- Eight node hexahedral elements are used. This element is designed for the small-strain plasticity model and uses the consistent material tangent operator (to preserve the quadratic convergence rate when the Newton nonlinear solver is called). The B-bar integration rule is used to avoid locking for near incompressible conditions.
- Dynamic problems are solved using the implicit Newmark time integrator.
- The linear matrix equations encountered at each time step are solved using a scalable parallel multigrid solver. The preconditioned conjugate gradient method is used as the relaxation scheme. All of the main components in the multigrid solver are implemented in an element-by-element framework. Matrix free element level computations are used to reduce the storage and CPU time.

- During each multigrid solution, interprocessor communications are performed in matrix-vector multiplications, scalar products and fine-to-coarse mesh restrictions. Nonblocking MPI communication routines are used to perform point-to-point communications.
- ALE formulation is used for moving interfaces.
- Mixed-enhanced elements are used for shell elements.
- ROCSOLID uses unstructured meshes. *Truegrid* is used to produce a sequence of nested, uniformly refined hexahedral meshes. Mesh partitioning is performed on the coarsest mesh using *Metis* to achieve perfect load balance between the processors. Uniform refinement of the coarsest mesh partitions produces the required partitions on all of the fine meshes. Thus, perfect element load balance is maintained through the mesh hierarchy, although the resulting communication pattern may not be optimum. The files generated by *Truegrid* are processed by another software to generate the necessary input files for ROCSOLID.

The following chapters describe algorithms and features of ROCSOLID, organization of the code, data structure, interface with GEN* and implementation details.

CHAPTER 2

NONLINEAR STRUCTURAL DYNAMICS

2.1 Solution Algorithm for Nonlinear Dynamics Problems

The weak formulation of equilibrium equations (virtual work statement) expressed on the current configuration is given by

$$\int_V \delta \boldsymbol{\varepsilon}^T \boldsymbol{\sigma}\, dV - \int_V \delta \mathbf{u}^T \mathbf{b}\, dV - \int_\Gamma \delta \mathbf{u}^T \mathbf{t}\, d\Gamma = 0, \tag{2.1}$$

where $\delta \boldsymbol{\varepsilon}$ and $\boldsymbol{\sigma}$ are the virtual rate of deformation vector and the Cauchy stress vector respectively, $\delta \mathbf{u}$ is the virtual displacement field, $\mathbf{b}$ is the body force vector per unit volume in the deformed configuration (which may well include the acceleration effects), $\mathbf{t}$ defines the tractions applied to the surface of the deformed model. V and Γ are the volume and the external surface at the current configuration. Inertial D'Alembert forces arising from accelerations are

$$\mathbf{b} = -\rho\, \ddot{\mathbf{u}} \tag{2.2}$$

where ρ and $\mathbf{u}$ are the mass density and displacement field in the deformed configuration, respectively. Discretization of the structure using the standard finite element method [20, 21] yields a set of equations of motion of the form

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{C}\dot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{f}\,, \tag{2.3}$$

where $\boldsymbol{M}$ is the mass matrix, $\boldsymbol{C}$ is the damping matrix and $\boldsymbol{K}$ is the stiffness matrix of the structure. The time dependent vectors $\boldsymbol{f}$ and $\boldsymbol{u}$ represent the external load applied to the structure and the resulting displacement response measured at the degrees-of-freedom of the model, respectively. The product $\boldsymbol{Ku}$ is the internal force at the current configuration and is given by

$$P(\boldsymbol{u}) = \boldsymbol{Ku} = \sum_e \int_{V^e} \delta \boldsymbol{\varepsilon}^{eT} \boldsymbol{\sigma}^e \, dV = \sum_e \int_{V^e} \boldsymbol{B}^{eT} \boldsymbol{\sigma}^e \, dV \,, \tag{2.4}$$

where the summation symbol implies a standard assembly process, the superscript e denotes element quantities, and $\boldsymbol{B}^e$ is the element strain-displacement operator.

Researchers have developed several explicit and implicit methods to solve equation (2.3) [1], all of which compute $\boldsymbol{u}$ at a sequence of time intervals Δt apart by making some assumptions regarding the changes in $\boldsymbol{u}$ and its derivatives over the time step. To establish an implicit formulation we adopt the Newmark method [19].

2.1.1 Newmark's Method Applied to the Governing Equations

This method is essentially a family of methods that are extensions of the linear acceleration method (which assumes acceleration is linear in each time step). We can write the velocity and displacement vectors at time $t + \Delta t$ as

$$\dot{\boldsymbol{u}}_{t+\Delta t} = \dot{\boldsymbol{u}}_t + \left[(1-\gamma)\ddot{\boldsymbol{u}}_t + \gamma \ddot{\boldsymbol{u}}_{t+\Delta t}\right]\Delta t \tag{2.5}$$

and

$$\boldsymbol{u}_{t+\Delta t} = \boldsymbol{u}_t + \dot{\boldsymbol{u}}_t \Delta t + \left[(1/2-\beta)\ddot{\boldsymbol{u}}_t + \beta \ddot{\boldsymbol{u}}_{t+\Delta t}\right]\Delta t^2, \tag{2.6}$$

where the parameters β and γ specify the method. For example, the linear acceleration method is produced by selecting $\gamma = 1/2$ and $\beta = 1/6$. The Newmark method is unconditionally stable when γ is greater than 1/2 and β is greater than $(1/2+\gamma)^2/4$. The most widely used choice is $\gamma = 1/2$ and $\beta = 1/4$, which is unconditionally stable and does not introduce any artificial damping into the solution.

When equation (2.3) is linear, manipulation of Equations (2.3) to (2.6) produces the following time integration algorithm:

i. Given the initial displacement and velocity, compute the initial acceleration from Equation (2.3).

ii. Compute the displacement at time $t+\Delta t$ from

$$\left[\frac{1}{\Delta t^2}\boldsymbol{M}+\frac{\gamma}{\Delta t}\boldsymbol{C}+\beta\boldsymbol{K}\right]\boldsymbol{u}_{t+\Delta t}=\beta\boldsymbol{f}_{t+\Delta t}+\left[\frac{1}{\Delta t^2}\boldsymbol{M}+\frac{\gamma}{\Delta t}\boldsymbol{C}\right]\boldsymbol{u}_t$$
$$+\left[\frac{1}{\Delta t}\boldsymbol{M}+(\gamma-\beta)\boldsymbol{C}\right]\dot{\boldsymbol{u}}_t+\left[(1/2-\beta)\boldsymbol{M}+\frac{\Delta t}{2}(\gamma-2\beta)\boldsymbol{C}\right]\ddot{\boldsymbol{u}}_t\,. \qquad (2.7)$$

iii. Compute the acceleration at time $t+\Delta t$ from

$$\ddot{\boldsymbol{u}}_{t+\Delta t}=\frac{1}{\beta\Delta t^2}\left[\boldsymbol{u}_{t+\Delta t}-\boldsymbol{u}_t\right]-\frac{1}{\beta\Delta t}\dot{\boldsymbol{u}}_t-\left[\frac{1}{2\beta}-1\right]\ddot{\boldsymbol{u}}_t\,. \qquad (2.8)$$

iv. Compute the velocity at time $t+\Delta t$ from Equation (2.5).

v. Advance the time, and repeat steps (ii) to (iv) until the required time history has been computed.

We can now incorporate the above algorithm into a conventional nonlinear solver, e.g., the Newton procedure, to be able to solve equation (2.3) when it is nonlinear. Assuming zero damping, this would yield the following system of linear equations at Newton iteration k to advance the solution from time t to time $t+\Delta t$

$$\left(\frac{1}{\beta\Delta t^2}\boldsymbol{M}+\boldsymbol{K}_T^{(k)}\right)\Delta\boldsymbol{u}^{(k)}=\boldsymbol{f}_{t+\Delta t}-\boldsymbol{i}_{t+\Delta t}^{(k)}-\frac{1}{\beta\Delta t^2}\boldsymbol{M}\boldsymbol{u}_{t+\Delta t}^{(k)}$$
$$+\frac{1}{\beta\Delta t^2}\boldsymbol{M}\boldsymbol{u}_t+\frac{1}{\beta\Delta t}\boldsymbol{M}\dot{\boldsymbol{u}}_t+(\frac{1}{2\beta}-1)\boldsymbol{M}\ddot{\boldsymbol{u}}_t\,. \qquad (2.9)$$

In this equation, $\boldsymbol{K}_T^{(k)}$ is the (consistent) tangent stiffness matrix and $\boldsymbol{i}_{t+\Delta t}^{(k)}$ is the internal forces defined by equation (2.4) for the current iteration and $\Delta\boldsymbol{u}^{(k)}$ is the correction to the displacement increment within the Newton solve. The right hand side represents the force imbalance (residual) from the previous iteration.

Figure 2.1 summarizes the steps involved to advance the solution in implicit nonlinear analysis. Before starting the Newton iterations, the effective load increment and its norm (to be used in the convergence check) need to be calculated. From Equation (2.9) the effective load increment is given by

$$\boldsymbol{f}_{t+\Delta t}^{eff} = \boldsymbol{f}_{t+\Delta t} + \frac{1}{\beta \Delta t^2}\boldsymbol{M}\boldsymbol{u}_t + \frac{1}{\beta \Delta t}\boldsymbol{M}\dot{\boldsymbol{u}}_t + (\frac{1}{2\beta} - 1)\boldsymbol{M}\ddot{\boldsymbol{u}}_t \tag{2.10}$$

The consistent tangent stiffness for the structure and the updated residual form a system of linear equations, which have to be solved to get the new correction to the displacement increment. Strains, stresses and the internal force vector can be updated next using the corrected displacement increment. A new residual can be calculated from the revised internal force vector and effective loads. When residuals meet the convergence criteria, velocities and accelerations for the current time step can be computed from the converged displacements and the solution for the last time step. An efficient implementation of different steps involved in this algorithm is discussed in Chapter 4. Matrix-vector multiplies are implemented using an element-by-element framework and a matrix-free approach is used to decrease storage and CPU time.

Most of the computational effort is required by the solution of the system of linear equations encountered in the solution algorithms described so far (e.g., Equation (2.9)). ROCSOLID is designed to solve theses equations using a parallel multigrid algorithm on a variety of multi-processor systems. For ease of notation, we rewrite these equations as

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{2.11}$$

where, in case of Equation (2.9) for example,

$$\boldsymbol{A} = \frac{1}{\beta \Delta t^2}\boldsymbol{M} + \boldsymbol{K}_T^{(k)} \tag{2.12}$$

$$\boldsymbol{x} = \Delta \boldsymbol{u}^{(k)} \tag{2.13}$$

and

$$\boldsymbol{b} = \boldsymbol{f}_{t+\Delta t}^{eff} - \boldsymbol{i}_{t+\Delta t}^{(k)} - \frac{1}{\beta \Delta t^2}\boldsymbol{M}\boldsymbol{u}_{t+\Delta t}^{(k)}.$$

In order to calculate the internal forces and the tangent stiffness matrix, the stresses must also be computed after each iteration. Further details are given in the next section.

2.2 Material State Determination

The next step after computing the incremental displacement is obtaining the new material state by calculating the strain and stress increments, and updating the stress-strain operator. The strains are obtained from displacements through the linear strain-displacement operator $\boldsymbol{B}$

$$\Delta \mathring{\boldsymbol{a}} = \boldsymbol{B}\Delta \boldsymbol{u} \,. \tag{2.14}$$

Computation of stress increments from strain increments (i.e. stress recovery) involves the material constitutive relationship. In this study, we restrict attention to a small-strain plasticity model based on rate independent von Mises material model with the associated flow rule utilizing a bilinear uniaxial material response, although other material models can easily be incorporated into this approach. Both isotropic and kinematic hardening are considered. The stresses are updated using the elastic predictor, radial return algorithm using a path independent strategy, which leads to greater accuracy and a reduction in the computational work.

2.2.1 Stress Recovery

The Mises yield surface is described by the equation

$$\frac{\xi'_{ij}\xi'_{ij}}{2} - r^2 = 0 \,, \tag{2.15}$$

where ξ'_{ij} is the deviator relative stress and r is proportional to the radius of the yield surface in the π plane (Figure 2.2). The deviator relative stress is given by

$$\xi'_{ij} = \sigma'_{ij} - a_{ij} \,, \tag{2.16}$$

where deviatoric stress σ'_{ij} is defined by $\sigma'_{ij} = \sigma_{ij} - \frac{\sigma_{kk}}{3}\delta_{ij}$ and a_{ij} designates the center of the yield surface (i.e., the back stress). The strain increment $\Delta\varepsilon_{ij}$ is decomposed into elastic and plastic components by the equation

$$\Delta\varepsilon_{ij} = \Delta\varepsilon^{e}_{ij} + \Delta\varepsilon^{p}_{ij} \,. \tag{2.17}$$

In an associated flow rule, the plastic potential and the yield function are equal [18, 33, 34]. Therefore, the plastic strain increment may be expressed as

$$\Delta\varepsilon_{ij}^{p} = \lambda n_{ij} \,, \tag{2.18}$$

where n_{ij} is the unit normal tensor and λ is a constant that depends on the stress increment. By defining $\left\| \left(\right)_{ij} \right\| = \sqrt{\left(\right)_{ij} \left(\right)_{ij}}$ as the norm associated with a tensor, the unit normal tensor is given by

$$n_{ij} = \frac{\xi'_{ij}}{\left\| \xi'_{ij} \right\|}. \tag{2.19}$$

Equation (2.18) shows that during plastic flow at a material point, the plastic strain increment is proportional to the outward normal to the yield surface defined by Equation (2.15). As a consequence of this flow rule, $\Delta\varepsilon_{kk}^{p}$, the change in plastic volume with time, is zero; the deviator plastic strain rate is therefore equal to the plastic strain rate (i.e., the yield function is independent of the hydrostatic stress state).

The equivalent plastic strain and the equivalent stress are

$$\Delta\bar{\varepsilon}^{p} = \sqrt{\frac{2}{3}\Delta\varepsilon_{ij}^{P}\Delta\varepsilon_{ij}^{P}} \tag{2.20}$$

$$\bar{\sigma} = \sqrt{3J_2'} \,, \tag{2.21}$$

where $J_2' = \frac{1}{2}\sigma'_{ij}\sigma'_{ij}$. The plastic modulus H' is defined as the derivative of the equivalent stress with respect to the equivalent strain, which for a Mises yield surface with a bilinear uniaxial stress-strain diagram is given by

$$H' = \frac{EE_T}{E - E_T}. \tag{2.22}$$

E and E_T are Young's modulus and the tangent modulus, respectively.

Along with Equation (2.18), the following defines the evolution equations for the material,

$$\Delta a_{ij} = \frac{2}{3}(1-\beta)H'\Delta\varepsilon_{ij}^{p} \tag{2.23}$$

$$\Delta r = \frac{\sqrt{2}}{3}\beta H'\lambda \tag{2.24}$$

$$\Delta\sigma'_{ij} = 2G(\Delta\varepsilon'_{ij} - \Delta\varepsilon'^{p}_{ij}) \tag{2.25}$$

$$\Delta p = K\Delta\varepsilon_{kk} \ . \tag{2.26}$$

The computational parameter β takes the values from zero to one; it is set to zero or one for purely kinematic or purely isotropic hardening, respectively. For mixed hardening, β takes values between zero and one. The parameters *K* and *G* are the bulk and shear moduli of the material and *p* is the hydrostatic stress.

The incremental constitutive equations must be numerically integrated to compute the new set of stresses corresponding to the new displacements. The elastic predictor, radial return algorithm is used to accomplish this [8, 18]. This procedure enforces the consistency condition, thereby constraining the stress point to remain on the yield surface during flow. Given the state of stress at step *n*, the hydrostatic stress and the elastic predictor trial deviator stress at step *n*+1 are computed as

$$^{n+1}p = {}^{n}p + K\Delta\varepsilon_{kk} \tag{2.27}$$

$$^{n+1}\sigma'^{t}_{ij} = {}^{n}\sigma'^{t}_{ij} + 2G\Delta\varepsilon'_{ij} \ , \tag{2.28}$$

and the trial deviator relative stress at step *n*+1 is defined by

$$^{n+1}\xi'^{t}_{ij} = {}^{n+1}\sigma'^{t}_{ij} - {}^{n}a_{ij} \ . \tag{2.29}$$

If the material point is elastic, the stress recovery is essentially complete at this stage. Only the trial deviator stress and the hydrostatic stress need to be recombined. But if the material point is in the state of plastic flow, the trial deviator stress is modified by a stress increment corresponding to a radial return to the yield surface. Using Equation (2.25), the updated deviator stress at step *n*+1 can be calculated;

$$^{n+1}\sigma'_{ij} = {}^{n+1}\sigma'^{t}_{ij} - 2G\lambda n_{ij} \ , \tag{2.30}$$

and Equations (2.23) and (2.24) give

$$^{n+1}a_{ij} = {}^{n}a_{ij} + \frac{2}{3}(1-\beta)H'\lambda n_{ij} \tag{2.31}$$

$$^{n+1}r = {}^{n}r + \frac{\sqrt{2}}{3}\beta H'\lambda\,. \tag{2.32}$$

Also, Equation (2.15) is rewritten as

$$\left\| {}^{n+1}\xi'_{ij} \right\| - \sqrt{2}\ {}^{n+1}r = 0\,. \tag{2.33}$$

The deviator relative stress at step n+1 is computed by combining Equations (2.30) and (2.31). The resulting equation is manipulated to give

$$\lambda = \frac{\left\| {}^{n+1}\xi'^{t}_{ij} \right\| - \sqrt{2}\ {}^{n}r}{2G\left(1+\dfrac{H'}{3G}\right)}\,. \tag{2.34}$$

It is possible to compute the parameter λ directly because H' is a constant which means that the equivalent stress is a linear function of the equivalent plastic strain. When this function is nonlinear, it would be necessary to iterate to determine λ. Figure 2.3 summarizes the steps involved in the stress recovery process using the elastic predictor, radial return algorithm.

Nonlinear problems can be either solved by linearizing the equations and using Newton's method or by solving the nonlinear equations directly using some appropriate nonlinear relaxation technique (e.g., the full approximation storage method, [15]). There are several potential difficulties associated with using nonlinear relaxation techniques especially when history-dependent nonlinear problems are considered. For example, spurious loading and unloading may be produced by nonlinear relaxation that would introduce unacceptably high errors. Also, when the multigrid method is considered, the material state variables would have to be continuously updated on all of the meshes after each relaxation sweep. Thus, the first approach, linearizing the equations and using Newton's method, will be considered in this research. A discussion of a full approximation storage multigrid method applied to elastoplasitcity problems can be found in reference [16].

Newton iterations can be used to trace the equilibrium path up to a critical point. Often, in order to understand the behavior of a structure, we need to be able to pass the critical

points and continue to trace the equilibrium path after these points. Arc-length continuation methods will be used in this study for this purpose [17]. The following section describes this method.

2.3 Arc-length Continuation Methods

The equilibrium equation for a structure can be written as

$$P(\boldsymbol{u}) = \lambda \boldsymbol{f}_0 \,, \tag{2.35}$$

where $P(\boldsymbol{u})$ is the internal force (a nonlinear history dependent function of $\boldsymbol{u}$, the displacements), λ is the load parameter and $\boldsymbol{f}_0$ is the normalized load pattern. Proportional loading is considered; the size of the external forces is controlled by the load parameter λ. Figure 2.4 shows the $(\boldsymbol{u},\lambda)$ relationship for a single degree-of-freedom demonstration problem. The goal is to determine the equilibrium state $(\boldsymbol{u}_{n+1},\lambda_{n+1})$ after the equilibrium state $(\boldsymbol{u}_n,\lambda_n)$ is known. Arc-length continuation methods treat both the magnitude of the loading and the displacements as unknowns and solve the problem in $N+1$ dimensional $(\boldsymbol{u},\lambda)$ space. We have N equilibrium equations, i.e.,

$$\boldsymbol{F}(\boldsymbol{u},\lambda) = P(\boldsymbol{u}) - \lambda \boldsymbol{f}_0 = 0 \,. \tag{2.36}$$

We need one more equation, a scalar constraint equation for $\boldsymbol{u}$ and λ, e.g.,

$$C(\boldsymbol{u},\lambda) = \left\| \boldsymbol{u} - \boldsymbol{u}_n \right\|^2 + (\lambda - \lambda_n)^2 - \Delta s^2 = 0 \,, \tag{2.37}$$

where Δs is the specified arc length. An iterative solver can be generated by linearizing (2.36) and (2.37) about $(\boldsymbol{u}_n,\lambda_n)$:

$$F(\boldsymbol{u},\lambda) \simeq F(\boldsymbol{u}_n,\lambda_n) + \frac{\partial F}{\partial \boldsymbol{u}} \Delta \boldsymbol{u} + \frac{\partial F}{\partial \lambda} \Delta\lambda \,, \tag{2.38}$$

and

$$C(\boldsymbol{u},\lambda) \simeq C(\boldsymbol{u}_n,\lambda_n) + \frac{\partial C}{\partial \boldsymbol{u}} \Delta \boldsymbol{u} + \frac{\partial C}{\partial \lambda} \Delta\lambda \,. \tag{2.39}$$

Noting that

$$\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{u}} = \frac{\partial \boldsymbol{P}}{\partial \boldsymbol{u}} = K_t \big|_{\boldsymbol{u}_n}, \tag{2.40}$$

where $K_t \big|_{\boldsymbol{u}_n}$ is the tangent stiffness matrix at the equilibrium configuration $(\boldsymbol{u}_n, \lambda_n)$,

$$\frac{\partial \boldsymbol{F}}{\partial \lambda} = -\boldsymbol{f}_0, \tag{2.41}$$

$$\frac{\partial C}{\partial \lambda} = 2(\lambda - \lambda_n), \tag{2.42}$$

and

$$\frac{\partial C}{\partial \boldsymbol{u}} = [2(u_1 - u_{n1}), 2(u_2 - u_{n2}), \ldots]_{1\times N} = 2(\boldsymbol{u} - \boldsymbol{u}_n)^T \tag{2.43}$$

we would need to solve the system of equations

$$\begin{bmatrix} K_t & -\boldsymbol{f}_0 \\ \dfrac{\partial C}{\partial u} & \dfrac{\partial C}{\partial \lambda} \end{bmatrix} \begin{Bmatrix} \Delta \boldsymbol{u}^{(i)} \\ \Delta \lambda^{(i)} \end{Bmatrix} = - \begin{Bmatrix} F(\boldsymbol{u}_n^{(i)}, \lambda_n^{(i)}) \\ C(\boldsymbol{u}_n^{(i)}, \lambda_n^{(i)}) \end{Bmatrix}. \tag{2.44}$$

Now we can update the displacement and load vectors,

$$\boldsymbol{u}^{(i+1)} = \boldsymbol{u}^{(i)} + \Delta \boldsymbol{u}^{(i)}, \tag{2.45}$$

$$\lambda^{(i+1)} = \lambda^{(i)} + \Delta \lambda^{(i)}. \tag{2.46}$$

Iteration is continued until $F(\boldsymbol{u}_n^{(i)}, \lambda_n^{(i)})$ and $C(\boldsymbol{u}_n^{(i)}, \lambda_n^{(i)})$ are small. In the above algorithm, equation (2.44) must be solved at each iteration, which involves an $(N+1)\times(N+1)$ non-symmetric matrix with a strange structure. By using the bordering algorithm the solution of equation (2.44) can be simplified. First we solve the following two equations for $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$:

$$K_t \boldsymbol{x}_1 = \boldsymbol{f}_0 \tag{2.47}$$

and

$$K_t \boldsymbol{x}_2 = \boldsymbol{F}(\boldsymbol{u}^{(i)}, \lambda^{(i)}), \tag{2.48}$$

then compute $\Delta \lambda^{(i)}$ and $\Delta \boldsymbol{u}^{(i)}$ from

$$\Delta\lambda^{(i)} = \frac{\left(-C(\boldsymbol{u}^{(i)}, \lambda^{(i)}) + \dfrac{\partial C}{\partial \boldsymbol{u}}\boldsymbol{x}_2\right)}{\dfrac{\partial C}{\partial \lambda} + \dfrac{\partial C}{\partial \boldsymbol{u}}\boldsymbol{x}_1} \tag{2.49}$$

and

$$\Delta\boldsymbol{u}^{(i)} = \Delta\lambda^{(i)}\boldsymbol{x}_1 - \boldsymbol{x}_2 \,. \tag{2.50}$$

2.4 Nonlinear Kinematics (ROCSOLID 3.2)

Large deformations are formulated using strains-stresses and their rates defined on an unrotated frame of reference. This model predicts physically acceptable responses for homogeneous deformations of exceedingly large magnitude. The implemented numerical algorithm is suitable for the large strain increments, which may arise in the implicit solution of the global equilibrium equations, employed in ROCSOLID. The finite rotation effects on strain-stress rates are separated from integration of the rates to update the material response over a time step. Consequently, all of the numerical routines developed previously for small-strain material models can be utilized without modification. In ROCSOLID 3.2 the j2 plasticity model can not yet use this large strain capability and for now only large deformations can be employed for linear elastic materials.

This formulation is also adopted in large-scale finite element codes, including NIKE, PRONTO, DYNA, ABAQUS-Standard and ABAQUS-Explicit. An extensive description of the numerical implementation details can be found in the monograph of Simo and Hughes, *Elastoplasticity and Viscoplasticity: Computational Aspects* (Stanford University, 1988).

2.4.1 Definitions

The deformation gradient is defined by

$$F = \partial x / \partial X, \qquad \det(F) = J > 0 \tag{2.51}$$

where x denotes the position vectors for material points at time t and X is the position vectors for material points defined on the configuration at t = 0. The displacements of the material points are thus given by

$$u = x - X \tag{2.52}$$

The polar decomposition of F yields

$$F = VR = RU \tag{2.53}$$

where V and U are the left and right-symmetric, positive definite stretch tensors, respectively and R is an orthogonal rotation tensor. The principal values of V and U are the stretch ratios of the deformation. We define an orthogonal reference frame at each material point such that the motion relative to these axes is only deformation throughout the loading history. For the RU decomposition for example, these axes do not follow the deformation (they are spatial) when applying tensor U and they do follow the deformation when tensor R is applied. Strain-stress tensors and their rates referred to these axes are said to be defined in the unrotated configuration.

The spatial gradient of the material point velocity $v = \dot{x}$ with respect to the current configuration is given by

$$L = \frac{\partial v}{\partial x} = \frac{\partial v}{\partial X}\frac{\partial X}{\partial x} = \dot{F}F^{-1} \tag{2.54}$$

We define D as the spatial rate of deformation tensor and the symmetric part of L and the skewsymmetric part, denoted W, is the spin rate or the vorticity tensor. W represents the rate of rotation of the principal axes of the spatial rate of deformation D.

$$L = D + W \tag{2.55}$$

where

$$D = \frac{1}{2}\left(L + L^T\right); \quad W = \frac{1}{2}\left(L - L^T\right) \tag{2.56}$$

When W is integrated over the loading history, the principal values of D are defined as the logarithmic (true) stains of infinitesimal fibers oriented in the principal directions if these directions do not rotate. D and W are instantaneous rates and do not sense the deformation history. Using the following relations,

$$\dot{F} = R\dot{U} + \dot{R}U \tag{2.57}$$

and

$$F^{-1} = \left(RU\right)^{-1} = U^{-1}R^{-1} = U^{-1}R^T \tag{2.58}$$

The spatial gradient L may be also written in the form

$$L = \dot{R}R^T + R\dot{U}U^{-1}R^T \tag{2.59}$$

The symmetric part of the second term in the last equation is called the unrotated deformation rate tensor or d

$$d = \frac{1}{2}\left(\dot{U}U^{-1} + U^{-1}\dot{U}\right). \tag{2.60}$$

Using the orthogonality property of R the unrotated deformation rate can also be shown in the following form

$$d = R^T DR. \tag{2.61}$$

The spatial rate of deformation, D, and the symmetric Cauchy (true) stress, σ are work conjugate in the sense that work per unit volume in the current configuration is given by $\sigma_{ij}D_{ij}$. Components of both D and σ are defined relative to the fixed, global axes. Therefore, the unrotated Caushy stress t may be given by

$$t = R^T \sigma R. \tag{2.62}$$

2.4.2 Computational Steps

Using an incremental iterative Newmark method the global solution is advanced from time t_n to t_{n+1}. Iterations denoted as i are needed to remove unbalanced nodal forces. At each iteration a new estimate for the total displacements at t_{n+1}, shown as $u_{n+1}^{(i)}$, is calculated. A mid-increment scheme is adopted in which deformation rates are evaluated on the intermediate configuration.

1- The deformation gradient at n + ½ and n + 1 is computed

2- Compute polar decomposition of the above deformation gradients

$$F_{n+1}^{(i)} = R_{n+1}^{(i)}.U_{n+1}^{(i)} \tag{2.63}$$

$$F_{n+1/2}^{(i)} = R_{n+1/2}^{(i)}.U_{n+1/2}^{(i)} \tag{2.64}$$

3- Using the B matrix for the element, compute the ith estimate for the spatial deformation

$$\Delta\varepsilon^{(i)} = B^{(i)}_{n+1/2}\left(u^{(i)}_{n+1} - u_n\right) \tag{2.65}$$

This procrdure enable us to use the B-bar formulation for finite strains thereby reducing volumetric locking in the element.

4- Rotate the increment of spatial deformation to the unrotated configuration

$$\Delta d^{(i)} = R^{(i)}_{n+1/2}{}^{T}.\Delta D^{(i)}.R^{(i)}_{n+1/2} \tag{2.66}$$

5- The conventional small strain models can now be used to compute the unrotated Cauchy stress at n + 1. The terms of the spatial deformation tensor is used as the strain increment.

6- The Cauchy stress is computed from transformation of the unrotated Cauchy stress at n + 1

$$\sigma_{n+1} = R_{n+1} t_{n+1} R^{T}_{n+1} \tag{2.67}$$

The calculated Cauchy stress is required for subsequent computation of element internal forces.

2.5 New Material Models (ROCSOLID 3.4)

Some architectural changes needed to be made in order to make the process of implementing new material models easier. The element level calculations included the material contributions in a matrix-free format that although were helpful in improving the performance, it did not have the required modularity. Therefore, by relaxing the matrix-free condition for the material contributions and adopting a UMAT approach achieved a significant improvement for material models implementation.

Three new material models are now available. They are compressible and incompressible Neo-Hookean and porous viscoelastic (with void growth) material models. The porous viscoelastic model can be used in the framework of the nonlinear kinematics described above which enables to simulate large rotations.

Loop over time:

Calculate the effective load increment and its norm

Loop over Newton iterations:

Solve

$$\left(\frac{1}{\beta\Delta t^2}\boldsymbol{M}+\boldsymbol{K}_T^{(k)}\right)\Delta\boldsymbol{u}^{(k)}=\boldsymbol{f}_{t+\Delta t}^{eff}-\boldsymbol{i}_{t+\Delta t}^{(k)}-\frac{1}{\beta\Delta t^2}\boldsymbol{M}\boldsymbol{u}_{t+\Delta t}^{(k)}$$

Increment displacements

$$\boldsymbol{u}_{t+\Delta t}^{(k+1)}=\boldsymbol{u}_{t+\Delta t}^{(k)}+\Delta\boldsymbol{u}^{(k)}$$

Update strains, stresses and internal force vector

Compute residual force vector and its norm

Test convergence

End loop over Newton iterations

Compute velocities and accelerations

$$\ddot{\boldsymbol{u}}_{t+\Delta t}=\frac{1}{\beta\Delta t^2}\left(\boldsymbol{u}_{t+\Delta t}-\boldsymbol{u}_t\right)-\frac{1}{\beta\Delta t}\dot{\boldsymbol{u}}_t-\left(\frac{1}{2\beta}-1\right)\ddot{\boldsymbol{u}}_t$$

$$\dot{\boldsymbol{u}}_{t+\Delta t}=\dot{\boldsymbol{u}}_t+\left((1-\gamma)\ddot{\boldsymbol{u}}_t+\gamma\ddot{\boldsymbol{u}}_{t+\Delta t}\right)\Delta t$$

End loop over time

Figure 2.1: Steps in Solution for Implicit Nonlinear Analysis.

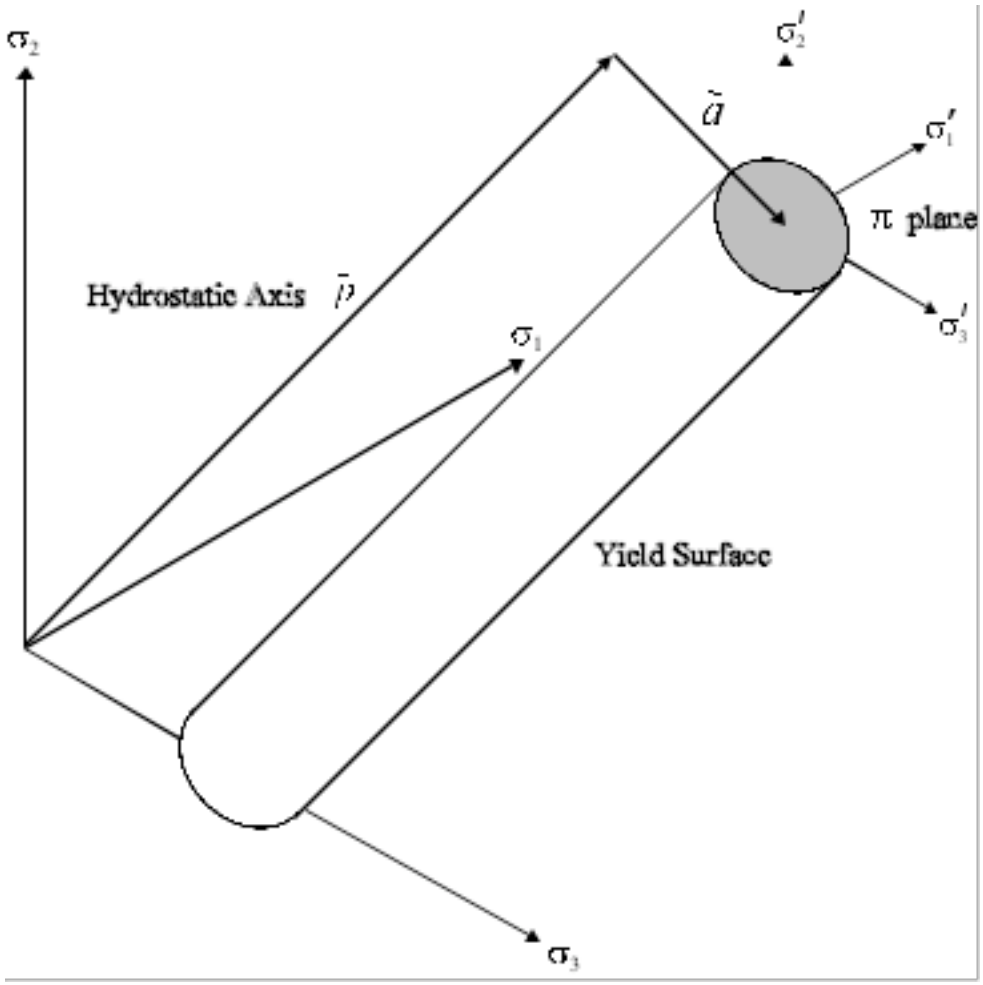


Figure 2.2: Mises Yield Surfaces in Principle Stress Space.

Compute strain increment $\Delta\varepsilon$ from displacement increment

Compute deviator strain increment $\Delta\varepsilon'_{ij} = \Delta\varepsilon_{ij} - \frac{\Delta\varepsilon_{kk}}{3}\delta_{ij}$

Compute trial deviator relative stress $^{n+1}\xi'^{t}_{ij} = {}^{n+1}\sigma'^{t}_{ij} - {}^{n}a_{ij}$

Where $^{n+1}\sigma'^{t}_{ij} = {}^{n}\sigma'^{t}_{ij} + 2G\Delta\varepsilon'_{ij}$

Evaluate yield function $f = \left\| {}^{n+1}\xi'_{ij} \right\| - \sqrt{2}\ {}^{n}r$

IF $f > 0$ THEN

Compute $\lambda = \dfrac{f}{2G\left(1 + \dfrac{H'}{3G}\right)}$

Update $^{n+1}r = {}^{n}r + \frac{\sqrt{2}}{3}\beta H'\lambda$

Update back stress $^{n+1}a_{ij} = {}^{n+1}\sigma'_{ij} - {}^{n+1}\xi'_{ij}$

Update deviator stress $^{n+1}\sigma'_{ij} = {}^{n+1}\sigma'^{t}_{ij} - 2G\lambda n_{ij}$

Where $n_{ij} = \dfrac{^{n+1}\xi'^{t}_{ij}}{\left\| {}^{n+1}\xi'^{t}_{ij} \right\|}$

Update hydrostatic stress $^{n+1}p = {}^{n}p + K\Delta\varepsilon_{kk}$

Update stress $^{n+1}\sigma_{ij} = {}^{n+1}\sigma'_{ij} + {}^{n+1}p\delta_{ij}$

ELSE

Update deviator stress = $^{n+1}\sigma'^{t}_{ij}$

Update hydrostatic stress $^{n+1}p = {}^{n}p + K\Delta\varepsilon_{kk}$

Update stress $^{n+1}\sigma_{ij} = {}^{n+1}\sigma'_{ij} + {}^{n+1}p\delta_{ij}$

END IF

Figure 2.3: Elastic Predictor, Radial Return Algorithm for Bi-linear (Mises) Material Model.

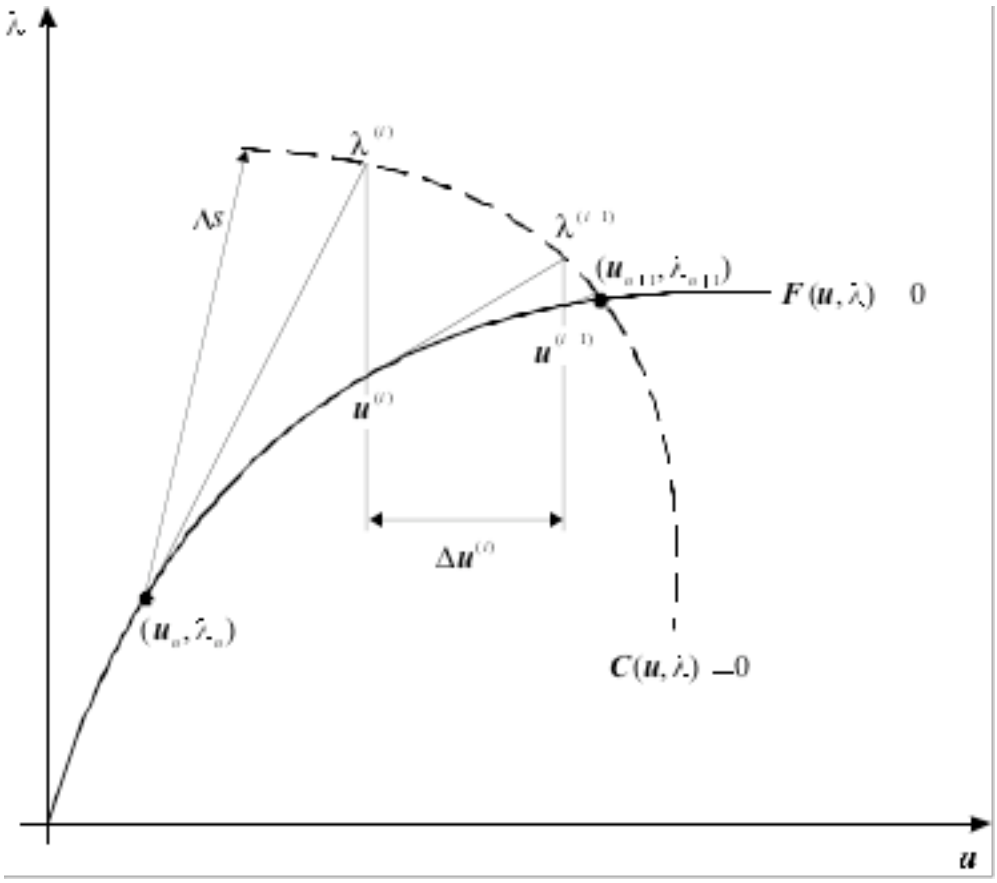


Figure 2.4: The Arc-Length Continuation Method for a Single Degree-of-Freedom System.

CHAPTER 3

THE MULTIGRID ALGORITHM FOR LINEAR EQUATIONS

This chapter describes the multigrid method we employ for solving the linear matrix equations encountered at each iteration in the solution algorithm explained in the previous chapter. Detailed descriptions of the method can be found in [12, 15, 13, 22]. Implementations of this method to perform nonlinear analysis of structural engineering problems are rather rare. Particularly, implementation of these methods on distributed memory machines presents additional challenges, which are the main issues in this research. A Proper communications strategy between processors is developed to maximize the performance of the solution algorithm.

Multigrid algorithms are particularly appealing because the computational effort (CPU time and storage) required to solve a problem is linearly proportional to the problem size. This property, which has been both proven (e.g., [12]) and observed (e.g., [13, 14]), implies that the method is algorithmically scalable. We first review basic multigrid methods, and then describe the components that have been successfully used to solve structural mechanics problems discretized using the finite element method.

3.1 The Basic Multigrid Algorithm

The multigrid method combines relaxation on a fine mesh with the approximate solution of residual equations on coarser grids. Smoothing properties of basic iterative methods are used to quickly produce a smooth fine mesh error. A coarse mesh is then used to cheaply ap-

proximate this error. In this section, multigrid methodology is briefly outlined by considering the solution of the generic linear matrix equation

$$\boldsymbol{A}_f \boldsymbol{x}_f = \boldsymbol{b}_f \,, \tag{3.1}$$

where $\boldsymbol{A}_f$ is a constant matrix, $\boldsymbol{b}_f$ is a constant vector and $\boldsymbol{x}_f$ contains the unknowns. Subscripts *f* and *c* indicate a fine or a coarse mesh respectively in this description. Figure 3.1 illustrates the steps involved in a simple two-grid method.

A small number (ν_1) of relaxation cycles are first applied to rapidly reduce the high frequency error associated with an initial approximate solution $\boldsymbol{x}_f^{(k)}$ on the fine mesh. The new approximate solution, $\overline{\boldsymbol{x}}_f^{(k)}$, produced by the relaxation cycles has a smooth error and therefore can be approximated on a coarser mesh. The fine mesh residual, $\boldsymbol{r}_f^{(k)} = \boldsymbol{b}_f - \boldsymbol{A}_f \overline{\boldsymbol{x}}_f^{(k)}$ is restricted to the coarse mesh to obtain the coarse mesh residual

$$\boldsymbol{r}_c^{(k)} = \boldsymbol{I}_f^c \boldsymbol{r}_f^{(k)}, \tag{3.2}$$

where $\boldsymbol{I}_f^c$ is the fine-to-coarse mesh restriction operator. The coarse mesh correction, $\Delta \boldsymbol{x}_c^{(k)}$, which is intended to be a reasonable representation of the smooth fine mesh error is computed by solving the coarse mesh correction equation

$$\boldsymbol{A}_c \Delta \boldsymbol{x}_c^{(k)} = \boldsymbol{r}_c^{(k)}. \tag{3.3}$$

To obtain the fine mesh correction, $\Delta \boldsymbol{x}_c^{(k)}$ is interpolated to the fine mesh using the coarse-to-fine mesh interpolation operator, $\boldsymbol{I}_c^f$, i.e.,

$$\Delta \boldsymbol{x}_f^{(k)} = \boldsymbol{I}_c^f \Delta \boldsymbol{x}_c^{(k)}. \tag{3.4}$$

The fine mesh correction is used to compute the new fine mesh approximate solution

$$\hat{\boldsymbol{x}}_f^{(k)} = \overline{\boldsymbol{x}}_f^{(k)} + \Delta \boldsymbol{x}_f^{(k)}. \tag{3.5}$$

A small number (ν_2) of relaxation cycles are performed on the fine mesh to reduce any high frequency errors introduced by interpolation. This produces $\boldsymbol{x}_f^{(k+1)}$, the new estimate to the solution of equation (3.1).

These steps are repeated until a converged solution is obtained. Usually, the following criteria is used to determine convergence

$$\frac{\| \boldsymbol{b}_f - \boldsymbol{A}_f \boldsymbol{x}_f^{(k)} \|}{\| \boldsymbol{b}_f \|} \leq \varepsilon_{tol} \tag{3.6}$$

where $\| \, . \, \|$ indicates the Euclidean norm of a vector and ε_{tol} is the convergence tolerance specified by the user.

The above two-grid algorithm can be extended to a true multigrid method by solving the coarse mesh correction equation (3.3) using one or more successively coarser meshes, i.e., the above two grid method can be recursively applied to this equation. It is generally necessary to perform only a few multigrid iterations on the coarse grid to obtain a good approximation to the required coarse grid correction. The parameter γ is used to denote the number of cycles used on each coarse mesh to solve coarse mesh correction equation. Figure 3.2 shows different multigrid algorithms that can be obtained depending on the number of meshes used and the value of γ.

3.1.1 Multigrid Components

The multigrid algorithm discussed above consists of four components: the fine mesh relaxation scheme, the interpolation operator, the restriction operator and the coarse mesh solution. The choice of these components is discussed in the following sections. These components will be visited again in Chapter 4 when we discuss the parallel implementation of the algorithm.

3.1.1.1 Relaxation Scheme

A crucial component of any multigrid implementation is the choice of the relaxation scheme. Many basic iterative procedures can be chosen to perform relaxation, which rapidly eliminates the high frequency error components from the current approximate fine mesh solution. The Jacobi preconditioned conjugate gradient method [37], outlined in Figure 3.3, is employed as the basic relaxation scheme in ROCSOLID. In this Figure, i is the iteration count, $\boldsymbol{r}$ specifies the residual, $\boldsymbol{x}$ represents the unknown vector, $\boldsymbol{p}$ defines the step direction, λ denotes the step length, and α specifies the correction factor. This scheme has been

found to be a robust smoother for a variety of nonlinear structural mechanics problems. Traditional smoothers such as Gauss-Seidel relaxation are unsuitable for the class of problems we are interested in (see [13, 14] for additional details). Usually, five to ten relaxation cycles would be enough to economically smooth fine mesh errors.

3.1.1.2 Interpolation and Restriction Operators

A sequence of nested meshes is used in ROCSOLID for the multigrid solution algorithm for which nodal averaging of displacements is a natural selection for interpolation between fine and coarse meshes. In other words, the interpolation operator can be defined by applying constraints to the fine mesh degree-of-freedom. In Figure 3.4, four coarse mesh quadrilateral elements are refined once to form sixteen fine mesh elements. The fine mesh degrees-of-freedom in terms of the coarse mesh degrees-of-freedom can be defined by the following constraints

$$\begin{aligned} u_f^a &= u_c^a, \\ u_f^e &= 1/2\,(u_c^a + u_c^b), \\ u_f^f &= 1/4\,(u_c^a + u_c^b + u_c^c + u_c^d), \end{aligned} \tag{3.7}$$

where u_f^α is a fine mesh degree-of-freedom at node α, and u_c^β denotes the corresponding coarse mesh degree-of-freedom at node β. The same approach can be extended to three-dimensional elements. Thus, the coarse-to-fine mesh interpolation operator $\boldsymbol{I}_c^f$ can be represented as a matrix $\boldsymbol{T}$ assembled on a coarse mesh element level using expressions such as Equation (3.7) so that

$$\Delta x_f^{(k)} = \boldsymbol{T}\Delta x_c^{(k)}. \tag{3.8}$$

Applying the principle of work equivalency between the fine and coarse meshes requires that the fine-to-coarse mesh restriction operator $\boldsymbol{I}_f^c$ be $\boldsymbol{T}^T$, thus

$$\boldsymbol{r}_c^{(k)} = \boldsymbol{T}^T \boldsymbol{r}_f^{(k)}. \tag{3.9}$$

3.1.1.3 Coarse Mesh Solution

The coarse mesh stiffness matrices must represent the deformation on the coarse mesh and can be computed by two different methods. Using the Galerkin coarse mesh stiffness matrix is the first choice:

$$\boldsymbol{K}_c = \boldsymbol{T}^T \boldsymbol{K}_f \boldsymbol{T}. \tag{3.10}$$

In other words, the coarse mesh stiffness matrix can be defined from the corresponding fine mesh stiffness matrix using the interpolation and restriction operators. This approach has some difficulties associated with it, which will be mentioned later. In ROCSOLID, the alternative approach is used for coarse mesh solution that is to assemble the coarse mesh stiffness matrix from individual element stiffness matrices. The construction of $\boldsymbol{K}_c$ in this case is independent of $\boldsymbol{K}_f$ and is given by

$$\boldsymbol{K}_c = \sum_e \int_{V^e} \boldsymbol{B}^{eT} \boldsymbol{D}^e \boldsymbol{B}^e dV \tag{3.11}$$

where the summation symbol implies a standard assembly process, the superscript e denotes element quantities, and $\boldsymbol{B}^e$ and $\boldsymbol{D}^e$ are the appropriate coarse mesh element strain-displacement and material matrices, respectively. This approach fits well in the element level implementation of the method discussed in Chapter 4. The Jacobi preconditioned conjugate gradient method is used to obtain the solution to the coarsest mesh correction equation.

3.1.2 Treatment of History Dependent Problems

As was mentioned before, we use the multigrid method as an equation solver embedded inside Newton (or arc-length) iteration. This approach involves linearization of the problem on the finest mesh and multigrid solution of the resulting linear matrix equation. Some modification of the linear multigrid solver described above are required.

After each Newton iteration, in order to calculate the internal forces and the tangent stiffness matrix, the stresses must also be computed. Section 2.2 explained the method used in this study for a specific material model. To compute the tangent stiffness matrices for the

meshes involved in the multigrid solution, considering small deformations, we can use the following equation

$$\boldsymbol{K}_T = \sum_e \int_{V^e} \bar{\boldsymbol{B}}^{eT} \boldsymbol{D}_T^e \bar{\boldsymbol{B}}^e dV \,. \tag{3.12}$$

To preserve the quadratic convergence rate present in the Newton iteration, the consistent element material tangent operator D_T^e described in (Dodds 1987, Simo & Taylor 1985 [8, 18]) must be used. Also, the $\bar{\boldsymbol{B}}$ operator is computed by the so-called $\bar{\boldsymbol{B}}$ method (Hughes [23]) to avoid locking for near incompressible conditions, which occurs in fully integrated elements. This method replaces dilatational terms of the original strain-displacement matrix by a volume-averaged set of dilatational terms.

The coarse mesh state variables are associated with the fine mesh straining history. Instead of interpolating the fine mesh displacements to the coarse mesh and then integrating the constitutive law, which requires a large amount of computational work, we directly interpolate the fine mesh state variables to the coarse mesh. This means that the stress recovery procedure is only done on the finest mesh employed. More details are available in reference [24].

3.2 General Behavior of Multigrid Methods

Multigrid methods are used for solving linear and nonlinear structural mechanics problems by some researchers and in this section, some important features of this algorithm are discussed. A more complete description of these features can be found in [12, 13, 15, 29].

Iterative solvers are generally sensitive to ill-conditioning and multigrid methods are no exception. An ill-conditioned problem can cause an increase in the number of cycles required for convergence, or even a complete failure to converge. The spectral condition number of a matrix $\boldsymbol{A}$, defined by the following equation can be used to measure ill-conditioning,

$$\kappa(\boldsymbol{A}) = \left\| \boldsymbol{A} \right\| \left\| \boldsymbol{A}^{-1} \right\|. \tag{3.13}$$

where $\| . \|$ denotes the norm of a matrix. The matrices with higher condition numbers are more ill-conditioned and by definition, $\kappa(\boldsymbol{A}) = \infty$ when $\boldsymbol{A}$ is singular. The stiffness matrix

of a structure can become ill-conditioned when stiffnesses of widely varying orders of magnitude are present. Nearly incompressible materials, small elements, thin shells and heterogeneous material properties are common causes of ill-conditioning. It is important to note here that successive refinements of a given mesh which generates smaller elements on each successive mesh and therefore higher condition numbers, produce little or no increase in the number of multigrid cycles required to converge.

The smoothing effect of the relaxation scheme can deteriorate in near incompressible conditions and this makes multigrid converge slower. Jacobi preconditioned conjugate gradient algorithm usually behaves better as a relaxation scheme in these cases and using more relaxation cycles on each mesh is also helpful. Using Equation (3.10) to compute the coarse mesh stiffness matrix can cause locking even when reduced integration is used to cure fine mesh locking [14, 24]. Therefore, explicit computation of the coarse mesh stiffness matrix is recommended.

Large amounts of bending deformation also slow multigrid convergence, because the coarse meshes are generally too stiff. The possible solutions are using better elements [25] or using a higher value for γ. Special treatment should also be considered for problems with heterogeneous material properties. Some examples can be found in [26]. The interpolation procedure described in Section 3.1.1.2 does not work in these cases because a coarse element consists of fine elements with widely varying material properties.

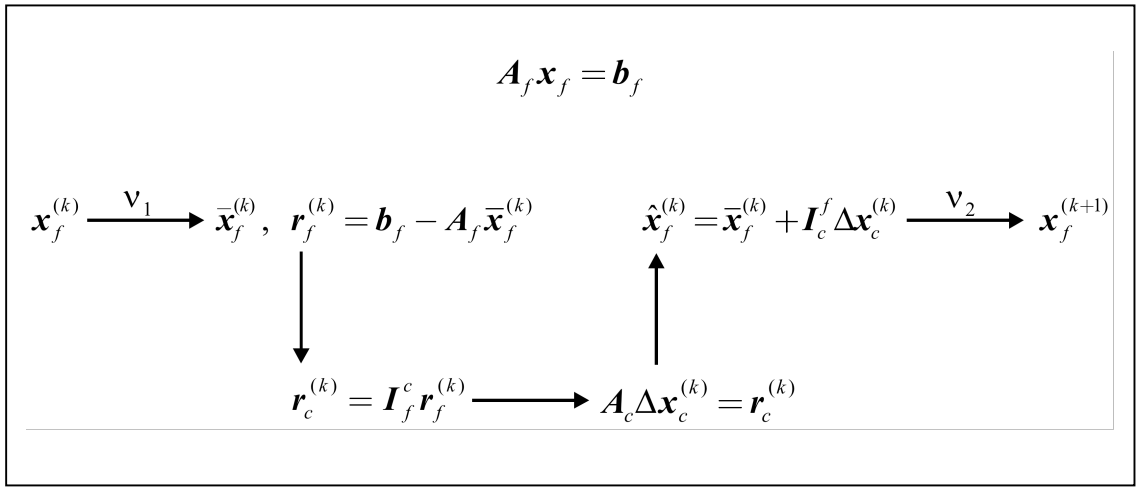


Figure 3.1: Steps Involved in a Simple Two-Grid Method.

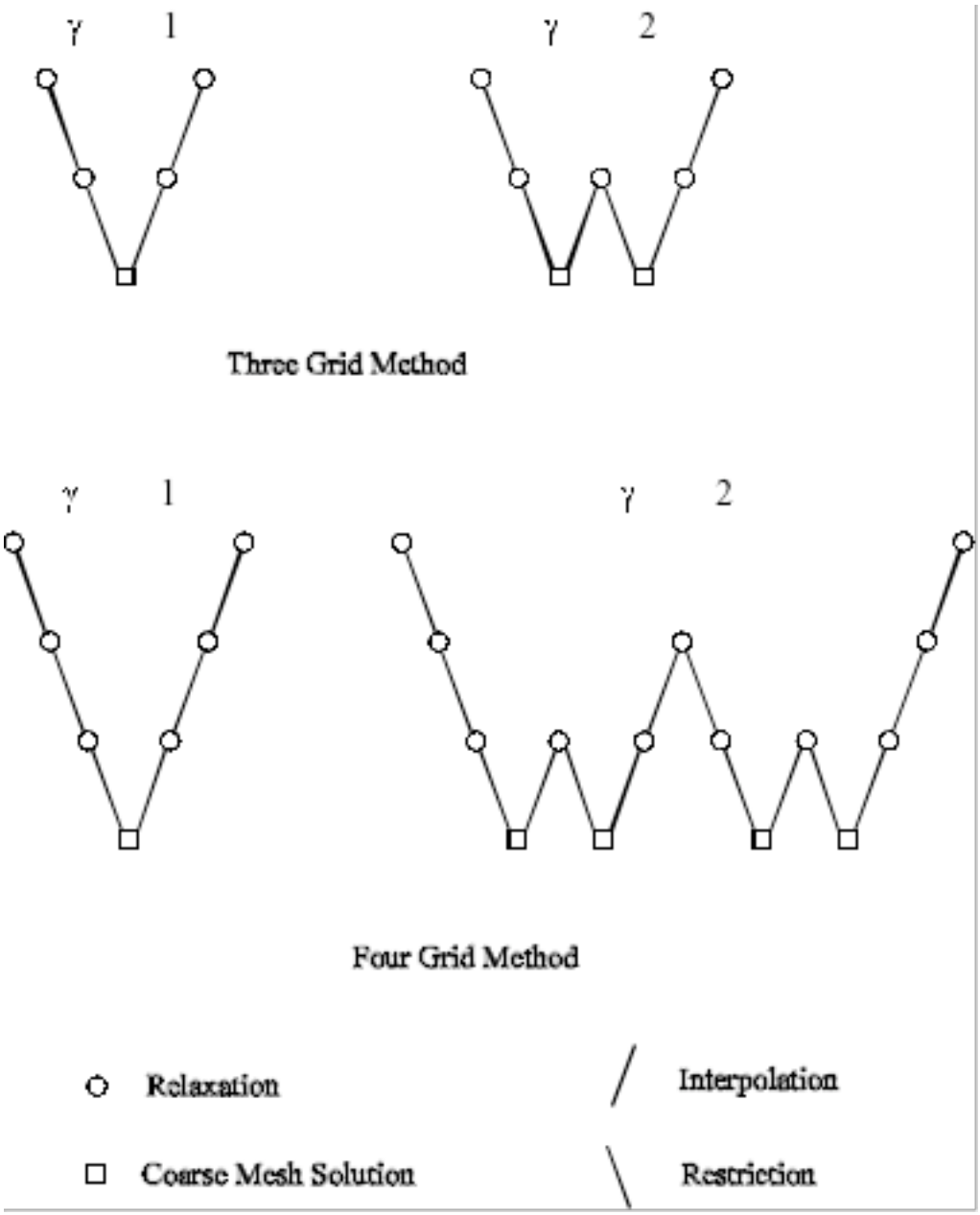


Figure 3.2: One Multigrid Cycle with Various Values of γ for Different Numbers of Meshes.

Given: $\boldsymbol{x}^{(0)}, \quad i = 0$

Initialize:

$$\boldsymbol{r}^{(0)} = \boldsymbol{f} - \boldsymbol{A}\boldsymbol{x}^{(0)}$$
$$\boldsymbol{d}^{(0)} = \boldsymbol{A}_D^{-1}\boldsymbol{r}^{(0)}$$
$$\boldsymbol{p}^{(0)} = \boldsymbol{d}^{(0)}$$

Iterate over:

$$\lambda^{(i)} = \frac{\left(\boldsymbol{r}^{(i)}, \boldsymbol{d}^{(i)}\right)}{\left(\boldsymbol{p}^{(i)}, \boldsymbol{A}\boldsymbol{p}^{(i)}\right)}$$
$$\boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{(i)} + \lambda^{(i)}\boldsymbol{p}^{(i)}$$
$$\boldsymbol{r}^{(i+1)} = \boldsymbol{r}^{(i)} - \lambda^{(i)}\boldsymbol{p}^{(i)}$$
$$\boldsymbol{d}^{(i+1)} = \boldsymbol{A}_D^{-1}\boldsymbol{r}^{(i+1)}$$
$$\alpha^{(i+1)} = \frac{\left(\boldsymbol{r}^{(i+1)}, \boldsymbol{d}^{(i+1)}\right)}{\left(\boldsymbol{r}^{(i)}, \boldsymbol{d}^{(i)}\right)}$$
$$\boldsymbol{p}^{(i+1)} = \boldsymbol{d}^{(i+1)} + \alpha^{(i+1)}\boldsymbol{p}^{(i)}$$
$$i = i + 1$$

Figure 3.3: The Jacobi Preconditioned Conjugate Gradient Algorithm.

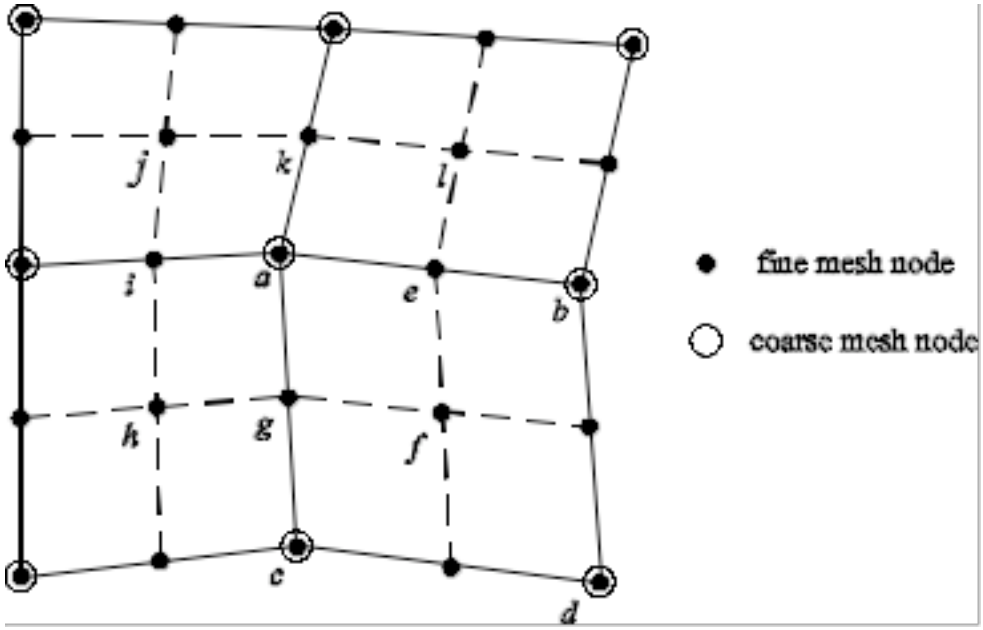


Figure 3.4: Coarse Mesh Definition for Four Node Quadrilateral Elements.

CHAPTER 4

MULTIGRID PARALLELIZATION

The nonlinear multigrid solution algorithm for solving structural mechanics problems was outlined in the previous chapter. In this chapter, we describe the details of multigrid parallelization on distributed memory systems. The parallel implementation strategy is explained in the following section. We then review the multigrid components to identify the primary operations that require parallelization. A suitable algebraic framework is developed in Section 4.3 that helps us to determine correct parallel algorithms to perform computations in different stages of multigrid solution. The parallel programming model used to implement the proposed algorithm is then described, followed by a discussion of some implementation issues. Finally, the mesh generation technique used to produce the required hierarchy of increasingly finer meshes and the partitioning method are discussed.

4.1 Parallel Implementation Strategy

The time to transfer data between processors is usually the most significant source of parallel processing overhead. An efficient algorithm for interprocessor communication needs to be adopted for a given program to minimize the amount of data that has to travel through the communication network of the machine, thereby reducing the total communication cost. The time taken for data transfer depends on the relative locations of the source and destination processors. Performance of a parallel program is determined by how well the location of data matches its use.

We adopt a domain decomposition strategy to parallelize the multigrid algorithm used in the implicit finite element solution method. This entails partitioning the finite element mesh into a number of domains and assigning the domains to individual processors of the parallel machine. Each processor is responsible for the elements within its own domain to perform the related computations concurrently with other processors. Appropriate communications are generally needed between the processors during the various multigrid components.

In this strategy, the communication cost is reduced by increasing computation and communication granularity, i.e., instead of performing the computation and communication one element at a time, computations for large number of elements inside domains is followed by communications between the domains. This is often called the surface-to-volume effect associated with the domain decomposition techniques, which results in improved efficiency by decreasing the communication to computation ratio. Mesh generation and partitioning is discussed later in this chapter.

4.2 Multigrid Components

The multigrid algorithm discussed in the previous chapter consists of four components: fine mesh relaxation, interpolation, restriction and coarse mesh solution. We now review each of these components in turn, identifying the primary operations that require parallelization.

4.2.1 Fine Mesh Relaxation

As explained in the previous chapter, the Jacobi preconditioned conjugate gradient method is our choice for the relaxation scheme. Figure 3.3 outlines this method and shows the following required primary operations at the *i*th iteration:

vi. Matrix-vector multiplications, i.e., $\boldsymbol{A}\boldsymbol{p}^{(i)}$;

vii. DAXPYs, e.g., $\boldsymbol{p}^{(i+1)} = \boldsymbol{d}^{(i+1)} + \alpha^{(i+1)}\boldsymbol{p}^{(i)}$;

viii. Scalar products, e.g., $\left(\boldsymbol{p}^{(i)}, \boldsymbol{A}\boldsymbol{p}^{(i)}\right)$, where $(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^T\boldsymbol{y}$;

ix. Preconditioning, i.e., $\boldsymbol{d}^{(i+1)} = \boldsymbol{A}_D^{-1}\boldsymbol{r}^{(i+1)}$, where $\boldsymbol{A}_D$ is the diagonal part of $\boldsymbol{A}$.

4.2.2 Interpolation and Restriction Operators

Section 3.1.1.2, explained that the coarse-to-fine mesh interpolation operator $\boldsymbol{I}_c^f$ and the fine-to-coarse mesh restriction operator $\boldsymbol{I}_f^c$, can be represented as matrices $\boldsymbol{T}$ and $\boldsymbol{T}^T$ respectively so that

$$\Delta \boldsymbol{x}_f^{(k)} = \boldsymbol{T} \Delta \boldsymbol{x}_c^{(k)} \tag{4.1}$$

and

$$\boldsymbol{r}_c^{(k)} = \boldsymbol{T}^T \boldsymbol{r}_f^{(k)} . \tag{4.2}$$

The primary operations for the intergrid transfer operations then reduce to the matrix-vector multiplications in Equations (4.1) and (4.2).

4.2.3 Coarse Mesh Solution

The coarse mesh stiffness matrices are assembled from the coarse mesh element matrices (Section 3.1.1.3), and the Jacobi preconditioned conjugate gradient method is used to obtain the solution to the coarsest mesh correction equation. Hence, the primary operations required by the coarse mesh solution are identical to those identified in Section 4.2.1.

4.3 Basic Mesh Operations

Before detailing the distributed implementation of the multigrid components, we first develop a suitable algebraic framework. Consider a finite element mesh R with a total of n degrees-of-freedom that is partitioned in an element-wise fashion into n_D domains, D_i, $i = 1,...,n_D$, such that domain D_i has n_i degrees-of-freedom, $\mathsf{R} = \bigcup_{i=1}^{n_D} \mathsf{D}_i$ and $\bigcap_{i=1}^{n_D} \mathsf{D}_i = \varnothing$ (e.g., Figure 4.1 for a 16 element mesh partitioned into 4 domains). Note that n_i is not necessarily the same for each domain. Assume that any vector, $\boldsymbol{x} \in \mathbb{R}^n$, defined on this mesh is partitioned so that each domain receives the components of the vector that belong to the degrees-of-freedom present in the domain. These components are stored in vectors $\boldsymbol{x}_i \in \mathbb{R}^{n_i}$ that are local to each domain D_i. We construct our algorithm so that we operate on the do-

main vectors independently, rather than the global vectors. For each domain, we can define mapping and weighting matrices ($\boldsymbol{M_i}$ and $\boldsymbol{W_i}$, respectively) such that

$$\boldsymbol{x_i} = \boldsymbol{M_i^T x} \tag{4.3}$$

and

$$\boldsymbol{x} = \sum_{i=1}^{n_D} \boldsymbol{M_i W_i x_i} \tag{4.4}$$

where $\boldsymbol{M_i} \in \mathbb{R}^{n \times n_i}$ and $\boldsymbol{W_i} \in \mathbb{R}^{n_i \times n_i}$. For example, consider Figure 4.1. Here

$$\boldsymbol{M}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{25 \times 9}$$

and

$$\boldsymbol{W}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/4 \end{bmatrix}_{9 \times 9}$$

The mapping matrix $\boldsymbol{M}_i$ is a Boolean operator that maps variables on D_i into their correct locations in the global vector. The weighting matrix $\boldsymbol{W}_i$ accounts for the number of times a variable is stored on different domains. Since we are considering an element-level partitioning of the mesh, the domain mapping matrices can also be used to assemble a matrix that has been computed on each of the domains, i.e.,

$$\boldsymbol{A} = \sum_{i=1}^{n_D} \boldsymbol{M}_i \boldsymbol{A}_i \boldsymbol{M}_i^T \tag{4.5}$$

where $\boldsymbol{A}$ is the global matrix and $\boldsymbol{A}_i$ are the domain contributions to $\boldsymbol{A}$ that are assembled from $\boldsymbol{A}_i^e$ in the usual way. This is identical to the standard procedure used to assemble element stiffness matrices.

4.4 Algorithm Implementation

In the following sections, the above algebraic framework is applied to the primary operations identified earlier. Equations (4.3), (4.4) and (4.5) allow us to determine correct parallel algorithms for computing matrix-vector products and scalar products on a single mesh; extensions of these ideas lead to similar algorithms for the intermesh transfer operators.

4.4.1 Matrix-Vector Multiplications

Consider the matrix-vector product $\boldsymbol{q} = \boldsymbol{A}\boldsymbol{x}$ that is required on a single partitioned mesh. Using Equations (4.3) and (4.5)

$$\begin{aligned} \boldsymbol{q} &= \boldsymbol{A}\boldsymbol{x} \\ &= \left(\sum_{i=1}^{n_D} \boldsymbol{M}_i \boldsymbol{A}_i \boldsymbol{M}_i^T \right) \boldsymbol{x} \\ &= \sum_{i=1}^{n_D} \boldsymbol{M}_i \boldsymbol{A}_i \boldsymbol{x}_i \end{aligned} \tag{4.6}$$

However, on D_i we need to store $\boldsymbol{q}_i$ where, from Equations (4.3) and (4.6),

$$\boldsymbol{q}_i = \boldsymbol{M}_i^T \boldsymbol{q}$$

$$= \sum_{j=1}^{n_D} \boldsymbol{M}_i^T \boldsymbol{M}_j \boldsymbol{A}_j \boldsymbol{x}_j \tag{4.7}$$

The product $\boldsymbol{M}_i^T \boldsymbol{M}_j$ is a communication matrix that transfers variables from $\mathsf{D_j}$ to $\mathsf{D_i}$. This can be seen through the following argument. The product $\boldsymbol{M}_j \boldsymbol{x}_j$ for any $\boldsymbol{x}_j \in \mathbb{R}^{n_j}$ that is defined on $\mathsf{D_j}$ produces $\boldsymbol{M}_j \boldsymbol{x}_j \in \mathbb{R}^n$, the vector that represents the global version of $\boldsymbol{x}_j$. Then $\boldsymbol{M}_i^T \boldsymbol{M}_j \boldsymbol{x}_j \in \mathbb{R}^{n_i}$ is a vector defined on $\mathsf{D_i}$ that contains the components of $\boldsymbol{x}_j$ that also appear on $\mathsf{D_i}$. This process is illustrated in Figure 4.2. Noting that $\boldsymbol{M}_i^T \boldsymbol{M}_i = \boldsymbol{I}_i \in \mathbb{R}^{n_i \times n_i}$, the identity matrix defined on $\mathsf{D_i}$, Equation (4.7) becomes

$$\boldsymbol{q}_i = \boldsymbol{A}_i \boldsymbol{x}_i + \sum_{\substack{j=1 \\ j \neq i}}^{n_D} \boldsymbol{M}_i^T \boldsymbol{M}_j \boldsymbol{A}_j \boldsymbol{x}_j \,. \tag{4.8}$$

Thus, the computation of domain versions of the matrix-vector product $\boldsymbol{Ax}$ is a two-stage process. First, the product $\boldsymbol{A}_i \boldsymbol{x}_i$ is computed on all of the domains, $i = 1,...,n_D$. Second, transfer of data between the domains is required to map inter-domain boundary data using the mapping matrices. This data transfer is local, i.e., data is transferred to $\mathsf{D_i}$ only from domains that share degrees-of-freedom with $\mathsf{D_i}$.

In the multigrid algorithm, most of the effort is spent in computing the matrix-vector products required in the PCG relaxations. These computations can account for as much as 95% of the total CPU time [24]. Therefore, this operation has to be implemented in the most efficient way.

In each domain, the element-by-element strategy is used to perform all of the necessary matrix-vector multiplications and to compute the residual loads on the finest mesh. This approach reduces the storage required because no structure matrices need to be assembled and stored. This method uses a node-element domain data structure, which requires gather-scatter procedures. Matrix-free element level computations are implemented in this element-by-

element framework. An alternative approach, which will not be discussed here, is to construct the sparse form of the system matrix locally for each domain and compute the matrix-vector products $\boldsymbol{A}_i\boldsymbol{x}_i$ followed by a gather operation over all domains to get the final result. This might reduce the operation count and the required CPU time but will increase the memory demand that might not be desirable, especially for distributed systems.

The computation of $\boldsymbol{A}_i\boldsymbol{x}_i$ for each domain depends on the system matrix of the problem at hand. For example, in case of a dynamics problem (Chapter 2),

$$\boldsymbol{A}_i = \frac{1}{\beta\Delta t^2}\boldsymbol{M}_i + \boldsymbol{K}_i . \tag{4.9}$$

Recognizing that $\boldsymbol{M}_i = \sum_e \boldsymbol{M}_i^e$ and $\boldsymbol{K}_i = \sum_e \boldsymbol{K}_i^e$, where the superscript e denotes element quantities and the summation implies assembly of element quantities, we can write

$$\boldsymbol{M}_i\boldsymbol{x}_i = \sum_e \boldsymbol{M}_i^e\boldsymbol{x}_i^e = \sum_{\mathsf{e}} \int_{\mathsf{R}^e} \rho\boldsymbol{N}^T\boldsymbol{N}\boldsymbol{x}_i^e dV \tag{4.10}$$

and

$$\boldsymbol{K}_i\boldsymbol{x}_i = \sum_e \boldsymbol{K}_i^e\boldsymbol{x}_i^e = \sum_{\mathsf{e}} \int_{\mathsf{R}^e} \boldsymbol{B}^T\boldsymbol{D}\boldsymbol{B}\boldsymbol{x}_i^e dV \tag{4.11}$$

where ρ is the material mass density, $\boldsymbol{N}$ is the element shape function, $\boldsymbol{B}$ is the element strain-displacement matrix, $\boldsymbol{D}$ is the element material matrix, and integration is performed over the region occupied by the element, R^{e}.

The stages necessary to perform $\boldsymbol{K}_i\boldsymbol{x}_i$ computation for example, are as follows (Figure 4.3). First, the element version of $\boldsymbol{x}_i$, $\boldsymbol{x}_i^e$, is formed by gathering information from the domain $\boldsymbol{x}_i$ vector. Second, $\boldsymbol{K}_i^e\boldsymbol{x}_i^e$ is computed at the element level and finally, the domain $\boldsymbol{K}_i\boldsymbol{x}_i$ vector is produced by scattering the element quantities $\boldsymbol{K}_i^e\boldsymbol{x}_i^e$ to this vector. Usually, the elements are processed in blocks chosen to maximize code performance. The gathering and scattering of domain and element quantities is accomplished using assembly arrays.

This approach also admits a matrix-free implementation of the element level computations involving the mass and stiffness matrices (i.e., $\boldsymbol{M}_i^e \boldsymbol{x}_i^e$ and $\boldsymbol{K}_i^e \boldsymbol{x}_i^e$), which offers storage and time savings [24, 29]. For example, the element level matrix-vector product $\boldsymbol{K}_i^e \boldsymbol{x}_i^e$ (Equation (4.11)), is computed in the following four-stage process:

i. Compute element pseudo-strains $\tilde{\boldsymbol{\varepsilon}} = \boldsymbol{B}\boldsymbol{x}_i^e$, at each element Gauss integration point;

ii. Compute element pseudo-stresses $\tilde{\boldsymbol{\sigma}} = \boldsymbol{D}\tilde{\boldsymbol{\varepsilon}}$, at the Gauss points;

iii. Compute element pseudo-internal forces $\tilde{\boldsymbol{f}} = \boldsymbol{B}^T \tilde{\boldsymbol{\sigma}}$, at the Gauss points;

iv. Compute the product $\boldsymbol{K}_i^e \boldsymbol{x}_i^e$ as

$$\boldsymbol{K}_i^e \boldsymbol{x}_i^e = \sum_l w_l \tilde{\boldsymbol{f}}_l , \tag{4.12}$$

where w_l are the Gauss quadrature weights, and summation is implied over all of the Gauss points. The prefix pseudo indicates that these quantities are not directly related to any physical deformation.

An alternative approach to perform the element level matrix-vector multiplications is computing and storing the upper halves of the element matrices and then performing the matrix-vector products. The computational work involved in this strategy and the matrix-free approach can be compared for example in a problem with eight node brick elements and linear elastic, homogenous, isotropic material [29]. To compute the upper half of the element stiffness matrix explicitly, 6,624 operations are required where one operation represents the combined work of one multiplication and one addition. Computation of $\boldsymbol{K}^e \boldsymbol{x}^e$ requires 576 operations. However, using the matrix-free technique, 1,248 operations are required to compute $\boldsymbol{K}^e \boldsymbol{x}^e$ if full integration is used and only 312 operations if reduced integration is employed. The storage and time savings gained by matrix-free implementation play an important role in producing a code for large-scale simulations.

4.4.2 Scalar Products

Consider the scalar product $(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^T \boldsymbol{y}$ where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ are defined over the finite element mesh R. From Equations (4.3) and (4.4),

$$(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{n_D} \boldsymbol{M}_i \boldsymbol{W}_i \boldsymbol{x}_i, \boldsymbol{y} \right)$$

$$= \sum_{i=1}^{n_D} \boldsymbol{x}_i^T \boldsymbol{W}_i^T \boldsymbol{M}_i^T \boldsymbol{y}$$

$$= \sum_{i=1}^{n_D} \boldsymbol{x}_i^T \boldsymbol{W}_i^T \boldsymbol{y}_i \tag{4.13}$$

Hence, scalar products are computed by first computing $\boldsymbol{x}_i^T \boldsymbol{W}_i^T \boldsymbol{y}_i$ on each domain, and then performing a global sum over all of the domains.

4.4.3 Preconditioner Computation

Using Equation (4.5), the global diagonal preconditioner can be calculated by the following equation

$$\boldsymbol{A}_{\boldsymbol{D}} = \sum_{i=1}^{n_D} \boldsymbol{M}_i \boldsymbol{A}_{\boldsymbol{D}_i} \boldsymbol{M}_i^T, \tag{4.14}$$

where $\boldsymbol{A}_{\boldsymbol{D}_i}$ are the domain contributions to $\boldsymbol{A}_{\boldsymbol{D}}$. The information in $\boldsymbol{A}_{\boldsymbol{D}}$ and $\boldsymbol{A}_{\boldsymbol{D}_i}$ can be stored in vectors $\boldsymbol{a} \in \mathbb{R}^n$ and $\boldsymbol{a}_i \in \mathbb{R}^{n_i}$ respectively using

$$\boldsymbol{a} = \boldsymbol{A}_{\boldsymbol{D}} \boldsymbol{1} \tag{4.15}$$

and

$$\boldsymbol{a}_i = \boldsymbol{A}_{\boldsymbol{D}_i} \boldsymbol{1}_i . \tag{4.16}$$

In these equations, vectors $\boldsymbol{1} \in \mathbb{R}^n$ and $\boldsymbol{1}_i \in \mathbb{R}^{n_i}$ are unit vectors. Equations (4.14), (4.16) and (4.3) allow us to modify Equation (4.15) as

$$\boldsymbol{a} = \sum_{i=1}^{n_D} \boldsymbol{M}_i \boldsymbol{A}_{\boldsymbol{D}_i} \boldsymbol{M}_i^T \boldsymbol{1}$$

$$= \sum_{i=1}^{n_D} \boldsymbol{M}_i \boldsymbol{A}_{\boldsymbol{D}_i} \boldsymbol{1}_i$$

$$= \sum_{i=1}^{n_D} \boldsymbol{M}_i \boldsymbol{a}_i \tag{4.17}$$

On the other hand, on each domain we need an updated version of $\boldsymbol{a}_i$, $\tilde{\boldsymbol{a}}_i$ that takes into account the shared nodes across the processors. This vector, which is the domain version of $\boldsymbol{a}$, can be calculated by

$$\tilde{\boldsymbol{a}}_i = \boldsymbol{M}_i^T \boldsymbol{a} . \tag{4.18}$$

Equation (4.17) can now be used to get

$$\tilde{\boldsymbol{a}}_i = \boldsymbol{M}_i^T \sum_{j=1}^{n_D} \boldsymbol{M}_j \boldsymbol{a}_j$$

$$= \boldsymbol{a}_i + \sum_{\substack{j=1 \\ j \neq i}}^{n_D} \boldsymbol{M}_i^T \boldsymbol{M}_j \boldsymbol{a}_j . \tag{4.19}$$

The data transfer across the domains is indicated by the product $\boldsymbol{M}_i^T \boldsymbol{M}_j$ in this equation.

The diagonal preconditioner computation is easily implemented in our element-by-element framework. Even though the system matrix is not assembled in our approach, using Equation (4.19) parallel computation of the diagonal preconditioner remains simple. By looping over the elements in its domain, each processor calculates the domain contribution to the diagonal of the global system matrix. Some inter-domain communication is then necessary using the communication operator $\boldsymbol{M}_i^T \boldsymbol{M}_j$ to update the values corresponding to the nodes shared across processors. This communication can be performed in the same manner explained above and shown in Figure 4.2. Once each processor has the updated version of the

diagonal terms for all of its nodes, the preconditioning process $\boldsymbol{d}^{(i+1)} = \boldsymbol{A}_D^{-1}\boldsymbol{r}^{(i+1)}$ can be performed independently on each domain since the preconditioner is the diagonal part of the system matrix.

4.4.4 Coarse-to-Fine Mesh Interpolation

On each domain, the interpolation operator $\boldsymbol{T}_i$ needs to be computed in order to interpolate the coarse mesh correction $\Delta\boldsymbol{x}_{c_i}$ using

$$\Delta\boldsymbol{x}_{f_i} = \boldsymbol{T}_i\Delta\boldsymbol{x}_{c_i}, \tag{4.20}$$

to obtain the corrections on the fine domain. Considering Equation (4.1), we should determine the expression for the global interpolation operator $\boldsymbol{T}$ to see if there is any need for data transfer between the domains to complete the interpolation process. Using Equations (4.4), (4.3) and (4.20) we can write

$$\begin{aligned}
\Delta\boldsymbol{x}_f &= \sum_{i=1}^{n_D} \boldsymbol{M}_{f_i}\boldsymbol{W}_{f_i}\Delta\boldsymbol{x}_{f_i} \\
&= \sum_{i=1}^{n_D} \boldsymbol{M}_{f_i}\boldsymbol{W}_{f_i}\boldsymbol{T}_i\Delta\boldsymbol{x}_{c_i} \\
&= \sum_{i=1}^{n_D} \boldsymbol{M}_{f_i}\boldsymbol{W}_{f_i}\boldsymbol{T}_i\boldsymbol{M}_{c_i}^{T}\Delta\boldsymbol{x}_c \\
&= \boldsymbol{T}\Delta\boldsymbol{x}_c
\end{aligned}$$

where $\boldsymbol{M}_{f_i}$ and $\boldsymbol{M}_{c_i}$ are the mapping matrices for the fine and coarse meshes on domain $\mathsf{D_i}$, respectively and $\boldsymbol{W}_{f_i}$ is the fine mesh weighting matrices for this domain. Therefore, the global interpolation operator is

$$\boldsymbol{T} = \sum_{i=1}^{n_D} \boldsymbol{M}_{f_i}\boldsymbol{W}_{f_i}\boldsymbol{T}_i\boldsymbol{M}_{c_i}^{T}. \tag{4.21}$$

Equation (4.21) indicates that there is no need for communications across the domains to form the interpolation operator $\boldsymbol{T}$. This makes the interpolation process straightforward to implement using our data structure. Since we are restricting attention to nested meshes, interpolation can proceed independently on each domain in the manner described in Chapter 3. This is implemented in the element-by-element framework by looping over the coarse mesh elements on each domain; no transfer of data between the domains is required.

4.4.5 Fine-to-Coarse Mesh Restriction

The restriction operator is the transpose of the interpolation operator (Equation (4.2)). However, its implementation is more complex. Equation (4.21) can be used to determine the restriction operator $\boldsymbol{T}^{\boldsymbol{T}}$;

$$\begin{aligned} \boldsymbol{T}^{\boldsymbol{T}} &= \sum_{i=1}^{n_D} \boldsymbol{M}_{c_i} \boldsymbol{T}_i^{\boldsymbol{T}} \boldsymbol{W}_{f_i}^{\boldsymbol{T}} \boldsymbol{M}_{f_i}^{\boldsymbol{T}} \\ &= \sum_{i=1}^{n_D} \boldsymbol{M}_{c_i} \boldsymbol{T}_i^{\boldsymbol{T}} \boldsymbol{W}_{f_i} \boldsymbol{M}_{f_i}^{\boldsymbol{T}} \end{aligned} \tag{4.22}$$

where $\boldsymbol{W}_{f_i}^{\boldsymbol{T}} = \boldsymbol{W}_{f_i}$ because the weighting matrices are diagonal. Then, using Equations (4.2) and (4.3),

$$\begin{aligned} \boldsymbol{r}_c &= \sum_{i=1}^{n_D} \boldsymbol{M}_{c_i} \boldsymbol{T}_i^{\boldsymbol{T}} \boldsymbol{W}_{f_i} \boldsymbol{M}_{f_i}^{\boldsymbol{T}} \boldsymbol{r}_f \\ &= \sum_{i=1}^{n_D} \boldsymbol{M}_{c_i} \boldsymbol{T}_i^{\boldsymbol{T}} \boldsymbol{W}_{f_i} \boldsymbol{r}_{f_i} \end{aligned} \tag{4.23}$$

Also,

$$\begin{aligned} \boldsymbol{r}_{c_i} &= \boldsymbol{M}_{c_i}^{\boldsymbol{T}} \boldsymbol{r}_c \\ &= \boldsymbol{M}_{c_i}^{\boldsymbol{T}} \sum_{j=1}^{n_D} \boldsymbol{M}_{c_j} \boldsymbol{T}_j^{\boldsymbol{T}} \boldsymbol{W}_{f_j} \boldsymbol{r}_{f_j} \end{aligned}$$

$$= \boldsymbol{T}_i^T \boldsymbol{W}_{f_i} \boldsymbol{r}_{f_i} + \sum_{\substack{j=1 \\ j \neq i}}^{n_D} \boldsymbol{M}_{c_i}^T \boldsymbol{M}_{c_j} \boldsymbol{T}_j^T \boldsymbol{W}_{f_j} \boldsymbol{r}_{f_j} \qquad (4.24)$$

Here, $\boldsymbol{M}_{c_i}^T \boldsymbol{M}_{c_j}$ represents data transfer from $\mathsf{D_j}$ to $\mathsf{D_i}$. Thus, $\boldsymbol{r}_{c_i}$, the coarse mesh residual on domain $\mathsf{D_i}$ can be computed by calculating $\boldsymbol{T}_i^T \boldsymbol{W}_{f_i} \boldsymbol{r}_{f_i}$ on each domain, and then transferring border data between domains on the coarse mesh in the same manner as shown in Figure 4.2.

The calculation of $\boldsymbol{T}_i^T \boldsymbol{W}_{f_i} \boldsymbol{r}_{f_i}$ requires careful consideration. To demonstrate, consider the two dimensional nested meshes in Figure 4.4. The figure shows four of the fine mesh elements that compose a coarse mesh element and the weighted transfer of data from the fine mesh to the coarse mesh required by the computation of $\boldsymbol{T}_i^T \boldsymbol{W}_{f_i} \boldsymbol{r}_{f_i}$. Each of the coarse mesh nodes receives the fine mesh values at that node, plus $\frac{1}{4}$ the values of the adjacent fine mesh center nodes (e.g., node *f* in Figure 4.4) and $\frac{1}{2}$ of the values of adjacent fine mesh edge nodes (e.g., node *g* in Figure 4.4). By considering all of the elements in $\mathsf{D_i}$, the result of this transfer is, for example,

$$\boldsymbol{r}_{c_a} = \boldsymbol{r}_{f_a} + \tfrac{1}{2}\left(\boldsymbol{r}_{f_i} + \boldsymbol{r}_{f_k} + \boldsymbol{r}_{f_e} + \boldsymbol{r}_{f_g}\right) + \tfrac{1}{4}\left(\boldsymbol{r}_{f_h} + \boldsymbol{r}_{f_j} + \boldsymbol{r}_{f_l} + \boldsymbol{r}_{f_f}\right) \qquad (4.25)$$

This calculation can be implemented by looping over the coarse mesh elements in the domain and transferring the values at the fine mesh corner, side and center nodes in turn. In order to ensure that each fine mesh nodal quantity is transferred only once as all of the coarse mesh elements are processed, each fine mesh value is set to zero after it has first been transferred.

4.4.6 DAXPY Operations

The remaining operations required by the algorithm (i.e., DAXPY's) can be performed independently on each domain by virtue of the mash partitioning. Since each domain receives the components of a vector defined for the mesh that belong to the degrees-of-freedom present in the domain, the DAXPY $\alpha \boldsymbol{x} + \boldsymbol{y}$ can be computed independently on each domain as $\alpha \boldsymbol{x}_i + \boldsymbol{y}_i$; this requires that α be stored on each domain.

4.5 Data Structure and Implementation Issues

The proposed algorithm is implemented in a portable code using most of the features of Fortran 90. One of our objectives is to develop a single code that can be used on a wide variety of parallel architectures. The efficiency of this work will be shown later by solving some benchmark problems on different parallel systems. Features of the Fortran 90 language that make this possible are described in the next section.

4.5.1 Object-Based Programming

The addition of modules, pointers and derived types to the Fortran standard significantly aided in the development and maintenance of the code written for this study. The necessary procedures can be written to operate on the various data objects, such as meshes and elements, without the overhead of memory management common to FORTRAN 77 implementations. User defined data types are employed to generate the necessary data structures on each processor and all storage is dynamically allocated using pointers.

The mesh hierarchy is managed using a doubly linked list that facilitates the intermesh switching required by the multigrid algorithm (Figure 4.5). A derived type named mesh_pointers contains pointers to the global vectors and element information required on each mesh. There are two stationary pointers that locate the data for the finest and coarsest meshes, and a moving pointer provides access to the data required on the mesh currently under consideration (e.g., when relaxation is being performed on one of the coarse meshes).

Figure 4.6 outlines the primary information that the mesh_pointers data type can include. Different element types are easily included through the definition of derived types such as mesh_brick8, a pointer to eight node brick element data or mesh_nl_brick8, a pointer to nonlinear eight node brick element data. Other pointers to handle inter-processor communications are also included which are explained in the next section. The pointers finest_mesh and coarsest_mesh are placed in a module that is accessible to all procedures. This global module also contains vectors used on the finest mesh and replaces the common blocks familiar to FORTRAN 77 programmers.

4.5.2 Interprocessor Communications

So far, we have not explicitly identified the domains with individual processors of a parallel computer. In some instances, such as mesh generation using adaptive refinement, multiple domains may be placed on a single processor in order to maintain load balance. However, these cases are not discussed here, and we make the assumption that each domain is assigned to a single processor (i.e., domain $\mathbf{D_i}$ is assigned to processor $\mathbf{P_i}$). We also assume that the mesh is partitioned so that the number of elements in each domain is the same on a given mesh (this is discussed further in the next section).

Interprocessor (or inter-domain) communication is performed using non-blocking communication procedures from the standard MPI library [6]. Two types of interprocessor communication need to be considered: local data transfer during the matrix-vector products (Equations (4.8) and (4.24)) and global data transfer during the scalar products (Equation (4.13)). The mesh_pointers data type that was described in the previous section also includes the necessary pointers to handle these communications. The pointers mesh_mapping_send and mesh_mapping_recv are used for local data transfers and dot_product_comm is used for global data transfers.

First, consider the local transfer via the communication matrix $\boldsymbol{M}_i^T \boldsymbol{M}_j$. Processor $\mathbf{P_i}$ contains a list of the n_{cp_i} processors that it communicates with, i.e., $\mathbf{P_j}$, j = 1,…, n_{cp_i} (these are the processors that contain degrees-of-freedom that also belong to $\mathbf{P_i}$). The type of the pointers mesh_mapping_send and mesh_mapping_recv is defined as inter_processor_mapping: a derived data type that provides all the required information (Figure 4.7). For example, on processor $\mathbf{P_i}$, mesh_mapping_send($\mathbf{P_j}$) includes the list of nodes on $\mathbf{P_i}$ that also belong to $\mathbf{P_j}$ (border_nodes) and the degrees-of-freedom associated with these border nodes on $\mathbf{P_i}$ (send_equ_num). Similarly, mesh_mapping_recv($\mathbf{P_j}$) contains the information about $\mathbf{P_j}$ the receiving processor, e.g., a list of the degrees-of-freedom on $\mathbf{P_j}$ that $\mathbf{P_i}$ sends information to (indx). Using this data structure, exchange of data between processors can be performed on each mesh in the multigrid solution algorithm. This interchange is depicted in Figure 4.8.

We are dealing with unstructured meshes and this means the neighboring nodes have non-contiguous equation numbers. To avoid short messages and therefore latency dominated transfer times, all of the sending values are first packed into the sending buffer on each processor. This data is received in a receiving buffer and then unpacked to update the appropriate vector. Using non-blocking communication procedures (MPI_Isend and MPI_Irecv) avoids deadlocks and makes concurrent transfer of data possible. Deadlock might happen in blocking communications when all of the sends block, waiting for a matching receive. Deferring synchronization is another reason for using non-blocking operations. A send-receive operation accomplishes two tasks: data transfer and synchronization. In many cases, blocking procedures would introduce more synchronization than required. Use of non-blocking operations and MPI_Waitall defer synchronization.

The global data transfer for scalar products is achieved using the MPI_AllReduce procedure. To compute $\alpha = (\boldsymbol{x}, \boldsymbol{y})$, processor $\mathsf{P_i}$ computes $\alpha_i = \boldsymbol{x}_i^T \boldsymbol{W}_i^T \boldsymbol{y}_i$. The global value of α is computed and becomes available to all of the processors using MPI_AllReduce as depicted in Figure 4.9. The weighting matrix $\boldsymbol{W}_i$ is stored as vector on $\mathsf{P_i}$ and in the mesh_pointers data type (Figure 4.6) is represented as dot_product_comm.

4.6 Mesh Generation and Partitioning

The multigrid solution algorithm requires a hierarchy of increasingly finer meshes. Adaptive mesh refinement schemes can be used to create the necessary meshes; the distributed implementation of these schemes requires a separate research effort. Here, we employ the *Truegrid* mesh generation software [35] to produce a sequence of nested, uniformly refined hexahedral meshes. *Truegrid* is based on a technique known as the projection method. This method allows faces, edges, and nodes of the mesh to be placed on surfaces. In addition, edges and nodes of the mesh can be placed along curves by this method. Creating geometry and creating a mesh are two separate phases in *Truegrid*. First, surfaces and curves are created (or imported) and then a block mesh is produced and molded to the shape of the geometry. Complex parts can be modeled in this manner. Figure 4.10 shows a sequence of nested, uniformly refined meshes for a solid rocket motor that were generated using this method.

In this study, in order to get a high quality decomposition of the finite element domain, we employ the METIS graph-partitioner software [27]. The resulting decomposition balances the computational load among the processors while simultaneously minimizing the communication cost between the processors. A graph representation of the element connectivity is generated by METIS, which is coupled with a multi-level partitioning scheme to attain uniform computational loads and to minimize domain boundaries. When the domain consists of different element types with different computational costs, non-uniform weights can be assigned to the domain to provide METIS with enough information (i.e., relative computational cost of elements) to produce a load-balanced decomposition.

Partitioning is performed on the coarsest mesh using METIS. Figure 4.11 illustrates the coarsest mesh partitions for eight processors of a solid rocket motor. Uniform refinement of these partitions produces partitions on all fine meshes. Using this method, perfect element load balance is maintained throughout the mesh hierarchy but communications may not be optimum for the fine meshes. In the next chapter, by showing the results from some benchmark runs, it will become clear these non-optimal communications does not introduce any significant overhead. *Truegrid* capabilities make the refinement process a trivial task and provide enough information about the coarse meshes to prepare all the necessary input data required in the multigrid solution method.

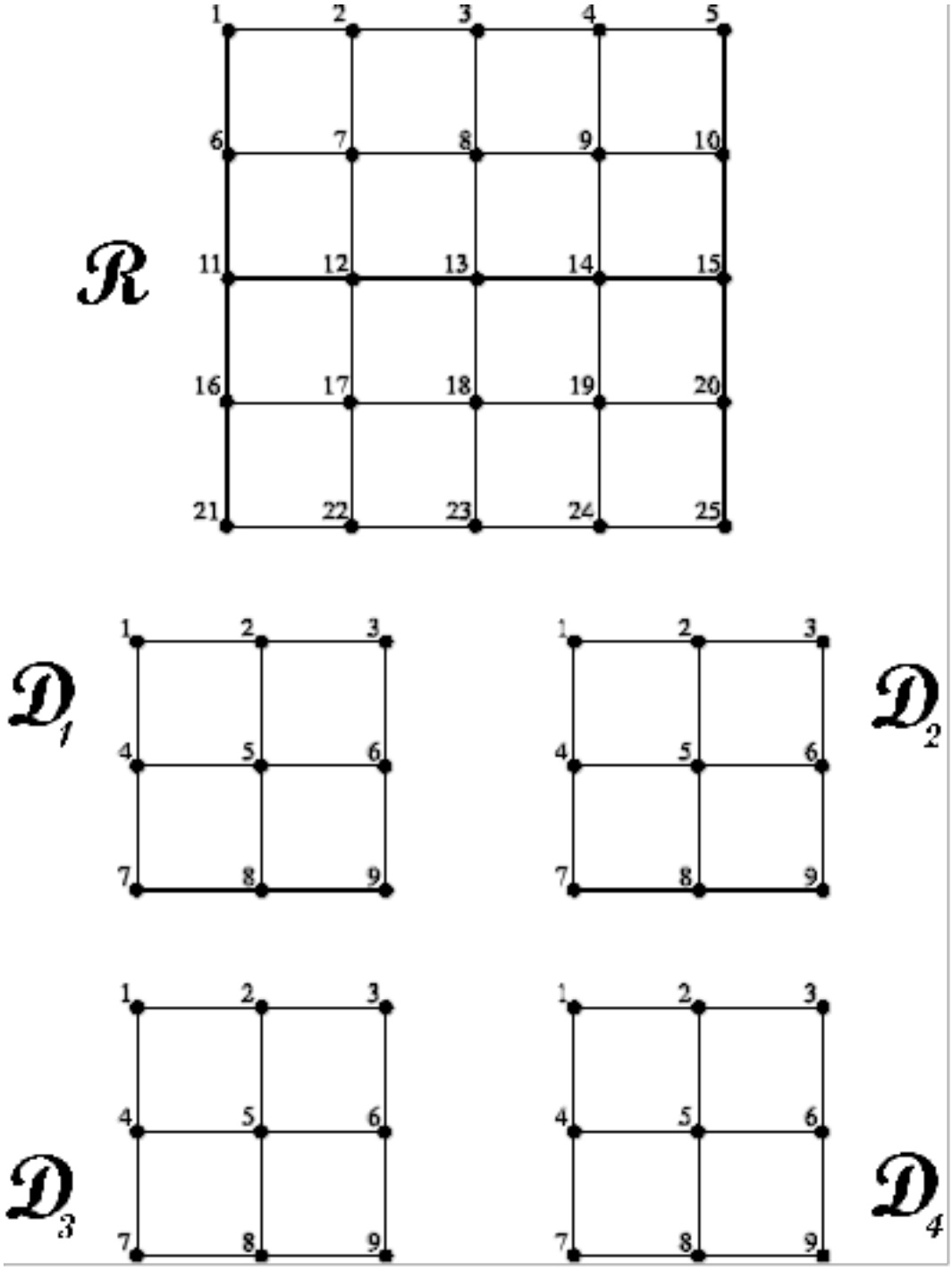


Figure 4.1: A Sixteen Element Mesh Partitioned into Four Domains.

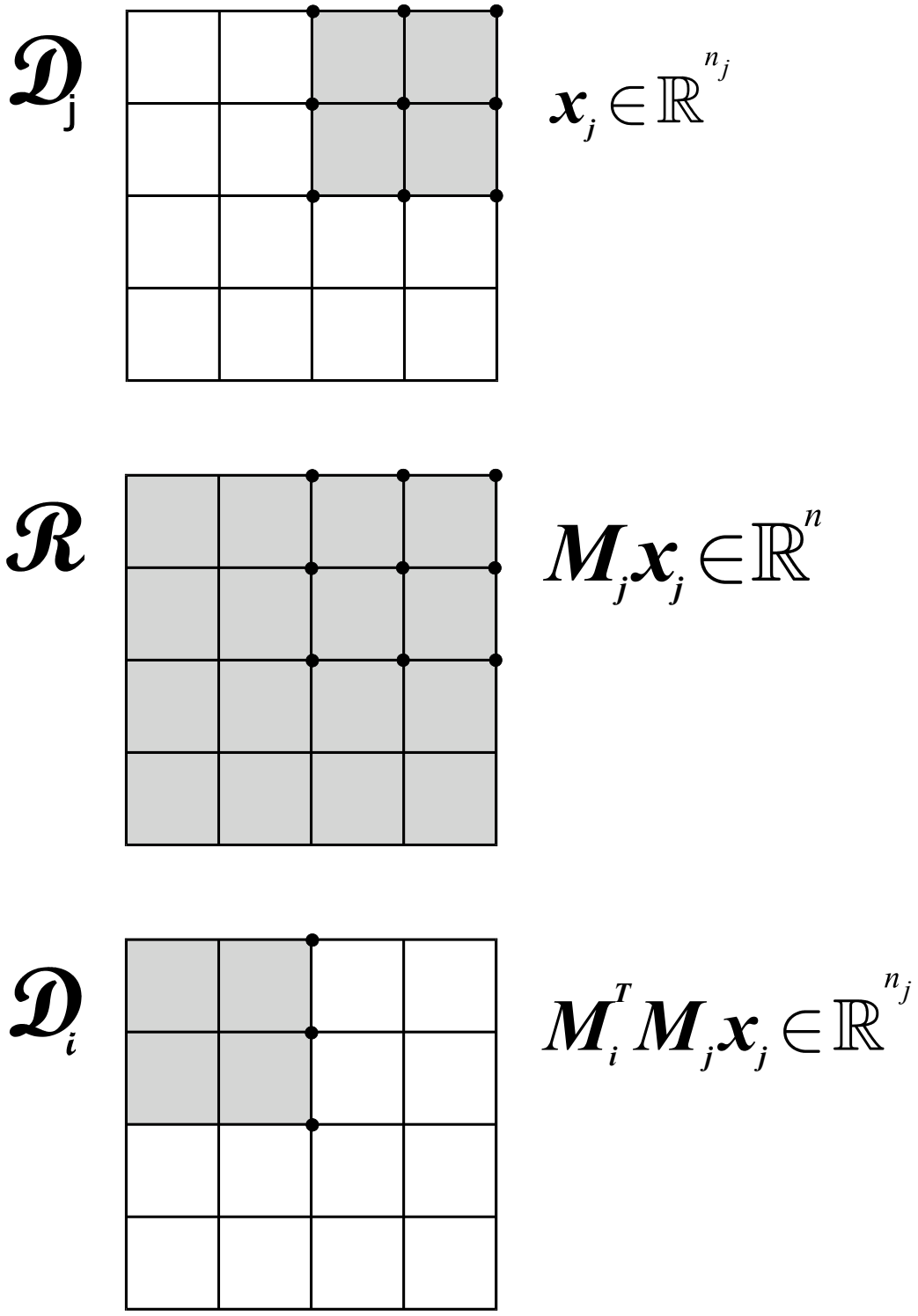


Figure 4.2: The Communication matrix $M_i^T M_j$ Transfers Variables from D_j to D_i.

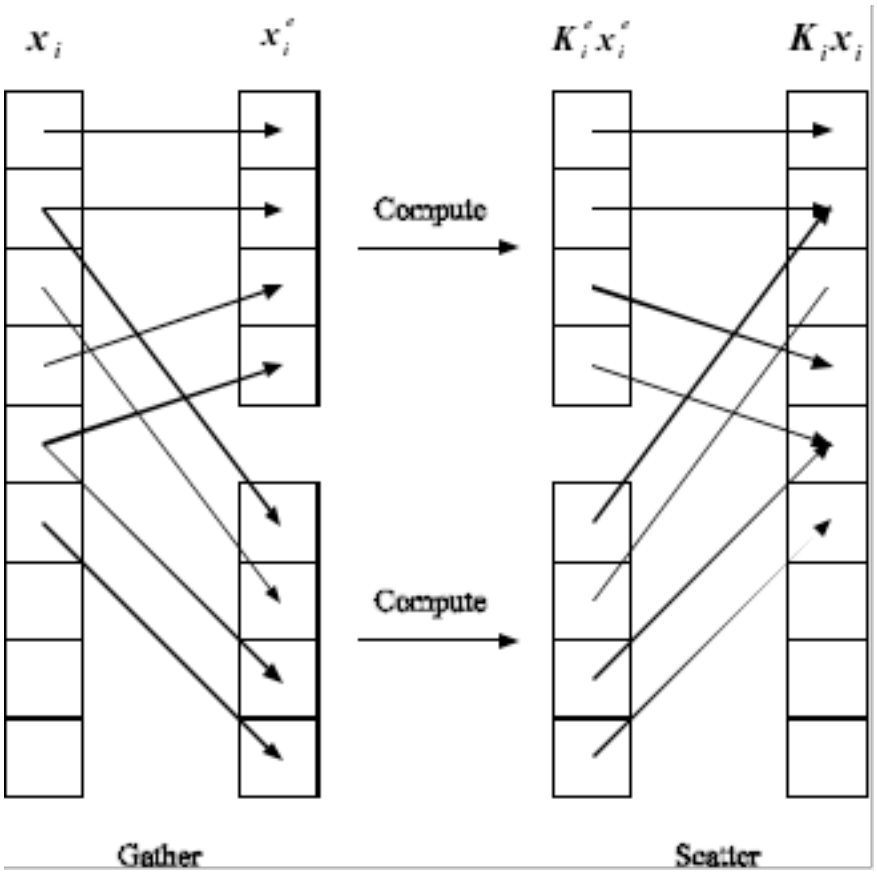


Figure 4.3: Gather-Scatter Operation for Matrix-Vector Products on each Domain.

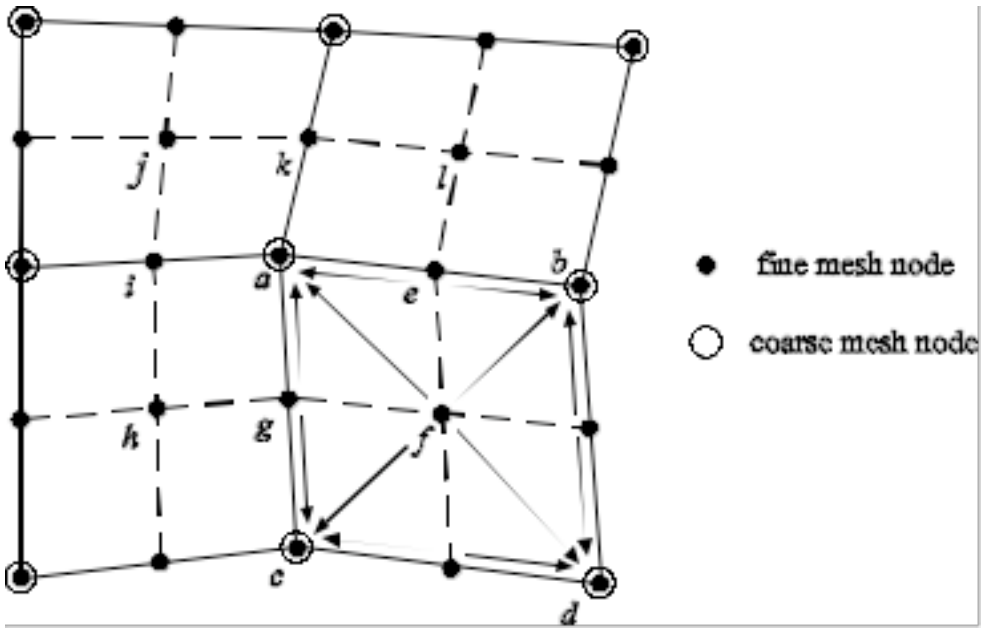


Figure 4.4: Calculation of $\boldsymbol{T}_i^T \boldsymbol{W}_{f_i} \boldsymbol{r}_{f_i}$ for a Two Dimensional Nested mesh.

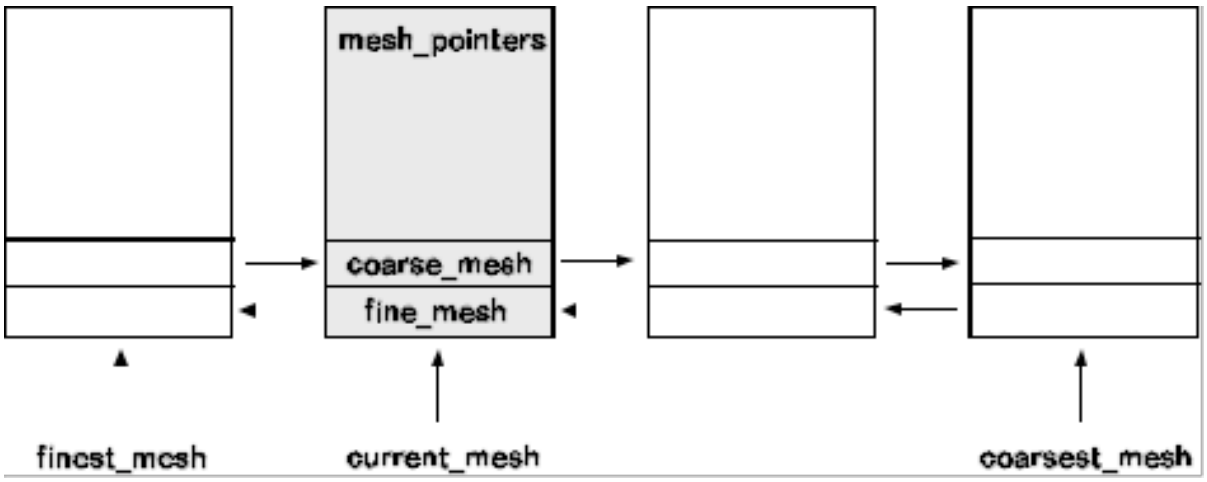


Figure 4.5: Double Linked List for the Hierarchy of Meshes.

```
TYPE  mesh_pointers

   INTEGER  : :  num_equ
   INTEGER  : :  mesh_num

   REAL(KIND=double),  DIMENSION( : ),  POINTER  : :  f
   REAL(KIND=double),  DIMENSION( : ),  POINTER  : :  k_diag_inv
   REAL(KIND=double),  DIMENSION( : ),  POINTER  : :  x

   TYPE(mesh_quad4),       POINTER  : :  quad4
   TYPE(mesh_brick8),      POINTER  : :  brick8
   TYPE(mesh_nl_brick8),  POINTER  : :  nl_brick6

   TYPE(mesh_pointers),   POINTER  : :  coarse_mesh
   TYPE(mesh_pointers),   POINTER  : :  fine_mesh

   TYPE(inter_processor_mapping), DIMENSION( : ), POINTER  : : mesh_mapping_send
   TYPE(inter_processor_mapping), DIMENSION( : ), POINTER  : : mesh_mapping_recv

   INTEGER, DIMENSION( : ),  POINTER  : : dot_product_comm

 END TYPE  mesh_pointers
```

Figure 4.6: Derived Type Used for Mesh Data.

```
TYPE  inter_processor_mapping

   INTEGER,  DIMENSION ( : ),  POINTER : :  border_nodes
   INTEGER,  DIMENSION ( : ),  POINTER : :  indx

   INTEGER,  DIMENSION ( : ),  POINTER : :  send_equ_num

   INTEGER : :  send_count

   INTEGER : :  num_border_np

END TYPE inter_processor_mapping
```

Figure 4.7: Derived Type That Provides All the Required Information for Communications Between Domains on Each Mesh.

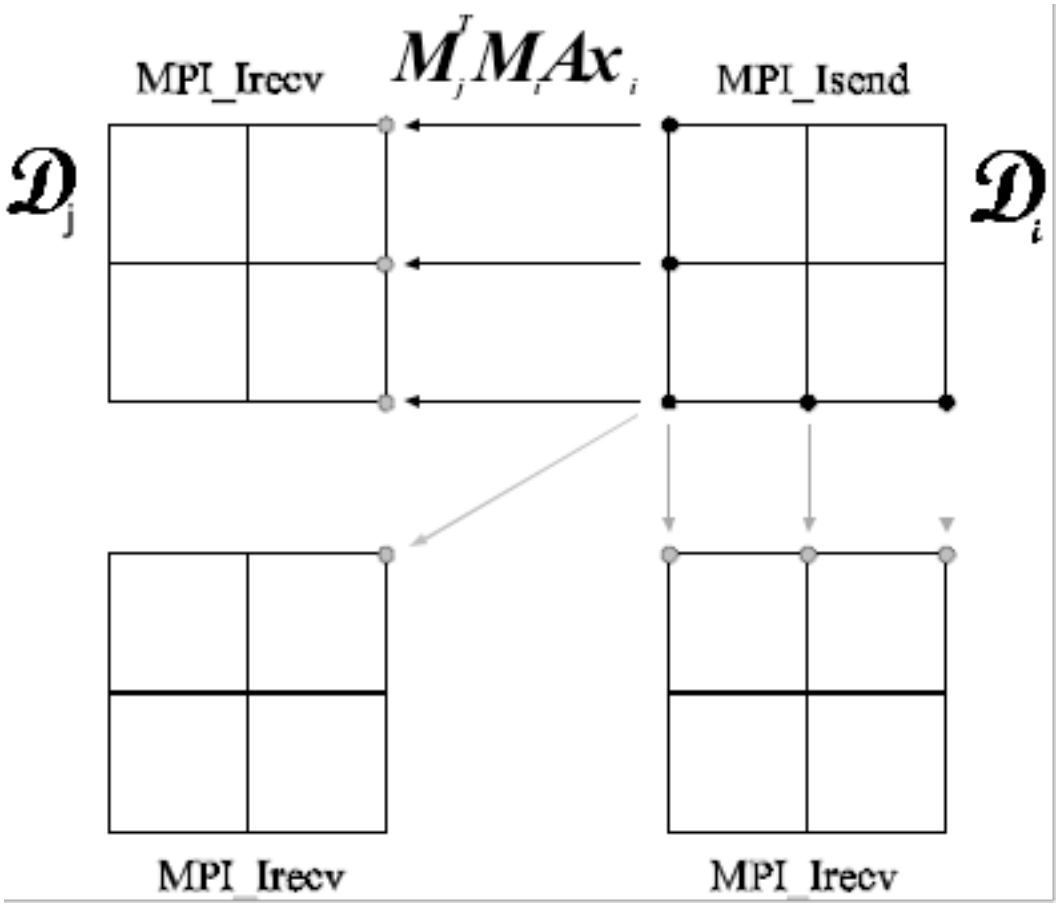


Figure 4.8: Local Inter-Domain Data Transfer Via the Communication Matrix $M_i^T M_j$.

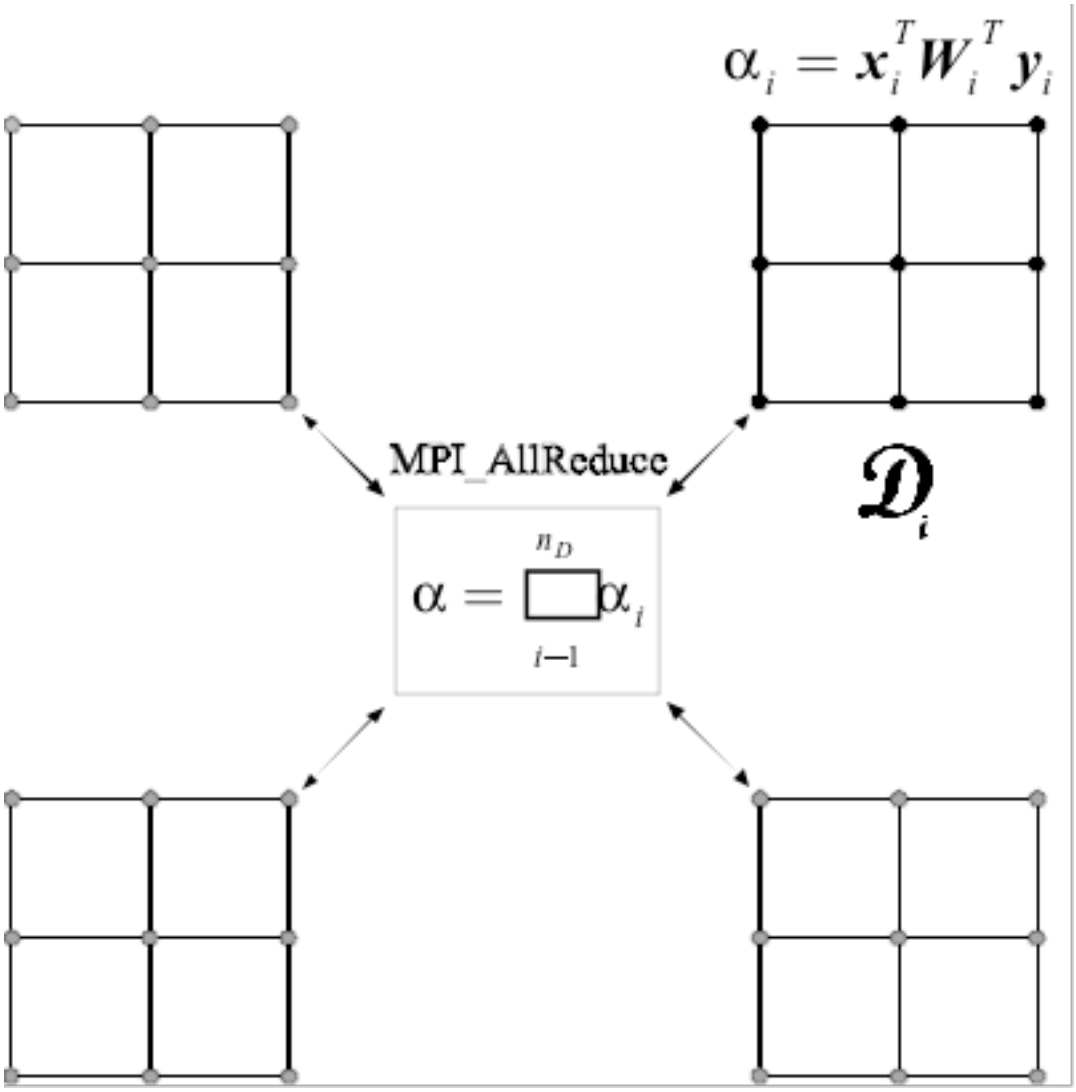


Figure 4.9: Global Data Transfer to Compute Scalar Products.

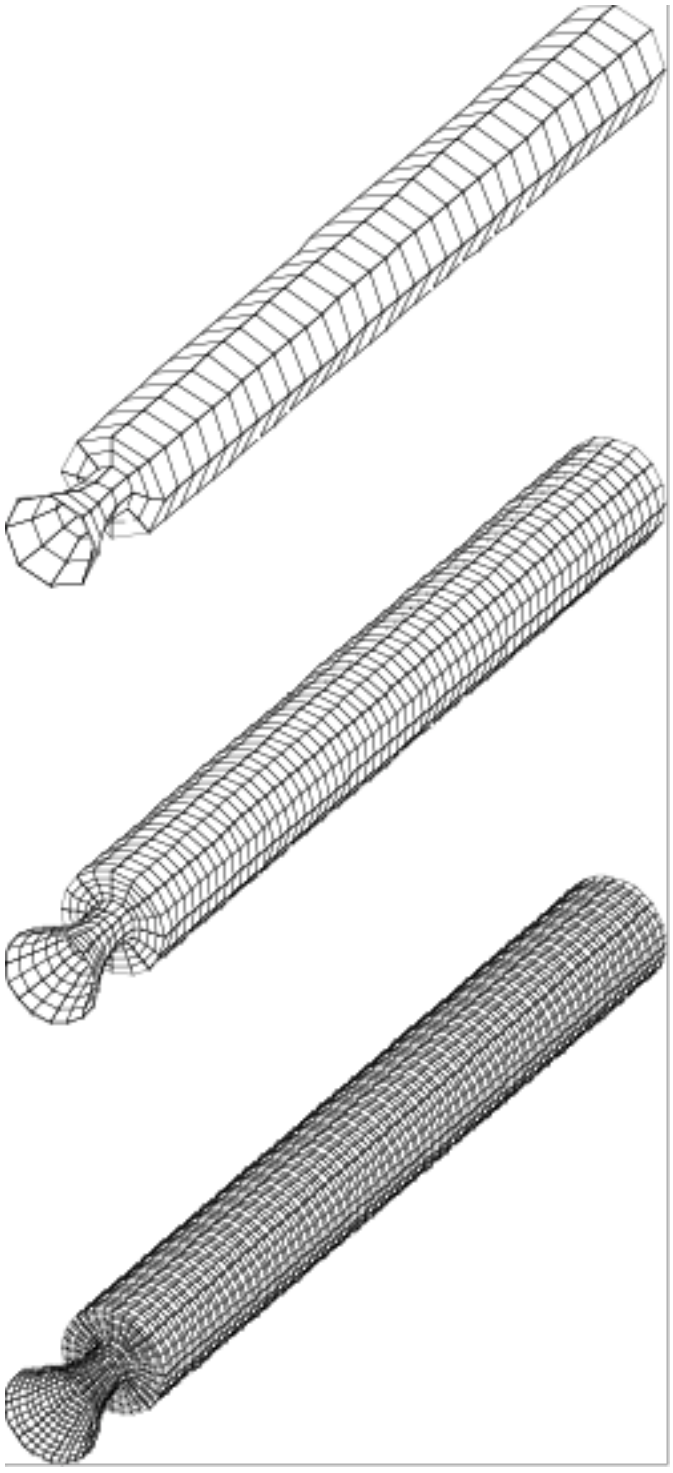


Figure 4.10: A Sequence of Nested, Uniformly Refined Meshes for a Solid Rocket Motor. The Finest Mesh has 262,144 Elements.

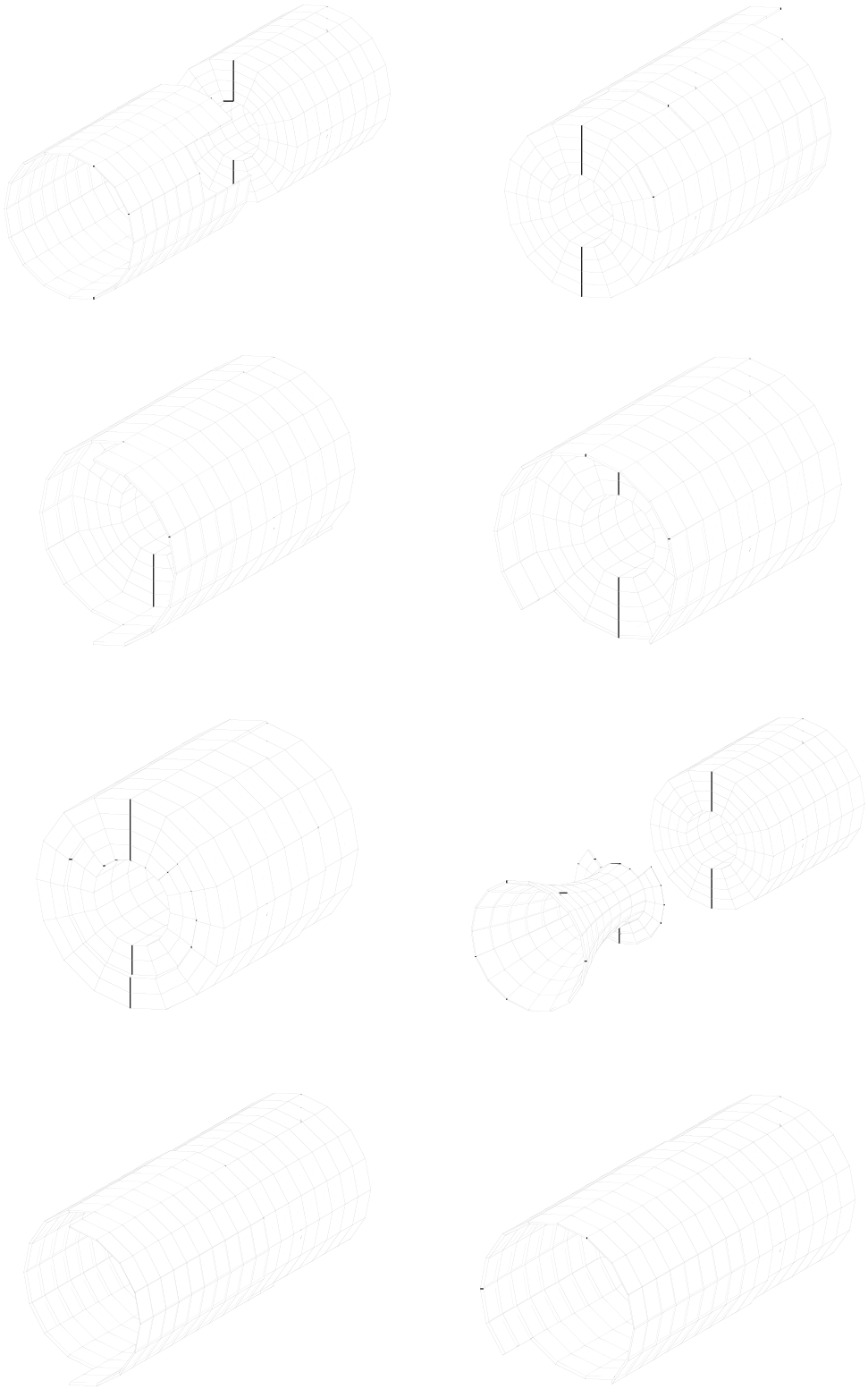


Figure 4.11: Coarsest Mesh Partitions for Eight Processors of a Solid Rocket Motor.

CHAPTER 5

PARALLEL PERFORMANCE

Parallelization of the multigrid solution algorithm based on the domain decomposition methodology was described in the previous chapter. Different components of this method were examined to detect any necessary data transfer between the domains and the implementation issues for an efficient communication algorithm were discussed. The matrix free approach embedded in the element-by-element framework was adopted to reduce the required memory and CPU time. Here, the performance of the parallel algorithm described in this document is studied using two sets of benchmark problems: fixed-size problems and scaled-size problems. In the following sections, these benchmark problems and the measures used to evaluate performance are explained. In order to demonstrate the portability of the code, analyses were conducted in dedicated mode on three different parallel machines: an SGI Origin2000, an IBM SP2 and a CRAY T3E. The performance of the algorithm on these three machines is compared. A cost analysis is then performed for the scaled-size problems by timing all the necessary communications separately and comparing with the total computation time.

5.1 Description of the Computer Architectures Employed

This section outlines the three parallel machines used to benchmark the algorithm described in this document: an SGI Origin2000 at the National Computational Science Alliance (NCSA) at the University of Illinois, an IBM SP2 at the Argonne National Laboratory and a CRAY T3E at Pittsburgh Supercomputing Center.

The SGI Origin2000 has a distributed shared-memory architecture with a number of processing nodes linked together by an interconnection fabric. Each processing node contains either one or two processors, a portion of shared memory, a directory for cache coherence, and two interfaces: one that connects to I/O devices and another that links system nodes through the interconnection fabric. The Origin2000 uses Silicon Graphics' Scalable Shared-Memory Multiprocessor (S2MP) architecture to distribute shared memory amongst the nodes. This shared memory is accessible to all processors through the interconnection fabric, which links nodes to each other and is a mesh of multiple, simultaneous, dynamically-allocated switch-connected links. The processors are MIPS R10000, a 64-bit superscalar processor which supports dynamic scheduling. A maximum of 128, 250 Mhz processors and 64 GB of memory were available for this research. In addition to shared memory programming model, message passing and data-parallel programming models are supported on this machine.

The IBM SP2 is a scalable parallel system based on distributed memory message passing architecture. The processor nodes consists of a POWER2 Super Chip (P2SC) microprocessor or PowerPC symmetric multiprocessor (SMP), memory, Micro Channel expansion slots for I/O and connectivity and disk devices. Three types of nodes (thin, wide and high) are available and can be mixed in a system. Internode communication is performed by the SP switch, which is a high-performance, multi-stage, packet-switched network that maintains the point-to-point communication time independently from the relative position of the nodes. We used an SP2 with 80 P2SC 120 Mhz thin nodes and 256 MB RAM per node.

The CRAY T3E is a scalable parallel system and for this research, we had access to a T3E with 512 processing elements (PE's), which were high-performance Digital Alpha 64-bit microprocessors, each running at 450 Mhz. CRAY T3E processing elements are tightly coupled by the Cray interconnect: a three-dimensional bi-directional torus capable of very low latency and high bandwidth. The memory is physically distributed, with each PE having 128 MB. CRAY T3E has another feature called STREAMS, which maximizes local memory bandwidth, allowing the microprocessor to run at full speed for vector-like data references. CRAY T3E systems support both explicit distributed memory parallelism through CF90 and C/C++ with message passing (MPI or PVM) and data-parallel programming models and implicit parallelism through HPF and the Cray CRAFT work-sharing features.

5.2 Fixed-Size Problems

we evaluate the scalability of our parallel algorithm by solving some fixed-size problems and observe how effectively our proposed scheme can use an increased number of processors. In these experiments, our approach to quantify parallel performance is to determine how elapsed wall clock time and efficiency vary with increasing processor count for a fixed problem size. We also measure the speedups for each of these runs. Speedup, S_p, and efficiency, E_p, are defined by the following equations:

$$S_p = \frac{t_1}{t_p} \tag{5.1}$$

and

$$E_p = \frac{S_p}{P} \tag{5.2}$$

where t_1 is the elapsed wall clock time on one processor and t_p is the time on P processors. Efficiency is a measure independent of problem size that can indicate how effective the algorithm uses the computational resources.

Amdahl's law [28] is also used in this section for our performance evaluation. This law states that the time required to run a process on P processors, t_p, is

$$t_p = (1-f)t_1 + \frac{f}{P}t_1, \tag{5.3}$$

where f is the fraction of the process running in parallel. From this equation, the speedup can be derived as

$$\frac{t_1}{t_p} = S_p = \frac{1}{(1-f)+\frac{f}{P}}, \tag{5.4}$$

which means the maximum possible speedup that can be achieved on a parallel computer (with infinite processors) is limited by $\frac{1}{(1-f)}$.

Amdahl's law should not be viewed as the only explanation for scalability limitations of a parallel algorithm because these limits may be due to communication costs, replicated

computation or idle time rather than the existence of sequential components. On the other hand, if we want to claim that our algorithm does not include any sequential parts, we should be able to find a set of problems for which the computed fraction of parallelism from the Amdahl's law equation is equal to one. Although Amdahl's law is based on a simple model of parallel computing, it still can reveal important features of the performance of a given implementation.

Since we only had access to multi-processor machines (i.e., we were unable to run our test problems on a single processor of either the three target machines due to restrictions on time or memory), we used a least squares fit to estimate the values of t_1 and f, and calculated S_p and E_p from Equations (5.1) and (5.2). The overall parallel performance is given for a single complete full multigrid solve that includes 10 fine mesh relaxations (five at the beginning and another five after the interpolation), interpolation, restriction and coarse mesh solution. In the next section, we describe the specifications of these fixed-size problems.

5.2.1 Problem Specifications

A simple three dimensional problem was used to test the performance of the parallel algorithm for fixed-size problems. A coarse mesh with 64 eight node trilinear brick elements was generated to discretize a cube. This mesh was then uniformly refined three and four times to produce two sets of problems with 32,768 elements and 262,144 elements at the finest level (Figure 5.1 shows the first three meshes of increasing refinement for this cube). These two sets of problems (each partitioned for different number of processors) have 35,645 nodes (or 110,435 degrees-of-freedom) and 275,846 nodes (or 765,983 degrees-of-freedom) respectively. One side of this cube was held fixed and some uniform tractions were applied at the other sides as shown in Figure 5.2. A Bilinear (Mises) material model with steel properties was assumed.

Decompositions of this model were done using the method described in Chapter 4. The coarsest mesh was decomposed into 8, 16, 32 and 64 partitions and each partition was then uniformly refined to produce the partitions for finer levels. Figure 5.3 shows the decomposition of the problem with 32,768 elements into eight partitions at the third level. On the SGI Origin 2000, a bigger problem with 2,097,152 elements (2,145,785 nodes and 6,749,745 degrees-of-freedom) was also generated to be solved on up to 128 processors by producing a

coarse mesh with 128 elements and refining it four times. This problem could not be solved on the CRAY T3E and IBM SP2 that we had access to because of the memory limits of these machines. The total amount of memory required by our code to solve these three sets of problems (with 32,768 elements, 262,144 elements and 2,097,152 elements) was 184 MB, 1.45 GB and 11.6 GB respectively. The memory needed to allocate the communication arrays in the algorithm is minimal in our implementation.

5.2.2 Results and Discussion

Figures 5.4, 5.5 and 5.6 show the wall clock times required by the test problems when run in dedicated mode on various numbers of processors of the SGI Origin2000. Note that these timings include all of the costs associated with the algorithm: computation, communication and idle time (if any). As mentioned before, these problems maintain perfect load balance across the processors and, ideally, there should be no idle time for any of the processors. However, as we see later in this chapter (for the scaled-size problems), in case of the SGI Origin2000, some idle time can be measured for some of the processors due to certain anomalies. On each of these figures we also report the values for t_1, the time required on one processor and f, the fraction of the process running in parallel, which were both computed using a least squares fit to Amdahl's law, equation (5.3). The curves shown in these figures are the result of this least squares fit. Excellent results are observed for the test problems, especially those with the larger number of degrees-of-freedom. The problem with 32,768 elements was solved efficiently up to 32 processors. Similar results for the wall clock times are shown for the IBM SP2 (Figures 5.7 and 5.8) and the CRAY T3E (Figures 5.9 and 5.10). Speedups are shown in Figure 5.11 for the SGI Origin2000, Figure 5.12 for the IBM SP2 and Figure 5.13 for the CRAY T3E. Parallel efficiencies are also shown in Figures 5.14, 5.15 and 5.16 for these three parallel machines.

Superlinear speedups and efficiencies exceeding 1.0 are observed in some cases due to cache effects, i.e., when a greater number of processors are used to solve a fixed-size problem, more of its data can be placed in the cache memory. As a result, total computation time will tend to decrease. If this decrease in computation time offsets increases in communication and idle times resulting from the use of additional processors, then speedup will be superlinear and efficiency and the fraction of parallelism will be greater than one. Among the three

machines employed in this study, the SGI Origin2000 has the largest cache memory size (32 KB primary and 4 MB secondary) and therefore shows the strongest cache effects.

Parallel performance of the three parallel machines is compared in Figure 5.17 where the elapsed wall clock time required by each of these machines to solve the problem with 262,144 elements are shown together. The CRAY T3E appears to be the fastest among the three, although on 64 processors, using its strong cache performance, the SGI Origin2000 is able to solve the problem in almost the same amount of time as the CRAY T3E.

In addition, the floating point performance of these machines is compared in Figure 5.18. Again, the CRAY T3E shows a better result, e.g., on 64 processors, 3587 MFLOPS (or 56 MFLOPS per processor) was obtained during the solution. Figure 5.19 shows the floating point performance of the SGI Origin2000 for all three problems where the largest problem was solved on 128 processors. In this case, 4108 MFLOPS (or 32 MFLOPS per processor) was achieved. We should note that the source code used on the three machines was identical; no fine tuning was done in each case to improve performance.

5.3 Scaled-Size Problems

Solving fixed-size problems on different numbers of processors does not usually demonstrate the scalability of an algorithm due to certain anomalies caused by the changes in the conditions in which processors function. When the number of processors for a given problem is increased, the amount of work and the memory requirements per processor can change dramatically. As noted in the previous section, cache effects are the most common anomaly witnessed in these circumstances that can interfere with scalability studies and sometimes make it impossible to obtain real performance results for the implemented algorithm. Also, for a large fixed-size problem, virtual memory paging might become necessary when smaller number of processors are used that can cause a considerable degradation in performance. The problem would then be solved faster on a larger number of processors in the absence of virtual memory paging, resulting in superlinear speedups. These observations encourage a different approach to parallel performance analysis called scaled problem analysis, whereby we use scaled-size problems, i.e., we increase the model size with the number of processors to maintain the same work load per processor.

Often, large parallel computers are used to solve larger problems rather than solving fixed-size problems faster. This is another motivation for doing a scaled problem analysis. The fixed-size problem analysis allows us to find out what is the fastest one can solve a certain problem on a certain machine using a proposed algorithm, whereas the scaled-size problem analysis reveals not only the parallel scalability but also the numerical scalability of the algorithm which demonstrates its ability to solve larger problems.

The scaled speedup is defined by the following equation,

$$S_p = \frac{Pt_1}{t_p}, \tag{5.5}$$

where t_1 is the elapsed wall clock time on one processor and t_p is the time on P processors when solving a scaled-size problem. In this section, we report parallel performance for a single complete full multigrid solve that includes 10 fine mesh relaxations per iteration (five at the beginning and another five after interpolation), interpolation, restriction and coarse mesh solution. Next section describes the specifications of our scaled-size test problems.

5.3.1 Problem Specifications

A section of the grain (propellant) of a solid rocket motor subjected to internal pressure was modeled to generate scaled-size test problems. This model has four meshes of increasing refinement with 8192 elements and 9792 nodes at the finest level on a single processor (Figure 5.20). The nodes along the curved outside surface were fixed to represent the high stiffness of a casing. A simple linear-elastic material model was assumed for the fuel with E = 3.4 MPa, $\nu = 0.49$. To generate scaled-size problems for 8, 16, 32, 64, 128, 256 and 512 processors, the length of the propellant was increased by putting together the same number of layers of the above mesh. Table 5.1 contains the number of elements and nodes of the resulting meshes for up to 512 processors and Figure 5.21 shows the decomposition of the model generated for 8 processors. There was 40 MB of memory per processor required to solve the scaled-size test problems.

5.3.2 Results and Discussion

Tables 5.2, 5.3 and 5.4 summarize the scalability and parallel performance results for a full multigrid solve on the CRAY T3E, the IBM SP2 and the SGI Origin2000 respectively.

These results include total elapsed wall clock time, scaled speedups and efficiency for different numbers of processors. On the CRAY T3E and the IBM SP2, wall clock times remain almost constant when the number of processors is increased up to the maximum number available on these machines, which shows our parallel algorithm is highly scalable. This also explains the excellent speedup and efficiency results. On the SGI Origin2000, parallel performance shows a noticeable drop-off in speedup and efficiency measures above 64 processors. The above results for the wall clock time, scaled speedups and efficiency from the different machines are combined in Figures 5.22, 5.23 and 5.24 for comparison purposes. The CRAY T3E has the shortest runtime and shows the best performance. The floating point performance of these machines are also compared in Figure 5.25 which shows 62, 30 and 43 MFLOPS per processor were achieved on the CRAY T3E, IBM SP2 and SGI Origin2000 respectively.

In order to investigate the costs of the different communications involved in our algorithm, timing was also done for important sections of our implementation. This included timings for the total computation and communications in the matrix-vector multiplications, scalar products and restriction operations. The ratio of communication times required for different operations to the total computation times versus number of processors are illustrated in Figures 5.26, 5.27 and 5.28 for the three parallel machines. This cost analysis shows that the total ratio of communication cost to computation cost remained under 3% for all cases. Also, the communication cost associated with the matrix-vector products shows excellent scalability on the CRAY T3E and IBM SP2. Due to the global data transfers, the cost of communications related to the scalar products increases when a large number of processors are used. The restriction operations require minimal communication cost.

5.3.3 Lazy Origin2000 Processors

As mentioned earlier, the SGI Origin2000 showed a noticeable drop-off in parallel performance measures for larger number of processors. A more careful look into the timings for the cost analysis described above revealed an interesting phenomenon for this parallel machine. Timings were conducted in dedicated mode and barriers were used at the necessary locations to ensure that the computations, communications and idle times were accurately measured. The computation time on all of the processors must be equal for a scaled-size test

problem where the processors have exactly the same amount of computational work. The timings on the CRAY T3E and the IBM SP2 followed this simple rule but the story was different for the SGI Origin2000. Figure 5.29 shows the total computation time for the matrix-vector multiplications on each processor for a scaled-size test problem executed on 128 processors. A few of the processors spend more time than the others to perform the same amount of computational work. This results in an increase in the total runtime because the rest of the processors have to wait for these *lazy* processors at any synchronization point. All of the processors available in the machine were used for this test. To try the case where more resources are available for the system, the same test was also performed on 124 processors of this machine and the same result was observed (Figure 5.30). Running this test several times showed that different processors would become *lazy* for different runs.

The SGI Origin2000 and other machines with a similar architecture are become more available to users. Experts predict this trend will continue into the future. This justified more investigation into this matter. The SGI Origin2000 features the Cache Coherent NonUniform Memory Access (CC-NUMA) memory subsystem, which provides the user with a simple shared memory programming model. However, the actual memory is physically distributed that could have a significant impact on the performance of applications on the Origin2000 system. A description of the CC-NUMA memory subsystem is given in [32]. We used the NUMA library [31] to investigate the perceived performance problems associated with process and memory placement on an SGI Origin2000. This library contains some Fortran callable routines, which provide a fairly flexible set of features for the analysis and management of memory and process placement on the Origin2000 system.

We wanted to ensure that there was no mismatch between the memory and the process location; that is, all the processes (running on their CPUs) should allocate all of the memory they need on the same nodes that contain their CPUs (the Origin2000 that we had access to had two CPUs on each node). In other words, any memory allocated for a process comes from the same node containing the CPU that runs the process. This was accomplished by calling the place_process( ) routine in the NUMA library. This routine has two separate actions. First, the process is linked with the specified node such that any memory allocated for the process comes from that node. Second, the process is associated with one of the CPUs

on the node so that as the process runs (on that CPU) it will be accessing memory that is local to that node.

Figure 5.31 shows similar timing data as Figure 5.29 after the process placement. There are some improvements in the maximum timings of the *lazy* processors (from 78 seconds to less than 74 seconds) but the average time is still at the same level as before. The total runtime after process placement was 79.3 seconds that is not a significant improvement (from 80.80 seconds without process placement). There is another possible explanation for this phenomena in which system processes apparently steal processor cycles from computing processes, which gives the appearance of slowing down one or two of them.

Number of Proces-	Number of Ele-	Number of
1	8,192	9,792
8	65,536	70,720
16	131,072	140,352
32	262,144	279,616
64	524,288	558,144
128	1,048,576	1,115,200
256	2,097,152	2,229,312
512	4,194,304	4,457,536

Table 5.1: Number of Elements and Nodes for the Generated Scaled-Size Test Problems.

Number of	Elapsed	Scaled	Efficiency
1	52.46	1.00	1.00
8	52.93	7.93	0.99
16	52.96	15.85	0.99
32	53.31	31.49	0.98
64	53.45	62.81	0.98
128	53.62	125.23	0.98
256	53.68	250.14	0.98
512	54.21	495.47	0.97

Table 5.2: Scalability and Parallel Performance Results for a Full Multigrid Solve on the CRAY T3E.

Number of	Elapsed	Scaled	Efficiency
1	109.31	1.00	1.00
8	112.24	7.79	0.97
16	112.93	15.49	0.97
32	113.40	30.85	0.96
64	117.52	59.53	0.93

Table 5.3: Scalability and Parallel Performance Results for a Full Multigrid Solve on the IBM SP2.

Number of	Elapsed	Scaled	Efficiency
1	62.94	1.00	1.00
8	63.91	7.88	0.99
16	63.99	15.74	0.98
32	65.24	30.87	0.96
64	66.62	60.46	0.94
128	80.80	99.71	0.78

Table 5.4: Scalability and Parallel Performance Results for a Full Multigrid Solve on the SGI Origin2000.

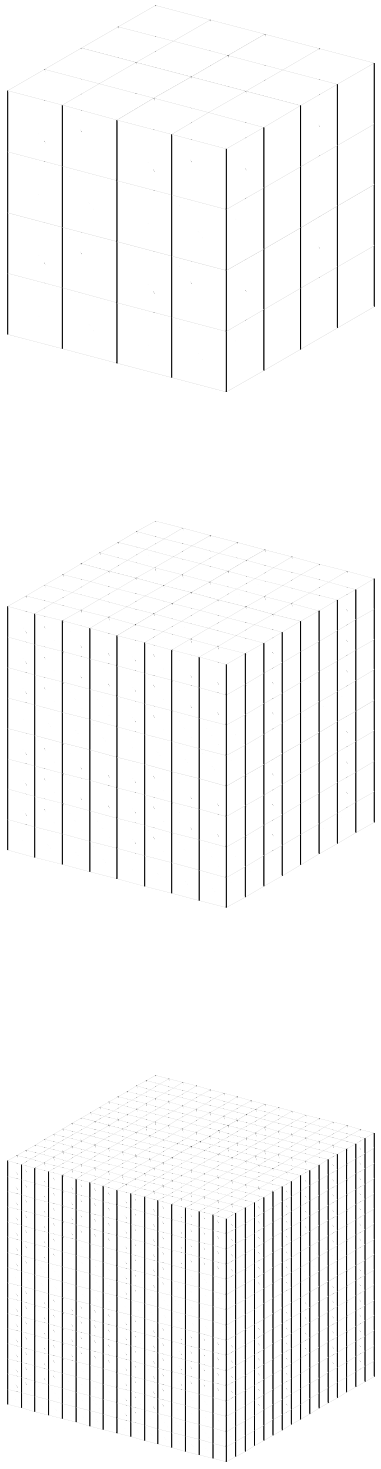


Figure 5.1 : The First Three Meshes of Increasing Refinement for the Cube.

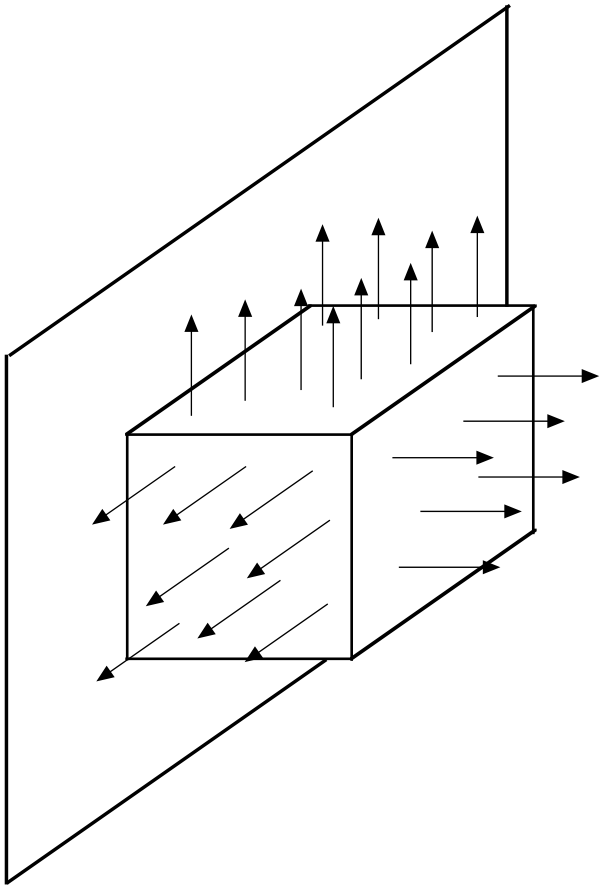


Figure 5.2: The Boundary Conditions for the Cube Benchmark Problem.

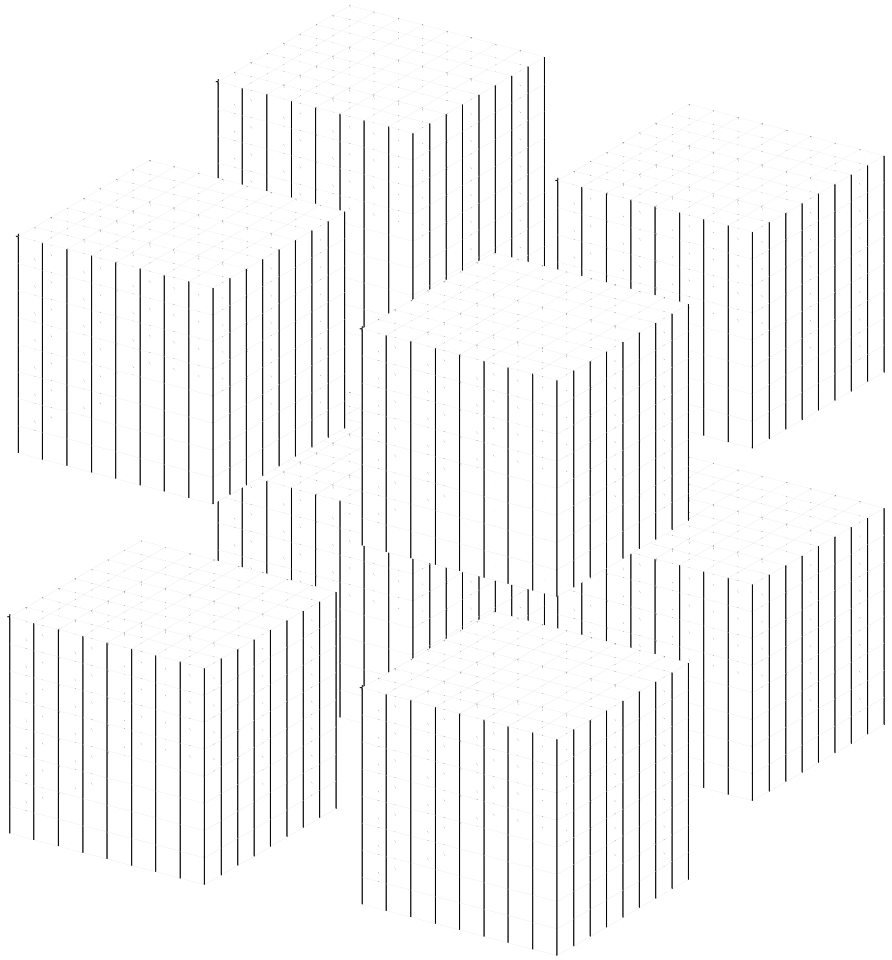


Figure 5.3: Decomposition of the Fixed-Sized Problem with 32,768 Elements into 8 Partitions at the Third Level.

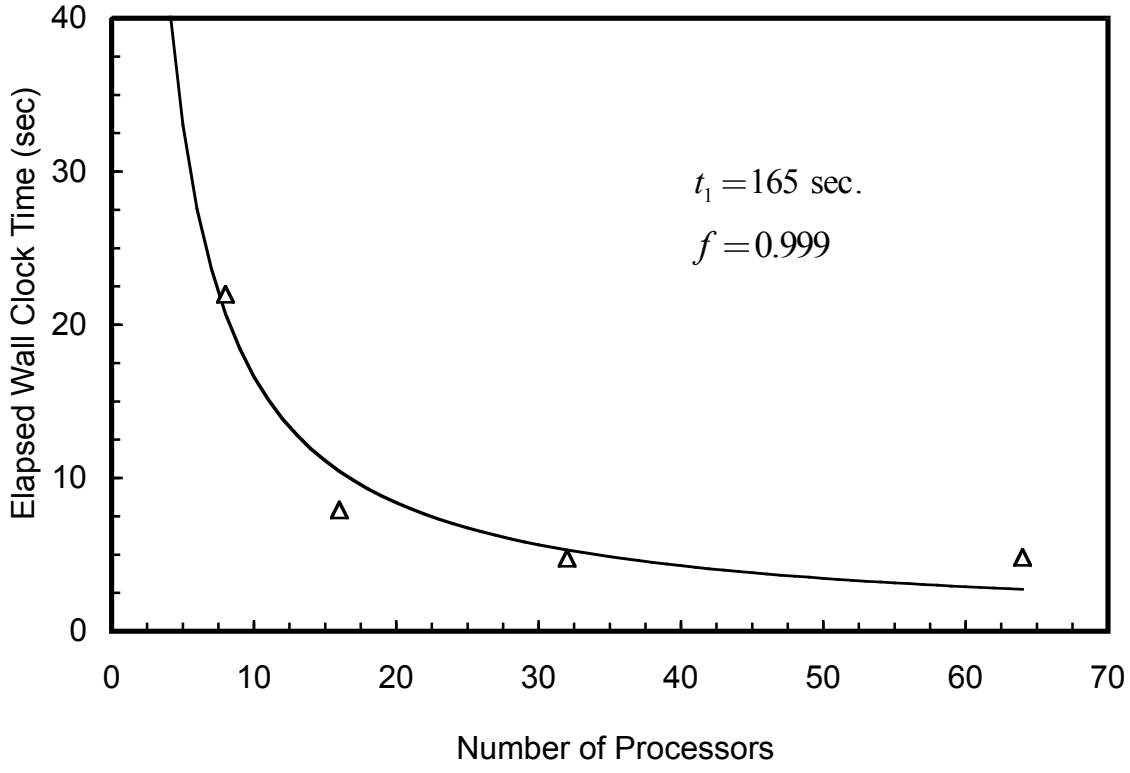


Figure 5.4: Total Elapsed Wall Clock Time Versus Number of Processors for the Fixed-Size Problem with 32,768 Elements on the SGI Origin2000. Time is Given for a Single Full Multigrid Solve. Time on One processor and the Fraction of Parallelism are also Reported.

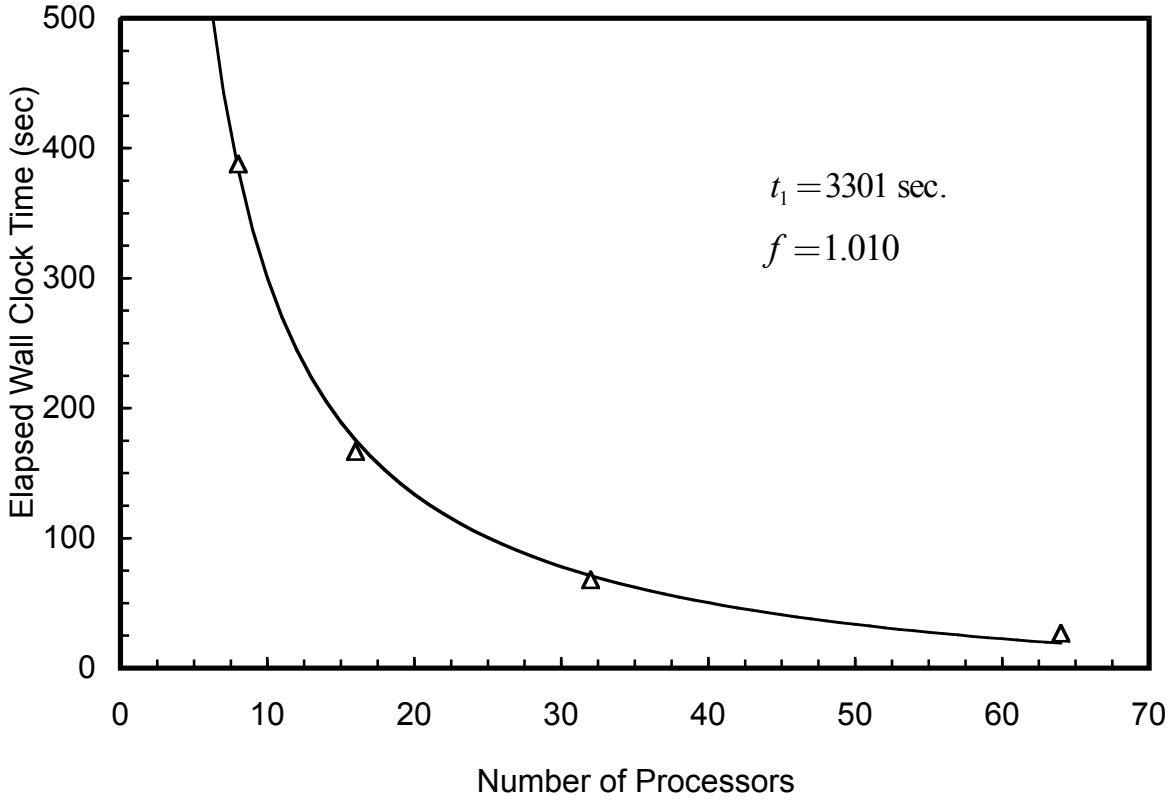


Figure 5.5: Total Elapsed Wall Clock Time Versus Number of Processors for the Fixed-Size Problem with 262,144 Elements on the SGI Origin2000. Time is Given for a Single Full Multigrid Solve. Time on One processor and the Fraction of Parallelism are also Reported.

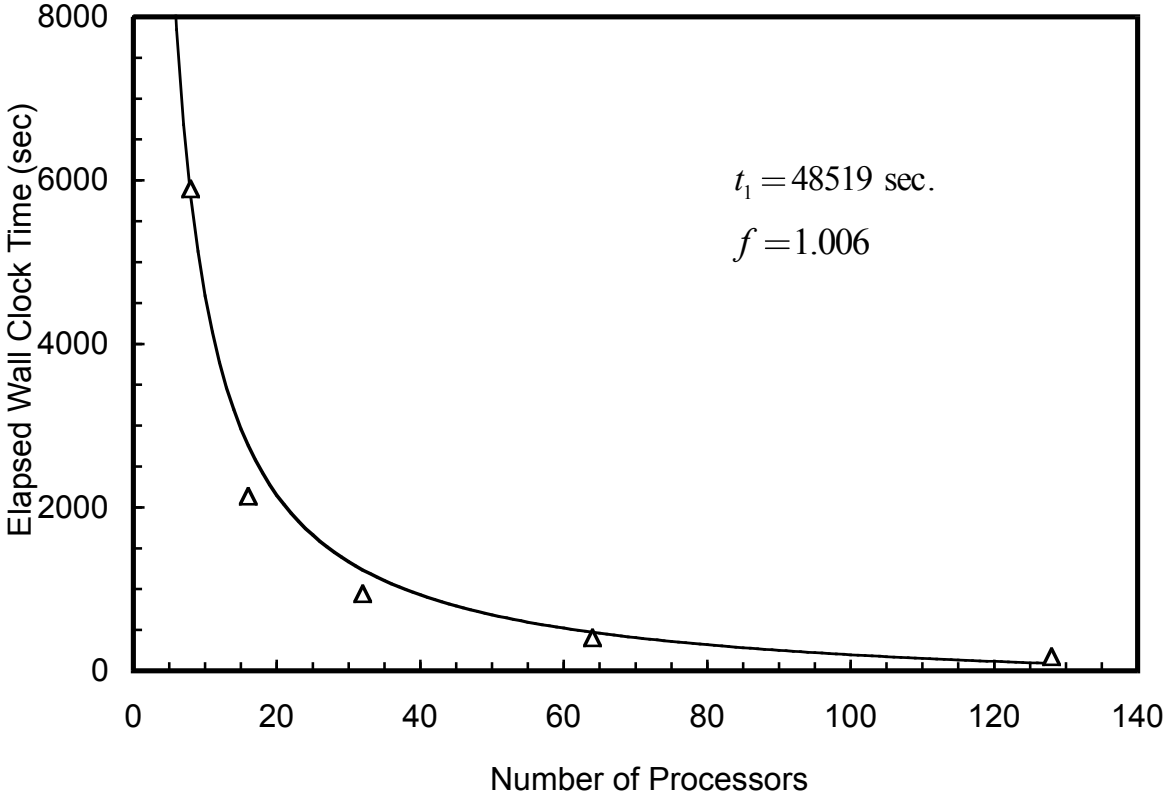


Figure 5.6: Total Elapsed Wall Clock Time Versus Number of Processors for the Fixed-Size Problem with 2,097,152 Elements on the SGI Origin2000. Time is Given for a Single Full Multigrid Solve. Time on One processor and the Fraction of Parallelism are also Reported.

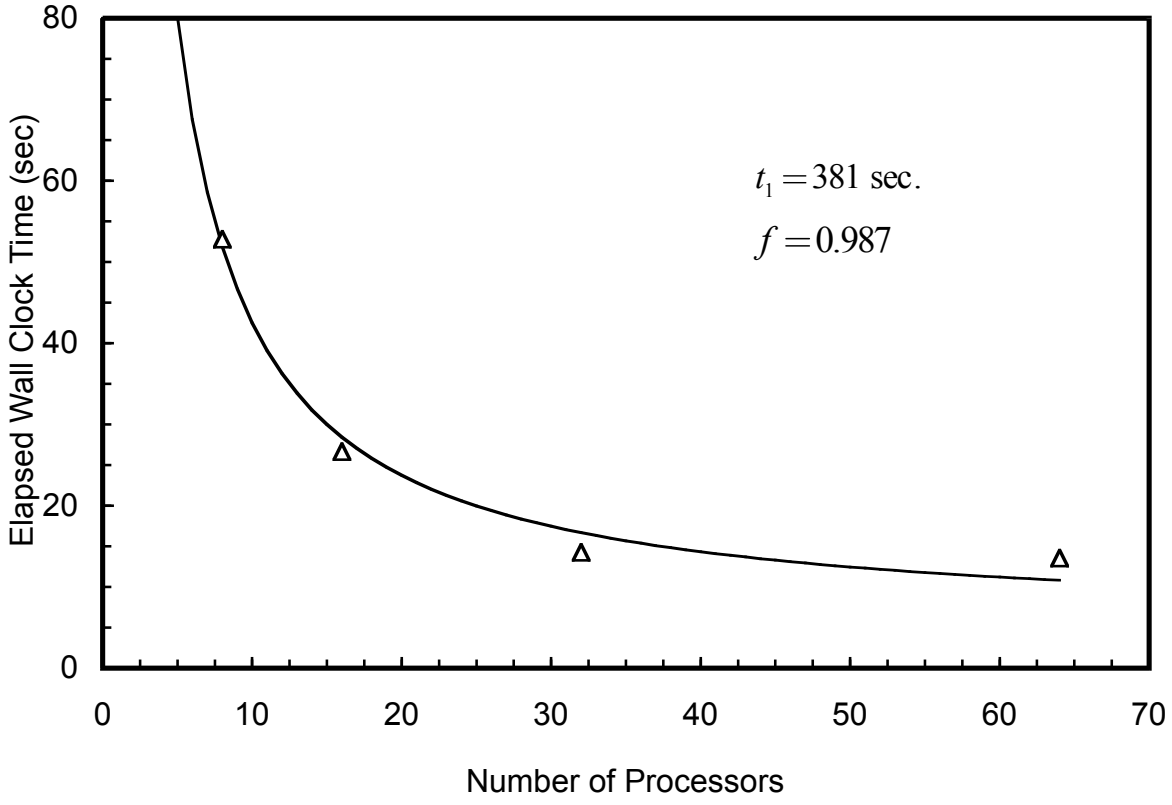


Figure 5.7: Total Elapsed Wall Clock Time Versus Number of Processors for the Fixed-Size Problem with 32,768 Elements on the IBM SP2. Time is Given for a Single Full Multigrid Solve. Time on One processor and the Fraction of Parallelism are also Reported.

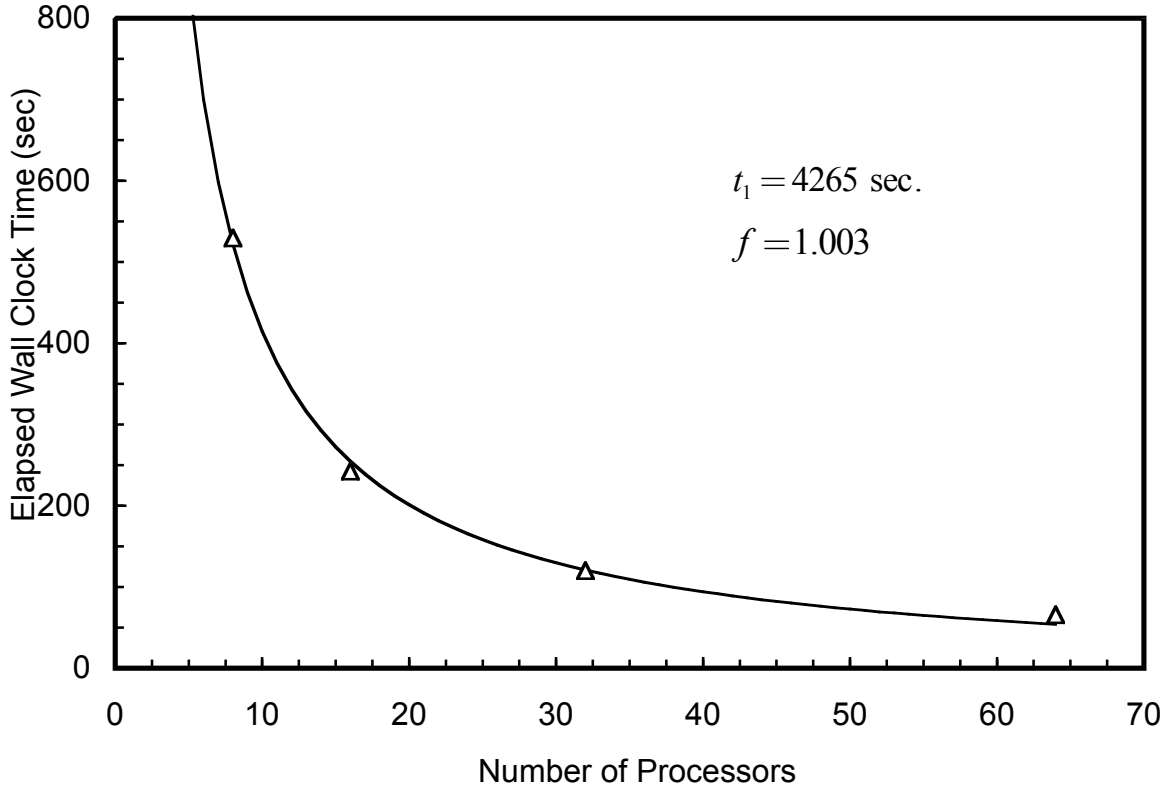


Figure 5.8: Total Elapsed Wall Clock Time Versus Number of Processors for the Fixed-Size Problem with 262,144 Elements on the IBM SP2. Time is Given for a Single Full Multigrid Solve. Time on One processor and the Fraction of Parallelism are also Reported.

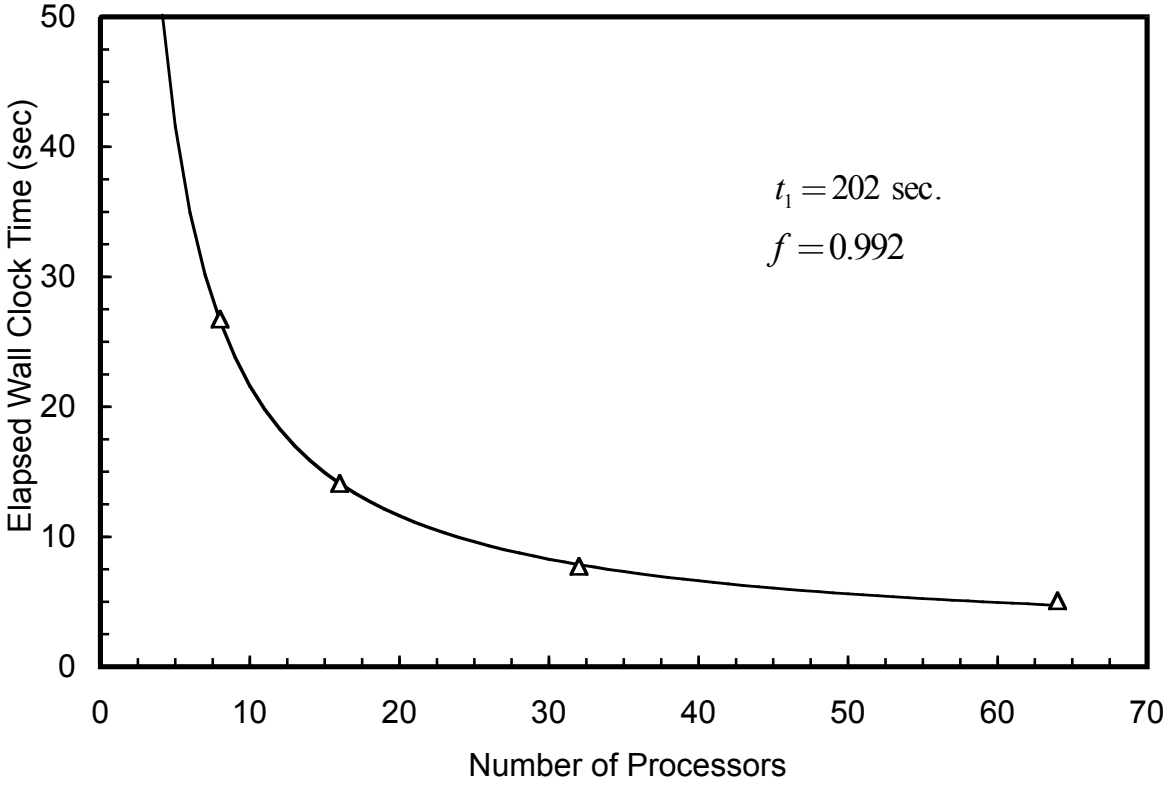


Figure 5.9: Total Elapsed Wall Clock Time Versus Number of Processors for the Fixed-Size Problem with 32,768 Elements on the CRAY T3E. Time is Given for a Single Full Multigrid Solve. Time on One processor and the Fraction of Parallelism are also Reported.

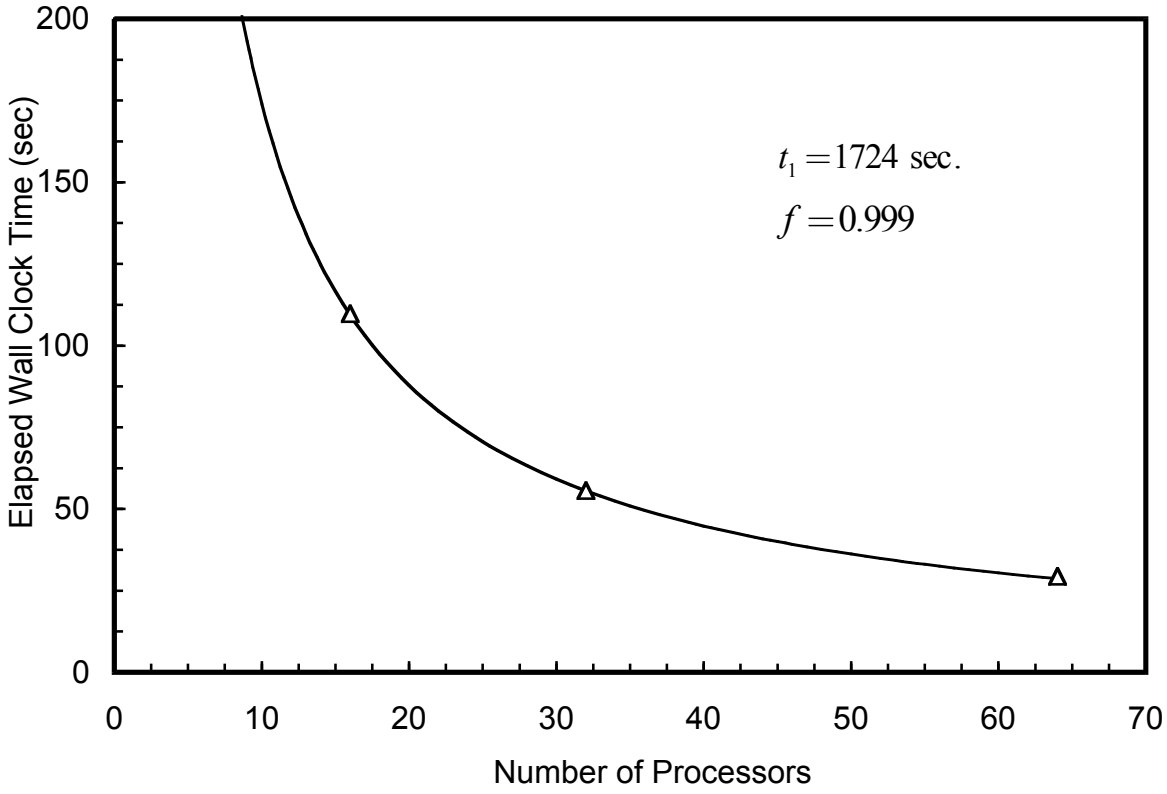


Figure 5.10: Total Elapsed Wall Clock Time Versus Number of Processors for the Fixed-Size Problem with 262,144 Elements on the CRAY T3E. Time is Given for a Single Full Multigrid Solve. Time on One processor and the Fraction of Parallelism are also Reported.

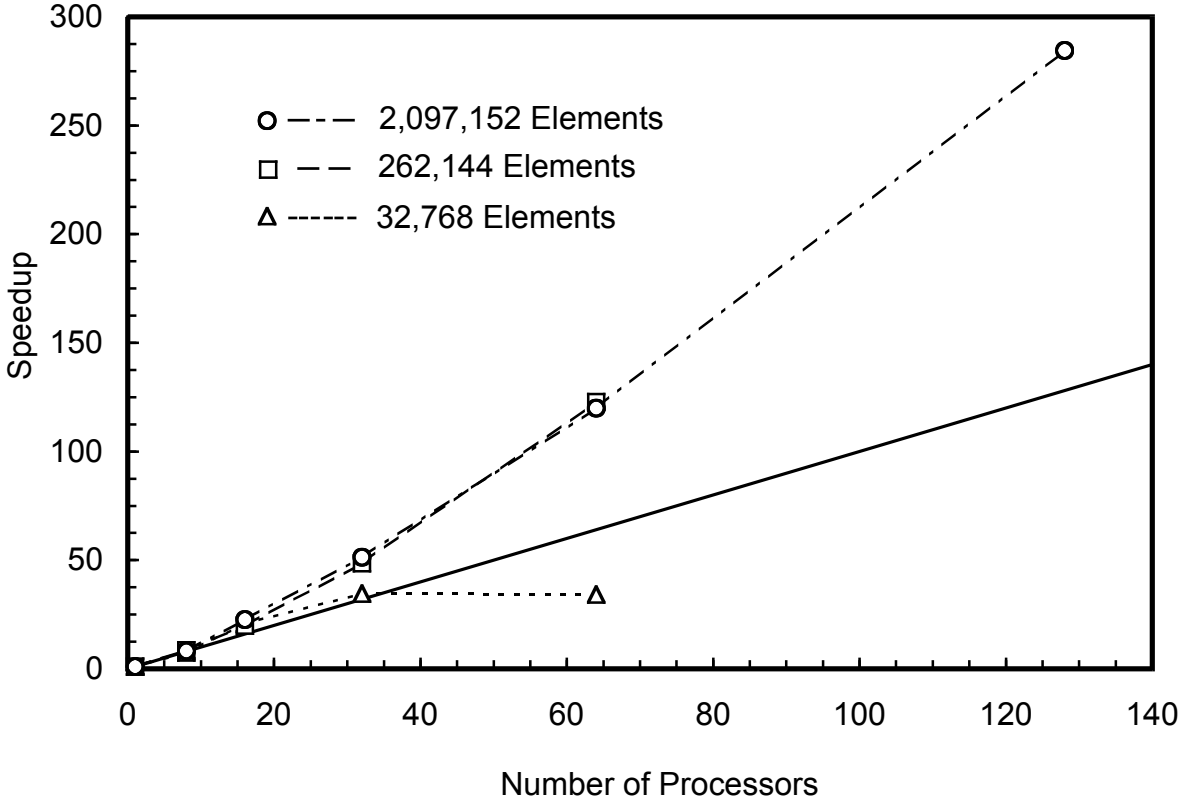


Figure 5.11: Speedups for the Different Fixed-Size Problems
on the SGI Origin2000.

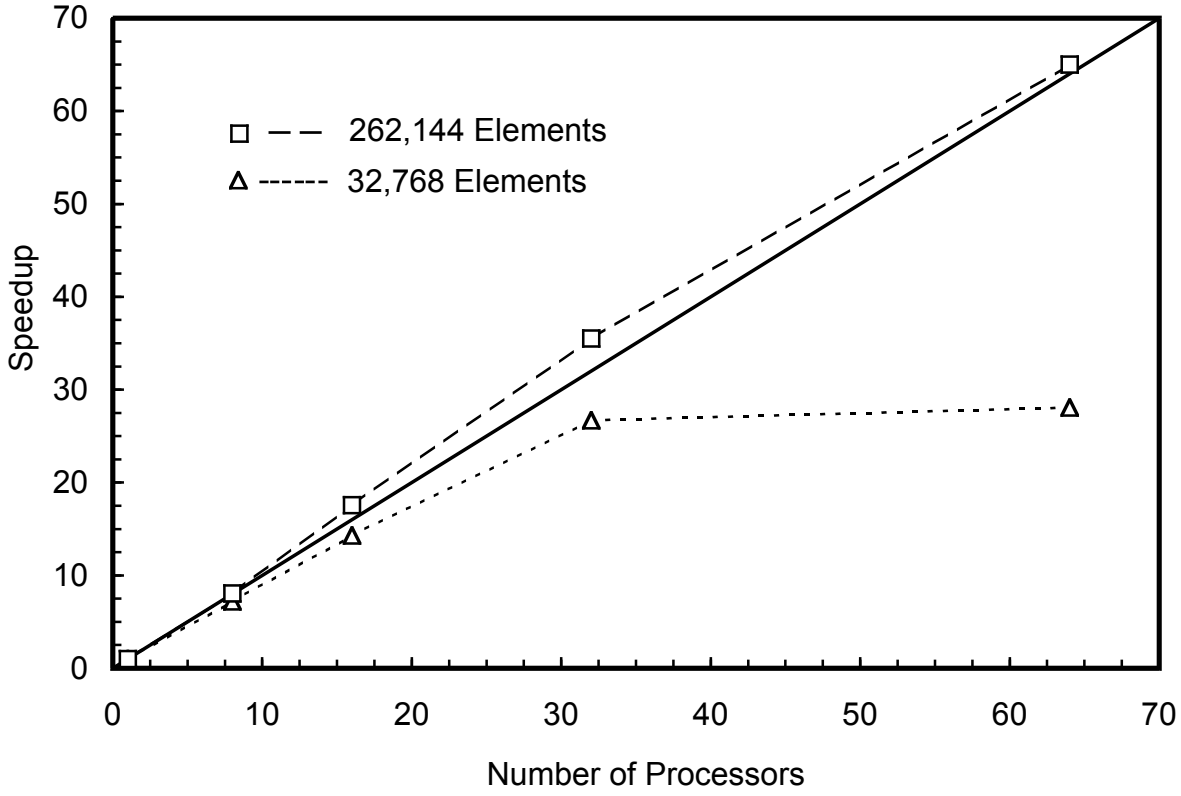


Figure 5.12: Speedups for the Different Fixed-Size Problems
on the IBM SP2.

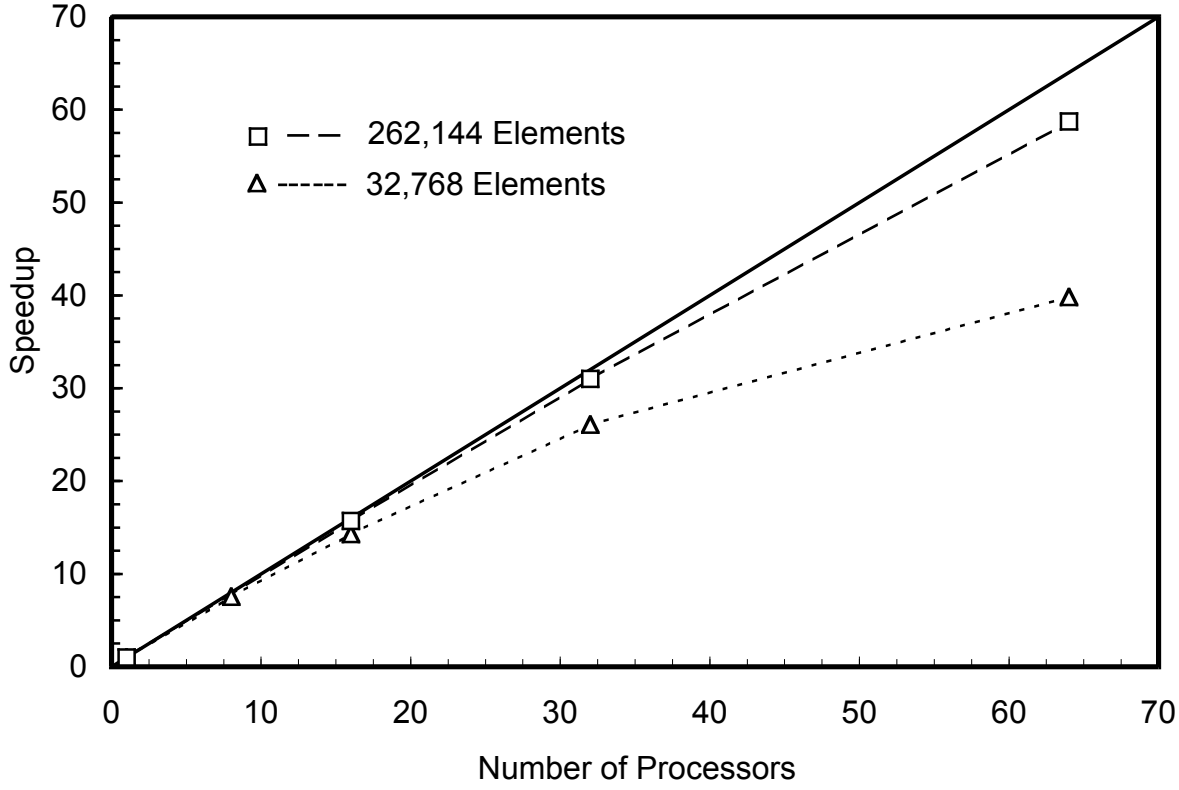


Figure 5.13: Speedups for the Different Fixed-Size Problems
on the CRAY T3E.

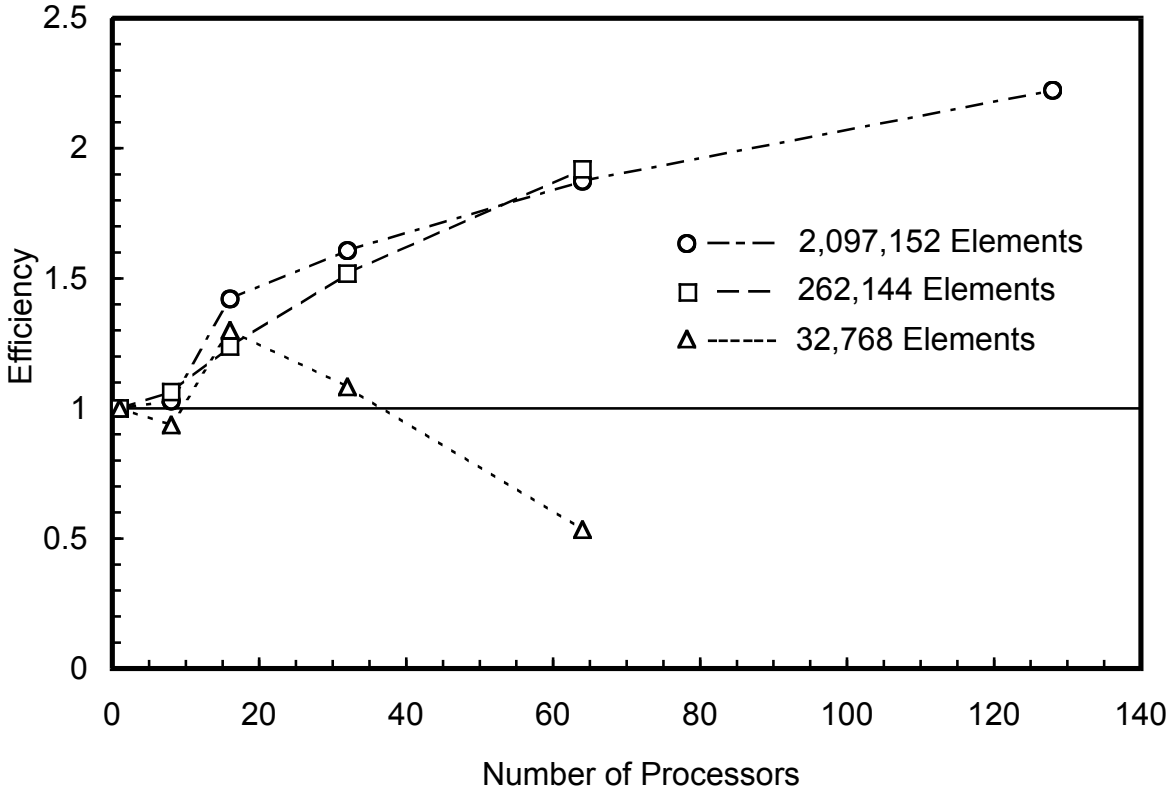


Figure 5.14: Efficiency for the Different Fixed-Size Problems
on the SGI Origin2000.

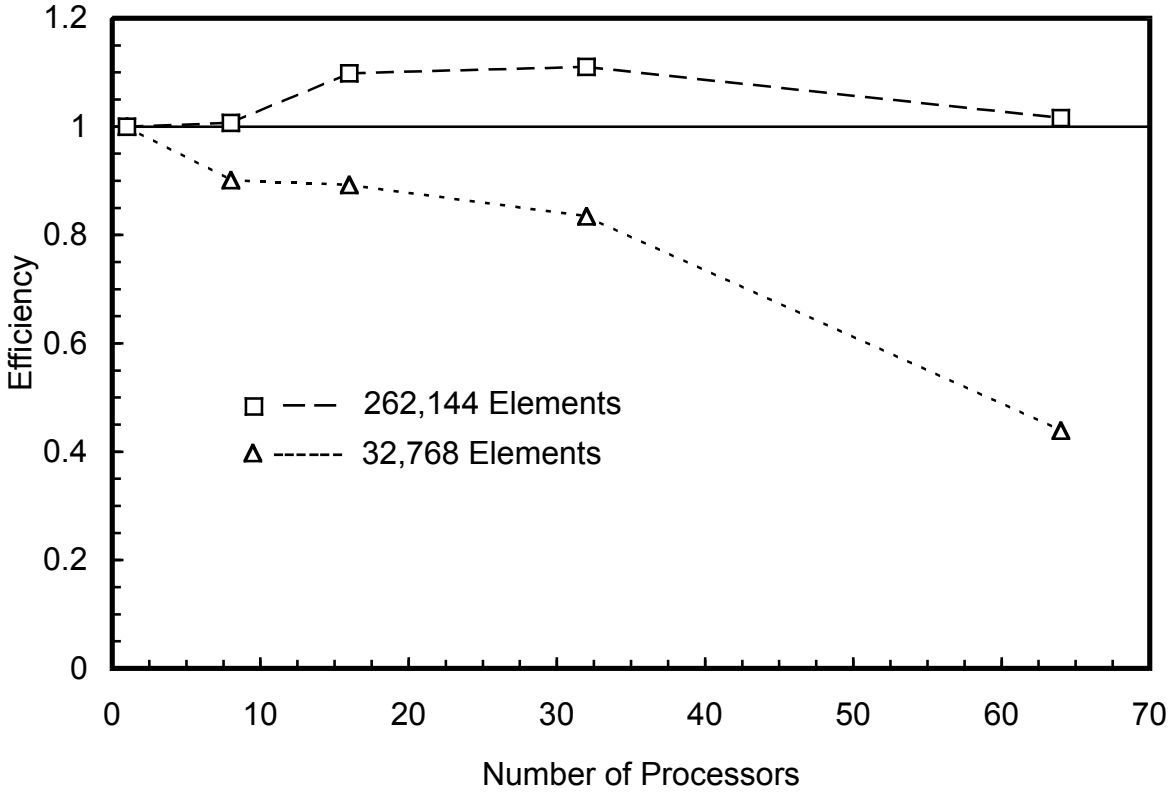


Figure 5.15: Efficiency for the Different Fixed-Size Problems
on the IBM SP2.

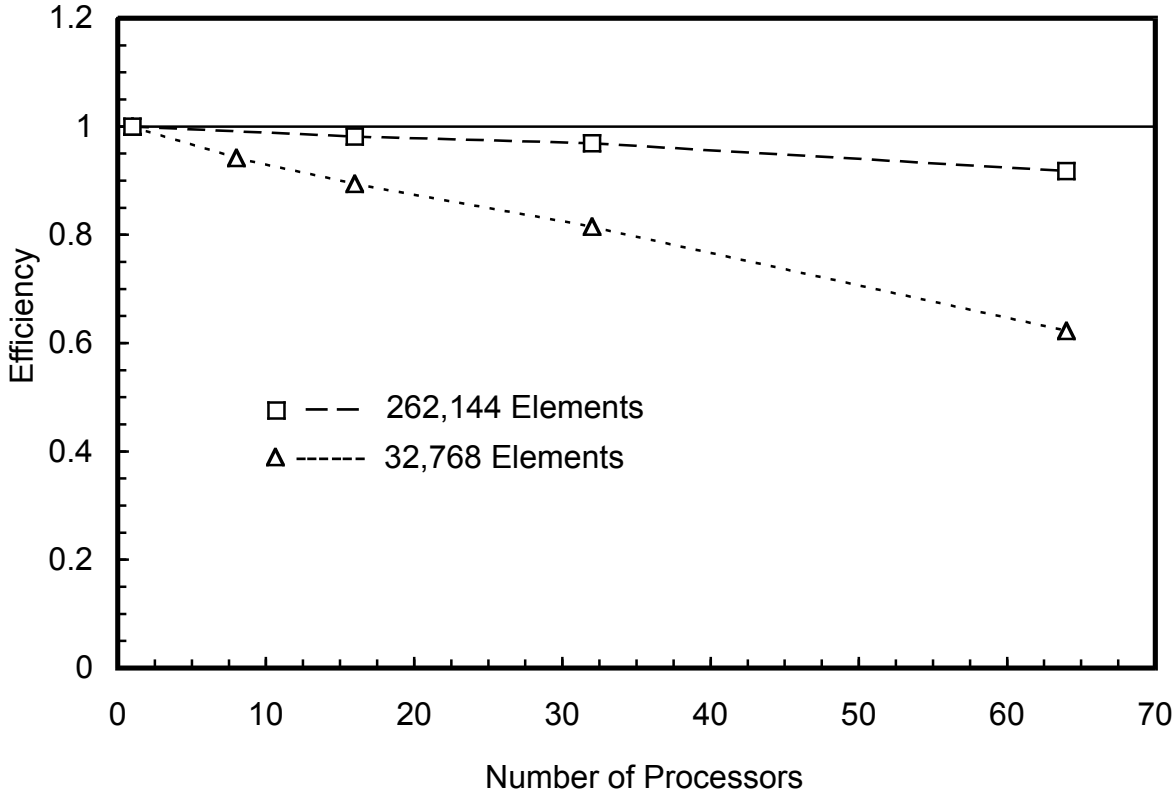


Figure 5.16: Efficiency for the Different Fixed-Size Problems
on the CRAY T3E.

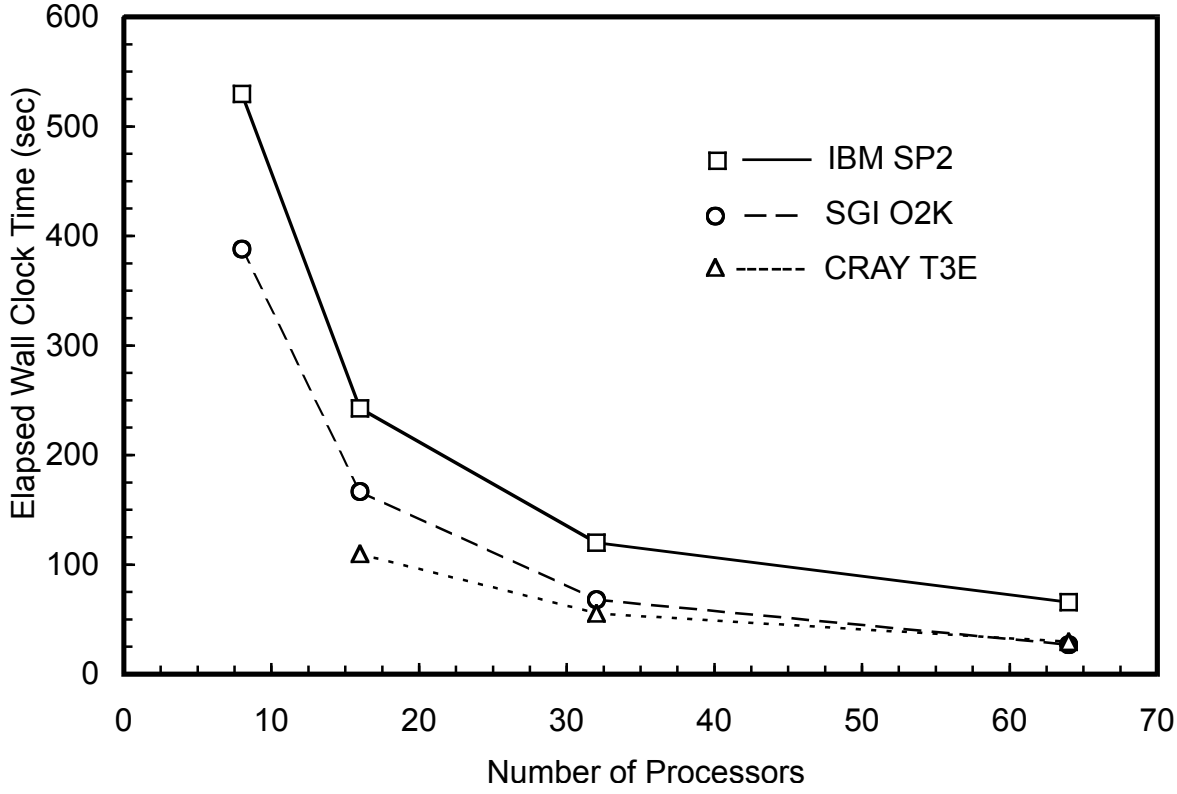


Figure 5.17: Comparison of the Total Elapsed Wall Clock Time on the 3 Different Parallel Machines Versus Number of Processors for the Fixed-Size Problem with 262,144 Elements.

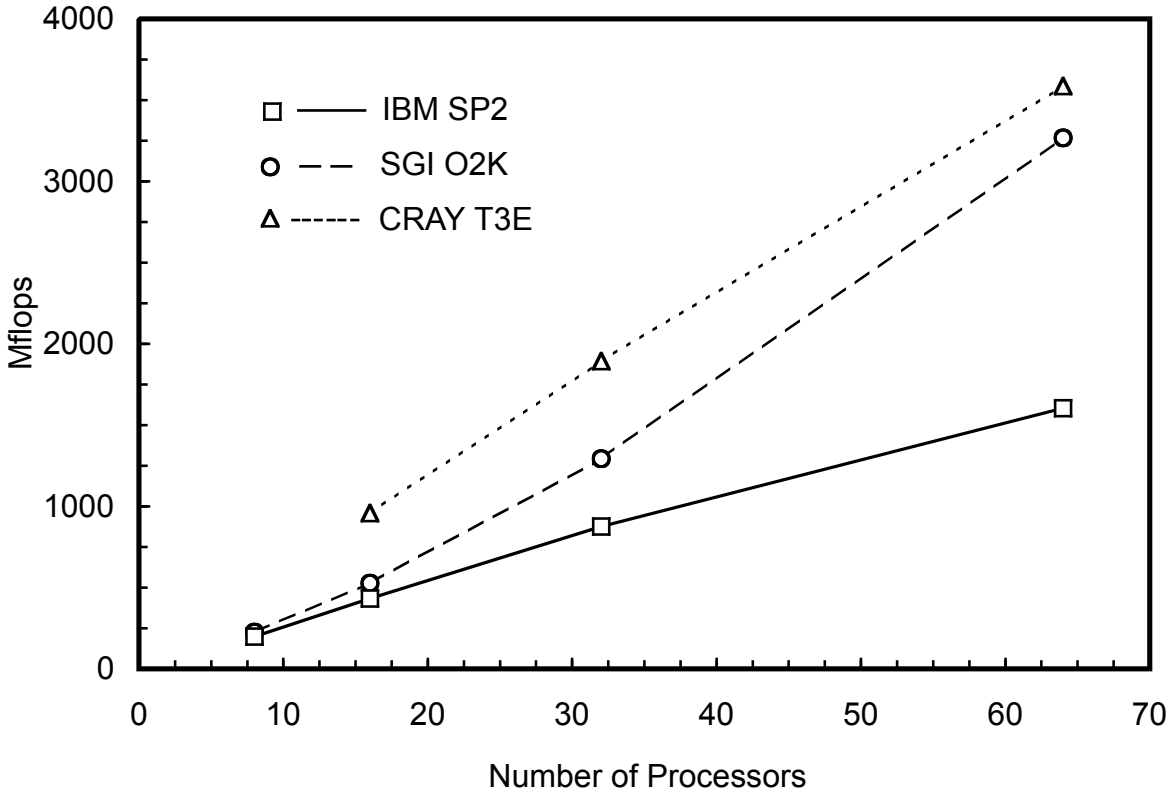


Figure 5.18: Comparison of the Floating Point Performance on the 3 Different Parallel Machines Versus Number of Processors for the Fixed-Size Problem with 262,144 Elements.

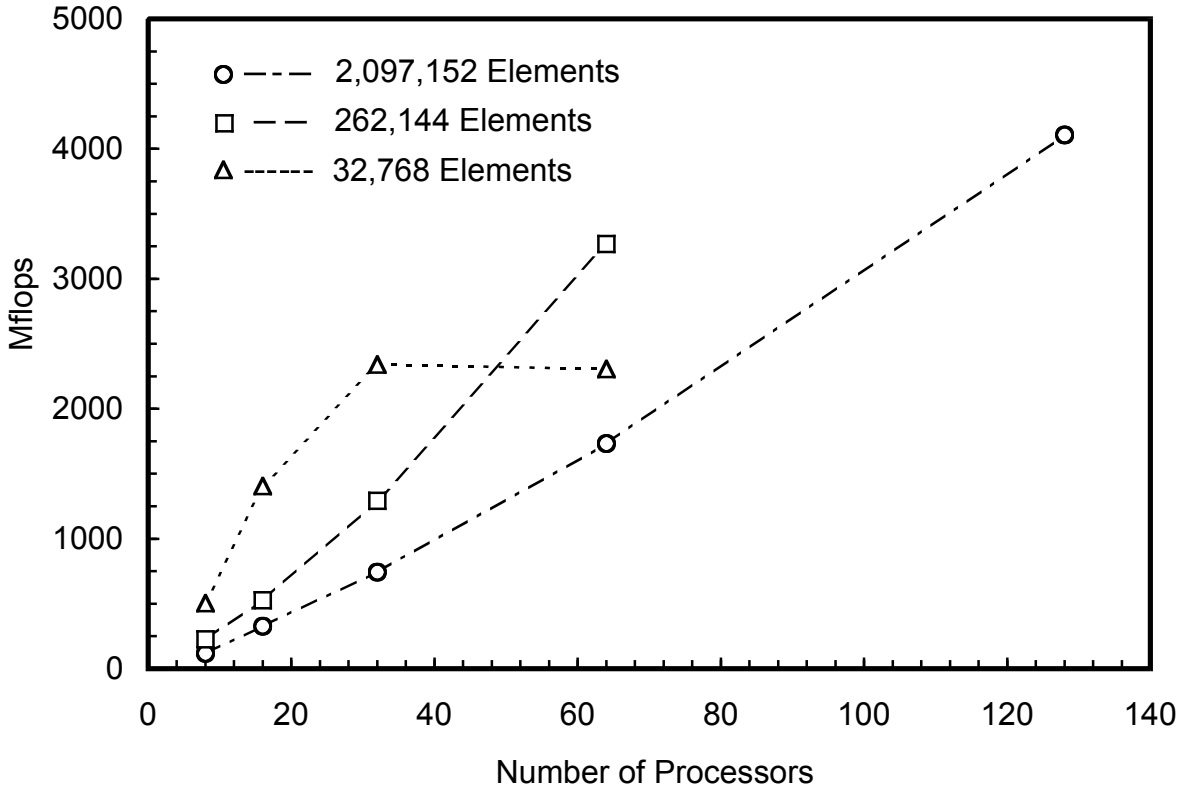


Figure 5.19: Floating Point Performance of the SGI Origin2000 Versus Number of Processors for Different Fixed-Size Problems.

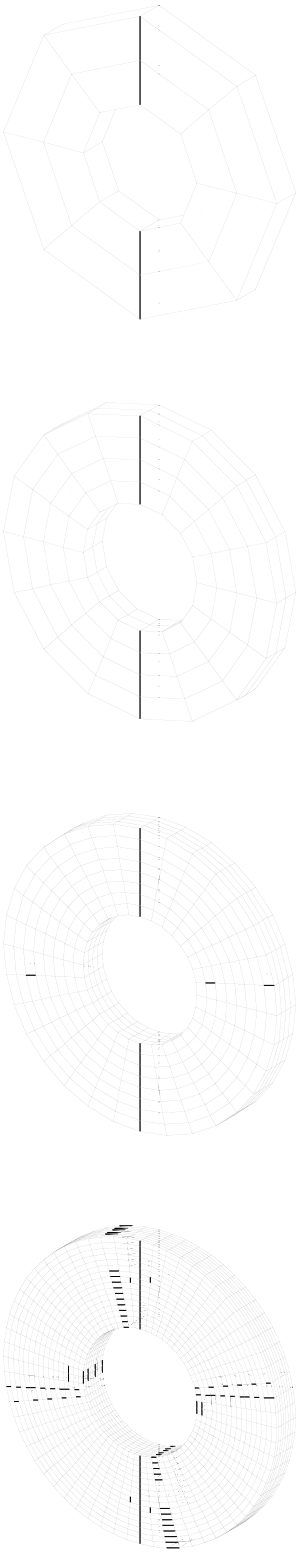


Figure 5.20: Four Meshes of Increasing Refinement Used for the Scaled-Size Benchmark Problems.

Figure 5.21: Decomposition of the Scaled-Size Problem Generated for 8 Processors.

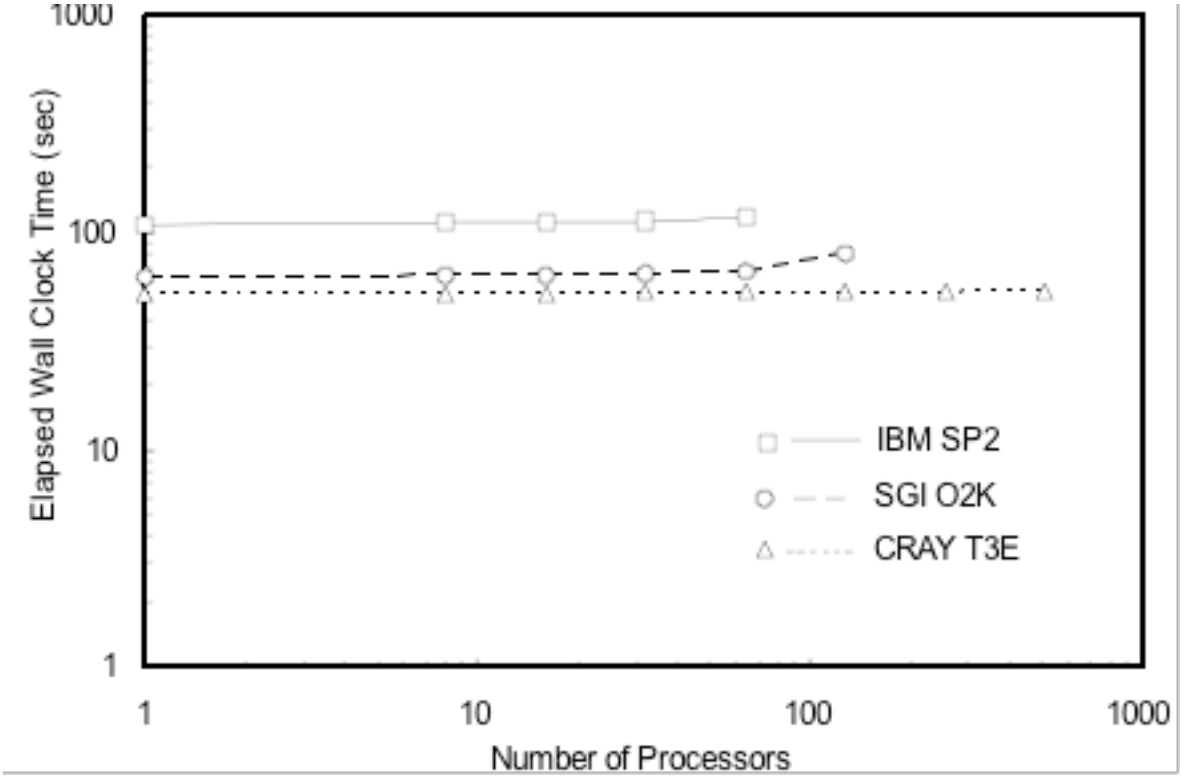


Figure 5.22: Total Elapsed Wall Clock Time Versus Number of Processors for the Scaled-Size Test Problems on Three Different Parallel Machines. Time is Given for a Single Full Multigrid Solve.

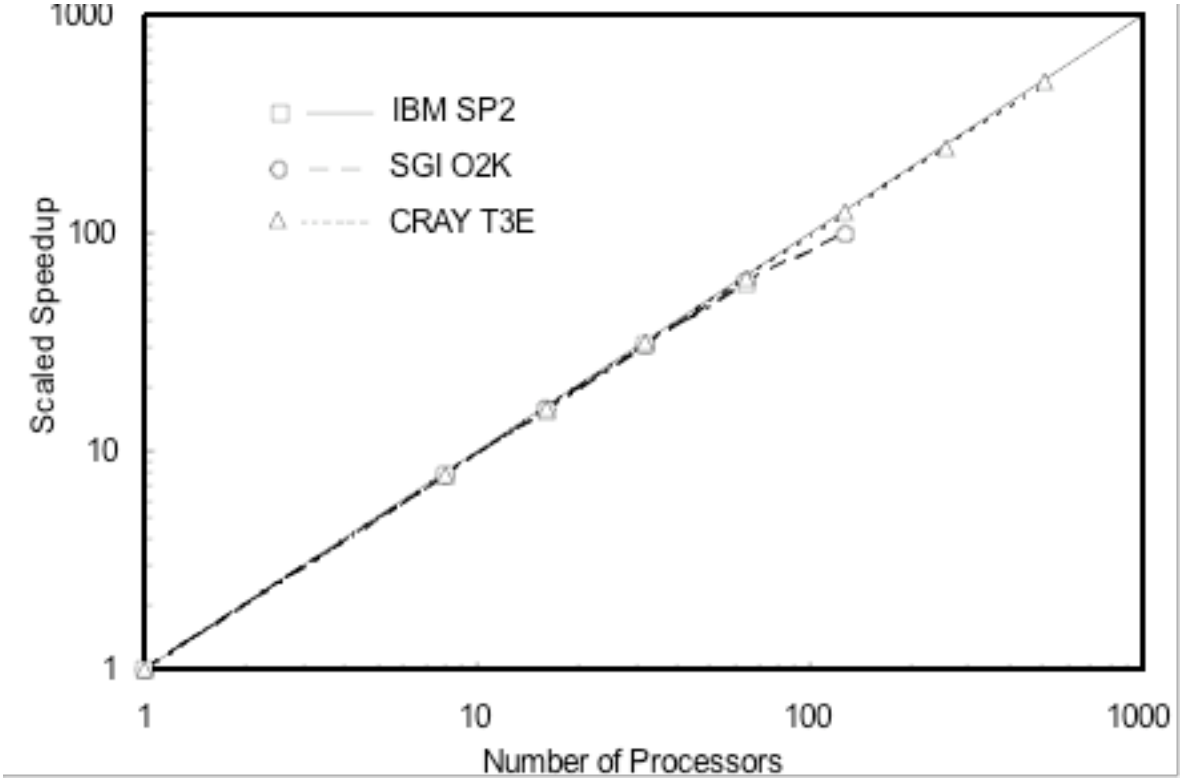


Figure 5.23: Comparison of Scaled Speedups for Three Different Parallel Machines.

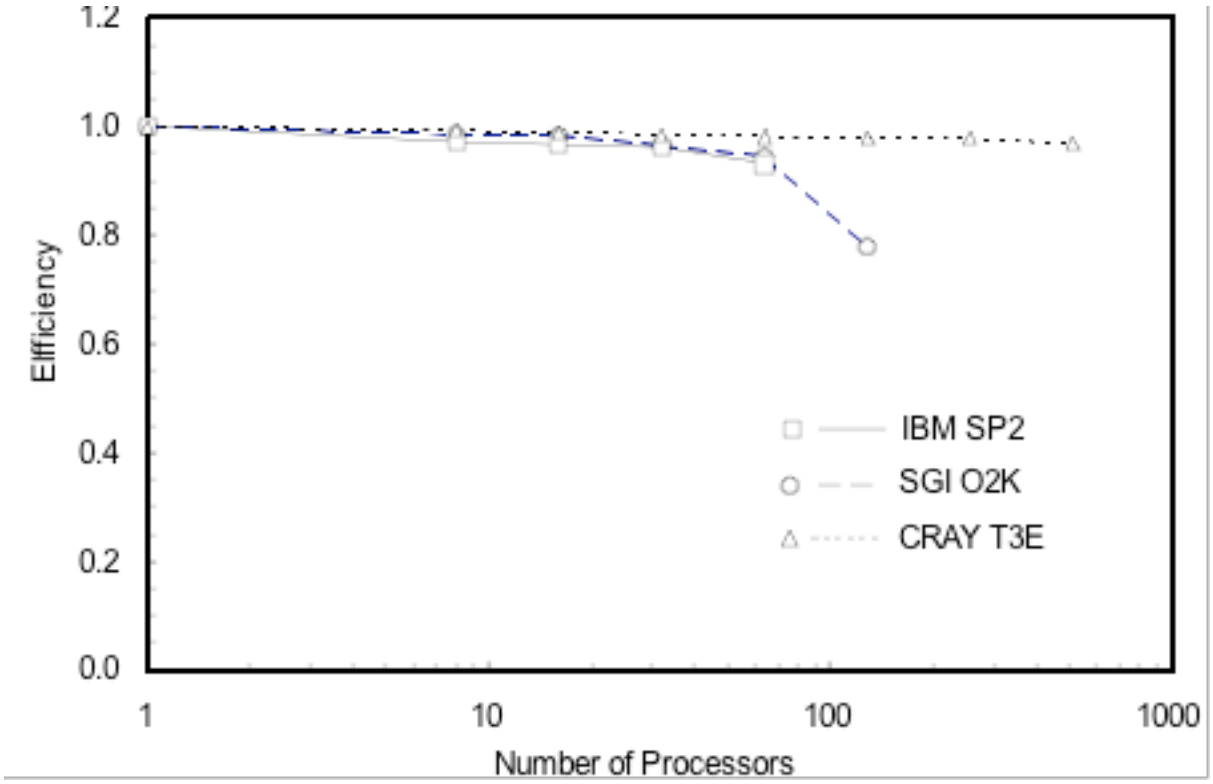


Figure 5.24: Comparison of Efficiency for Three Different Parallel Machines to Solve the Scaled-Size Test Problems.

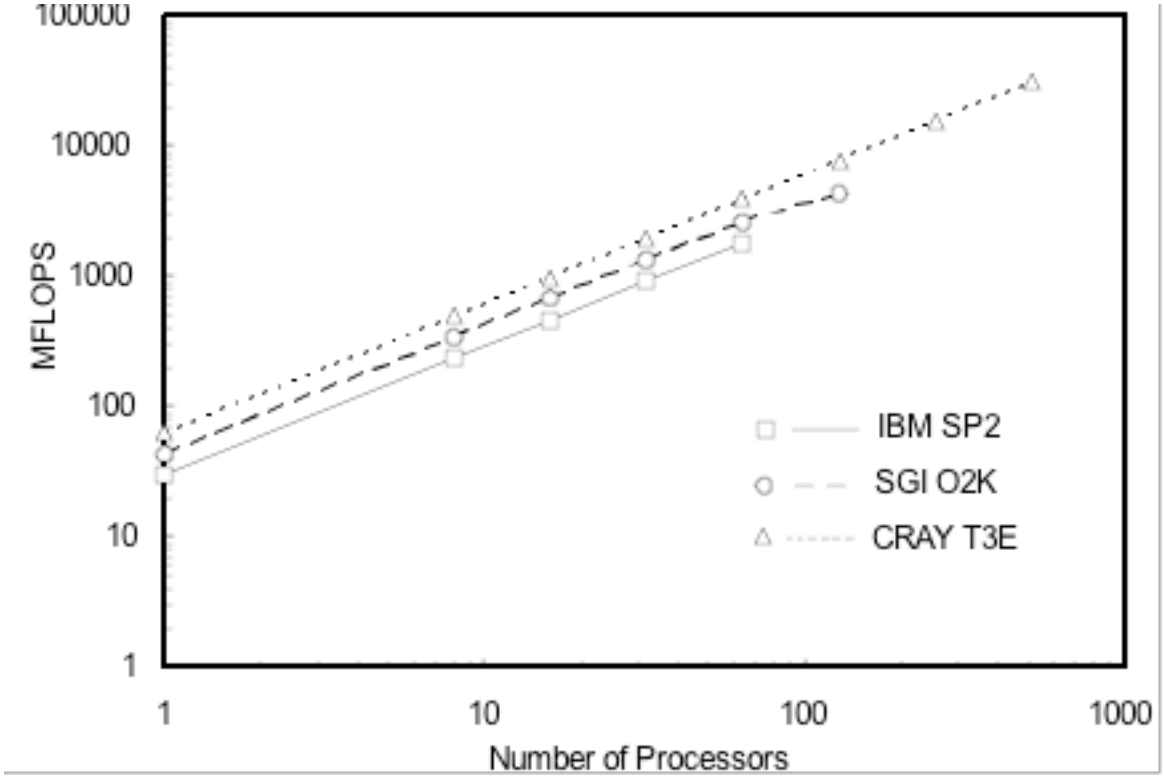


Figure 5.25: Comparison of the Floating Point Performance for Three Different Parallel Machines to Solve the Scaled-Size Test Problems.

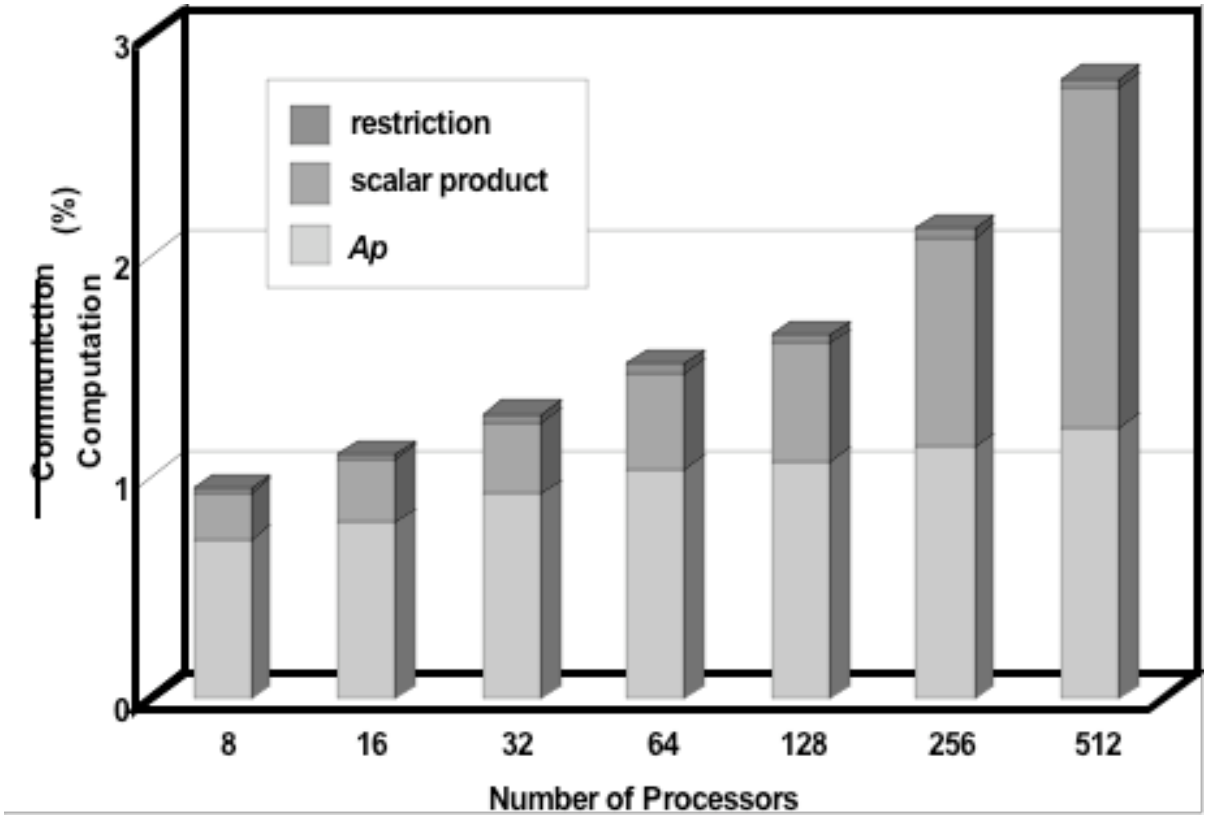


Figure 5.26: Cost Analysis for the CRAY T3E.

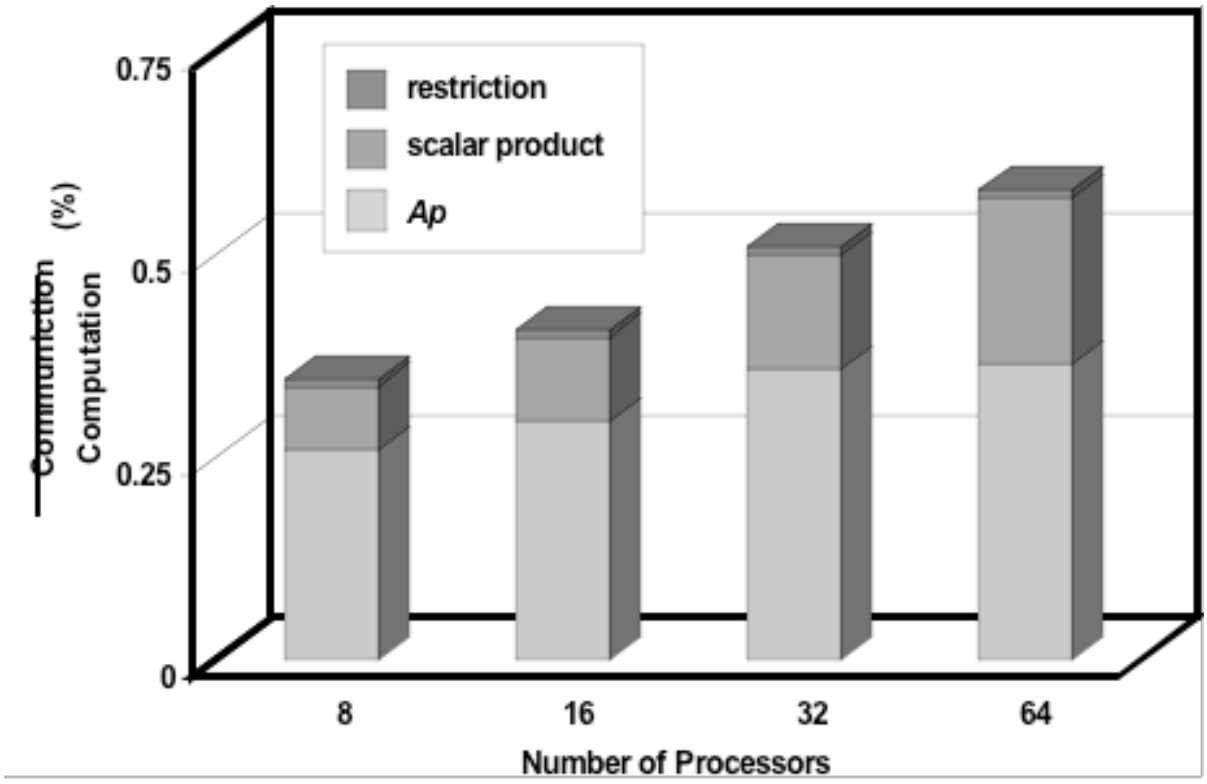


Figure 5.27: Cost Analysis for the IBM SP2.

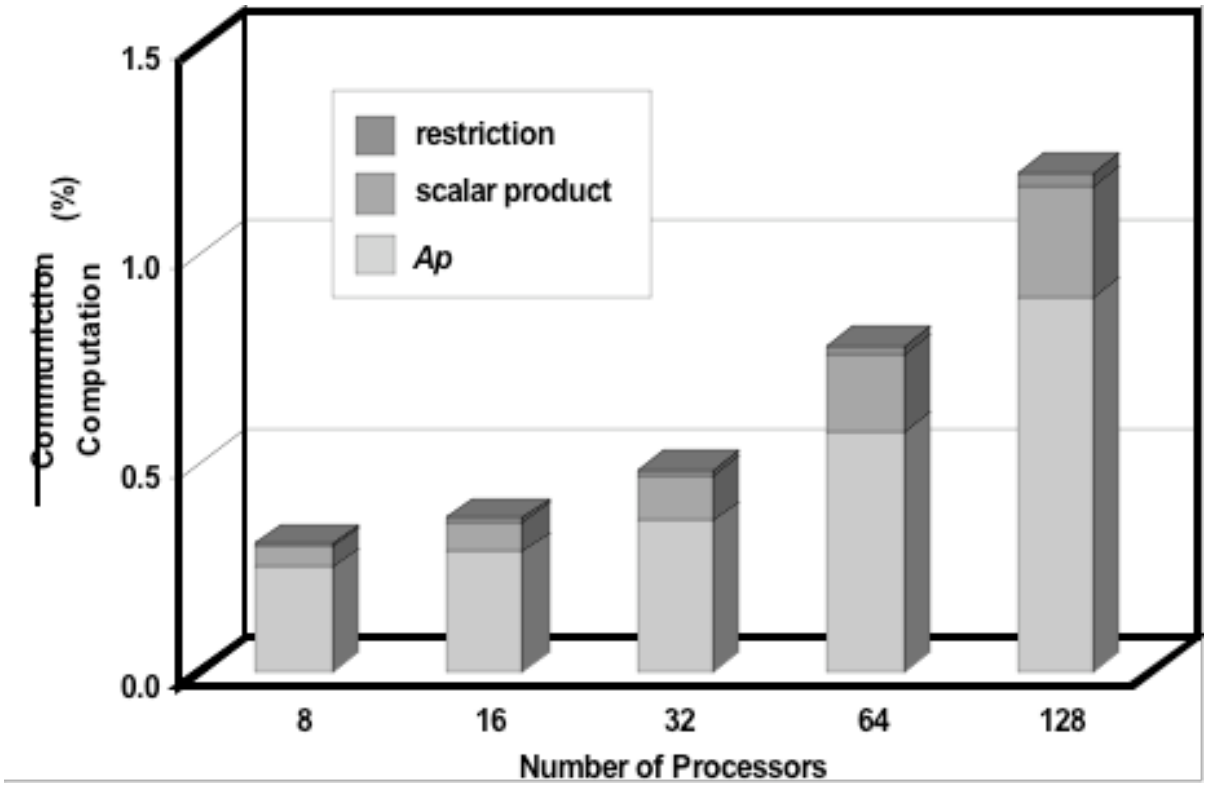


Figure 5.28: Cost Analysis for the SGI Origin2000.

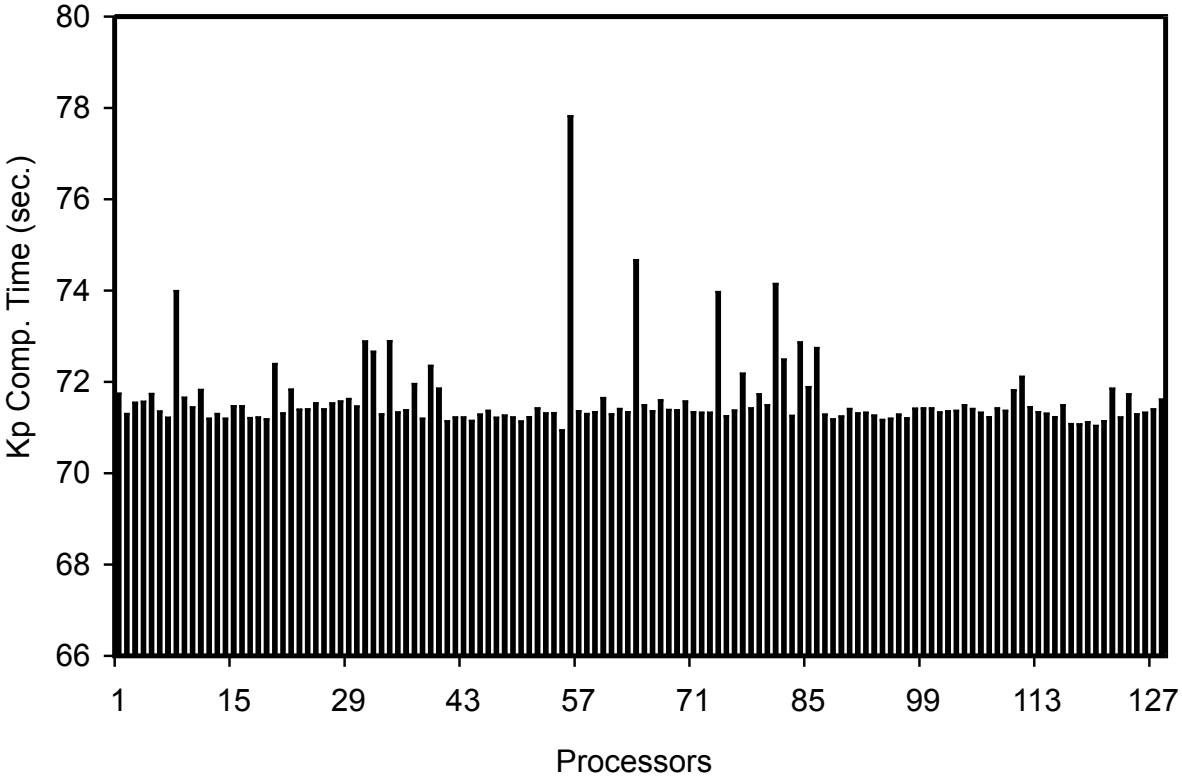


Figure 5.29: The *Lazy Processors* Phenomena for the SGI Origin2000 in a Scaled-Size Test Problem (8192 Elements per Processor) on 128 Processors.

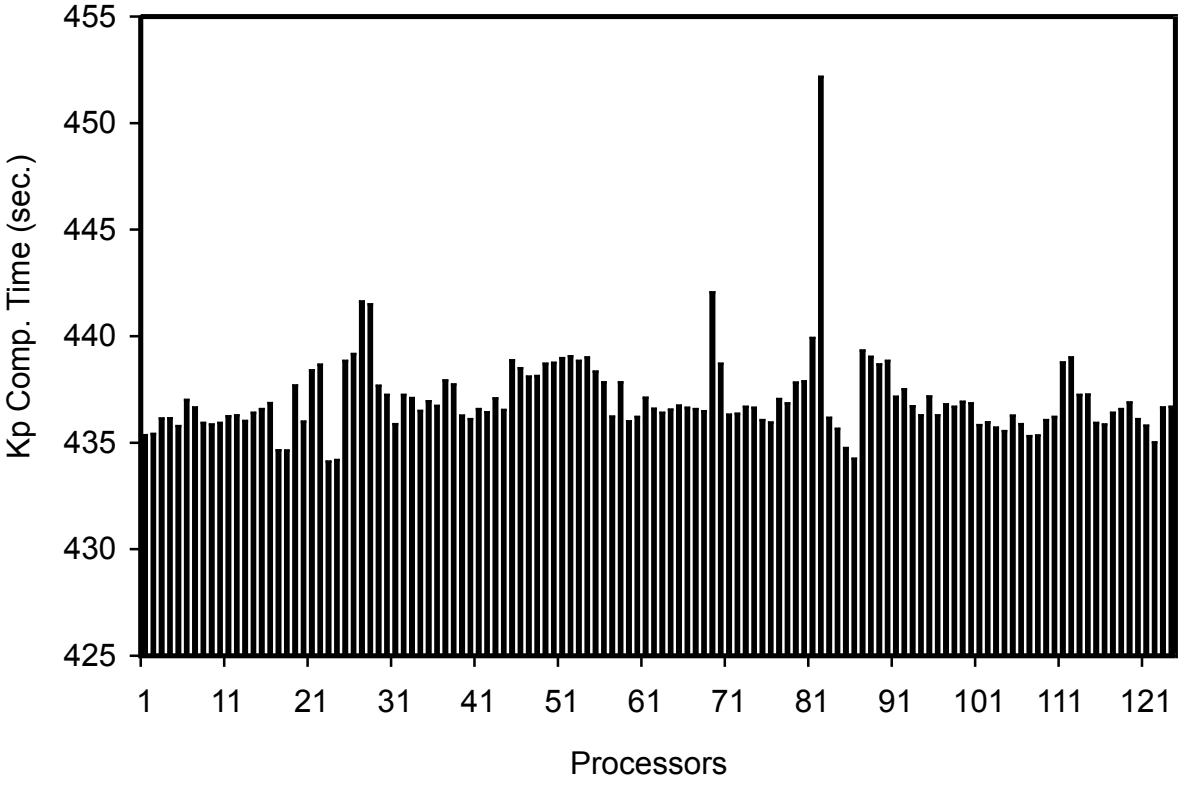


Figure 5.30: The *Lazy Processors* Phenomena for the SGI Origin2000 in a Scaled-Size Test Problem (32768 Elements per Processor) on 124 Processors.

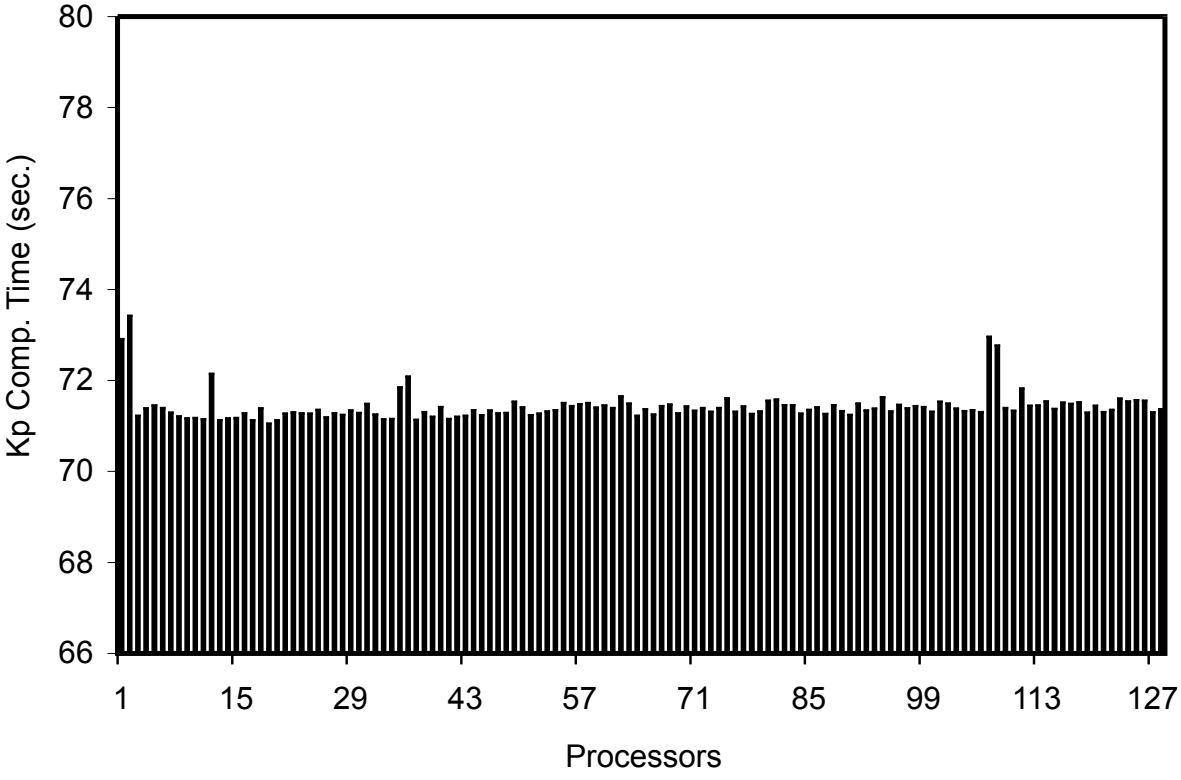


Figure 5.31: The Effect of Process Placement to Diminish the *Lazy Processors* for a Scaled-Size Test Problem (8192 Elements per Processor) on 128 Processors.

C H A P T E R 6

ORGANIZATION

The following describes ROCSOLID's modules and important subroutines. Inside each of these modules Elemtype indicates one of the element types implemented in ROCSOLID, i.e., b8_ale or b8_bbar or b8_me b8_ld (ROCSOLID 3.2).

ROCSOLID 3.2 can be used either with other ROC* modules through GENX framework or in the stand-alone mode.

Module communication: contains subroutines to generate the arrays needed for inter-processor communication and to perform the global scatter operation and the necessary communication between the processors.

Contains: communicate, comm_allocate_read_data, comm_arrays_calc

Module data_structure: contains subroutines to manipulate the data structure.

Contains: storage_allocate

Modute data_types: contains the data types used in the code.

Contains: data types

Module global_data: contains the global data used in the code.

Contains: global data

Module input: contains the subroutines for reading and writing the input data

Contains: input_data, read_mat, io_unit_open

Module mat_vec_ops: contains a library of generic subroutines operating on matrices and vectors.

Contains: A_times_p_ebe, m_times_p_ebe, c_times_p_ebe and scalar_product.

Module mesh_data: contains the subroutines for generating the data on each mesh.

Contains: mesh_data_generate, force_initialize, InterfaceSolidsMapBuild, force_calc, LumpedMassCalc, mesh_array_calc and equ_num.

Module mpi_include: contains the MPI include line.

Module output: contains the subroutines for writing out the results on a given mesh.

Contains: output_results, mesh_patran_out, write_input_out, mesh_calc_out, mesh_motion_output_results, nodal_disp_out and patran_neutral.

Module resource_calc: contains subroutines for timing and memory usage calculation.

Contains: timer and mem_calc.

Module rocsolid: contains the main driver.

Module elemtype_A_diag_calc: contains the subroutine to calculate and assemble the inverse of diagonal part of system matrix.

Contains: elemtype_A_diag.

Module elemtype_Ap_calc: to perform [A]{p} calculations.

Contains: elemtype_Ap, elemtype_mp_calc and elemtype_cp_calc.

Module elemtype_calc: to calculate the element arrays.

Contains: elemtype_conn_calc, elemtype_assem_calc, elemtype_load_calc and elemtype_id_calc.

Module elemtype_element: to perform calculations for elemtype.

Contains: elemtype_k_diag_calc, elemtype_m_diag_calc, elemtype_c_diag_calc, elemtype_kp, elemtype_mp and elemtype_cp.

Module elemtype_gather_scatter: to perform gather-scatter operations.

Contains: elemtype_gather and elemtype_scatter.

Module elemtype_in: to read in data and allocate storage for elemtype.

Contains: elemtype_allocate_read_data.

Module elemtype_out: to write out data for elemtype.

Contains: elemtype_stress_strain_out, elemtype_write_neutral, elemtype_disp_build, elemtype_write_input_out, elemtype_calc_arrays_out and elemtype_id_rebuild.

Module elemtype_solve: contains subroutines to solve problems with elemtype.

Contains: elemtype_t_x_calc and elemtype_t_trans_r_calc.

Module elemtype_util: contains utility subroutines for elemtype.

Contains: elemtype_derivatives_calc.

Module elastic_mat: to calculate material dependent variables for elastic materials.

Contains: elastic_stress_calc.

Module j2_plasticity_mat: to calculate material dependent variables for J2 plasticity materials.

Contains: j2_plasticity_stress_calc.

Module ale_mod: to use the ALE formulation for solving problems with moving interface.

Contains: ale_steps, mesh_motion_solve_laplace, interface_position_read, solids_interface_coor_scatter, interface_tractions_read, SolidsTractionScatter.

Module arc_length_mod: to perform arc length continuation method to trace the equilibrium path.

Contains: arc_length_steps, predictor_solve, corrector_solve, stability_check and convergence_check.

Module newmark_mod: to integrate the equation of motion of the structure using the Newmark method.

Contains: newmark and newmark_steps.

Module newton_mod: to perform Newton iterations to trace the equilibrium path.

Contains: newton_steps and convergence_check.

Module nl_utilities: contains subroutines used in the nonlinear procedures.

Contains: initialize, residual_calc, converged_state_store, restore_converged_state, stress_recovery and elem_internal_force.

Module bicgstab_solver: to solve nonsymmetric linear systems using biconjugate gradient stabilized method.

Contains: bicgstab_solve.

Module mg_solver: contains the subroutines for full multigrid solution algorithm.

Contains: fmg_solve, mg_cycle, t_x_calc and t_trans_r_calc.

Module pcg_solver: contains the subroutines for preconditioned conjugate gradient solution method.

Contains: pcg_solve and pcg_relax.

Module preconditioners: contains the subroutines for preconditioning during conjugate gradient iterations.

Contains: precondition, preconditioner_calc and LumpedMassPreconditioner.

Module solver_mod: to call different solvers for solving a system of linear equations.

Contains: solve.

CHAPTER 7

ROCSOLID INTERFACE WITH ROCSTAR

In addition to the usual ROCSOLID input files, interface mesh data is also needed to specify the mapping information between the interface and the ROCSOLID database. This information is then used to generate the data required by ROCFACE. ROCFACE uses this data to perform interpolation and data transfer between ROCSOLID and the other physics module, e.g., ROCFLO.

The following information is registered with ROCCOM:

- Interface nodal coordinates
- Interface elements connectivities
- Interface communication map
- Interface nodal displacements (during solution)
- Interface nodal velocities (during solution)
- Interface number of nodes and elements

In GENX, interface pressure values are received from ROCCOM during the solution. These tractions are then scattered to the ROCSOLID database using the mapping arrays. Interface nodal velocities are used by ROCFLO or ROCFLU only for the first predictor-corrector cycle when no converged displacements are available from ROCSOLID. ROCFLO or ROCFLU use the solids interface displacements for the rest of the predictor-corrector cycles.

The capabilities of GENX is used for output and restart features. The HDF output files can then be used by ROCKETEER for visualization. The required information for output and restart are also registered with ROCCOM.

BIBLIOGRAPHY

[1] Bathe K. J., 1982. *Finite element procedures in engineering analysis*. Prentice Hall.

[2] Papadrakakis M., 1993. *Solving large-scale problems in mechanics: the development and application of computational solution methods*. New York: Wiley.

[3] 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.

[4] Farhat C., Sobh N. and Park K. C., 1990. Transient finite element computations on 65536 processors: the connection machine. *International Journal for Numerical Methods in Engineering*, 30, 27-55.

[5] I. Foster, 1995. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Reading: Addison Wesley.

[6] Gropp W., Lusk E. and Skjellum A., 1994. *Using MPI: portable parallel programming with the message passing interface*. The MIT Press.

[7] Crisfield, M. A., 1979. A faster modified Newton-Raphson iteration. *Computer Methods in Applied Mechanics and Engineering,* 20, 267-278.

[8] Dodds, R. H., 1987. Numerical techniques for plasticity computations in finite element analysis. *Computers and Structures*, 26, No 5, 767-779.

[9] Farhat C. 1991. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, 32, 1205-27.

[10] Farhat C., Crivelli L. 1994. A transient FETI methodology for large-scale parallel implicit computations in structural mechanics. *International Journal for Numerical Methods in Engineering*, 37, 1945-75.

[11] Farhat C., Chen P. 1995. A scalable Lagrange multiplier based domain decomposition method for time dependent problems. *International Journal for Numerical Methods in engineering*, 38, 3831-53.

[12] Hackbusch, W. 1985. *Multigrid methods and applications*, Spriger-Verlag, Berlin.

[13] Parsons, I.D. and Hall, J. F. 1990. The multigrid method in solid mechanics: part I – Algorithm description and behaviour. *International Journal for Numerical Methods in Engineering,* 29, 719-737.

[14] Parsons, I.D. and Hall, J. F. 1990. The multigrid method in solid mechanics: part II – Practical applications, *International Journal for Numerical Methods in Engineering,* 29, 739-753.

[15] Stuben K. and Trottenberg U., 1982. Multigrid methods: fundamental algorithms, model problem analysis and applications. *Multigrid Methods*, W. Hackbush and U. Trottenberg, eds., Lecture notes in mathematics 960, Springer, 1-176.

[16] Fish J., Pandheeradi M., and Belsky V., 1995. An effective multilevel solution scheme for large-scale nonlinear systems. *International Journal for Numerical Methods in engineering*, 38, 1597-1610.

[17] Crisfield, M. A., 1991. *Nonlinear finite element analysis of solids and structures, volume 1: Essentials*. John Wiley & Sons.

[18] Simo, J. C. and Taylor, R. L., 1985. Consistent tangent operators for rate-independent elasto-plasticity, *Computer Methods in Applied Mechanics and Engineering*, 48, 1101-1118.

[19] Newmark N. M., 1959. A method of computation for structural dynamics. *Journal of the Mechanics Division*, ASCE, 32, 67-94.

[20] Zienkiewicz O. C., Taylor R. L., 1989. *The finite element method, fourth edition volume 1: basic formulations and linear problems.* New York: McGraw Hill.

[21] Zienkiewicz O. C., Taylor R. L., 1991. *The finite element method, fourth edition volume 2: solid and fluid mechanics and non-linearity.* New York: McGraw Hill.

[22] Brandt, A., 1977. Multi-level adaptive solutions to boundary-value problems, *Mathematics of Computation*, 31, 333-390.

[23] Hughes, T. J., 1980. Generalization of selective integration procedures to anisotropic and nonlinear media. *International Journal for Numerical Methods in Engineering,* 15, 1413-18.

[24] Kacou S., Parsons I. D., 1993. A parallel multigrid method for history-dependent elastoplasticity computations. *Computer Methods in Applied Mechanics and Engineering*, 108, 1-21.

[25] Belytschko T. and Bindeman L. P., 1993. Assumed strain stabilization of the eight node hexahedral element. *Computer Methods in Applied Mechanics and Engineering*, 105, 225-260.

[26] Behie A. and Forsyth P. A., 1983. Multigrid solution of the pressure equation in reservoir simulation. *Society of Petroleum Engineers Journal* 23, 623-632.

[27] Karypis G., Kumar V., 1995. Analysis of multilevel graph partitioning. *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, 1, 658-77.

[28] Amdahl G., 1967. Validity of single processor approach to achieving large-scale computing capabilities. *Proceeding of the 1967 AFIPS Conference*, 30, 483, AFIPS Press.

[29] Papadrakakis M. 1997. *Parallel solution methods in computational mechanics*. New York: Wiley.

[30] Adams M. 1998. Heuristics for the automatic construction of coarse grids in multigrid solvers for finite element matrices. *Technical Report UCB//CSD-98-994*, University of California, Berkeley, 1998.

[31] Faulkner T. R., 1999. Performance implications of process and memory placement using a multi-level parallel programming model on the CRAY Origin2000. This document is available online at www.nas.nasa.gov/~faulkner.

[32] Origin 2000 & Onyx 2 performance tuning and optimization guide, Chapter 1 & 2. This document is available online through the IRIS InSight Online Documentation Viewer and at SGI's technical publication web site.

[33] Chakrabarty J., 1987. *Theory of plasticity*. New York: McGraw Hill.

[34] Hunsaker J. R., Vaughan D. K. and Stricklin J. A., 1976. A comparisonof the capability of four hardening rules to predict a material's plastic behavior. *Journal of Pressure Vessel Technology*, 66-74.

[35] XYZ Scientific Applications, Inc., 1999. *TrueGrid* user's manual.

[36] Golub G. H., Van Loan C. F., 1983. *Matrix computations*. Baltimore: The Johns Hopkins University Press.

[37] Shewchuk J. R., 1994. An introduction to the conjugate gradient method without the agonizing pain. CMU-CS-94-125. Pittsburgh: Carnegie Mellon University.

[38] Wessling P., 1992. *An introduction to multigrid methods*. New York: John Wiley and Sons.

[39] Farris C. and Misra M., 1988. Distributed algebraic multigrid for finite element computations. *Mathematical and Computer Modeling*, 27(8), 41-67.

[40] Farhat C., Chen P. and Stern P., 1994. Toward the ultimate iterative substructuring method: combined numerical and parallel scalability, and multiple load cases. *Computing Systems in Engineering*, 5(4-6), 337-50.

[41] Carey G. F. and Jiang B. N., 1986. Element-by-element linear and nonlinear solution schemes. *Communications in Applied Numerical Methods*, 2, 145-54.

[42] Hughes T. J. R. and Winget J. M., 1985. Solution algorithms for nonlinear transient heat conduction analysis employing element-by-element iterative strategies. *Computer Methods in Applied Mechanics and Engineering*, 52, 711-815.

[43] Noor-Omid B. and Parlett B. N., 1985. Element preconditioning using splitting techniques. *SIAM Journal of Scientific and Statistical Computing*, 6, 761-71.

[44] Topping B. H. V. and Khan A. I., 1996. *Parallel finite element computations*. Edinburgh: Saxe-Coburg Publications.

[45] Papadrakakis M., 1986. Accelerating vector iterations methods. *Journal of Applied Mechanics*, 53(2), 291-97.

[46] Kershaw D. S., 1987. The incomplete cholesky conjugate gradient method for the iterative solution of systems of linear equations. *Journal of Computational Physics*, 6, 43-65.

[47] Papadrakakis M. and Dracopoulos M. C., 1991. Improving the efficiency of preconditioning for iterative methods. *Computers and Structures*, 41(6), 1263-72.

[48] Khan A. I. and Topping B. H. V., 1996. Parallel finite element analysis using Jacobi-conditioned conjugate gradient algorithm. *Advances in Engineering Software*, 25, 309-19.

[49] Papadrakakis M. and Bitzarakis S., 1996. Domain decomposition PCG methods for serial and parallel processing. *Advances in Engineering Software*, 25, 91-307.

[50] Hughes T. J. R., Ferenez 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. *Computer Methods in Applied Mechanics and Engineering*, 61, 215-48.

[51] Axelsson O., Carey G. and Lindskog G., 1989. On a class of preconditioned iterative methods on parallel computers. *International Journal for Numerical Methods in Engineering*, 27, 637-54.

[52] King R. B. and Sonnad V., 1987. Implementation of an element-by-element solution algorithm for the finite element method on a coarse-grained parallel computer. *Computer Methods in Applied Mechanics and Engineering*, 65, 47-59.

[53] El Attar M., 1992. Finite element analysis on distributed memory architectures. *Applied Mathematics and Computation*, 52, 309-16.

[54] Concus P., Golub G. H. and O'Leary D. P., 1965. A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations. *Sparse Matrix Computations*. J. R. Bunch and D. J. Rose, Eds. New York: Academic Press, 307-22.

[55] Kumar V., Gramma A., Gupta A. and Karypis G., 1994. *Introduction to parallel computing: design and analysis of algorithms*. Redwood City: Benjamin-Cummings.

[56] Koelbel C. H., Loveman D. B., Schreiber R. S., Steele Jr. G. L. and Zosel M. E., 1994. *The high performance Fortran handbook.* The MIT Press.

[57] Johan Z., 1992. *Data parallel finite element techniques for large-scale computational fluid dynamics*. PhD Thesis, Department of Mechanical Engineering, Stanford university.

[58] Simon H. D., 1991. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2, 135-148.

[59] Belytschko T., Plaskacz E. J., Kennedy J. M. and Greenwell D. L., 1990. Finite element analysis on the connection machine. *Computer Methods in Applied Mechanics and Engineering*, 81, 229-254.

[60] Tezduyar T., Aliabadi S., Behr M., Johnson A. and Mittal, S., 1993. Parallel finite element computation of 3D flows – Computation of moving boundaries and interfaces, and mesh update strategies. *University of Minnesota Supercomputer Institute Research report UMSI 93/65*.

[61] Barrett K. E., Butterfield D. M., Ellis S. E., Judd C. J. and Tabor J. H., 1985. Multigrid analysis of linear elastic stress problems. *Multigrid methods for integral and differential equations*, D. J. Paddon and H. Holstein, eds., The Institute of Mathematics and its Applications Conference Series 3, Clarendon Press, 263-82.

[62] Briggs W. L., 1987. *A multigrid tutorial*. SIAM.

[63] Kocvara M. and Mandel J., 1987. A multigrid method for three-dimensional elasticity and algebraic convergence estimates. *Applied Mathematics and Computation* 23, 121-35.

[64] Mahnken R., 1995. Newton-multigrid algorithm for elasto-plastic/viscoplastic problems. *Computational Mechanics* 15, 408-25.